SAT 2009 competitive events booklet: preliminary version

Organizers
SAT competition: Daniel Le Berre, Olivier Roussel, Laurent Simon
PB competition: Vasco Manquinho, Olivier Roussel
Max-SAT competition: Josep Argelich, Chu Min Li, Felip Manyà, Jordi Planes

September 30, 2009

Table of Contents

SAT Solver Descriptions

SAT Benchmarks Descriptions

PB Solver Descriptions

PB Benchmarks Descriptions

Max-SAT Solver Descriptions

Max-SAT Benchmarks Descriptions

Solver name BinMiniSat

Authors Kiyonori Taniguchi, Miyuki Koshimura, Hiroshi Fujita, and
Ryuzo Hasegawa

Track Minisat hack track

Description BinMiniSat is a SAT solver which prefers to select literals
in binary clauses as decision variables so as to produce unit propagation. We introduce a method to detect binary clauses by adding
an operation to the 2-literal watching function.

CirCUs 2.0 - SAT Competition 2009 Edition[*]

Hyojung Han[1], HoonSang Jin[2], Hyondeuk Kim[1], and Fabio Somenzi[1]

[1] University of Colorado at Boulder
[2] Cadence Design Systems
{Hyojung.Han,Hyondeuk.Kim,Fabio}@colorado.edu
{hsjin}@cadence.com

CirCUs is a SAT solver based on the DPLL procedure and conflict clause recording [7, 5, 2]. CirCUs includes most current popular techniques such as two-watched literals scheme for Boolean Constraint Propagation (BCP), activity-based decision heuristics, clause deletion strategies, restarting heuristics, and first UIP-based clause learning. In this submission we focus on the search for a balance between the ability of a technique to detect implications (the *deductive power* of [3]) and its cost.

This version of CirCUs adopts *strong conflict analysis* [4], one that often learns better clauses than those of the first *Unique Implication Point* (UIP). Just as different clauses may be derived from the same implication graph, different implication graphs may be obtained from the same sequence of decisions, depending on the order in which implications are propagated. Strong conflict analysis is a method to get more compact conflict clauses by scrutinizing the implication graph. It attempts to make future implication graphs simpler without expending effort on reshaping the current implication graph. The clauses generated by strong conflict analysis tend to be effective particularly in escaping *hot spots*, which are regions of the search space where the solver lingers for a long time. A new restarting heuristics added in this version can also help the SAT solver to handle hot spots. This new scheme is an ongoing work.

Detecting whether the resolvent of two clauses subsumes either operand is easy and inexpensive. Therefore, checking *on-the-fly* for subsumption can be added with almost no penalty to those operations of SAT solvers that are based on resolution. This detection is used to improve three stages of CirCUs: variable elimination, clause distillation, and conflict analysis. The idea and applications of the on-the-fly subsumption check is proposed in our full paper submitted to SAT 2009.

Conflict analysis in CirCUs is extended to keep checking the subsumption condition whenever a new resolvent is produced. In conflict analysis, if the new resolvent contains fewer literals than one of its operands, and the operand exists in the clause database, the operand is strengthened by removing the pivot variable. Then, the strengthened operand is established as the new resolvent. When both operands are subsumed, only one of them is selected to survive, and the other is deleted. Only direct antecedents can be strengthened in resolution steps. At the end of the resolution step, if the final resolvent containing the UIP is identified as an existing clause, the conflict analysis algorithm refrains from adding a new conflict clause to the clause database. Whether a new conflict clause is added or not, the DPLL procedure backtracks to the level returned by conflict analysis, and asserts the clause finally learned from the latest conflict. On-

---

the-fly subsumption check can be applied to strong conflict analysis as well as regular conflict analysis.

Simplifying the CNF clauses leads to fast BCP and to earlier detection of conflicts in practice. CirCUs is incremented with preprocessing based on subsumption, variable elimination [1, 6], and distillation [3]. Resolution is the main operation in preprocessing. Therefore, on-the-fly susbsumption is also applied to the preprocessors for variable elimination and clause distillation. During eliminating variables, at each resolution operation, we can check if one of the operands is subsumed by the resolvent, like the on-the-fly subsumption check in conflict analysis. A clause can be simplified by on-the-fly subsumption, regardless of whether the variable is eventually eliminated. Conflict analysis in clause distillation also performs resolutions steps as conflict analysis in DPLL. Therefore we can increase efficiency in the distillation procedure by using on-the-fly simplification.

CirCUs is written in C. An ANSI C compiler and GNU make are required to build it. It is supposed to be compiled for 32-bit machines.

References

[1] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[2] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 142–149, Paris, France, Mar. 2002.

[3] H. Han and F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *Proceedings of the Design Automation Conference*, pages 582–587, San Diego, CA, June 2007.

[4] H. Jin and F. Somenzi. Strong conflict analysis for propositional satisfiability. In *Design, Automation and Test in Europe (DATE'06)*, pages 818–823, Munich, Germany, Mar. 2006.

[5] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[6] URL: http://www.cs.chalmers.se/Cs/Research/ FormalMethods/MiniSat/MiniSat.html.

[7] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

clasp: A Conflict-Driven Answer Set Solver

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub[*]

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482
Potsdam, Germany

clasp combines the high-level modeling capacities of Answer Set Programming (ASP; [1]) with state-of-the-art techniques from the area of Boolean constraint solving. Hence, it is originally designed and optimized for conflict-driven ASP solving [2–4]. However, given the proximity of ASP to SAT, *clasp* can also deal with formulas in CNF via an additional DIMACS frontend. As such, it can be viewed as a chaff-type Boolean constraint solver [5].

From a technical perspective, *clasp* is implemented in C++ and was submitted in its source code, which is publicly available at [6]. Formulas in CNF are pre-processed internally relying on concepts borrowed from SatElite [7] as used in MiniSat 2.0 but implemented in a rather different way. Most innovative algorithms and data structures aim at ASP solving and are thus outside the scope of SAT solving. Among them, *clasp* supports further ASP-oriented preprocessing techniques and native support of aggregates, such as cardinality and weight constraints [8].

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press (2007) 386–392
3. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. [9] 260–265
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. [9] 136–148
5. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01), ACM Press (2001) 530–535
6. http://www.cs.uni-potsdam.de/clasp
7. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: Proceedings of the Eigth International Conference on Theory and Applications of Satisfiability Testing (SAT'05). Volume 3569 of Lecture Notes in Computer Science., Springer-Verlag (2005) 61–75
8. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
9. Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Volume 4483 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2007)

---

[*] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

GLUCOSE: a solver that predicts learnt clauses quality[*].

Gilles Audemard
Univ. Lille-Nord de France
CRIL/CNRS UMR8188
Lens, F-62307
audemard@cril.fr

Laurent Simon
Univ. Paris-Sud
LRI/CNRS UMR 8623 / INRIA Saclay
Orsay, F-91405
simon@lri.fr

Abstract

GLUCOSE is based on a new scoring scheme for the clause learning mechanism, based on the paper [Audemard and Simon, 2009]. This short competition report summarizes the techniques embedded in the competition 09 version of GLUCOSE. Solver's name comes from *glue clauses*, a particular kind of clauses that GLUCOSE detects and preserves during search. The web page for GLUCOSE is http://www.lri.fr/~simon/glucose.

1 Introduction

Since the breakthrough of Chaff [Moskewicz *et al.*, 2001], a lot of effort has been made in the design of efficient Boolean Constraint Propagation (BCP), the heart of all modern SAT solvers. The global idea is to reach conflicts as soon as possible, but with no direct guarantees on the new learnt clause usefulness. Following the successful idea of the Variable State Independent Decaying Sum (VSIDS) heuristics, which favours variables that were often – and recently – used in conflict analysis, future learnt clause usefulness is supposed to be related to its activity in recent conflicts analyses.

In this context, detecting what is a good learnt clause in advance was still considered as a challenge, and from first importance: deleting useful clauses can be dramatic in practice. To prevent this, solvers have to let the maximum number of learnt clauses grow exponentially. On very hard benchmarks, CDCL solvers hangs-up for memory problems and, even if they don't, their greedy learning scheme deteriorates their heart: BCP performances.

If a lot of effort has been put in designing smart restart policies, only a few work targetted smart clause database management. In [Audemard and Simon, 2009], we show that a very simple static measure on clauses can dramatically improves the performances of MINISAT [Eén and Sörensson, 2003], the solver on which GLUCOSE is based. GLUCOSE is based on the last plubicly available version of MINISAT.

2 Identifying good clauses in advance

During search, each decision is often followed by a large number of unit propagations. All literals from the same level are what we call "blocks" of literals in the later. Intuitively, at the semantic level, there is a chance that they are linked with each other by direct dependencies. Our idea is that a good learning schema should add explicit links between independent blocks of propagated (or decision) literals. If the solver stays in the same search space, such a clause will probably help reducing the number of next decision levels in the remaining computation. Staying in the same search space is one of the recents behaviors of CDCL solvers, due to phase-saving [Pipatsrisawat and Darwiche, 2007] and rapid restarts.

Definition 1 (Literals Blocks Distance (LBD)) *Given a clause C, and a partition of its literals into n subsets according to the current assignment, s.t. literals are partitioned w.r.t their decision level. The LBD of C is exactly n.*

From a practical point of view, we compute and store the LBD score of each learnt clause when it is produced. This measure is thus static, even if update it during search (LBD score of a clause can be re-computed when the clause is used in unit-propagation). Intuitively, it is easy to understand the importance of learnt clauses of LBD 2: they only contain one variable of the last decision level (they are FUIP), and, later, this variable will be "glued" with the block of literals propagated above, no matter the size of the clause. We suspect all those clauses to be very important during search, and we give them a special name: "Glue Clauses".

From a theoretical point of view, it is interesting to notice that LBD of FUIP learnt clauses is optimal over all other possible UIP learning schemas [Jabbour and Sais, 2008]. If GLUCOSE efficiency in the 2009 competition clearly demonstrates our scoring accuracy, this theoretical result will cast a good explanation of the efficiency of First UIP over all other UIP mechanisms: FUIP efficiency would then be partly explained by its ability to produce clauses of small LBD (in addition to its optimality in the size of the backjump [Jabbour and Sais, 2008]).

Property 1 (Optimality of LBD for FUIP Clauses) *Given a conflict graph, any First UIP asserting clause has the smallest LBD value over all other UIPs.*

3 Agressive clauses deletion

Despite its crucial importance, only a few works focus on the learnt clause database management. However, keeping too many clauses may decrease solver BCP performances, but deleting too many clauses may decrease the overall learning benefit. Nowadays, the state of the art is to let the clause database size follow a geometric progression (with a small common ratio of 1.1 for instance in MINISAT). Each time the limit is reached, the solver deletes at most half of the clauses, depending on their score (however, binary clauses are never deleted).

Because we wanted to really emphasize our learning schema, we propose to reduce drastically the number of learnt clauses in the database. We chose the following strategy: every $20000 + 500 * x$ conflicts, we remove at most half of the learnt clause database where x is the number of times this action was already performed before. It can be noticed that this strategy does not take the initial size of the formula into account (as opposite of most current solvers). Our first hope was only to demonstrate that even a static measure on clause usefulness could be *as efficient* as the past-activity one. However, our results were far beyond our initial hope.

4 Other embedded techniques

The GLUCOSE version submitted to the contest differs from the one used in [Audemard and Simon, 2009] on some very particular points that we review here.

First of all, we upgraded MINISAT for a special handling of binary clauses. We also used a phase-caching schema for variable polarity [Pipatsrisawat and Darwiche, 2007].

4.1 Restarts

One of the best restart strategy in CDCL solver is based on the Luby series, which exactly means that "we don't know when to restart". Recently, first steps have been done to find a dynamic (computed during the search) restart strategy [Biere, 2008; Ryvchin and Strichman, 2008]. Our restart strategy is based on the decreasing of the number of decisions levels during search. If the decreasing is stalling, then a restart is triggered. This is done by a moving average over the last 100 conflicts. If 0.7 times this value is greater than the global average of the number of decision levels, then a restart is forced (at least 100 conflicts are needed before any restart). This strategy should encourage the solver to keep searching at the right place, and to escape from wrong places.

4.2 Reward good variables

The state-of-the-art VSIDS [Moskewicz *et al.*, 2001] heuristic bumps all variables which participated to the resolution steps conducting to the assertive clause. This heuristic favors variables that are often and recently used in conflict analysis. Since we want to help the solver to generate clauses with small LBD values, we propose to reward a second time variables that help to obtain such clauses.

We bump once again all variables from the last decision level, which were used in conflict analysis, but which were propagated by a clause of small LBD (smaller than the new learnt clause).

5 Conclusion

GLUCOSE is based on a relatively old version of MINISAT, which is very well known, and well established. Only a relatively small amount of changes has been made in MINISAT: we tried to reduce the modifications as much as possible in order to identify what are the crucial techniques to add to a 2006 winning code to win the UNSAT category of the 2009 SAT competition. A lot of improvements can be awaited by more up-to-date datastructures (like the use of blocked literals).

References

[Audemard and Simon, 2009] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *proceedings of IJCAI*, 2009.

[Biere, 2008] A. Biere. Adaptive restart strategies for conflict driven sat solvers. In *proceedings of SAT*, pages 28–33, 2008.

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. An extensible SAT-solver. In *proceedings of SAT*, pages 502–518, 2003.

[Jabbour and Sais, 2008] S. Jabbour and L. Sais. personnal communication, February 2008.

[Moskewicz *et al.*, 2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *proceedings of DAC*, pages 530–535, 2001.

[Pipatsrisawat and Darwiche, 2007] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *proceedings of SAT*, pages 294–299, 2007.

[Ryvchin and Strichman, 2008] V. Ryvchin and O. Strichman. Local restarts. In *proceedings of SAT*, pages 271–276, 2008.

gNovelty$^+$ (v. 2)

Duc Nghia Pham[1] and Charles Gretton[2]

[1]ATOMIC Project, Queensland Research Lab, NICTA Ltd., Australia
duc-nghia.pham@nicta.com.au
[2]School of Computer Science, University of Birmingham, UK
c.gretton@cs.bham.ac.uk

1 Preface

We describe an enhanced version of gNovelty$^+$ [Pham *et al.*, 2008], a stochastic local search (SLS) procedure for finding satisfying models of satisfiable propositional CNF formulae. Version 1 of gNovelty$^+$ was a Gold Medal winner in the random category of the 2007 SAT competition. In this version 2, we implemented an explicit mechanism to handle implications derived from binary clauses. We also made this version of gNovelty$^+$ multithreaded.

This abstract is organised as follows: we provide an overview of gNovelty+ and then describe the new enhancements added to this new version. Finally, we describe the technical settings of its contest implementation.

2 gNovelty$^+$

The original version of gNovelty$^+$ draws on the features of two other WalkSAT family algorithms: R+AdaptNovelty$^+$ [Anbulagan *et al.*, 2005] and G^{2}WSAT [Li and Huang, 2005], while also successfully employing a hybrid clause weighting heuristic based on the features of two dynamic local search (DLS) algorithms: PAWS [Thornton *et al.*, 2004] and (R)SAPS [Hutter *et al.*, 2002]. This version is sketched out in Algorithm 1 and depicted diagrammatically in Figure 2.

Algorithm 1 gNovelty$^+$(F)

1: **for** $try = 1$ to $maxTries$ **do**
2:     initialise the weight of each clause to 1;
3:     randomly generate an assignment A;
4:     **for** $step = 1$ to $maxSteps$ **do**
5:         **if** A satisfies F **then**
6:             **return** A as the solution;
7:         **else**
8:             **if** within a walking probability wp **then**
9:                 randomly select a variable x that appears in a false clause;
10:             **else if** there exist *promising* variables **then**
11:                 *greedily* select a promising variable x, breaking tie by selecting the least recently flipped one;
12:             **else**
13:                 select a variable x according to the *weighted AdaptNovelty* heuristic;
14:                 update the weights of false clauses;
15:                 with probability sp smooth the weights of all clauses;
16:             **end if**
17:             update A with the flipped value of x;
18:         **end if**
19:     **end for**
20: **end for**
21: **return** 'no solution found';

At every search step, gNovelty$^+$ selects the most promising variable that is also the least recently flipped, based on our *weighted* objective function. Our objective is to minimise the sum of weights of all false clauses. If no such promising variable exists, the next variable is selected using a heuristic based on AdaptNovelty that utilises the *weighted* objective function. After the Novelty step, gNovelty$^+$ increase the weights of all current false clauses by 1.[1] In order to keep the control of the level of greediness of the search flexible, we also incorporates into gNovelty$^+$ a new linear version of the probabilistic weight smoothing from SAPS [Hutter *et al.*, 2002]. Every time gNovelty$^+$ updates its clause weights, with a *smoothing probability sp* the weights of all *weighted clauses* (a clause is weighted if its weight is greater than one) are subject to a reduction of 1. Finally, we also added a probabilistic walk heuristic (i.e. the *plus* heuristic from Hoos [1999]) to gNovelty$^+$ to further improve the balance between the level of randomness (resp. greediness) of the search.

3 gNovelty$^+$ version 2

DPLL-based SAT procedures have for some time exploited the presence of binary clauses – Both present in the problem at hand, and derived during search [Gelder and Tsuji, 1995; Zheng and Stuckey, 2002]. In particular, they incorporate linear time algorithms based on graph theory or unit propagation into the DPLL search. In developing gNovelty$^+$ V.2, one of our objectives was to exploit 2SAT procedures in the setting of local search. In particular, gNovelty$^+$ V2 cannot make local moves that violate constraints implied by the 2SAT fragment of the problem at hand. Moreover, we maintain consistency of the current valuation with the 2SAT fragment. Finally, we heuristically punish the local search – via the DLS clause weighting scheme – for failing to satisfy clauses that contain literals occurring in the 2SAT fragment.

4 Contest Implementation

For the 2009 SAT competition, the parameter sp of gNovelty$^+$ is fixed at .4 for the 3-SAT problems and at 1 for other problems. Also, wp was always set to 0.01.

---

[1]We decided to use the additive weight increase at each local minimum as it is cheaper to maintain than its counterpart multiplicative weighting [Thornton *et al.*, 2004].
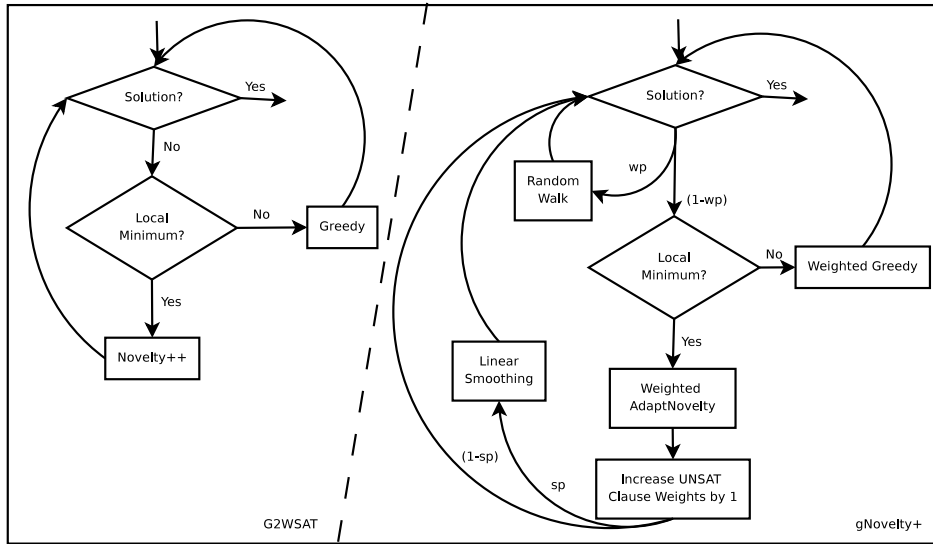
Figure 1: Flow-chart comparison between the two procedures G²WSAT and gNovelty⁺.

We submitted 6 variants of gNovelty⁺ to the competition. These differ in whether they: (1) exploit the presence of binary clauses, (2) support multithreading, and (3) incorporate a hash-based *tabu*. The details of our submission is summarised in Table 1. Threaded versions of the respective solvers run multiple concurrent and independent searches. The number of threads is an algorithm parameter, thus is not dynamically decided. The *tabu* is a simple mechanism we have implemented to escape local minima quickly in structured problems.

	2SAT	Multithreaded	Tabu
gNovelty⁺	no	no	no
gNovelty⁺-T	no	yes	no
gNovelty⁺-V.2	yes	no	no
gNovelty⁺-V.2-T	yes	yes	no
gNovelty⁺-V.2-H	yes	no	yes
gNovelty⁺-V.2-T-H	yes	yes	yes

Table 1: Versions of gNovelty⁺ submitted at the 2009 SAT competition.

Acknowledgements

References

[Anbulagan *et al.*, 2005] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Old resolution meets modern SLS. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 354–359, 2005.

[Gelder and Tsuji, 1995] Allen Van Gelder and Yumi K. Tsuji. Satisfiability testing with more reasoning and less guessing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:559–586, 1995.

[Hoos, 1999] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666, 1999.

[Hutter *et al.*, 2002] Frank Hutter, Dave A.D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, pages 233–248, 2002.

[Li and Huang, 2005] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, pages 158–172, 2005.

[Pham *et al.*, 2008] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Combining adaptive and dynamic local search for satisfiability. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:149–172, 2008.

[Thornton *et al.*, 2004] John R. Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-04)*, pages 191–196, 2004.

[Zheng and Stuckey, 2002] Lei Zheng and Peter J. Stuckey. Improving sat using 2sat. In *Proceedings of the 25th Australasian Computer Science Conference*, pages 331–340, 2002.

$Hybrid2$: A Local Search Algorithm that Switches Between Two Heuristics

Wanxia Wei[1] and Chu Min Li[2] and Harry Zhang[1]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada, E3B 5A3
{wanxia.wei,hzhang}@unb.ca
[2] MIS, Université de Picardie Jules Verne 33 Rue St. Leu, 80039 Amiens Cedex 01, France
chu-min.li@u-picardie.fr

1 Review of Algorithms $adaptG^2WSAT_P$, VW, $Hybrid$, and $adaptG^2WSAT+$

The local search algorithm $adaptG^2WSAT_P$ [2, 3] flips the promising decreasing variable with the largest computed promising score if there are promising decreasing variables. It selects a variable to flip from a randomly chosen unsatisfied clause using heuristic $Novelty$++$_P$ [2, 3] otherwise.

The local search algorithm VW [5] introduces variable weighting. This algorithm initializes the weight of a variable x, $variable\_weight[x]$, to 0 and updates and smoothes $variable\_weight[x]$ each time x is flipped, using the following equation:

$$variable\_weight[x] = (1 - s)(variable\_weight[x] + 1) + s \times t \qquad (1)$$

where s is a parameter and $0 \leq s \leq 1$, and t denotes the time when x is flipped. This algorithm uses a unique variable selection rule. We call this rule the *low variable weight favoring rule*. If a randomly selected unsatisfied clause c contains freebie variables,[3] VW randomly flips one of them. Otherwise, with probability p, it flips a variable chosen randomly from c, and with probability $1 - p$, it flips a variable in c according to the low variable weight favoring rule.

A switching criterion, namely the evenness or unevenness of the distribution of variable weights, was proposed in [6, 7]. It is defined in [6, 7] as follows. Assume that γ is a number. If the maximum variable weight is at least γ times as high as the average variable weight, the distribution of variable weights is considered *uneven*, and the step is called *an uneven step* in terms of variable weights. Otherwise, the distribution of variable weights is considered *even*, and the step is called *an even step* in terms of variable weights. An uneven or an even distribution of variable weights is used as a means to determine whether a search is undiversified in a step in terms of variable weights.

$Hybrid$ [6, 7] switches between *heuristic $adaptG^2WSAT_P$* and *heuristic VW* according to the above switching criterion.[4] More precisely, in each search step, $Hybrid$ chooses a variable to flip according to *heuristic VW* if the distribution of variable weights is uneven, and selects a variable to flip according to *heuristic $adaptG^2WSAT_P$* otherwise. In $Hybrid$, the default value of parameter γ is 10.0. $Hybrid$ updates variable weights using Formula 1, and parameter s in this formula is fixed to 0.0.

The local search algorithm $adaptG^2WSAT+$ was improved from $adaptG^2WSAT$ [2–4]. This new algorithm is different from $adaptG^2WSAT$ in two respects. First, when there is no promising decreasing variable, $adaptG^2WSAT+$ uses $Novelty+$ instead of $Novelty$++ [1], to select a variable to flip from a randomly chosen unsatisfied clause c. Second, when promising decreasing variables exist, $adaptG^2WSAT+$ no longer flips the promising deceasing variable with the highest score

---

[3] A freebie variable is a variable with a break of 0.

[4] The ways in which algorithms $adaptG^2WSAT_P$ [3] and VW [5] select a variable to flip, are referred to as *heuristic $adaptG^2WSAT_P$* and *heuristic VW*, respectively [6, 7].

among all promising decreasing variables, but chooses the least recently flipped promising decreasing variable among all promising decreasing variables to flip. We refer to the way in which algorithm $adaptG^2WSAT+$ selects a variable to flip, as *heuristic adaptG^{2}WSAT+*.

2   Local Search Algorithm $Hybrid2$

$Hybrid$ can solve a broad range of instances. In this algorithm, parameters γ is fixed for a broad range of instances and is not optimized for specific types of instance. We improve $Hybrid$ for random instances and obtain algorithm $Hybrid2$, which switches between *heuristic adaptG^{2}WSAT+* and *heuristic VW*. This algorithm is described in Fig. 1. In $Hybrid2$, γ is set to 1.025. In this algorithm, variable weights are updated using Formula 1, and parameter s in this formula is adjusted during the search in the same way as in VW ($s > 0.0$). That is, unlike $Hybrid$, $Hybrid2$ smoothes variable weights.

Algorithm: $Hybrid2$(SAT-formula $\mathcal{F}$)

1: $A \leftarrow$ randomly generated truth assignment;
2: **for** each variable x **do** initialize $flip\_time[x]$ and $variable\_weight[x]$ to 0;
3: initialize p, wp, max_weight, and ave_weight to 0; initialize s to 0.1;
4: store promising decreasing variables in stack *DecVar*;
5: **for** *flip*=1 **to** *Maxsteps* **do**
6:     **if** A satisfies $\mathcal{F}$ **then return** A;
7:     **if** $max\_weight \geq \gamma \times ave\_weight$
8:     **then** $y \leftarrow$ heuristic $VW(p)$;
9:     **else** $y \leftarrow$ heuristic $adaptG^2WSAT+(p, wp)$
10:     $A \leftarrow A$ with y flipped; adapt p and wp; adjust s;
11:     update $flip\_time[y]$, $variable\_weight[y]$, max_weight, ave_weight, and *DecVar*;
12: **return** Solution not found;

Fig. 1. Algorithm $Hybrid2$

References

1. C. M. Li and W. Q. Huang. Diversification and Determinism in Local Search for Satisfiability. In *Proceedings of SAT-2005*, pages 158–172. Springer, LNCS 3569, 2005.
2. C. M. Li, W. Wei, and H. Zhang. Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In *Proceedings of LSCS-2006*, pages 2–16, 2006.
3. C. M. Li, W. Wei, and H. Zhang. Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, *Trends in Constraint Programming*, chapter 14, pages 261–267. ISTE, 2007.
4. C. M. Li, W. Wei, and H. Zhang. Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In *Proceedings of SAT-2007*, pages 121–133. Springer, LNCS 4501, 2007.
5. S. Prestwich. Random Walk with Continuously Smoothed Variable Weights. In *Proceedings of SAT-2005*, pages 203–215. Springer, LNCS 3569, 2005.
6. W. Wei, C. M. Li, and H. Zhang. Criterion for Itensification and Diversification in Local Search for SAT. In *Proceedings of LSCS-2007*, pages 2–16, 2007.
7. W. Wei, C. M. Li, and H. Zhang. Criterion for Intensification and Diversification in Local Search for SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:219–237, June 2008.

hybridGM

Adrian Balint, Michael Henn, and Oliver Gableske

University of Ulm
Institute of Theoretical Computer Science
89069 Ulm, Germany
`adrian.balint,michael.henn,oliver.gableske@uni-ulm.de`

hybridGM is a hybrid SAT solver based on the SLS solver *gNovelty+* and the DPLL Solver *march_ks* which are combined with the help of search space partitions. First we give a brief description of the components of hybridGM.

1. *gNovelty+*:
   gNovelty+ is itself a hybrid SLS solver [2], combining the solvers *G2WSAT*, *PAWS* and *AdaptNovelty+*. The basis of the solver is *G2WSAT*, which computes the scores for all variables in the unsatisfied clauses before each flip and selects the most promising one. *gNovelty+* uses an additive clause weighting scheme like PAWS to better guide the search within *G2WSAT*. If the solver gets stuck in a local minimum (i.e: there are no promising variables) it uses *AdaptNovelty+* [3], which selects a random unsatisfied clause and flips the best or second best variable in this clause.

2. *march_ks*:
   march_ks is a look-ahead DPLL solver, that utilizes equivalency reasoning, adaptive double look-ahead and distribution jumping [4, 5]. It was chosen for the hybridization, because it is currently the fastest solver for unsatisfiable formulas.

3. *Search space partition*:
   Given a SLS solver S and a complete assignment α reached at step k by the SLS solver during its search, then a *search space partition* (SSP) of size r is defined as a partial assignment α_p built in the following way: First $\alpha_p = \alpha$, then the variables that are flipped by the SLS solver at step $k+i$ and $k-i$ get unassigned in α_p for $i = 1, 2, ..$ until the number of unassigned variables is $\geq r$.

hybridGM uses *gNovelty+* as its basis. When a good local minimum is reached (number of unsatisfied clauses $\leq barrier$), it starts to build up a search space partition. When the size of the search space partition reaches half the number of variables, *march_ks* is called to solve the problem, starting with the partial assignment α_p. A solution is found if *march_ks* can extend the partial assignment to a complete assignment. Otherwise, the search of *gNovelty+* continues. If the unassigned variables in α_p result in unary conflicts detected by *march_ks* with unit propagation, the size of a search space partition is increased by $n/20$ where n is the number of variables.

The intuition behind this concept is based on the observations made in [6]. Zhang showed that there is a correlation between the hamming distance of local minimum and their nearest solution and the quality of the local minimum. So, if *gNovelty+* finds a good local minimum, there will be a solution in the near hamming neighborhood of the local minimum very likely. But because this neighborhood is far too large to be completely searched we looked for a way to find another neighborhood relation. So we came up with the concept of search space partitions. Preliminary tests showed that this approach is promising. *hybridGM* was able to solve more 3-SAT instances in less time than the original *gNovelty+*.

gNovelty+ was modified slightly compared to the SAT 2007 version [1]. The smoothing probability for the *PAWS* component was set to $sp = 0.33$. When solving large 5-SAT or 7-SAT instances the *G2WSAT* component and clause weighting is completely turned off and only the *AdaptNovelty+* part is active. Smaller 5-SAT and 7-SAT instances are still solved by the G2WSAT component. The number of unsatisfied clauses *barrier* is always one.

We propose 3 solvers for the competition:

1. *hybridGM*: calls *march_ks* only for 3-SAT. For 5-SAT and 7-SAT it uses *AdaptG2WSAT*.
2. *hybridGM3*: calls *march_ks* only for 3-SAT. For 5-SAT and 7-SAT it uses *AdaptNovelty+* if the number of clauses is larger than 10000.
3. *hybridGM7*: always tries to build search space partitions and then to call *march_ks*. For 5-SAT and 7-SAT it acts like *hybridGM3*.

References

1. Pham, D. N., Gretton, C.: *gNovelty+*. Available at the SAT Competition homepage: http://www.satcompetition.org/2007/gNovelty+.pdf (2007)
2. Pham, D. N., Thornton, J. R., Gretton, C., Sattar, A.: Advances in Local Search for Satisfiability. Australian Conference on Artificial Intelligence 2007: 213-222 (2007)
3. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of AAAI 2002, 635–660 (2002)
4. Heule, M., van Maaren, H.: Whose side are you on? Finding solutions in a biased search-tree. Proceedings of Guangzhou Symposium on Satisfiability In Logic-Based Modeling, 82–89 (2006)
5. Heule, M., van Maaren, H.: Effective incorporation of Double Look-Ahead Procedures. Lecture Notes in Computer Science. 4501, 258–271 (2007)
6. Zhang, Weixiong: Configuration landscape analysis and backbone local search. Part I: Satisfiability and maximum satisfiability. Artificial Intelligence Volume 158, 1–26 (2004)

HydraSAT 2009.3 Solver Description

Christoph Baldow, Friedrich Gräter, Steffen Hölldobler,
Norbert Manthey, Max Seelemann, Peter Steinke,
Christoph Wernhard, Konrad Winkler, Erik Zenker

sat-praktikum@janeway.inf.tu-dresden.de

Technische Universität Dresden

1 Introduction

HydraSAT is a CDCL SAT solver which can be applied to satisfiable as well
as unsatisfiable formulas. The original motivation for writing the system was to
learn about techniques of advanced DPLL-based SAT systems. So it started out
basically as a re-implementation of MiniSat [3] that was done from scratch, with
emphasis on a component-based design which facilitates extension by further
techniques.

2 Features

Component-Based Architecture. HydraSAT includes alternate implementations
of core components such as unit propagation, decision heuristics or maintenance
of learned clauses, which are related internally through common interfaces and
a notification mechanism. Which components get activated is controlled at run-
time by configuration parameters. Alternate implementations of low-level data
structures such as representations of clauses or assignments can be selected with
compilation flags.

Detection of Multiple Conflicts. Optionally, unit propagation can be continued
after a conflict has been found, such that multiple conflicts and thus multiple
candidate lemma clauses can be derived. Depending on the solver configuration,
some of these are asserted as learned clauses. For example, a single lemma clause
which effects backjumping to the lowermost decision level.

Context Lemmas. HydraSAT includes a deletion scheme for learned clauses,
where deletion is triggered by backjumping: A learned clause is deleted, if the
backjumping step effects that the number of its unasserted literals becomes larger
than a certain threshold.

Representation of Low-Level Data Structures. The solver uses techniques from [1]
such as packed assignment representation, storage of the first literal of a clause
within the watched list, and split clause data structures to improve locality of
memory accesses. The exact choice and parametrization of these data structures
can be controlled with configuration options.

Preprocessing. The preprocessor of HydraSAT implements a portion of the simplifications employed in the 2007 contest version of MiniSat (see also [2]) in combination with probing techniques [4].

Competition Versions. Different variants of HydraSAT will be submitted to the SAT 2009 competition: The standard version, a *"fixed"* version that is hand optimized for a specific hard-coded component and parameter selection, and a *"multi"* version that invokes the standard version with different parameter settings, each with a timeout that is the fraction of the contest time limit.

Availability. The system is written in C++. The source code of the 2009 SAT competition version is publicly available for research purposes, in accord with the competition rules. We plan to release a revised and more extensively documented version for a wider public under the GPL license in Summer 2009.

References

1. G. Chu, A. Harwood, and P. J. Stuckey. Cache conscious data structures for Boolean satisfiability solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:99–120, 2009.
2. N. Eeen and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing: 8th International Conference, SAT 2005*, 2005.
3. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.
4. I. Lynce and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *ICTAI'03*, 2003.

IUT_BMB_SIM Preprocessor: System Description

Abdorrahim Bahrami, Seyed Rasoul Mousavi, Kiarash Bazargan

Isfahan University of Technology
March 22, 2009

1. Introduction

The IUT_BMB_SIM is a Preprocessor which converts a CNF formula to an equivalent but simpler CNF formula. It performs a simplification task using unary and binary clauses of the formula. The simplification task consists of three parts which are described next.

2. Resolution on binary Clauses

The first part of the simplification is to use the resolution rule on binary clauses to extract unary ones. If two binary clauses exist in the CNF formula in the forms $(l_i \vee l_j)$ and $(\bar{l}_i \vee l_j)$, a unary clause in the form (l_i) can be extracted from them. In this part of preprocessing, all binary clauses in the given formula are checked to find every pair from which a unary clause can be extracted.

For speed up, this part is performed at the same time the CNF formula is being read and prepared into the data structures.

3. Unit-propagation

All the unary clauses are stored in a queue while the CNF formula is being read. This includes the new unary clauses extracted from the binary ones. Then, a unit-propagation task is performed to further simplify the formula and also to eliminate all the unary clauses.

4. Finding Strongly Connected Component

After eliminating all unary and binary clauses in the forms $(l_i \vee l_j)$ and $(\bar{l}_i \vee l_j)$, an Implication Graph with all of the remaining binary clauses are constructed. As in 2-SAT, each strongly-connected component can be used for the purpose of binary equivalence of some literals. For example, if a strongly-connected component with three nodes l_i, l_j and l_k exist in the implication graph, then there will be paths from l_i to l_j and from l_j to l_k and from l_i to l_k, and vice versa. Recall that in an implication graph, if a literal l_i has a path to another literal l_j then l_i implies l_j or if $l_i = true$ then $l_j = true$ either. So if a strongly-connected component with three nodes l_i, l_j and l_k exist in an implication graph, then all these literals will be equivalent. So l_j and l_k can be replaced with l_i, and also $\bar{l}_j$ and $\bar{l}_k$ can be replaced with l_i. Therefore, by finding strongly-connected components, some variables are eliminated and the CNF formula is simplified and is expected to be solved in less time than the original one.

IUT_BMB_SAT SAT Solver: System Description

Abdorrahim Bahrami, Seyed Rasoul Mousavi, Kiarash Bazargan

Isfahan University of Technology
March 22, 2009

1. Introduction
The IUT_BMB_SAT is a two-phase SAT solver. The first phase is a simplification phase which converts a CNF formula to an equivalent but simpler CNF formula, and the second phase is to call a SAT solver to solve the simplified CNF formula. The emphasis is on the first phase; any sat solver may be used as for the second phase. This solver uses the latest available version of Minisat [1] which can be downloaded from [2]. Actually, the first phase is a preprocessing phase and tries to simplify the CNF formula using unary and binary clauses of the formula. This phase has three parts which are described next.

2. Resolution on binary Clauses
The first part of the simplification is to use the resolution rule on binary clauses to extract unary ones. If two binary clauses exist in the CNF formula in the forms $(l_i \vee l_j)$ and $(\bar{l}_i \vee l_j)$, a unary clause in the form (l_j) can be extracted from them. In this part of preprocessing, all binary clauses in the given formula are checked to find every pair from which a unary clause can be extracted.

For speed up, this part is performed at the same time the CNF formula is being read and prepared into the data structures.

3. Unit-propagation
All the unary clauses are stored in a queue while the CNF formula is being read. This includes the new unary clauses extracted from the binary ones. Then, a unit-propagation task is performed to further simplify the formula and also to eliminate all the unary clauses.

4. Finding Strongly Connected Component
After eliminating all unary and binary clauses in the forms $(l_i \vee l_j)$ and $(\bar{l}_i \vee l_j)$, an Implication Graph with all of the remaining binary clauses are constructed. As in 2-SAT, each strongly-connected component can be used for the purpose of binary equivalence of some literals. For example, if a strongly-connected component with three nodes l_i, l_j and l_k exist in the implication graph, then there will be paths from l_i to l_j and from l_j to l_k and from l_i to l_k, and vice versa. Recall that in an implication graph, if a literal l_i has a path to another literal l_j then l_i implies l_j or if $l_i = true$ then $l_j = true$ either. So if a strongly-connected component with three nodes l_i, l_j and

l_k exist in an implication graph, then all these literals will be equivalent. So l_j and l_k can be replaced with l_i , and also $\bar{l}_j$ and $\bar{l}_k$ can be replaced with l_i. Therefore, by finding strongly-connected components, some variables are eliminated and the CNF formula is simplified and is expected to be solved in less time than would otherwise be the case.

Having this accomplished, the simplification phase is completed, and a sat solver, Minisat in this version of IUT_BMB_SAT, must be invoked to solve the simplified formula.

References
[1] Een, N. and Sorensson, N., "*An Extensible SAT-solver*", In SAT '03 (2003).
[2] Minisat Homepage, http://minisat.se/
[3] http://en.wikipedia.org/wiki/2-satisfiability

The SAT Solver kw − 2009 version

by Johan Alfredsson, Oepir

johan@oepir.com

Introduction

kw is a logic reasoning machinery. It includes a satisfiability solver described in [1] (which is also called kw). It also includes a bounded model checker, an SMT solver, native support for more elaborate constraints than clauses; it supports circuit-based reasoning etc. The aim when creating it has been to build a flexible framework that can easily be extended in several directions. Therefore, the architecture is extremely modular and it is easy to create new logic reasoning components.

The SAT solver kw is based upon a number of strateies together with a strategy selection mechanism that time-slices different strategies to simplify and solve the problem instances. kw supports solving incremental SAT problems and proof generation.

kw is written in C++, Scheme and Python. The version of kw entering the SAT competition 2009 was submitted as a 64 bit binary.

Changes versus the SAT race 2008 version

Development during 2008 has mainly been centered around non-SAT functionality. Instead, kw has been developed in a tangential direction, for instance by extending it to an SMT solver and building a BMC engine on top of the SAT functionality. However, some core SAT-specific improvements have also been made which are described below.

Variable instantiation

kw now includes a variable instantiation strategy [2], [3]. It can be viewed as an extension to the pure literal rule and may transform a SAT instance to an equisatisfiable instance with fewer models and less unknown variables.

Elimination improvements

The elimination strategy [1] has been reworked to be stronger and more efficient, and is now able to eliminate more variables than before.

Polarity caching

kw now uses polarity caching initially described in [4].

Blocking literals

The blocking literals technique introduced in MiniSat 2.1 [5] has been added.

Automatic strategy improvements

The automatic strategy selection strategy is now smarter, resulting in shorter runtimes for most instances. It heuristically avoids scheduling strategies which are unlikely to be beneficial at the current stage of the solution process.

Optimized code

Several parts of the code has been improved performance-wise for competition-like conditions. For instance, the proof-logging code doesn't impact runs where logging is turned off any more, and some callback-heavy parts have been refactored.

Future work

More work is planned on several fronts. Continued development in SMT-related areas and more higher-level constraints are items high on the todo list. Regarding the SAT specific parts, two separate development tracks are intended: improving the circuit-based capabilities and adding more strategies. Development during 2008 produced several new strategies most of which, however, are not ready for prime-time. The potential in those strategies needs to be harvested.

References

[1] Johan Alfredsson – The SAT Solver KW, in *The SAT race 2008: Solver descriptions, 2008*

[2] Johan Alfredsson – The SAT Solver Oepir, in *The SAT competition 2004: Solver descriptions*, *2004*

[3] Gunnar Andersson, Per Bjesse, Byron Cook and Ziyad Hanna – A Proof Engine Approach to Solving Combinatorial Design Automation Problems, in *Proceedings of the Design Automation Conference, 2002*

[4] Knot Pipatrisawat and Adnan Darwiche – A Lightweight Component Caching Scheme for Satisfiability Solvers, in *Proceedings of the International Conference on the Theory and Applications of Satisfiability Testing, 2007*

[5] Niklas Sörensson and Niklas Eén – MiniSat 2.1 and MiniSat++ 1.0 — SAT Race 2008 Editions, in *The SAT race 2008: Solver descriptions, 2008*

LySAT: solver description

Youssef Hamadi[1], Said Jabbour[2], and Lakhdar Sais[2,3]

[1] Microsoft Research
7 J J Thomson Avenue, Cambridge, United Kingdom
youssefh@microsoft.com
[2] CRIL-CNRS, Université d'Artois
Rue Jean Souvraz SP18, F-62307 Lens Cedex France
{jabbour, sais}@cril.fr
[3] INRIA - Lille Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley Bt.A, Park Plaza, F-59650 Villeneuve d'Ascq, France

Overview

LySAT is a DPLL-based satisfiability solver which includes all the classical features like lazy data-structures and activity-based decision heuristics. It differs from well known satisfiability solvers such as Rsat [6] and MiniSAT [3] on many important components such as restart strategies and clause learning. In addition to the classical first-UIP scheme, it incorporates a new technique which extends the classical implication graph used during conflict-analysis to exploit the satisfied clauses of a formula [1]. It also includes a dynamic restart strategy, where the cut-off value of the next restart is computed using information gathered in the two previous runs. Finally, a new phase-learning [4, 6] policy based on the computed occurrences of literals in the learnt clauses is used.

Additionally, LySAT exploits a new dynamic subsumption technique for Boolean CNF formulae[5]. It detects during conflict analysis, clauses that can be reduced by subsumption. During the learnt clause derivation, and at each step of the resolution process, the solver checks for backward subsumption between the current resolvent and clauses represented in the implication graph. This dynamic subsumption approach gives rise to a strong and dynamic simplification technique which exploits learning to eliminate literals from the original clauses.

Code

The system is written in C++ and has about 2000 lines of code. It was submitted to the race as a 32 bit binary. It is written on top of minisat 2.02 [3]. SatElite was also applied systematically as a pre-processor [2].

References

1. Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. In *11th International Conference*

  on Theory and Applications of Satisfiability Testing - SAT'2008, volume 4996 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008.

2. Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

3. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

4. Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *AAAI*, pages 301–306, 1994.

5. Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. Learning for dynamic subsumption. In *12th International Conference on Theory and Applications of Satisfiability Testing - SAT'2009*, 2008 (submitted).

6. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

ManySAT 1.1: solver description

Youssef Hamadi[1], Said Jabbour[2], and Lakhdar Sais[2]

[1] Microsoft Research
7 J J Thomson Avenue, Cambridge, United Kingdom
`youssefh@microsoft.com`
[2] CRIL-CNRS, Université d'Artois
Rue Jean Souvraz SP18, F-62307 Lens Cedex France
`{jabbour,sais}@cril.fr`
[3] INRIA - Lille Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley Bt.A, Park Plaza, F-59650 Villeneuve d'Ascq, France

Overview

ManySAT is a parallel DPLL-engine which includes all the classical features like two-watched-literal, unit propagation, activity-based decision heuristics, lemma deletion strategies, and clause learning [4, 6]. In addition to the classical first-UIP scheme, it incorporates a new technique which extends the classical implication graph used during conflict-analysis to exploit the satisfied clauses of a formula [1].

When designing ManySat we decided to take advantage of the main weakness of modern DPLLs: their sensitivity to parameter tuning. For instance, changing parameters related to the restart strategy or to the variable selection heuristic can completely change the performance of a solver on a particular problem. In a multi-threading context, we can easily take advantage of this lack of robustness by designing a system which will run different incarnation of a core DPLL-engine on a particular problem. Each incarnation would exploit a particular parameter set and their combination should represent a set of orthogonal strategies.

To allow ManySAT to perform better than any of the selected strategy, conflict-clause sharing was added. Technically, this is implemented through lockless shared data structures. The version 1.1 implements innovative dynamic clause sharing policies [5].

Code

The system is written in C++ and has about 4000 lines of code. It is written on top of minisat 2.02 [3], which was extended to accommodate the new learning scheme, the various strategies, and our multi-threading clause sharing policy. SatElite was also applied systematically by the treads as a pre-processor [2].

References

1. Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. In *11th International Conference on Theory and Applications of Satisfiability Testing - SAT'2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008.
2. Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
3. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
4. Y. Hamadi, S. Jabbour, and L. Sais. Manysat: solver description. Technical Report MSR-TR-2008-83, Microsoft Research, May 2008.
5. Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel SAT solving. In *IJCAI under submission*, 2009.
6. Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, under submission, 2009.

march_hi

Marijn Heule and Hans van Maaren

{m.j.h.heule, h.vanmaaren}@ewi.tudelft.nl

1 introduction

The march_hi SAT solver is an upgraded version of the successful march_ks , march_dl and march_eq Sat solvers, which won several awards at the SAT 2004, 2005 and 2007 competitions. For the latest detailed description, we refer to [3, 2]. Like its predecessors, march_hi integrates equivalence reasoning into a DPLL architecture and uses look-ahead heuristics to determine the branch variable in all nodes of the DPLL search-tree. The main improvements in march_hi are:

- an improved guided jumping strategy: instead of the conventional depth-first search, march_hi uses a jumping strategy based on the distribution of solutions measured on random 3-Sat instances. It jumps more aggressively then the march_ks implementation [1].

- a more accurate look-ahead evaluation function for 3-SAT formulae.

2 pre-processing

The pre-processor of march_dl , reduces the formula at hand prior to calling the main solving (DPLL) procedure. Earlier versions already contained unit-clause and binary equivalence propagation, as well as equivalence reasoning, a 3-Sat translator, and finally a full - using all free variables - iterative root look-ahead. However, march_hi (as well as march_ks) does not use a 3-Sat translator by default (although it is still optional). The motivation for its removal is to examine the effect of (not) using a 3-Sat translator on the performance. Because the addition of resolvents was only based on the ternary clauses in the formula (after the translation) we developed a new algorithm for this addition which uses all clauses with at least three literals.

3 partial lookahead

The most important aspect of march_eq is the PartialLookahead procedure. The pseudo-code of this procedure is shown in Algorithm 1.

Algorithm 1 PartialLookahead( )

1: Let $\mathcal{F}'$ and $\mathcal{F}''$ be two copies of $\mathcal{F}$
2: **for** each variable x_i in $\mathcal{P}$ **do**
3:    $\mathcal{F}' :=$ IterativeUnitPropagation$(\mathcal{F} \cup \{x_i\})$
4:    $\mathcal{F}'' :=$ IterativeUnitPropagation$(\mathcal{F} \cup \{\neg x_i\})$
5:    **if** empty clause $\in \mathcal{F}'$ **and** empty clause $\in \mathcal{F}''$ **then**
6:      **return** "unsatisfiable"
7:    **else if** empty clause $\in \mathcal{F}'$ **then**
8:      $\mathcal{F} := \mathcal{F}''$
9:    **else if** empty clause $\in \mathcal{F}''$ **then**
10:     $\mathcal{F} := \mathcal{F}'$
11:    **else**
12:     $H(x_i) = 1024 \times$ Diff$(\mathcal{F}, \mathcal{F}') \times$ Diff$(\mathcal{F}, \mathcal{F}'')$ $+$ Diff$(\mathcal{F}, \mathcal{F}') +$ Diff$(\mathcal{F}, \mathcal{F}'')$
13:    **end if**
14: **end for**
15: **return** x_i with greatest $H(x_i)$ to branch on

---

4 additional features

- Prohibit equivalent variables from both occurring in $\mathcal{P}$: Equivalent variables will have the same Diff, so only one of them is required in $\mathcal{P}$.

- Timestamps: A timestamp structure in the lookahead phase makes it possible to perform PartialLookahead without backtracking.

- Cache optimisations: Two alternative data-structures are used for storing the binary and ternary clauses. Both are designed to decrease the number of cache misses in the PartialLookahead procedure.

- Tree-based lookahead: Before the actual lookahead operations are performed, var-

ious implication trees are built of the binary clauses of which both literals occur in $\mathcal{P}$. These implications trees are used to decrease the number of unit propagations.

- Necessary assignments: If both $x_i \rightarrow x_j$ and $\neg x_i \rightarrow x_j$ are detected during the lookahead on x_i and $\neg x_i$, x_j is assigned to true because it is a necessary assignment.

- Resolvents: Several binary resolvents are added during the solving phase. Those resolvents that are added have the property that they are easily detected during the lookahead phase and that they could increase the number of detected failed literals.

- Restructuring: Before calling procedure PARTIALLOOKAHEAD, all satisfied ternary clauses of the prior node are removed from the active data-structure to speed-up the lookahead.

References

[1] Marijn J.H. Heule and Hans van Maaren. Whose side are you on? Finding solutions in a biased search-tree. *Journal on Satisfiability, Boolean Modeling and Computation* **4**:117–148, 2008.

[2] Marijn J. H. Heule and Hans van Maaren. March dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation* **2**:47–59, 2006.

[3] Marijn J. H. Heule, Mark Dufour, Joris E. van Zwieten, and Hans van Maaren. March eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Holger H. Hoos and David G. Mitchell, editors, SAT (Selected Papers, *Lecture Notes in Computer Science* **3542**, pages 345–359. Springer, 2004.

MiniSAT 09z for SAT-Competition 2009

Markus Iser
iser@ira.uka.de

1 Introduction

The submitted version of MiniSAT is the result of a student research project
that I did at the Research Group Verification Meets Algorithm Engineering
at the KIT (Karlsruhe Institute of Technology). Part of the project was to
experiment with problem sensitive restart strategies. Since I used the provided
version of MiniSAT [3] to do my experiments, I decided to take part at the SAT
Competition 2009, MiniSAT Hack Track.

2 A Problem Sensitive Restart Heuristic

MiniSAT 09z applies a Problem Sensitive Restart Heuristic called Avoidance of
Plateaux. When the local minimum of backtracking levels stays equal during
the watched interval of backtracking levels we call this a Plateau. As soon as
the number of equal minima exceeds a constant threshold the algorithm triggers
a restart. This Restart Strategy had a good effect on the solvers runtime and
the number of conflicts on unstatisfiable SAT-Instances. A brief description of
our experiments on Problem Sensitive Restart Heuristics can be found in [1].

3 Side Effects

Since the Restart Strategy and the Strategy that reduces the number of learnt
clauses were closely linked in the provided MiniSAT version I picked another
Strategy to decide when to reduce the number of learnt clauses. I chose the
initial Learntsize-Limit to be

$$300000/cvr \qquad (1)$$

where cvr is the initial Clause-Variable-Ratio.
Increment of Learntsize-Limit takes place on every call to ReduceDB. Originally
this was done on every Restart. Since the Avoidance of Plateaux Restart Strategy
often produces many more Restarts this was not applicable any more.

4 Value Caching

Since it was not implemented yet in the provided version of MiniSAT, I did
some kind of Value Caching [2] on Variables to enhance the performance of our
competition version even more.

References

[1] Carsten Sinz and Markus Iser, *Problem Sensitive Restart Heuristics for the
    DPLL Procedure*, SAT Conference 2009, Research Group Verification meet
    Algorithm Engineering, KIT (Karlsruhe Institute of Technology)

[2] K. Pipatsrisawat and A. Darwiche, *A lightweight component caching scheme for satisfiability solvers*, In J. ao Marques-Silva and K. A. Sakallah, editors, SAT, volume 4501 of Lecture Notes in Computer Science, pages 294-299, (Springer, 2007).

[3] Niklas Een and Niklas Sörensson, *An Extensible SAT-solver*, (SAT, 2003)

MiniSat 2.1 and MiniSat++ 1.0 — SAT Race 2008 Editions

Niklas Sörensson

Chalmers University of Technology, Göteborg, Sweden.
`nik@cs.chalmers.se`

Niklas Eén

Cadence Berkeley Labs, Berkeley, USA.
`niklas@cadence.com`

1   Introduction

MiniSat is a SAT solver designed to be easy to use, understand, and modify while still being efficient. Originally inspired by ZChaff [10] and Limmat [1], MiniSat features the now commonplace two-literal watcher scheme for BCP, first-UIP conflict clause learning, and the VSIDS variable heuristic (see [5] for a detailed description). Additionally, it has support for incremental SAT solving, and it exists in variations that support user defined Boolean constraints and proof-logging. Since it's inception, the most important improvements have been the heap-based VSIDS implementation, conflict clause minimization [4], and variable elimination based pre-processing [2].

2   MiniSat 2.1

This version is largely an incremental update that brings MiniSat more in line with the current most popular heuristics, but also introduces a number of data structure improvements. Most are rather well known and of lesser academic interest but mentioned in Section 4 for completeness.

Heuristics   During the last couple of years it has been made clear that using a more aggressive restart strategy [7] is beneficial overall, in particular if it is used in combination with a polarity heuristic based on caching the last values of variables [12]. MiniSat uses the Luby-sequence [9] for restarts, multiplied by a factor of 100. For polarity caching it stores the last polarity of variables during backtracking, except for variables from the last decision level.

Blocking Literals   It can be observed that when visiting a watched clause during unit propagation, it is most commonly the case that the clause is satisfied in the current context. Detecting this without actually having to read from the clause's memory turns out to be a big win as indicated by [13], [8].

However, these techniques require an extra level of indirection which makes the win less clear cut. Instead, one can pair each clause in the watcher lists with one copy of a literal from the clause, and whenever this literal is true, the corresponding clause can be skipped. This is very similar to the approach used in the implementation of the SAT solver from Barcelogic Tools [11], but differs crucially in the sense that the auxiliary *blocking literal* does not have to be equal to the other watched literal of the clause, and thus there is no extra cost for updating it.

3   MiniSat++ 1.0

This tool is envisioned as a rewrite of MiniSat+ [6], but contains so far only the circuit framework necessary to participate in the AIG track. As an AIG solver it is currently rather simple: the circuit is first

simplified with DAG-aware rewriting (inspired by [3], but far less powerful at the moment), then clausified using the improved Tseitin transformation (see [3] for an overview), and finally MiniSat 2.1 is run on the result, including CNF based pre-processing.

4   SAT-Race Hacks

The versions submitted to the SAT-Race contains two data structure improvements designed to improve memory behaviour of the solvers: Firstly, binary clauses are treated specially as in MiniSat 1.14 [4]. In combination with blocking literals this is slightly more natural to implement, but on the other hand, there is some overlap in their beneficial effects and the difference thus becomes smaller. Secondly, a specialized memory manager is used for storing clauses. This was introduced to allow 32-bit references to clauses even on 64-bit architectures, but it also gives a small to modest performance benefit on 32-bit architectures depending on the quality of the system's `malloc` implementation.

Finally, even though the pre-processing of MiniSat scales relatively well, there are still cases were it takes too much time or memory. As a simple safe-guard measure, pre-processing is inactivated if the problem has more than 4 million clauses.

References

[1] A. Biere. The evolution from limmat to nanosat. In *Technical Report 444, Dept. of Computer Science, ETH Zürich*, 2004.

[2] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of the 8th Int. Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *LNCS*, 2005.

[3] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up sat. In *SAT*, pages 272–286, 2007.

[4] N. Eén and N. Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization. *System description for the SAT competition 2005.*

[5] N. Eén and N. Sörensson. An extensible sat solver. In *Proc. of the 6th Int. Conference on Theory and Applications of Satisfiability Testing*, 2003.

[6] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:1–26, 2006.

[7] J. Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pages 2318–2323, 2007.

[8] H. Jain and E. Clarke. Sat solver descriptions: Cmusat-base and cmusat. *System description for the SAT competition 2007.*

[9] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. In *Israel Symposium on Theory of Computing Systems*, pages 128–133, 1993.

[10] M. W. Moskewizc, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of 12th Int. Conference on Computer Aided Verification*, volume 1855 of *LNCS*, 2001.

[11] R. Nieuwenhuis and A. Oliveras. Barcelogic for smt. `http://www.lsi.upc.edu/~oliveras/bclt-main.html`.

[12] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. ao Marques-Silva and K. A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[13] T. Schubert, M. Lewis, N. Kalinnik, and B. Becker. Miraxt – a multi-threaded sat solver. *System description for the SAT competition 2007.*

minisat_cumr

Kazuya MASUDA and Tomio KAMADA

Dept. of Computer Sci. and Sys. Eng., Grad. School of Eng., Kobe Univ.
{kazy,kamada}@cs26.scitec.kobe-u.ac.jp

March 18, 2009

1   Introduction

minisat_cumr is hacked version of MiniSat[2]. this employs frequently restart such as RSat[4] or PicoSAT[1], process saving technique[3] and new learned clause amount control heuristic. minisat_cumr_r version, the restart algorithm is almost same as RSat 2.02. minisat_cumr_p version, the restart algorithm is almost same as PicoSat 846.

Conflict driven learning is one of the most effective speeding-up technique for DPLL solver, but a huge amount of learned clauses slows down BCP (boolean constraint propagation) terribly. Existing implementation set an upper limit of learned clauses, and increase it gradually. But when there are few effective clauses, also increase the limit. minisat_cumr's new heuristic control a learned clause amount by a amount of effective clauses.

2   Learned Clause Amount Control

To control a learned clause amount, minisat_cumr compute CUMR, clauses utilization mean ratio [1] in BCP. Counters of CUMR is periodically scaled down for time locality.

$$CUMR = \frac{number\ of\ clause\ utilized}{number\ of\ clause\ checked}$$

minisat_cumr check CUMR periodically, and if CUMR is higher than the threshold then it reduce the amount, otherwise it increase the amount. This will keep CUMR around the threshold, because learned clauses are deleted from the low activity one (leads low CUMR) sequentially.

The threshold is computed from the early value of CUMR, it intend to adjusts to the characteristic of individual instance. And to avoid an extreme value, the threshold doesn't exceed 0.1.

$$threshold = \min(\ (early\_CUMR * 0.5),\ 0.1\ )$$

---

[1] If a clause is conflict or propagatable, consider it is utilzable.

References

[1] Armin Biere. Picosat essentials. *JSAT*, 4:75–97, 2008.

[2] Niklas Eén and Niklas Sörensson. The minisat page. http://minisat.se/.

[3] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing(SAT)*, pages 294–299, 2007.

[4] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

MoRsat: Solver description

Jing-Chao Chen

School of Informatics, Donghua University, 1882 Yan-An West Road,
Shanghai, 200051, P. R. China
chen-jc@dhu.edu.cn

Overview

MoRsat is a new hybrid SAT solver. The framework of the solver is based on a look-ahead technique, and its core is a conflict-driven search. A look-ahead technique is used to split the original problem to sub-problems, each of which is either solved or aborted by a conflict-driven DPLL. The aborted sub-problems are solved recursively. We present new properties of XOR clauses, by which we incorporate XOR reasoning into our conflict-driven DPLL so that MoRsat can solve also some well-structured instances. In many places where the sub-problems are solved, the conflict-driven technique used in our solver is the same as Rsat. For example, the conflict resolution scheme (clause learning scheme) and decision heuristic used in our solver are *firstUIP* (unique implication points) + conflict clause minimization and VSIDS (Variable State Independent Decaying Sum), which are the same as those used in Rsat 2.01. However, in some places, our solver is different from Rsat. We made a slight modification and optimization on some strategies such as restart strategies, clause learning database maintenance etc.

Performance evaluation

Based on our empirical results, the performance of MoRsat is significantly better than Rsat and March, which won Gold Medals in the industrial and handmade SAT category at the SAT 2007 competition, respectively. On the handmade category, MoRsat can outperform March. On the industrial category, MoRsat is superior to Rsat, and can solve some industrial instances that were not solved in the SAT 2007 competition. On the random category, MoRsat outperformed Rsat, and was slightly slower than March. However, the number of instances solved by MoRsat was almost the same as that solved by March within 5000s.

The SAT Solver MXC, Version 0.99

(2009 SAT Competition Version)

David R. Bregman
Simon Fraser University
drb@sfu.ca

March 11, 2009

1. Introduction

MXC is a complete, clause-learning SAT+Cardinality solver, written in C++. MXC is open source, and may be obtained at http://www.cs.sfu.ca/research/groups/mxp/MXC/. MXC accepts an extended version of the DIMACS CNF format which contains cardinality constraints interleaved with regular clauses. Unit propagation on cardinality constraints is implemented using a simple counting based method. When a cardinality constraint is used as an antecedent or a conflict, some heuristics are applied to extract a weaker clausal version, which is then used in the standard learning process. The SAT part of the solver is relatively standard, using the two watched literals scheme (with occurrence stacks [5]) for unit propagation, 1-UIP cuts for clause learning, conflict clause minimization [2], activity based variable ordering and clause deletion, progress caching [4], and aggressive nested restarts [5]. When functioning in pure SAT mode (i.e. if no cardinality constraints are present) then the SatELite algorithm [3] is used for preprocessing. The high-level search algorithm itself is implemented as "repeated probing" [6].

2. History

Development of MXC began in 2006, to support the MX project (see: http://www.cs.sfu.ca/research/groups/mxp/). The first version, MXC 0.1, received the "best student solver" award at Sat Race '06. The second version, MXC 0.5, received a bronze medal in the "handmade" category at Sat Competition '07, and the third version, MXC 0.75, placed 5th in Sat Race '08, the highest ranking by an open-source solver (as of the date of writing, the top 4 entries remain unavailable for download).

Solver	Solved (out of 100)	Total time (min.)
MXC 0.1 (sat race '06)	49	976.8
MXC 0.5 (sat comp '07)	66	709.0
MXC 0.75 (sat race '08)	82	444.3
MXC 0.99 (sat comp '09)	85	380.3
Minisat 2.0 Beta	71	668.5

Figure 1: performance comparison of MXC 0.99 and previous versions of MXC on the Sat Race '06 benchmark set, with a 15 minute time limit. Instances that time out count the full 15 minutes towards the total time. Minisat 2.0 Beta is included as a point of reference. All tests were run on a 2.4 GHz Opteron 250 with 1MB L2 cache.

3. New in MXC 0.99

Adaptive restart control.

In MXC 0.75 [1], a classification heuristic was introduced to control restart frequency. The classifier was learnt using the Weka machine learning suite on a training set of 300 instances. In MXC 0.99, this is replaced by Biere's ANRFA heuristic [7], which achieves roughly the same result using less computational resources and requiring no training.

Blocking literals.

It has been noticed (independently?) by several authors (e.g. [8,9]) that on typical industrial instances, when scanning a clause for a new watch, the clause is already satisfied by the first literal with high probability - as often as 50-90% of the time. Because visiting the clause requires following a pointer, there will often be an expensive cache miss involved. If a copy of the first literal is stored with the watch, then it can be checked without dereferencing the clause. If it is true, then no additional work needs to be done. This extra literal stored with the watch is called a blocking literal. There is no requirement that it be the same as the first literal in the clause, and it is also possible to have more than one blocking literal. (MXC 0.99 uses a single blocking literal.)

Implementation details.

The source release of MXC 0.99 includes a visual studio project file, and implementation of the getopt() function for windows, allowing native compilation on that platform. Version 0.99 is intended to be the last "monolithic" version of MXC. Versions 1.0 and up will be modularized, allowing easy use as an API for interactive solving. Some refactoring towards that end has already been done.

References

[1] David R. Bregman and David G. Mitchell. The SAT solver MXC, version 0.75. Solver Description for SAT Race 2008.

[2] Niklas Een and Niklas Sörensson. MiniSat - A SAT solver with conflict-clause minimization. In Proc. SAT'05.

[3] Niklas Een and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Proc. SAT'05.

[4] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In Proc. SAT'07.

[5] Armin Biere. PicoSAT essentials. JSAT, 2008.

[6] Jinbo Huang. A case for simple SAT solvers. In Proc. CP'07.

[7] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Proc. SAT'08.

[8] Himanshu Jain and Edmund Clarke. Sat solver descriptions: Cmusat-base and cmusat. Solver description for SAT competition 2007.

[9] Geoffrey Chu and Aaron Harwood and Peter J. Stuckey. Cache Conscious Data Structures for Boolean Satisfiability Solvers. JSAT, 2009. (to be published)

$NCVW_r$: A Specific Version of $NCVW$

Wanxia Wei[1] and Chu Min Li[2] and Harry Zhang[1]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada, E3B 5A3
{wanxia.wei,hzhang}@unb.ca
[2] MIS, Université de Picardie Jules Verne 33 Rue St. Leu, 80039 Amiens Cedex 01, France
chu-min.li@u-picardie.fr

1 Review of Algorithms $adaptG^2WSAT+$, $RSAPS$, and VW

The local search algorithm $adaptG^2WSAT+$ [5] combines the use of promising decreasing variables [3] and the adaptive noise mechanism [1]. $SAPS$ [2] scales the weights of unsatisfied clauses and smoothes the weights of all clauses probabilistically. It performs a greedy descent search in which a variable is selected at random to flip, from the variables that appear in unsatisfied clauses and that lead to the maximum reduction in the total weight of unsatisfied clauses when flipped. $RSAPS$ [2] is a reactive version of $SAPS$ that adaptively tunes smoothing parameter P_{smooth} during the search. In VW [4], the weight of a variable reflects both the number of flips of this variable and the times when this variable is flipped. This algorithm initializes the weight of a variable x, $vw[x]$, to 0 and updates and smoothes $vw[x]$ each time x is flipped, using the following formula:

$$vw[x] = (1-s)(vw[x]+1) + s \times t \tag{1}$$

where s is a parameter and $0 \leq s \leq 1$, and t denotes the time when x is flipped, i.e., t is the number of search steps since the start of the search. VW always flips a variable from a randomly selected unsatisfied clause c. If c contains freebie variables,[3] VW randomly flips one of them. Otherwise, with probability p (noise p), it flips a variable chosen randomly from c, and with probability $1-p$, it flips a variable in c according to a unique variable selection rule.

2 Local Search Algorithm $NCVW$

A switching criterion, namely the evenness or unevenness of the distribution of variable weights, is defined in [7] as follows. If the maximum variable weight is at least γ times as high as the average variable weight, the distribution of variable weights is considered *uneven*, and the step is called *an uneven step* in terms of variable weights. Otherwise, the distribution of variable weights is considered *even*, and the step is called *an even step* in terms of variable weights.

Another switching criterion, namely the evenness or unevenness of the distribution of variable weights, was proposed in [6]. This criterion is defined in [6] as follows. Assume that δ is a number and $\delta > 1$. If the maximum clause weight is at least δ times as high as the average clause weight, the distribution of clause weights is considered *uneven*, and the step is called *an uneven step* in terms of clause weights. Otherwise, the distribution of clause weights is considered *even*, and the step is called *an even step* in terms of clause weights. An uneven distribution and an even distribution of clause weights correspond to the situations in which clause weights are unbalanced and balanced, respectively.

The ways in which algorithms $adaptG^2WSAT+$, $RSAPS$, and VW select a variable to flip, are referred to as *heuristic $adaptG^2WSAT+$*, *heuristic $RSAPS$*, and *heuristic VW*, respectively [6]. Local search algorithm $NCVW$ [6] adaptively switches among *heuristic $adaptG^2WSAT+$*, *heuristic $RSAPS$*, and *heuristic VW* in every search step according to the distributions of variable and clause weights, to intensify or diversify the search when necessary. This algorithm is described in Fig. 1.

---

[3] Flipping a freebie variable will not falsify any clause.

Algorithm: $NCVW$(SAT-formula $\mathcal{F}$)

1: $A\leftarrow$ randomly generated truth assignment;
2: **for** each variable i **do** initialize $flip\_time[i]$ and $vw[i]$ to 0;
3: initialize max_vw and ave_vw to 0;
4: **for** each clause j **do** initialize $cw[j]$ to 1; initialize max_cw and ave_cw to 1;
5: **for** $flip\leftarrow1$ **to** *Maxsteps* **do**
6:     **if** A satisfies $\mathcal{F}$ **then return** A;
7:     **if** $(max\_vw \geq \gamma \times ave\_vw)$
8:     **then** $heuristic\leftarrow$"VW";
9:     **else**
10:       **if** $((ave\_cw \leq \pi)$ or $(max\_cw \geq \delta \times ave\_cw))$
11:       **then** $heuristic\leftarrow$"$RSAPS$";
12:       **else** $heuristic\leftarrow$"$adaptG^2WSAT+$";
13:     $y\leftarrow$ use *heuristic* to choose a variable;
14:     **if** $(y \neq -1)$
15:     **then** $A\leftarrow A$ with y flipped; update $flip\_time[y]$, $vw[y]$, max_vw, and ave_vw;
16:     **if** $(heuristic = $"$RSAPS$"$)$
17:     **then if** $(y = -1)$ **then** update clause weights, max_cw, and ave_cw;
18: **return** Solution not found;

Fig. 1. Algorithm $NCVW$

In $NCVW$, parameter γ determines whether the distribution of variable weights is uneven, δ determines whether the distribution of clause weights is uneven, and π represents a threshold for average clause weight. Like VW, $NCVW$ updates variable weights using Formula 1. In $NCVW$, the default values of γ, δ, π, s, and wp are $(\gamma, \delta, \pi, s, wp) = (7.5, 3.0, 15.0, 0.0, 0.05)$.

3   A Specific Version of $NCVW$

In $NCVW$, parameters γ and δ are fixed for a broad range of instances and are not optimized for specific types of instance. $NCVW_r$ is a specific version of $NCVW$ that uses optimized γ and δ for random instances. In $NCVW_r$, γ and δ are set to 1.0122 and 2.75, respectively. In $NCVW_r$, variable weights are updated using Formula 1, and parameter s in this formula is adjusted during the search ($s > 0.0$) in the same way as in VW. That is, unlike $NCVW$, $NCVW_r$ smoothes variable weights.

References

1. H. H. Hoos. An Adaptive Noise Mechanism for WalkSAT. In *Proceedings of AAAI-2002*, pages 655–660. AAAI Press, 2002.
2. F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and Probabilistic Smoothing: Efficient Dynamical Local Search for SAT. In *Proceedings of CP-2002*, pages 233–248. Springer, LNCS 2470, 2002.
3. C. M. Li and W. Q. Huang. Diversification and Determinism in Local Search for Satisfiability. In *Proceedings of SAT-2005*, pages 158–172. Springer, LNCS 3569, 2005.
4. S. Prestwich. Random Walk with Continuously Smoothed Variable Weights. In *Proceedings of SAT-2005*, pages 203–215. Springer, LNCS 3569, 2005.
5. W. Wei, C. M. Li, and H. Zhang. Deterministic and Random Selection of Variables in Local Search for SAT. http://www.satcompetition.org/2007/contestants.html.
6. W. Wei, C. M. Li, and H. Zhang. Switching Among Non-Weighting, Clause Weighting, and Variable Weighting in Local Search for SAT. In *Proceedings of CP-2008*, pages 313–326. Springer, LNCS, 2008.
7. W. Wei, C. M. Li, and H. Zhang. Criterion for Intensification and Diversification in Local Search for SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:219–237, June 2008.

P{re,i}coSAT@SC'09

Armin Biere

Johannes Kepler University, Linz, Austria

Abstract. In this note we describe the new features of PicoSAT version 913 as it was submitted to the SAT competition 2009. It also contains a description of our new solver PrecoSAT version 236, which tightly integrates various preprocessing techniques into a PicoSAT like core engine.

PicoSAT 193

The results of the SAT Race 2008 [15] showed that the old version of PicoSAT which is mostly covered in [7] was actually doing very well up to a certain point where its ability to solve more instances stagnated. This is particularly apparent in the cactus plots [15]. Our analysis revealed that the garbage collection schedule to reduce the number of learned clauses was much more aggressive than in earlier versions which did not use rapid restarts. In essence, PicoSAT in the last SAT Race did not keep enough learned clauses around. In order to use the original reduce policy of PicoSAT, which is similar to the one in MiniSAT [9], we separated the reduce scheduler from the restart scheduler.

To simplify comparison with other solvers, we also use Luby [13] style restart scheduling [11] instead of our inner/outer scheme [7]. Beside this clean-up work, we added two new features, which we have not seen described in the literature before. First, we employed a new literal watching scheme, that uses a literal move-to-front strategy for the literals in visited clauses instead of just swapping the new watched literal with the head respectively tail literal. In our experiments this reduces the average number of traversed literals in visited clauses considerably.

While minimizing learned clauses [17], it seems to be counter-productive, as also explained in [17], to resolve with binary clauses extensively. Learned clauses can be shortened this way, even without decreasing backjumping (backward pruning). But using these learned clauses shortened by extensive binary clause reasoning in a conflict driven assignment loop [14] results in less propagation (forward pruning). This argument can be turned around as follows. Maybe it is better to continue propagating along binary clauses and not stop at the first conflict, but at the last. We experimented with some variations of this idea. It turns out that a conflict that occurs while visiting a longer clause should stop BCP immediately. But for binary clause we run propagation until completion and only record the last conflict that occurred, which is then subsequently used for conflict analysis.

PrecoSAT 236

In the last SAT Race it became apparent, that in order to be successful in these competitions, the integration of a preprocessor, such as SATeLite [8] is mandatory. Last year we experimented, with an external simplifier, called PicoPrep [4], which shares many ideas with SATeLite [8] and Quantor [6]. As in Quantor we used signatures heavily and also functional substitution whenever possible instead of clause distribution. A new feature was to use signatures in forward subsumption as well. Our new solver PrecoSAT is a prototype that allows us to experiment with tight integration of these ideas into a more or less standard PicoSAT/MiniSAT like core engine. This year in PrecoSAT with respect to SATeLite-like preprocessing, we additionally support, functional substitution of XOR and ITE gates. XOR gates with an arbitrary number of inputs are extracted. Gate extraction uses signatures as in Quantor.

We also implemented all the old and new features of PicoSAT discussed before and in addition revived and old idea from PicoSAT 2006 [5], which learns binary clauses during BCP, whenever a forcing clause can be replaced by a new learned binary clause. This can be checked and implemented with negligible overhead in the procedure that assigns a forced variable, by maintaining and computing a dominator tree for the binary part of the implication graph. As a new feature of PrecoSAT during simplification of the clause data base, we decompose the binary clause graph into strongly connected components and merge equivalent literals. We conjecture that the combination of these two techniques allows to simulate equivalence reasoning with hyper-binary resolution [2] and structural hashing.

Most of the binary clauses in PrecoSAT are learned during failed literal preprocessing, which is the only preprocessing technique currently available in plain PicoSAT. Equivalent literals are also detected during failed literal preprocessing and in addition with the help of a hash table. The hash table also allows fast self-subsuming resolution for binary clauses.

The reduce scheduler was simplified and in addition to enlarge the reduce limit on learned clauses in a geometric way, as in PicoSAT/MiniSAT, we also shrink it proportionally to the number of removed original clauses eliminated during simplification and preprocessing phases. We maintain a doubly linked list of all learned clauses, which together with a move-to-front policy [10] allows us to remove the least active learned clause during conflict analysis. Full reduction as in PicoSAT/MiniSAT is only needed if too many inactive clauses are used as reasons.

For the decision heuristic we use a low-pass filter on the number of times a variable is involved in producing a conflict, implemented as an infinite impulse response filter of order 3. This order is configurable at run-time. An order of 1 gives similar characteristics as the exponential VSIDS scheme described in [3].

The most important aspect of PrecoSAT is however, that all three preprocessing techniques, using strongly connected components, failed literals, and SATeLite style preprocessing are tightly integrated in the main loop of the solver, and can be run after new top level units or new binary clauses are derived. The

scheduling of these preprocessors during the search is rather complex and leaves place for further optimizations.

We also integrated blocking literals [16] to reduce the number of visited clauses during BCP and also experimented with more general implication graph analysis [1]. Finally, we flush the phase-saving-cache in regular intervals, also controlled by a Luby strategy, and "rebias" the search by recomputing new phase scores from scratch taking also learned clauses into account. This in contrast to PicoSAT, where we compute a static two-sided Jeroslow-Wang [12] score as phase bias once using the original clauses only.

The tight integration of all these optimizations was very difficult to implement. We spent considerable time in debugging very subtle bugs, also because PrecoSAT can not produce proof traces yet. Accordingly, PrecoSAT is still considered to be in an early stage of development. Moreover, there is only a partial understanding how these optimizations interact.

References

1. G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *Proc. SAT'08*.
2. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proc. SAT'03*.
3. A. Biere. Adaptive restart control for conflict driven SAT solvers. In *Proc. SAT'08*.
4. A. Biere. PicoSAT and PicoAigerSAT entering the SAT-Race 2008.
5. A. Biere. (Q)CompSAT and (Q)PicoSAT at the SAT'06 Race.
6. A. Biere. Resolve and expand. In *Proc. SAT'04*, 2004.
7. A. Biere. PicoSAT essentials. *JSAT*, 4, 2008.
8. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT'05*, 2005.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT'03*, 2003.
10. R. Gershman and O. Strichman. HaifaSat: A new robust SAT solver. In *Proc. Haifa Verification Conference*, 2005.
11. J. Huang. The effect of restarts on the eff. of clause learning. In *Proc. IJCAI'07*.
12. R. Jeroslow and J. Wang. Solving propositional satisability problems. *Annals of mathematics and AI*, 1, 1990.
13. M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47, 1993.
14. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*. IOS Press, 2009.
15. C. Sinz. SAT-Race'08. http://baldur.iti.uka.de/sat-race-2008.
16. N. Sörensson. MS 2.1 and MS++ 1.0 SAT Race 2008 editions.
17. N. Sörensson and A. Biere. Minimizing learned clauses. 2009. Submitted.

Rsat Solver Description for SAT Competition 2009

Knot Pipatsrisawat and Adnan Darwiche

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095 USA
{thammakn,darwiche}@cs.ucla.edu

1 Introduction

This version of Rsat is based on the version of Rsat that participated in the SAT Competition 2007 [2].[1] This version of Rsat still utilizes the SatELite preprocessor [1]. The solver is written in C/C++ and is compiled as a 64-bit binary. The following sections describe changes from the 2007 version.

2 Bi-asserting clause Learning

This version of Rsat tries to learn 1–empowering bi-asserting clauses [3] whenever possible. A conflict clause is 1–empowering if it allows (empowers) unit resolution to derive a new implication, while a bi-asserting clause is a conflict clause with exactly two literals falsified at the conflict level (as opposed to exactly one in the case of asserting clause). The standard algorithm for deriving conflict clauses can be easily modified to detect any occurrence of a 1–empowering bi-asserting clause. Once a 1–empowering bi-asserting clause is found, Rsat will learn it *instead* of the normal (FUIP) asserting clause if it induces an assertion level that is smaller than the one induced by the asserting clause.[2] Empirically, this new learning scheme tends to improve the performance of our solver on unsatisfiable problems. See [3] for more details.

3 Decision Heuristic

The decision heuristic used is a slight variation of the VSIDS heuristic commonly used by leading solvers such as MiniSat and Picosat. Normally, each variable's score is incremented at most once during conflict analysis. This version of Rsat increments variables' scores based on the variables' involvement in conflict analysis. If a variable appears more during the derivation of the conflict clause, its score is incremented more. Moreover, this version of Rsat also increment the scores of variables and clauses participating in conflict clause minimization.

---

[1] See http://reasoning.cs.ucla.edu/rsat for information on previous versions.

[2] The assertion level of a bi-asserting clause is defined to be the second highest level of any literal in it.

4 Restart Policy

This version of Rsat still employs the restart policy based on Luby's series (unit=512). Moreover, it tries to detect periods of slow progress. This is indicated by consecutive small backtracks and roughly stationary levels of conflicts. Whenever this situation arises, Rsat also restarts.

5 Clause Deletion Policy

This version of Rsat deletes conflict clauses more aggressively than the previous version. In particular, it uses a smaller factor to increment the maximum number of conflict clauses. This results in more frequent clause deletions.

6 Data Structure

A new data structure is used to organize information about variables. For each variable, Rsat (like some other solvers) keeps track of its current status, level of assignment, and reason. In the past, this information is usually stored in a number of arrays (or vectors). In this version of Rsat, we group these properties of each variable together, because they are accessed at the same time in most cases. As a result, we only need to maintain one array of variable information. This optimization appears to decrease the running time of Rsat by 10-20%, depending on the access pattern.

References

1. EÉN, N., AND BIERE, A. Effective preprocessing in sat through variable and clause elimination. In *SAT* (2005), pp. 61–75.
2. LE BERRE, D., SIMON, L., AND ROUSSEL, O. SAT'07 Competition Homepage, http://www.satcompetition.org/2007/.
3. PIPATSRISAWAT, K., AND DARWICHE, A. A new learning scheme for efficient unsatisfiability proofs. In *Proceedings of 23rd National Conference on Artificial Intelligence (AAAI) (to appear)* (2008).

SApperloT

Description of two solver versions submitted for the SAT-competition 2009

Stephan Kottler

Eberhard Karls Universität Tübingen, Germany
kottlers@informatik.uni-tuebingen.de

Abstract

In this paper we briefly describe the approaches realised in the two versions of SApperloT. The first version SApperloT-base primarily implements the state-of-the-art techniques of conflict-driven Sat-solvers with some extensions. SApperloT-hrp enhances the base version to a new hybrid three-phase approach that uses reference points for decision making.

1 The main issues of SApperloT

This chapter sketches the main ideas that are implemented in SApperloT-base and are also contained in SApperloT-hrp for the most parts. Both versions are complete Sat-solvers written in C++ using the functionality offered by the standard template library.

Solver basics SApperloT-base is a conflict-driven solver that implements state-of-the-art techniques like clause learning, non chronological backtracking and the two watched literal scheme that were originally introduced by GRASP [10] and CHAFF [11]. For the first version of SApperloT-base Minisat 2.0 [14, 4] was used as a guideline for efficient implementation.
Most decisions are made according to the previous assignment as in RSAT [12]. Moreover, we implemented the extension to the watched literal data-structure as described in [3]. Hence, instead of pointing directly from literals to clauses an indirection object is used. Binary and ternary clauses (both original and learnt clauses) are stored within this object. This has an impact (among other things) on the garbage collection of inactive learnt clauses since binary and ternary learnt clauses cannot be deleted using an activity value (which is applied for clauses with size > 3). To avoid the deletion of valuable long learnt clauses the garbage collection reduces the size of the learnts database by only one quarter and is therefore invoked more frequently. During the garbage collection the learnts database is split in two pieces by applying a variant of the linear median algorithm[1] not to waste time with sorting the learnts database at each call.

Activity values Many decisions in SApperloT are based on activity values like the VSIDS heuristic [11] and the garbage collection of learnt clauses. Also during the minimisation of learnt clauses [1, 14, 13] literals are ordered regarding their activity values. SApperloT-hrp uses the activity of clauses and variables even more extensive. To get the same results on different machines and with different optimization levels of the compiler we implemented a representation of activity values, as it is also done in PicoSAT [2].
Activity values are implemented as (restricted) fractions where the denominator is always a power of some predefined constant. Let $v = n/d$ be any activity value with $d = c^k$. The main operations done with activity related values are addition and multiplication. Since the results

---

[1]It is implemented in the function nth_element of the standard template library

of both operations will have a denominator $c^{k'}$ with some value k' the constant c can be omitted and just kept implicitly. For the above value v our data structure will just store the values n and k. This allows for storing very small numbers within a few bits. Since activity values are only used for relative comparison with each other small values are completely sufficient.

We use 16 bits for the nominator and 16 bits for the denominator for the activity values of variables and clauses. c is set to 128. Choosing, for instance, the reciprocal of the decaying factor as $r := 135/128$ and an initial activity addend as $a := 1/128^{65530}$ guarantees more than 6 million decay and add operations ($s\mathrel{+}=a$; $a\mathrel{*}=r$;) without having to perform an expensive decay of all activity values (worst case). Using the double size for activity values reduces expensive decay operations practically completely.

Preprocessing   SApperloT does not perform any preprocessing on the input formula. Instead the solver has two features to simplify an instance during the solving process. At the first decision level always both polarities of a decision variable d are propagated. If there is a variable u that is assigned by unit propagation in both cases then either a unit clause is learnt (if u was assigned the same value twice) or two binary clauses can be learnt if they are not already contained in the formula [9, 8].

Moreover, after each period of 15 restarts asymmetric branching is applied for all clauses below average length. This helps to further shrink short clauses in order to prune the search space.

2   SApperloT-hrp – a hybrid version with reference points

Our motivation behind SApperloT-hrp is to develop a solver that utilises more information during the solving process and we intend to extend the solver by incorporating more structural information. The three-phase approach realised by the submitted version of SApperloT-hrp can be sketched as follows:

|base| Within this phase usual conflict-driven Sat-solving is applied. The solver gathers information about which clauses occur most frequently in conflicting assignments. Thus, we hold activity values for all clauses. If the solver cannot find a solution within a certain number of conflicts a subset $P \subseteq C$ of clauses is initialised holding the most active clauses.

|pcl| If P is a proper subset of C the solver aims to compute a model that satisfies all clauses in P. If a model is found the solver continues with the third phase. If P contains all clauses of C or if no model can be found within a certain number of conflicts the solver restarts (after simplification of the formula) with the first phase again. Obviously, if the clauses in P are unsatisfiable we conclude unsatisfiability of the entire formula.

|rp| If the solver enters this phase a model M is known that satisfies all clauses in P. This model is taken as a reference point for a variant of the DMRP approach [6, 7]. Thus, the solver tries to modify M so that all clauses in C are satisfied. If there are still some unsatisfied clauses $U \subset C$ after a certain number of conflicts a new set P is initialised: The new set P contains all clauses of U and the most active clauses of C. Also the size of P is remarkably increased and the solver continues with the second phase.

As already shown in [5] hybrid approaches can improve the performance of Sat-solvers. In SApperloT-hrp it seems that alternating the two phases |pcl| and |rp| gives the solver a quite good direction to find a solution for satisfiable instances or to resolve an empty clause if a formula is unsatisfiable. We first implemented an approximation of the break-count of variables as a basis for decisions in the DMRP approach. However, experiments showed that a fast and lazy implementation of the make-count of variables clearly outperforms the break-count approximation. We also achieved good speed-ups by optimising the data-structures to realise delta as defined in [7].

The current version of SApperloT-hrp already performs quite well on many families of instances. However, there are many parameters and magic constants that still have to be figured out by experiments.

References

[1] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *IJCAI*, pages 1194–1201, 2003.

[2] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[3] Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6:99–120, 2009.

[4] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing - SAT 2003*, pages 502–518, 2003.

[5] Lei Fang and Michael S. Hsiao. Boosting sat solver performance via a new hybrid approach. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 5:243–261, 2008.

[6] Eugene Goldberg. Determinization of resolution by an algorithm operating on complete assignments. In *Theory and Applications of Satisfiability Testing - SAT 2006*, 2006.

[7] Eugene Goldberg. A decision-making procedure for resolution-based SAT-solvers. In *Theory and Applications of Satisfiability Testing - SAT 2008*, 2008.

[8] Marijn Heule. *SmArT Solving*. PhD thesis, Technische Universiteit Delft, 2008.

[9] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Principles and Practice of Constraint Programming*, pages 341–355, 1997.

[10] Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, 1999.

[11] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, 2001.

[12] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0. In *Solver Description*, 2007.

[13] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *SAT 2009*, pages 237–243, 2009.

[14] Niklas Sörensson and Niklas Eén. Minisat - a SAT solver with conflict-clause minimization. In *Solver Description*, 2005.

satake: solver description

Kota Tsuyuzaki, Kei Ohmura and Kazunori Ueda

Dept. of Computer Science and Engineering, Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan

1   Overview

We introduce our SAT solver *satake* implemented using Pthread to parallelize
MiniSat v1.14. This solver is designed to operate efficiently on workstations with
8 or more CPU cores.

Satake has a standard master-worker structure consisting of a single master
and multiple workers. Each worker has its own clause database and the master
has a database of learned clauses. Each worker has most of the mechanisms that
MiniSat employs, including the DPLL algorithm, non-chronological backtrack-
ing, conflict-driven learning, restart, 2-literal watching and decision heuristics,
to search for solutions efficiently. Workers communicate with the master to share
learned clauses used for pruning branches of search trees. The master controls the
number of learned clauses by deleting duplicative clauses and some of redundant
learned clauses. These schemes achieve the following benefits:

- Each worker can use and modify its own clause database freely without any
  lock or synchronization because the database is not shared.
- Each worker can concentrate on search for solutions because the master
  manages shared learned clauses.
- The communication overhead is reduced by controlling the sharing of learned
  clauses.

We introduce the key ideas of data sharing of *satake* in the next section.

2   Effective Data Exchanging

The key issue of *satake* is to reduce communication overhead to achieve
parallel speedup. To decrease latency in data exchange, we have added some
mechanisms to the solver.

One is the qualified communication of learned clauses. Although sharing
learned clauses is effective in the reduction of the search space, exchanging too
many learned clauses will result in increasing the cost of operations such as unit
propagation. *Satake* has reduced the communication overhead that may cause
the bottleneck of parallel computation by sharing only shorter learned clauses.
The upper bound of the length of learned clauses exchanged is determined by
dynamic profiling and the number of threads.

Another point is the tuning of the transmission of learned clauses. To access shared objects frequently might increase synchronization overhead even when the communication bandwidth between the master and worker is high enough. To decrease the idle time for synchronization, workers send learned clauses to the master after a certain number of learned clauses are accumulated.

Moreover, each worker has a different decision heuristic and a different random seed. This can reduce the duplication of search space and still benefits from the sharing of effective learned clauses. We have confirmed that our implementation achieves parallel speedup with up to 8 threads on a shared-memory workstation with four Intel Xeon 7350 quad-core processors. Unfortunately, we have also experienced that using 16 threads does not improve parallel speedup. This phenomenon is due to the limited memory bandwidth; it is not specific to *satake* but is common to many parallel applications we have evaluated.

3 Code

Satake is written in the C language and is about 3000 lines long. *Satake* uses Pthread and the IPC (InterProcess Communication) system call. *Satake* cannot use more than 16 threads due to the specification of IPC at the moment.

References

1. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: *Proc. SAT 2003*, LNCS 2919, Springer, pp. 502–518 (2004)
2. Zhang, L., Malik, S.: The Quest for Efficient Boolean Satisfiability Solvers. In: *Proc. CAV'02*, LNCS 2404, Springer, pp.17–36 (2002)

SATzilla2009: an Automatic Algorithm Portfolio for SAT

Lin Xu, Frank Hutter, Holger H. Hoos and Kevin Leyton-Brown
Computer Science Dept., University of British Columbia
Vancouver, BC, Canada
{xulin730, hutter, hoos, kevinlb}@cs.ubc.ca

1 Introduction

Empirical studies often observe that the performance of algorithms across problem domains can be quite uncorrelated. When this occurs, it seems practical to investigate the use of algorithm portfolios that draw on the strengths of multiple algorithms. SATzilla is such an algorithm portfolio for SAT problems; it was first deployed in the 2004 SAT competition [12], and recently an updated version, SATzilla2007, won a number of prizes in the 2007 SAT competition [21], including the gold medals for the SAT+UNSAT categories of both the random and handmade categories. SATzilla2008, submitted to the 2008 SAT Race, did not perform as well. We attribute this mainly to the lack of publicly available high-performance component solvers as well as to overheads in computing instance features for huge industrial instances; we addressed this latter point in SATzilla2009.

SATzilla is based on *empirical hardness models* [10, 13], learned predictors that estimate each algorithm's performance on a given SAT instance. Over the years, we have added several features to SATzilla. We integrated regression methods based on partly censored data, probabilistic prediction of instance satisfiability, and hierarchical hardness models [21, 22]. We also almost entirely automated the portfolio construction process based on automatic procedures for selecting pre-solvers and candidate component solvers [23].

The new features in SATzilla2009 are as follows:

- New instance features
- Prediction of feature computation time
- New component algorithms

Due to the automatic procedures we used since SATzilla2008, after obtaining candidate solvers and measuring their runtime for our training and validation instances, the construction of our SATzilla2009 solvers took very little time after we knew the scoring function. SATzilla2009's methodology can be outlined as follows:
Offline, as part of algorithm development:

1. Identify a target distribution of problem instances.
2. Select a set of candidate solvers that are known or expected to perform well on at least a subset of the instances in the target distribution.
3. Use domain knowledge to identify features that characterize problem instances. To be usable effectively for automated algorithm selection, these features must be related to instance hardness and relatively cheap to compute.
4. On a training set of problem instances, compute these features and run each algorithm to determine its running times. We use the term *performance score* to refer to the quantity we aim to optimize.
5. Automatically determine the best-scoring combination of pre-solvers and their corresponding performance scored. Pre-solvers will later be run for a short amount of time before features are computed (step 1 below), in order to ensure good performance on very easy instances and to allow the predictive models to focus exclusively on harder instances.
6. Using a validation data set, determine which solver achieves the best performance for all instances that are not solved by the pre-solvers and on which the feature computation times out. We refer to this solver as the *backup solver*.
7. **New:** Construct a predictive model for feature computation time, given the number of variables and clauses in an instance.
8. Construct a model for each algorithm in the portfolio, predicting the algorithm's performance score on a given instance based on instance features.
9. Automatically choose the best-scoring subset of solvers to use in the final portfolio.

Then, online, to solve a given problem instance, the following steps are performed:

1. Run the presolvers in the predetermined order for up to their predetermined fixed cutoff times.
2. **New:** Predict time required for feature computation. If that prediction exceeds two minutes, run the backup solver identified in step 6 above; otherwise continue with the following steps.
3. Compute feature values. If feature computation cannot be completed due to an error, select the backup solver identified in step 6 above; otherwise continue with the following steps.
4. Predict each algorithm's performance score using the predictive models from step 8 above.

5. Run the algorithm predicted to be the best. If a solver fails to complete its run (e.g., it crashes), run the algorithm predicted to be next best.

2 SATzilla2009 vs SATzilla2008

SATzilla2009 implements a number of improvements over SATzilla2008.

New instance features. We introduced several new classes of instance features: 18 features based on clause-learning [11], 18 based on survey propagation [9], and five based on graph diameter [8]. For the *Industrial* category, we discarded 12 computationally expensive features based on unit propagation, lobjois probing, and graph diameter.

Prediction of feature computation time. In order to predict the feature computation time for an instance based on its number of variables and clauses, we built a simple linear regression model with quadratic basis functions. This was motivated by the fact that in the industrial category of the 2007 SAT competition, as well as in the SAT Race 2008, SATzilla's feature computation timed out on over 50% of the instances, forcing SATzilla to use a default solver; we also discarded some expensive features trading off cost *vs* benefit.

New component algorithms. We updated the component solvers used in SATzilla2008 with the newest publicly-available versions and included a number of local search solvers based on the SATenstein solver framework [1]. For the *Industrial* category, one limiting factor is that many high-performance industrial solvers are not publicly available, such that we cannot use them as component solvers.

3 The SATzilla2009 solvers

SATzilla's performance depends crucially on its component solvers. We considered a number of state-of-the-art SAT solvers as candidate solvers, in particular the eleven complete solvers March_dl04[8], March_pl[7], Minisat 2.0[6], Vallst[19], Zchaff_Rand[11], Kcnfs04[5], TTS 4.0[18], Picosat8.46[2], MXC08[3], Minisat 2007[17] and Rsat 2.0[16].

We also considered the five local search solvers gnovelty+[15], Ranov[14], Ag2wsat0[4], Ag2wsat+[20] and SATenstein[1] (seven automatically configured versions).

As training data, we used all available SAT instances from previous SAT competitions (2002 until 2005, and 2007) and from the SAT Races 2006 and 2008. Based on these instances, we built three data sets:

- *Random*: all 2,821 random instances;

- *Crafted*: all 1,686 handmade/crafted instances;
- *Industrial*: all 1,376 industrial instances.

For each training instance we ran each solver for one hour and recorded its runtime. (Local search solvers were only run on unsatisfiable instances.) Unlike in previous SATzilla versions, we did not use any preprocessing. We computed 96 features for each instance in categories *Random* and *Crafted*, and 84 features for category *Industrial*. In each category, as a training set we used all previously mentioned instances, and as a validation set the 2007 SAT competition instances from that category (note that this validation is a subset of the training; this was motivated by the relative scarcity of available data and our expectation that the 2009 SAT competition instances resemble more closely those from the 2007 competition than those from earlier competitions).

For presolving, we committed in advance to using a maximum of two presolvers. We allowed a number of possible cutoff times, namely 5, 10, and 30 CPU seconds, as well as 0 seconds (i.e., the presolver is not run at all) and considered all orders in which to run the three presolvers. Automated presolver selection then chose the following presolving strategies:

- *Random*: SATenstein(T7) for 30 seconds, then MXC08 for 30 seconds;
- *Crafted*: March_dl04 for 5 seconds, then MXC08 for 5 seconds;
- *Industrial*: MXC08 for 10 seconds, then Picosat8.46 for 5 seconds.

Automated solver subset selection [23] chose the following component solvers:

- *Random*: Kcnfs04, March_dl04, Picosat8.46, Ag2wsat0, Ag2wsat+, gnovelty+, SATenstein(QCP)
- *Crafted*: March_dl04, Minisat 2.0, Minisat 2007, Vallst, Zchaff_Rand, TTS 4.0, MXC08
- *Industrial*: March_dl04, Minisat 2007, Zchaff_Rand, Picosat8.46, MXC08

The automatically-selected backup solvers were Ag2wsat0, Minisat 2007, and MXC08 for *Random*, *Handmade*, *Industrial*, respectively.

4 Expected Behaviour

We submit three different versions of SATzilla, specifically designed to perform well in each of the categories: SATzilla2009_R (*Random*), SATzilla2009_C (*Crafted*), and SATzilla2009_I (*Industrial*). In order to run properly, subdirectory satzilla_Solvers should contain all binaries for SATzilla's component solvers and its feature computation.

References

[1] Anonymous. Satenstein: Automatically building local search sat solvers from components. Under double-blind review., 2009.

[2] A. Biere. Picosat version 535. Solver description, SAT competition 2007, 2007.

[3] D. R. Bregman and D. G. Mitchell. The SAT solver MXC, version 0.5. Solver description, SAT competition 2007, 2007.

[4] W. Wei C. M. Li and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description, SAT competition 2007, 2007.

[5] G. Dequen and O. Dubois. kcnfs. Solver description, SAT competition 2007, 2007.

[6] N. Eén and N. Sörensson. Minisat v2.0 (beta). Solver description, SAT Race 2006, 2006.

[7] M. Heule and H. v. Maaren. March_pl. http://www.st.ewi.tudelft.nl/sat/download.php, 2007.

[8] M. Heule, J. Zwieten, M. Dufour, and H. Maaren. March_eq: implementing additional reasoning into an efficient lookahead SAT solver. pages 345–359, 2004.

[9] E. I. Hsu and S. A. McIlraith. Characterizing propagation methods for boolean satisfiability. pages 325–338, 2006.

[10] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proc. of CP-02*, pages 556–572, 2002.

[11] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: an efficient SAT solver. pages 360–375, 2005.

[12] E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. Hoos. SATzilla: An algorithm portfolio for SAT, 2004.

[13] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proc. of CP-04*, pages 438–452, 2004.

[14] D. N. Pham and Anbulagan. Resolution enhanced SLS solver: R+AdaptNovelty+. Solver description, SAT competition 2007, 2007.

[15] D. N. Pham and C. Gretton. gnovelty+. Solver description, SAT competition 2007, 2007.

[16] K. Pipatsrisawat and A. Darwiche. Rsat 1.03: SAT solver description. Technical Report D-152, Automated Reasoning Group, UCLA, 2006.

[17] N. Sörensson and N. Eén. Minisat2007. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/, 2007.

[18] I. Spence. Ternary tree solver (tts-4-0). Solver description, SAT competition 2007, 2007.

[19] D. Vallstrom. Vallst documentation. http://vallst.satcompetition.org/index.html, 2005.

[20] W. Wei, C. M. Li, and H. Zhang. Deterministic and random selection of variables in local search for SAT. Solver description, SAT competition 2007, 2007.

[21] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Satzilla2007: a new & improved algorithm portfolio for SAT. Solver description, SAT competition 2007, 2004.

[22] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Proc. of CP-07*, pages 712–727, 2007.

[23] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, June 2008.

Switching Between Two Adaptive Noise Mechanisms in Local Search for SAT

Wanxia Wei[1] and Chu Min Li[2]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada, E3B 5A3
`wanxia.wei@unb.ca`[*]
[2] MIS, Université de Picardie Jules Verne 33 Rue St. Leu, 80039 Amiens Cedex 01, France
`chu-min.li@u-picardie.fr`

1   New Mechanism for Adaptively Adjusting Noise

The adaptive noise mechanism was introduced in [1] to automatically adjust noise during the search. We refer to this mechanism as Hoos's noise mechanism. This mechanism adjusts noise based on search progress and applies the adjusted noise to variables in any clause in a search step.

We propose a new mechanism for adaptively adjusting noise during the search. This mechanism uses the history of the most recent consecutive falsifications of a clause. During the search, for the variables in each clause, we record both the variable that most recently falsifies this clause and the number of the most recent consecutive falsifications of this clause due to the flipping of this variable. For a clause c, we use $var\_fals[c]$ to denote the variable that most recently falsifies c and use $num\_fals[c]$ to denote the number of the most recent consecutive falsifications of c due to the flipping of this variable. Assume that c is falsified most recently by variable x in c and so far x has consecutively falsified clause c m times. So, for c, $var\_fals[c] = x$ and $num\_fals[c] = m$. If c is falsified again, there are two cases. One is that x falsifies c again. In this case, $var\_fals[c]$ is still x, and $num\_fals[c]$ becomes *(m + 1)*. The other is that another variable y in c falsifies c. In this case, $var\_fals[c]$ becomes y, and $num\_fals[c]$ becomes *1*. Assume that clause c is the currently selected unsatisfied clause and that a variable in c will be chosen to flip. We use $best\_var$ to represent the best variable in clause c measured by the scores of all variables in c. If $best\_var$ is not $var\_fals[c]$, this mechanism sets noise to its lowest value 0.00 in order to choose $best\_var$ to flip. If $best\_var$ is $var\_fals[c]$, this mechanism determines noise according to $num\_fals[c]$. Specifically, the higher $num\_fals[c]$ is, the higher the noise value is.

Our mechanism for adjusting noise is different from Hoos's noise mechanism in two respects. First, our mechanism uses the history of the most recent consecutive falsifications of a clause due to the flipping of one variable in this clause, while Hoos's noise mechanism observes the improvement in the objective function value. Second, the noise adjusted by our mechanism is clause-specific, whereas the noise adjusted by Hoos's noise mechanism is not.

2   New Local Search Algorithm TNM

Variable weighting was introduced in [4]. The weight of a variable x, $vw[x]$, is initialized to 0 and is updated and smoothed each time x is flipped, using the following formula:

$$vw[x] = (1 - s)(vw[x] + 1) + s \times t \tag{1}$$

where s is a parameter and $0 \leq s \leq 1$, and t denotes the time when x is flipped, i.e., t is the number of search steps since the start of the search [4].

If all variables in all clauses have roughly equal chances of being flipped, all variables should have approximately equal weights. In this case, the same noise can be applied to any variable in any clause at a search step. Otherwise, our proposed mechanism can be used to adjust noise for the variables in each specific clause in order to break stagnation.

---

[*] The first author can be reached via e-mail at weiwanxia@gmail.com after graduation.

A switching criterion, namely, the evenness or unevenness of the distribution of variable weights, was introduced in [6]. We propose a new local search algorithm called TNM (Two Noise Mechanism), which switches between Hoos's noise mechanism and our proposed noise mechanism according to this criterion. This algorithm is described in Fig. 1. Hoos's noise mechanism was integrated in G^2WSAT [2], resulting in $adaptG^2WSAT$ [3]. The local search algorithm $adaptG^2WSAT+$ [5] was improved from $adaptG^2WSAT$. We integrate our proposed noise mechanism to G^2WSAT [2] and obtain $adaptG^2WSAT'$. In Fig. 1, parameter γ ($\gamma > 1$) determines whether the distribution of variable weights is uneven. TNM sets γ to its default value 10.0. Parameters $p1$ and $p2$ represent the noise values adjusted by Hoos's noise mechanism and by our proposed mechanism, respectively. TNM updates variable weights using Formula 1.

Algorithm: TNM(SAT-formula $\mathcal{F}$)

1: $A\leftarrow$ randomly generated truth assignment;
2: **for** each clause j **do** initialize $var\_fals[j]$ and $num\_fals[j]$ to -1 and 0, respectively;
3: **for** each variable x **do** initialize $flip\_time[x]$ and $var\_weight[x]$ to 0;
4: initialize $p1$, wp, s, max_weight, and ave_weight to 0; initialize dp to 0.05;
5: store promising decreasing variables in stack *DecVar*;
6: **for** $flip\leftarrow 1$ **to** *Maxsteps* **do**
7:     **if** A satisfies $\mathcal{F}$ **then return** A;
8:     **if** $max\_weight \geq \gamma \times ave\_weight$
9:     **then**
10:         **if** there is no promising decreasing variable
11:         **then**
12:             randomly select an unsatisfied clause c;
13:             adjust $p2$ for variables in c according to $var\_fals[c]$ and $num\_fals[c]$;
14:         $y\leftarrow$*heuristic adapt*$G^2WSAT'(p2, dp)$;
15:     **else** $y\leftarrow$*heuristic adapt*$G^2WSAT+(p1, wp)$;
16:     $A\leftarrow A$ with y flipped;
17:     **if** flippin of y falsifies a clause j **then** update $var\_fals[j]$ and $num\_fals[j]$;
18:     adjust $p1$ according to Hoos's noise mechanism; $wp = p1/10$;
19:     update $flip\_time[y]$, $var\_weight[y]$, max_weight, ave_weight, and *DecVar*;
20: **return** Solution not found;

Fig. 1. Algorithm TNM

Our first implementation of the proposed noise mechanism in algorithm TNM is simple. Assume that the currently selected unsatisfied clause is falsified most recently by variable *x* and so far *x* has consecutively falsified this clause *m* times. If the best variable measured by the scores of all variables in this clause is not *x*, we set noise $p2$ to 0.00. Otherwise, we set $p2$ to a reasonable value according to *m*.

References

1. H. H. Hoos. An Adaptive Noise Mechanism for WalkSAT. In *Proceedings of AAAI-2002*, pages 655–660. AAAI Press, 2002.
2. C. M. Li and W. Q. Huang. Diversification and Determinism in Local Search for Satisfiability. In *Proceedings of SAT-2005*, pages 158–172. Springer, LNCS 3569, 2005.
3. C. M. Li, W. Wei, and H. Zhang. Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In *Proceedings of SAT-2007*, pages 121–133. Springer, LNCS 4501, 2007.
4. S. Prestwich. Random Walk with Continuously Smoothed Variable Weights. In *Proceedings of SAT-2005*, pages 203–215. Springer, LNCS 3569, 2005.
5. W. Wei, C. M. Li, and H. Zhang. Deterministic and Random Selection of Variables in Local Search for SAT. http://www.satcompetition.org/2007/contestants.html.
6. W. Wei, C. M. Li, and H. Zhang. Criterion for Intensification and Diversification in Local Search for SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:219–237, June 2008.

Ternary Tree Solver (tts-5-0)

Ivor Spence

School of Electronics, Electrical Engineering and Computer Science

Queen's University Belfast

i.spence@qub.ac.uk

March 2009

1 Introduction

The Ternary Tree Solver (tts) algorithm is a complete, deterministic solver for CNF satisfiability. This note describes the operation of version 5.x. Version 5.0 was entered into the SAT 2009 competition. The solver is very loosely based on the well-known Davis-Putnam model and has five phases, namely: Minimization; Variable ordering; Tree building; Tree walking, Rebuilding.

The solver cannot compete with the state-of-the-art solution of large industrial and random benchmarks but appears to have good worst-case performance on hand-crafted benchmarks (such as hgen8, holen, xor-chain etc.) that others find difficult[1].

Brief descriptions of the five phases follow.

2 Minimization

Here, if possible, the problem is first partitioned into disjoint sub-problems in which any variable can be reached from any other. The rest of the algorithm processes each sub-problem separately.

Any clauses that are tautologies, i.e. contain both v and $\overline{v}$, are removed. If all occurrences of a particular variable are of the same sign it is safe to assign the corresponding value to the variable and hence remove any clauses containing it. This is combined with unit clause propagation.

3 Variable ordering

Unlike most Davis-Putnam solvers the variables are processed according to a <u>static</u> ordering. The overall performance depends critically on this ordering, in which variables that occur in the same clause should be processed near to each other. If the variables are regarded as nodes and the clauses as hyperedges, this corresponds to the *minimum linear arrangement* problem for hypergraphs. A perfect solution to this problem is known to be NP-hard, and so an approximation algorithm is used. Note that this approximation affects the overall performance, but not the correctness of the solver.

The approximation algorithm used for small inputs (of the order of fewer than 3000 literals) combines:

1. Simulated Annealing - this is generally regarded as providing the best approximations for MLA, but the execution time is significant;

2. A local search to see whether the simulated annealing result can be improved.

For larger inputs this algorithm is too slow and a more direct algorithm is used in which variables are chosen in turn according to weights which are derived from the number of clauses in common with variables already chosen. This is much faster so at least the solver has a chance of processing larger, easier inputs but does not give such a good ordering.

It should be noted that the minimum linear arrangement does not always lead to the smallest overall execution time and it is an open question whether there is a better metric.

4 Tree building

At the heart of the algorithm is a 3-tree which represents the proposition to be solved. Each node of this tree corresponds to a proposition

and each level corresponds to a variable according to the variable ordering which was determined in the previous phase. The tree is constructed as follows:

- The root of the tree corresponds to the proposition to be solved.

- Each node of the tree has three children, *left*, *middle* and *right*. The left child consists of those clauses from the current proposition that contain the literal v except that the v is removed (where v is the current level). The right child consists of clauses that contain $\overline{v}$ except that the $\overline{v}$ is removed. The middle child consists of those clauses that don't contain v or $\overline{v}$.

- When the removal of a v or $\overline{v}$ leaves a clause empty this generates the proposition *false*. When there are no clauses to be included in a child this generates the proposition *true*.

A hash table of derived propositions is maintained during the tree building process. This is ensures that if different partial assignments lead to the same proposition only one corresponding node is created. The data structure thus contains cycles and is no longer a tree, but can be interpreted as a tree for the purposes of the next stage.

5 Tree walking

This phase is where the bulk of the computation occurs. Starting from the root, the walk has in principle a false/true choice to make at each level, representing an assignment to the corresponding variable, which would by itself lead to 2^n routes (where n is the number of variables). Each node of the tree represents only a portion of the proposition, and so a set of nodes is maintained to record progress. For the false branch from a particular set of nodes, the new set of nodes consists of the union of all left and middle children of the current set. For the true branch, the new set consists of all right and middle children.

There are three outcomes which can result in a path being pruned, i.e. abandoned before the full depth of variables has been explored:

- When a set of nodes contains *false*, no further computation is performed on that branch because there can be no satisfying assignment built from the choices up to this point;

- If a set contains all *true* nodes a satisfying assignment has been found, regardless of the choice of values for subsequent variables;

- When a set of nodes has been found to correspond to an unsatisfiable proposition a record of this is made. If a subsequent request is made for the same set (or indeed a superset) it is known immediately that this is unsatisfiable without having to repeat the previous analysis. This corresponds to *clause memoization* and in this form is perhaps the main contribution of this solver.

In the multi-threaded version of the solver it is the tree walking which is done in parallel, with different threads making different choices about whether to assign *false* or *true* to a variable first. Each thread writes to the database the sets of nodes corresponding to unsatisfiable propositions and thus benefits from the discoveries of other threads.

6 Rebuilding

If any of the sub-problems is found to be unsatisfiable then the overall problem is unsatisfiable. Otherwise, if all the sub-problems have been found to be satisfiable, some of the actions of the initial minimization have to be undone to construct the overall model. Variables removed because they only occur with one sign are inserted with the appropriate value and the models for each of the sub-problems are renumbered as required.

7 Conclusions

Improving the results from the variable ordering phase is expected to be the best way to improve the overall algorithm.

References

[1] tts: A sat-solver for small, difficult instances. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:173–190, 2008.

The VARSAT Satisfiability Solver
SAT Competition 2009

Eric I. Hsu
eihsu@cs.toronto.edu
University of Toronto

Abstract

Here we describe the VARSAT solver as entered in the 2009 SAT Competition. VARSAT integrates probabilistic methods for estimating *variable bias* (the proportion of solutions in which a variable appears positively or negatively) with a modern backtracking solver (MINISAT). There is one entry for each of the three competition divisions: VARSAT-crafted, -industrial, and -random, plus a -small version specialized for problems in all three categories that have smaller numbers of variables. The entries share a common overall design, but differ in choice of estimator and in the settings of parameters that govern the threshold for deactivating the bias estimators. All versions are complete solvers, designed to handle satisfiable and unsatisfiable instances alike.

VARSAT uses probabilistic methods as variable- and value-ordering heuristics within a full-featured backtracking soler, MINISAT(Eén & Sörensson 2003). Each such method computes a *survey* estimating the *bias* of each variable in a problem or sub-problem. The bias represents the probability of finding the variable set positively or negatively if we were to somehow sample from the space of solutions to the problem. The most successful way of using such information, to date, is to identify the most "strongly" biased variable (*i.e.*, the one with the most difference in probability of being positive and probability of being negative), and setting it to the polarity of its stronger bias. The goal is to prefer the more "constrained" variables in our variable-ordering, and to avoid conflicts in our value-ordering.

We calculate one survey after every iteration of fixing a variable and performing unit propagation. Because surveys are expensive, though, we only want to identify the few most important variables, ostensibly simplifying the problem to a smaller subproblem that can be solved by traditional means. Thus, when the maximum strength across variables in a survey drops below a certain *threshold*, we deactivate the entire mechanism and revert to default heuristics until the next restart. This threshold parameter is the main distinguishing feature between the four entries. In addition, the entire survey apparatus is bypassed should the number of clauses in a problem exceed $4,000,000$.

At this point the solver is still subject to experimentation, and a single overall description of the system has yet to be published officially. However, the current design is detailed in (Hsu *et al.* 2008), and the bias estimation techniques are described in a technical report available at the VARSAT homepage (Hsu 2008). Below we simply list the parameter settings that distinguish the four entries:

- VARSAT-crafted: bias estimator is "EMBPG", threshold is 0.8.
- VARSAT-industrial: bias estimator is "EMBPG", threshold is 0.95.
- VARSAT-random: bias estimator is "EMSPG", threshold is 0.75.
- VARSAT-small: bias estimator is "EMSPG", threshold is 0.6.

References

Eén, N., and Sörensson, N. 2003. An extensible SAT-solver. In *Proc. of 6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Portofino, Italy*.

Hsu, E.; Muise, C.; Beck, J. C.; and McIlraith, S. 2008. Probabilistically estimating backbones and variable bias: Experimental overview. In *Proc. of 14th International Conference on Constraint Processing (CP '08), Sydney, Australia*.

Hsu, E. 2008. VARSAT SAT-Solver homepage. http://www.cs.toronto.edu/~eihsu/VARSAT/.

SAT Instances for Termination Analysis with AProVE*

Carsten Fuhs

LuFG Informatik 2, RWTH Aachen University, Germany
fuhs@informatik.rwth-aachen.de

Abstract. Recently, SAT solving has become the backbone for tackling the search problems in automated termination analysis for term rewrite systems and for programming languages. Indeed, even since the last SAT competition in 2007, many new termination techniques have been published where automation heavily relies on the efficiency of modern SAT solvers. Here, a successful satisfiability proof of the SAT instance results in a step in the modular termination proof and simplifies the termination problem to be analyzed.

The present SAT benchmark submission was created using the automated termination prover AProVE. The CNFs stem from termination proof steps using various recent termination techniques. All instances of this submission are satisfiable, and any speed-up for SAT solvers on these instances will directly lead to performance improvements also for automated termination provers.

1 Introduction

Termination is one of the most important properties of programs. Therefore, there is a need for suitable methods and tools to analyze the termination behavior of programs automatically. In particular, there has been intensive research on techniques for termination analysis of *term rewrite systems* (TRSs) [2]. Instead of developing many separate termination techniques for different programming languages, it is a promising approach to transform programs from different languages into TRSs instead. Then termination tools for TRSs can be used for termination analysis of many different programming languages, cf. e.g. [11,19,20].

The increasing interest in termination analysis for TRSs is also demonstrated by the *International Competition of Termination Tools*,[1] held annually since 2004. Here, each participating tool is applied to the examples from the *Termination Problem Data Base (TPDB)*[2] and gets 60 seconds per termination problem to prove or disprove termination. Thus, in order for a termination prover to be competitive, one needs efficient search techniques for finding termination (dis)proofs automatically.

---

* Description of benchmark instances submitted to the *SAT Competition 2009*.
[1] See http://termination-portal.org/wiki/Termination_Competition.
[2] The current version 5.0.2 of this standard database for termination problems is available at http://dev.aspsimon.org/projects/termcomp/downloads/.

However, many of the arising search problems in automated terminating analysis for TRSs are NP-complete. Due to the impressive performance of modern SAT solvers, in recent years it has become common practice to tackle such problems by encoding them to SAT and by then applying a SAT solver on the resulting CNF. This way, performance improvements by orders of magnitude over existing dedicated search algorithms have been achieved, and also for new termination techniques, SAT solving is the method of choice for automation (cf. e.g. [3,4,5,6,7,8,9,14,16,17,21,23]).

Nowadays, techniques like the *Dependency Pair framework* [1,12,13,15] allow for *modular* termination proofs. This means that it is not necessary to show termination of a term rewriting system in a single proof step, but instead one can show termination of the different functions of the system *separately* and *incrementally*. In this setting, one can use SAT solving in such a way that a successful satisfiability proof of the encoded SAT instance results in an incremental step in the modular termination proof which allows to simplify the termination problem to be analyzed.

On the other hand, also speed-ups on unsatisfiable instances are beneficial for automated termination analysis. The faster one finds out that a particular termination technique does not succeed on a given termination problem (e.g., by a SAT solver returning UNSAT for an encoding of this technique for the termination problem), the more time is left to apply other techniques from the plethora of available termination analysis methods.

Nevertheless, this benchmark submission only contains satisfiable instances which contribute directly to successful termination proofs.

2 Benchmark Instances

The present SAT benchmark submission was created using the automated termination prover AProVE [10], which can be used to analyze the termination behavior of term rewriting systems, logic programs [20], and Haskell 98 programs [11].

AProVE was the most powerful termination prover for TRSs in all the termination competitions from 2004 – 2008. In AProVE, SAT encodings are performed in two stages:

1. First, the search problem is encoded into a propositional formula with arbitrary junctors. The formula is represented via a directed acyclic graph such that identical subformulas are shared.
2. Afterwards, this propositional formula is converted into an equisatisfiable formula in CNF. This is accomplished using SAT4J's [18] implementation of Tseitin's algorithm [22].

The submitted CNFs are named `AProVE09-`n`.dimacs`. For the analyzed termination problems from the TPDB, Fig. 1 provides details on the encoded termination technique and on the termination problem for each n.

Fig. 1. Details on the submitted SAT instances from TPDB problems

n	Encoded technique	Termination problem
01	Recursive Path Order [3,4,21]	`TRS/Cime/mucrl1.trs`
02	Recursive Path Order [3,4,21]	`TRS/TRCSR/inn/PALINDROME_complete_noand_C.trs`
03	Recursive Path Order [3,4,21]	`TRS/TRCSR/PALINDROME_complete_iGM.trs`
04	Matrix Order [5,16]	`SRS/secret06/matchbox/3.srs`
05	Matrix Order [5,16]	`SRS/Trafo/hom01.srs`
06	Matrix Order [5,16]	`SRS/Waldmann07b/size-12-alpha-3-num-535.srs`
07	Matrix Order [5,16]	`SRS/Zantema/z049.srs`
08	Matrix Order [5,16]	`SRS/Zantema/z053.srs`
09	Matrix Order [5,16]	`TRS/secret05/cime5.trs`
10	Polynomial Order [6]	`TRS/CSR_Maude/bool/RENAMED-BOOL_nokinds.trs`
11	Max-Polynomial Order [7]	`TRS/secret05/cime1.trs`
12	Max-Polynomial Order [7]	`TRS/Zantema/z09.trs`
13	Non-Monotonic Max-Pol. Order [7]	`TRS/aprove08/log.trs`
14	Rational Polynomial Order [9]	`SRS/Zantema/z117.srs`
15	Rational Polynomial Order [9]	`TRS/endrullis08/morse.trs`
16	Rational Polynomial Order [9]	`TRS/SchneiderKamp/trs/thiemann17.trs`
17	Rational Polynomial Order [9]	`TRS/TRCSR/inn/Ex49_GM04_C.trs`
18	Rational Polynomial Order [9]	`TRS/TRCSR/inn/Ex5_DLMMU04_C.trs`
19	Bounded Increase [14]	`TRS/SchneiderKamp/trs/cade14.trs`
20	Arctic Matrix Order [17]	`SRS/Endrullis/04.srs`
21	Arctic Matrix Order [17], alt. enc.	`SRS/Endrullis/04.srs`

For termination analysis, TRSs are a very suitable representation of algorithms on user-defined data structures. However, another main challenge in termination analysis of programs are algorithms on pre-defined data types like integers. Using standard representations of integers as terms leads to problems in efficiency and power for termination analysis with termination tools for TRSs.

Therefore, very recently we extended TRSs by built-in integers [8]. This combines the power of TRS techniques on user-defined data types with a powerful treatment of pre-defined integers. To automate the corresponding constraint-based termination techniques for this new formalism in AProVE, we again perform a reduction to SAT. For the empirical evaluation of these contributions, we collected a set of integer termination problems from the literature and from applications. This collection can be found on the web page of the evaluation at http://aprove.informatik.rwth-aachen.de/eval/Integer/.

Fig. 2 again provides details on the technique and on the analyzed problems.

3 Conclusion

SAT solving has become a key technology for automated termination provers. Thus, any improvements in efficiency of SAT solvers on the submitted SAT instances will also have a direct impact on efficiency and power of the respective termination tool.

Fig. 2. Details on the SAT instances from Integer TRSs

n	Encoded technique	Termination problem
22	Integer Max-Polynomial Order [8]	`Beerendonk/19.itrs`
23	Integer Max-Polynomial Order [8]	`CADE07/A14.itrs`
24	Integer Max-Polynomial Order [8]	`patrs/pasta/a.10.itrs`
25	Integer Max-Polynomial Order [8]	`VMCAI05/poly4.itrs`

References

1. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
2. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Solving partial order constraints for LPO termination. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:193–215, 2008.
4. Michael Codish, Peter Schneider-Kamp, Vitaly Lagoon, René Thiemann, and Jürgen Giesl. SAT solving for argument filterings. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006)*, volume 4246 of *LNAI*, pages 30–44, Phnom Penh, Cambodia, 2006.
5. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.
6. Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, René Thiemann, Peter Schneider-Kamp, and Harald Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *LNCS*, pages 340–354, Lisbon, Portugal, 2007.
7. Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, René Thiemann, Peter Schneider-Kamp, and Harald Zankl. Maximal termination. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 110–125, Hagenberg, Austria, 2008.
8. Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009)*, LNCS, Brasília, Brazil, 2009. To appear.
9. Carsten Fuhs, Rafael Navarro-Marset, Carsten Otto, Jürgen Giesl, Salvador Lucas, and Peter Schneider-Kamp. Search techniques for rational polynomial orders. In *Proceedings of the 9th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2008)*, volume 5144 of *LNAI*, pages 109–124, Birmingham, UK, 2008.
10. Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*, volume

4130 of *LNAI*, pages 281–286, Seattle, WA, USA, 2006. See also http://aprove.informatik.rwth-aachen.de.

11. Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006)*, volume 4098 of *LNCS*, pages 297–312, Seattle, WA, USA, 2006.

12. Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The DP framework: Combining techniques for automated termination proofs. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2004)*, volume 3452 of *LNAI*, pages 301–331, Montevideo, Uruguay, 2005.

13. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

14. Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp. Proving termination by bounded increase. In *Proceedings of the 21st International Conference on Automated Deduction (CADE 2007)*, volume 4603 of *LNAI*, pages 443–459, Bremen, Germany, 2007.

15. Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

16. Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006)*, volume 4098 of *LNCS*, pages 328–342, Seattle, WA, USA, 2006.

17. Adam Koprowski and Johannes Waldmann. Arctic termination ... below zero. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 202–216, Hagenberg, Austria, 2008.

18. Daniel Le Berre and Anne Parrain. SAT4J: The Java SAT Library. http://www.sat4j.org, 2009.

19. Enno Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):73–116, 2001.

20. Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*. To appear.

21. Peter Schneider-Kamp, René Thiemann, Elena Annov, Michael Codish, and Jürgen Giesl. Proving termination using recursive path orders and SAT solving. In *Proceedings of the 6th International Symposium on Frontiers of Combining Systems (FroCoS 2007)*, volume 4720 of *LNAI*, pages 267–282, Liverpool, UK, 2007.

22. Gregory Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. 1968. Reprinted in *Automation of Reasoning*, volume 2, pages 466–483, Springer, 1983.

23. Harald Zankl and Aart Middeldorp. Satisfying KBO constraints. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA 2007)*, volume 5117 of *LNCS*, pages 389–403, Paris, France, 2007.

Solving edge-matching problems
with satisfiability solvers[*]

Marijn J.H. Heule[**]

Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Sciences
Delft University of Technology
`marijn@heule.nl`

Abstract. Programs that solve Boolean satisfiability (SAT) problems have become powerful tools to tackle a wide range of applications. The usefulness of these solvers does not only depend on their strength and the properties of a certain problem, but also on how the problem is translated into SAT. This paper offers additional evidence for this claim.

To show the impact of the translation on the performance, we studied encodings of edge-matching problems. The popularity of these problems was boosted by the release of Eternity II in July 2007: Whoever solves this 256 piece puzzle first wins \$ 2,000,000. There exists no straightforward translation into SAT for edge-matching problems. Therefore, a variety of possible encodings arise.

The basic translation used in our experiments and described in this paper, is the smallest one that comes to mind. This translation can be extended using redundant clauses representing additional knowledge about the problem. The results show that these redundant clauses can guide the search – both for complete and incomplete SAT solvers – yielding significant performance gains.

1   Introduction

The Boolean satisfiability (SAT) problem deals with the question whether there exists an assignment –a mapping of the Boolean values to the Boolean variables– that satisfies a given formula. A formula, in Conjunctive Normal Form (CNF), is a conjunction of clauses, each clause being a disjunction of literals. Literals refer either to a Boolean variable x or to its negation $\overline{x}$.

SAT solvers have become very powerful tools to solve a wide range of problems, such as Bounded Model Checking and Equivalence Checking of electronic circuits. These problems are first translated into CNF, solved by a SAT solver, and a possible solution is translated back to the original problem domain.

---

[*] A slightly different version of this paper with the same title appeared in Proceedings of the Second International Workshop on Logic and Search (LaSh 2008).

Translating a problem into CNF in order to solve it does not seem optimal: Problem specific information, which could be used to develop specialized solving methods, may be lost in the translation. However, due to the strength of modern SAT solvers, it could be very fruitful in practice: Problem specific methods to beat the SAT approach may take years to develop.

SAT solvers have been successfully applied to various combinatorial problems ranging form lower bounds to Van der Waerden numbers [3] to Latin Squares. However, on many other combinatorial problems, such as Traveling Salesman and Facility Allocation [8], SAT solvers cannot compete with alternative techniques such as Linear Programming. A possible explanation is that the former (successful) group can be naturally translated into CNF, while the latter, due to arithmetic constraints cannot.

For most problems, there is no straight-forward translation into CNF. Whether SAT solvers can efficiently solve such problems does not only depend on the strength of the solvers, but also on the translation of the problem into CNF. This paper offers an evaluation of the influence of a translation on the performance of SAT solvers. The translation of edge-matching problems into CNF serves as this papers experimental environment. The problem at hand appears both challenging and promising; because 1) there is no "natural" translation into CNF, yielding many alternative translations, and 2) there are no arithmetic constraints that seem hard for SAT solvers.

The focus of this paper will be on the influence of *redundant* clauses – those clauses which removal / addition will not increase / decrease the number of solutions. Notice that redundancy as stated above should be interpreted in the *neutral* mathematical sense of the word and not in the *negative* connotation of day-to-day talk. In fact, as we will see, redundant clauses can improve the performance of SAT solvers. Furthermore, all presented encodings will use the same set of Boolean variables.

After introducing edge-matching problems (Section 2), this paper presents the smallest translation into CNF that comes to mind. First the choice of the variables (Section 3), followed by the required clauses (Section 4). This translation can be extended with clauses representing additional knowledge about the problem (Section 5). Then it reflects on the influence of the translation (with and without extensions) on the performance (Section 6) and concludes that encoding is crucial to solve the hardest instances (Section 7).

2 Edge-Matching Problems

Edge-matching problems [5] are popular puzzles, that appeared first in the 1890's. Given a set of pieces and a grid, the goal is to place the pieces on the grid such that the edges of the connected pieces match. Edge-matching problems are proved to be NP-complete [2]. Most edge-matching problems have square pieces and square grids. Yet, there exists a large variety of puzzles[1] with triangle or 3D pieces and irregular grids.

---

[1] See for instance `http://www.gamepuzzles.com/edgemtch.htm`

There are two main classes of edge-matching problems. First, the edges are colored and connected edges much have the same color. These problems are called *unsigned*. Second, instead of colors, edges can have a partial image. These edges match if they have complementary parts of the same image. These problems are called *signed*. A famous signed edge-matching problem is Rubik's Tangle.

Edges on the border of the grid are not connected to pieces, so they cannot match as the other edges. In case there are no constraints placed on these edges, we call the problem *unbounded*. On the other hand, problems are *bounded* if these edges are constraint. A common constraint is that these edges must have the same color. Throughout this paper, when we refer to bounded edge-matching problems, we assume this constraint and that their is a special color only for these border edges.

The popularity of edge-matching problems was boosted by the release of the Eternity II puzzle in July 2007: Whoever solves it first wins $2,000,000. Eternity II is a 16×16 bounded unsigned edge-matching problem invented by Christopher Monckton and published by Tomy. Apart from the large 256 piece puzzle, also four smaller clue puzzles have been released.

3   Choosing the Variables

The selection of Boolean variables for the translation is an important first step to construct an efficient encoding. This section introduces the variables used in the proposed translation of edge-matching problems into CNF. These consist of two types: Variables representing a mapping from pieces to squares (Section 3.1) and variables describing the colors of the diamonds (Section 3.2). Apart from these variables, this section describes the clauses relating to variables of the same type. Clauses that consist of both types will be discussed in Section 4.

Throughout this paper, no auxiliary variables are introduced. Using only the variables in this section, one can already construct dozens of alternative translations. Therefore, evaluating these translations seems a natural starting point. That said, related work such as [7] shows that auxiliary variables can be very helpful to reduce the computational costs of solving the problem at hand.

3.1   Mapping Pieces to Squares

Arguably the most intuitive way to translate edge-matching problems into CNF would be a mapping of the pieces to the squares of the grid. A similar approach has been proposed to translate edge-matching problems into a Constraint Satisfaction Problem [9]. This requires the following Boolean variables:

$$x_{i,j} \quad \begin{cases} 1 & \text{if piece } p_i \text{ is placed on square } q_j \\ 0 & \text{otherwise} \end{cases}$$

Notice that there is no rotation embedded in the variable encoding. As we will see in Section 4, rotation does not require additional variables and can be achieved by clauses.
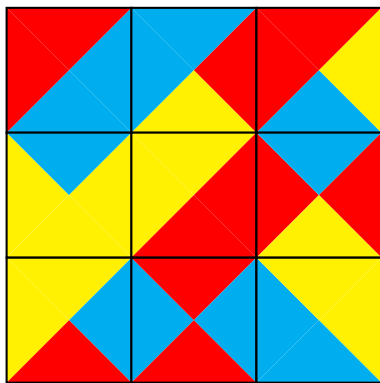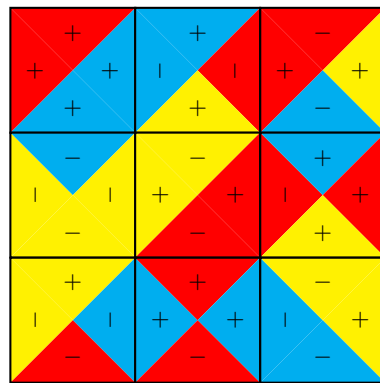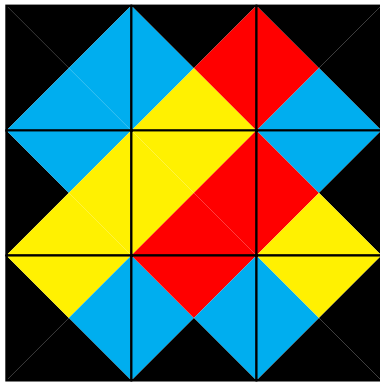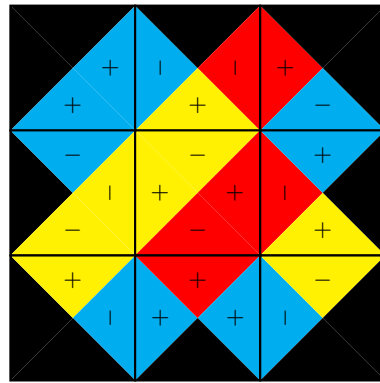
Fig. 1. Examples of an (*a*) unbounded unsigned edge-matching problem, (*b*) unbounded signed edge-matching problem, (*c*) bounded unsigned edge-matching problem, and (*d*) bounded signed edge-matching problem.

For bounded edge-matching problems, *zero edges* refer to those edges that should be placed along the boundary of the grid. Given a $n \times n$ grid, these problems contain four corner pieces with two zero edges (denoted by set P_{corner}) and $4n - 8$ border pieces with one zero edge (denoted by set P_{border}). The pieces with no zero edges are denoted by set P_{center}. For unbounded edge-matching problems, all pieces are in set P_{center}.

Likewise, corner pieces can only be placed in the corners of the grid (denoted by set Q_{corner}), border pieces only along the border (denoted by set Q_{border}), and the other pieces can only be placed in the center (denoted by set Q_{center}). So, the mapping variables are related as follows:

$$(\bigvee_{p_i \in P_{\text{corner}}} x_{i,a}) \wedge (\bigvee_{p_i \in P_{\text{border}}} x_{i,b}) \wedge (\bigvee_{p_i \in P_{\text{center}}} x_{i,c}) \text{ for } q_a \in Q_{\text{corner}}, q_b \in Q_{\text{border}}, q_c \in Q_{\text{center}} \quad (1)$$

$$(\bigvee_{q_j \in Q_{\text{corner}}} x_{a,j}) \wedge (\bigvee_{q_j \in Q_{\text{border}}} x_{b,j}) \wedge (\bigvee_{q_j \in Q_{\text{center}}} x_{c,j}) \text{ for } p_a \in P_{\text{corner}}, p_b \in P_{\text{border}}, p_c \in P_{\text{center}} \quad (2)$$

The above encoding requires $|P_{\text{corner}}|^2 + |P_{\text{border}}|^2 + |P_{\text{center}}|^2$ variables and $2|P_{\text{corner}}| + 2|P_{\text{border}}| + 2|P_{\text{center}}|$ clauses. Notice that the encoding only forces each piece on *at-least-one* square and each square to hold *at-least-one* piece. In fact, in any valid placement, this should be *exactly-one*. Forcing them exactly-one *explicitly* – each mapping of one piece on two squares would violate a specific (binary) clause – is very expensive (in terms of additional (binary) clauses), as we will discuss in Section 5.2. Instead, the clauses presented in Section 4 force the one-on-one mapping *implicitly* which makes the explicit encoding redundant.
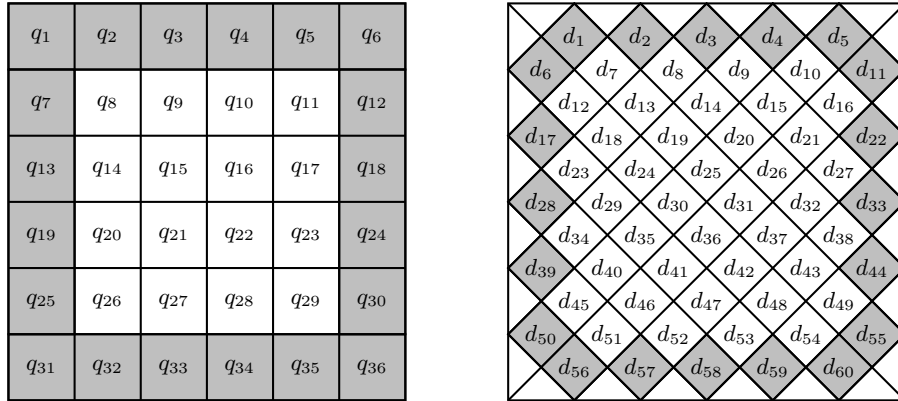

Fig. 2. The numbering of squares q_j (left) and diamonds d_k (right) for a 6x6 edge-matching problem. Gray squares are corner and border squares, gray diamonds are border diamonds.

Implicit encoding assumes that all pieces are unique. In case two pieces are equivalent (modulo rotation), then a few additional clauses have to be added to force equivalent pieces to be placed on different squares. To ensure a valid mapping, we therefore need some additional clauses for each square q_j:

$$(\overline{x}_{i,j} \vee \overline{x}_{l,j}) \quad \text{for } p_i \text{ equivalent to } p_l \text{ and } i < l \tag{3}$$

3.2 Colored Diamonds

The only constraint forced on a placement is that colors of connecting edges must match. Edges are represented as triangles and connected edges as diamonds. Given a $n \times n$ grid, there are $n^2 - 2n$ diamonds. Diamonds are numbered from left to right, from top to bottom, see Figure 2. This brings us to the second type of variables.

$$y_{k,c} \quad \begin{cases} \texttt{1} & \text{if diamond } d_k \text{ has color } c \\ \texttt{0} & \text{otherwise} \end{cases}$$

The colored edges can be partitioned into *border edges* (those directly next to zero edges) and *center edges* (those not directly next to zero edges). Set C_{border} consists of all colors of border edges and set C_{center} consists of all colors of center edges. Likewise, diamonds are partitioned into two sets, one for the border edges, called D_{border}, and one for the center edges, called D_{center}. Figure 2 shows the partition for a 6×6 grid. The disjunction of C_{border} and C_{center} could be empty, but that is not as a rule. The number of variables $y_{k,c}$ equals $|C_{\text{border}}| \cdot |D_{\text{border}}| + |C_{\text{center}}| \cdot |D_{\text{center}}|$. In case either $|C_{\text{border}}|$ or $|C_{\text{center}}|$ is large, the number of required binary clauses will be enormous.

$$(\bigvee_{c \in C_{\text{border}}} y_{k,c}) \wedge (\bigvee_{c \in C_{\text{center}}} y_{l,c}) \wedge \bigwedge_{c,c' \in C_{\text{border}}, c < c'} (\overline{y}_{k,c} \vee \overline{y}_{k,c'}) \wedge \bigwedge_{c,c' \in C_{\text{border}}, c < c'} (\overline{y}_{l,c} \vee \overline{y}_{l,c'}) \text{ for } \begin{cases} d_k \in D_{\text{border}} \\ d_l \in D_{\text{center}} \end{cases} \tag{4}$$

Example 1. Given an edge-matching problem with $C_{\text{border}} = \{\text{blue}, \text{green}, \text{red}\}$ and $C_{\text{center}} = \{\text{cyan}, \text{green}, \text{pink}, \text{yellow}\}$. The following clauses will encode that each diamond has exactly one color:

$$\left. \begin{matrix} (y_{k,\text{red}} \vee y_{r,\text{green}} \vee y_{k,\text{blue}}) \wedge \\ (\overline{y}_{k,\text{red}} \vee \overline{y}_{k,\text{green}}) \wedge (\overline{y}_{k,\text{red}} \vee \overline{y}_{k,\text{blue}}) \wedge (\overline{y}_{k,\text{green}} \vee \overline{y}_{k,\text{blue}}) \end{matrix} \right\} \ d_k \in D_{\text{border}} \tag{5}$$

$$\left. \begin{matrix} (y_{k,\text{cyan}} \vee y_{r,\text{green}} \vee y_{k,\text{pink}} \vee y_{k,\text{yellow}}) \wedge \\ (\overline{y}_{k,\text{cyan}} \vee \overline{y}_{k,\text{green}}) \wedge (\overline{y}_{k,\text{cyan}} \vee \overline{y}_{k,\text{pink}}) \wedge (\overline{y}_{k,\text{cyan}} \vee \overline{y}_{k,\text{yellow}}) \wedge \\ (\overline{y}_{k,\text{green}} \vee \overline{y}_{k,\text{pink}}) \wedge (\overline{y}_{k,\text{green}} \vee \overline{y}_{k,\text{yellow}}) \wedge (\overline{y}_{k,\text{pink}} \vee \overline{y}_{k,\text{yellow}}) \end{matrix} \right\} \ d_k \in D_{\text{center}} \tag{6}$$

4 Essential Clauses

This section deals with the question of how to connect the mapping variables $x_{i,j}$ with the colored diamond variables $y_{k,c}$. The encoding presented here is one of many alternatives. This one uses only a few clauses per mapping variable $x_{i,j}$. All constraints have the format "if p_i is mapped on q_j ..., then d_k has color c". Or as clause $(\overline{x}_{i,j} \lor ... \lor y_{k,c})$. The number of these clauses and their sizes depend on the type of piece p_i. Besides corner and border pieces (discussed in Section 4.1), the center pieces are grouped in seven types (see Section 4.2).

4.1 Corner and Border Pieces

First the easy part. Recall that the zero edges are known. So, corner and border pieces can only be placed on a square with a specific rotation. Therefore, only one binary clause is required for each non-zero edge of the center and border pieces.

Example 2. Given a corner piece p_A with a red east edge and a blue south edge which should be placed on a $n \times n$ grid (see Figure 2). Then the eight clauses below should be added (per corner piece depending on the colors). Notice that $q_1, q_n, q_{n^2-n+1}, q_{n^2}$ are the corresponding corner squares.

$$
\begin{array}{lclc}
(\overline{x}_{A,1} \lor y_{1,\text{red}}) & \land & (\overline{x}_{A,1} \lor y_{n,\text{blue}}) & \land \\
(\overline{x}_{A,n} \lor y_{2n-1,\text{red}}) & \land & (\overline{x}_{A,n} \lor y_{n-1,\text{blue}}) & \land \\
(\overline{x}_{A,n^2-n+1} \lor y_{2n^2-4n+2,\text{red}}) & \land & (\overline{x}_{A,n^2-n+1} \lor y_{2n^2-3n+2,\text{blue}}) & \land \\
(\overline{x}_{A,n^2} \lor y_{2n^2-2n,\text{red}}) & \land & (\overline{x}_{A,n^2} \lor y_{2n^2-3n+1,\text{blue}}) &
\end{array}
$$

Similarly, given a border piece p_B with a pink east edge, a yellow south edge and a green west edge, that should be placed on the same grid, the following clauses should be added for $j \in \{1, \ldots, n-2\}$:

$$
\begin{array}{lclc}
(\overline{x}_{B,j+1} \lor y_{j,\text{green}}) & \land & (\overline{x}_{B,j+1} \lor y_{j+n+1,\text{yellow}}) & \land \\
(\overline{x}_{B,j+1} \lor y_{j+1,\text{pink}}) & \land & (\overline{x}_{B,nj+1} \lor y_{(2n-1)j+n,\text{green}}) & \land \\
(\overline{x}_{B,nj+1} \lor y_{(2n-1)j+1,\text{yellow}}) & \land & (\overline{x}_{B,nj+1} \lor y_{(2n-1)j-n+1,\text{pink}}) & \land \\
(\overline{x}_{B,n(j+2)-1} \lor y_{(2n-1)j,\text{green}}) & \land & (\overline{x}_{B,n(j+2)-1} \lor y_{(2n-1)j+n-1,\text{yellow}}) & \land \\
(\overline{x}_{B,n(j+2)-1} \lor y_{(2n-1)(j+1),\text{pink}}) & \land & (\overline{x}_{B,j+n^2-n+1} \lor y_{j+2n^2-3n+2,\text{green}}) & \land \\
(\overline{x}_{B,n^2-n+1} \lor y_{j+2n^2-4n+2,\text{yellow}}) & \land & (\overline{x}_{B,j+n^2-n+1} \lor y_{j+2n^2-3n+2,\text{pink}}) &
\end{array}
$$

Concluding, for each variable $x_{i,j}$ with $p_i \in P_{\text{corner}}$ we only need two binary clauses, while for each $x_{i,j}$ with $p_i \in P_{\text{border}}$, we need three binary clauses. The next section will discuss which clauses to add for those $x_{i,j}$ with $p_i \in P_{\text{center}}$.

4.2 Center Pieces

Given the choice of the variables presented in Section 3, the encoding of corner and border pieces (as above) is quite straight-forward. However, encoding the

center pieces efficiently is much more tricky. The crux is that if a certain mapping variable $x_{i,j}$ of a center piece is true, we cannot directly color the corresponding diamonds[2]. We need to know how p_i is rotated ($0°, 90°, 180°$, or $270°$).

Rotation can be encoded using two kinds of clauses: *positive rotation clauses* and *negative rotation clauses*. First, positive rotation clauses consist of only positive literals $y_{k,c}$ and the negative mapping literal $\overline{x}_{i,j}$. These clauses force a subset of the corresponding diamonds to be colored in correspondence with one of the edges. The number of these clauses and their sizes depend on how many times a color occurs on a piece. If a color occurs only once then this is encoded as a single clause of length five. If all edges have the same color then a binary clause is required per edge. In the other cases, these clauses have length three and the number depends on the relative location of the edges with the same color. All positive rotation clauses are used in the proposed encoding.

Second, for negative rotation clauses, all literals are negated except for one literal $y_{k,c}$. The negated literals represent the conditions to force diamond d_k to color c. Most negative rotation clauses are ternary clauses. For instance, $\overline{x}_{1,5} \vee \overline{y}_{7,\text{yellow}} \vee y_{8,\text{red}}$, which could be read as "if p_1 is mapped on q_5 and d_7 is yellow, then d_8 is red". In case a piece contains three or four different colors, some negative rotation clauses are required to make the encoding valid.

Center pieces can be partitioned into seven types: 1) All edges have the same color; 2) three edges have the same color; 3) two neighbouring pairs of edges have the same color; 4) both opposite pairs of edges have the same color; 5) one neighbouring pair of edges has the same color; 6) one opposite pair of edges has the same color; 7) all edges have a different color. The number of clauses that should be added for each variable $x_{i,j}$ depends on the type of piece p_i – ranging from 4 (type 1) to 20 (type 7). Figure 3 lists the combination of clauses that should be added per type of each piece.

Notice that the positive rotation clauses of length five are not listed in Figure 3 for types 5, 6, and 7. First, piece type 5 does not need the long positive rotation clauses because the shown clauses are enough to force a valid encoding. For piece types 6 and 7 it is required to add at least of these long clauses. We omitted it in Figure 3, because there is a choice – anyone of them will make the encoding valid. To make the encoding independent of the choice, as stated before, all the positive rotation clauses will be used.

5 Redundant Clauses

Translation of edge-matching problems into CNF as presented in Sections 3 and 4 is the smallest one that came to mind. This translation is such that each satisfying assignment corresponds with a unique valid positioning of the pieces. Moreover, the translation is satisfiable if and only if there exists a valid placement of the the original problem.

---

[2] Expect for the special case in which all edges have the some color (and the same sign, for signed problems).

#	Type	Implications	Clauses

1 — $(\overline{x}_{i,15} \lor y_{25,\text{blue}})^b$

2
$(\overline{x}_{i,15} \lor y_{24,\text{blue}} \lor y_{19,\text{blue}})^b$
$(\overline{x}_{i,15} \lor y_{24,\text{blue}} \lor y_{25,\text{blue}})^a$
$(\overline{x}_{i,15} \lor y_{19,\text{red}} \lor y_{24,\text{red}} \lor y_{25,\text{red}} \lor y_{30,\text{red}})$

3
$(\overline{x}_{i,15} \lor y_{24,\text{blue}} \lor y_{25,\text{blue}})^a$
$(\overline{x}_{i,15} \lor y_{19,\text{red}} \lor y_{30,\text{red}})^a$

4
$(\overline{x}_{i,15} \lor y_{19,\text{blue}} \lor y_{24,\text{blue}})^b$
$(\overline{x}_{i,15} \lor y_{25,\text{red}} \lor y_{30,\text{red}})^b$

5
$(\overline{x}_{i,15} \lor \overline{y}_{24,\text{yellow}} \lor y_{30,\text{red}})^b$
$(\overline{x}_{i,15} \lor y_{24,\text{yellow}} \lor \overline{y}_{30,\text{red}})^b$
$(\overline{x}_{i,15} \lor y_{24,\text{blue}} \lor y_{25,\text{blue}})^a$
$(\overline{x}_{i,15} \lor \overline{y}_{19,\text{blue}} \lor \overline{y}_{25,\text{blue}} \lor y_{24,\text{yellow}})^b$

6
$(\overline{x}_{i,15} \lor \overline{y}_{24,\text{yellow}} \lor y_{25,\text{red}})^b$
$(\overline{x}_{i,15} \lor \overline{y}_{25,\text{red}} \lor y_{24,\text{yellow}})^b$
$(\overline{x}_{i,15} \lor y_{19,\text{blue}} \lor y_{24,\text{blue}})^b$

7
$(\overline{x}_{i,15} \lor \overline{y}_{24,\text{green}} \lor y_{19,\text{blue}})^b$
$(\overline{x}_{i,15} \lor \overline{y}_{19,\text{blue}} \lor y_{25,\text{red}})^b$
$(\overline{x}_{i,15} \lor \overline{y}_{25,\text{red}} \lor y_{30,\text{yellow}})^b$
$(\overline{x}_{i,15} \lor \overline{y}_{30,\text{yellow}} \lor y_{24,\text{green}})^b$

[a] two clauses; apply permutation $\{(y_{19,c}, y_{25,c}), (y_{25,c}, y_{30,c}), (y_{30,c}, y_{24,c}), (y_{24,c}, y_{19,c})\}$ to obtain the other one.

[b] four clauses; apply permutation $\{(y_{19,c}, y_{25,c}), (y_{25,c}, y_{30,c}), (y_{30,c}, y_{24,c}), (y_{24,c}, y_{19,c})\}$ iteratively to obtain the other three.

Fig. 3. The translation of the seven types of center pieces to CNF. The most frequent occurring color is represented by blue, followed by red, yellow and green. Each arrow (implication) is encoded a (set of) clause(s). Black diamonds refer to the complement of an edge. The last column shows one clause per arrow for a piece p_i placed on q_{15} on a 6×6 grid. The corresponding diamonds are d_{19} (north), d_{25} (east), d_{30} (south), d_{24} (west).

.

Although the translation is sufficient, it may not be optimal in case one wants to solve it with a SAT solver. With the addition of some (or even many) clauses and variables, some SAT solvers may find a solution much faster. This section discusses two extensions of the compact translation. Both represent additional knowledge about the problem and require only some extra clauses.

5.1 Forbidden Color Clauses

Once a diamond is given a certain color, then several pieces are not allowed to be placed on the corresponding squares. This knowledge can be added to the formula with several binary clauses. For each diamond, if assigned to a color, then all pieces without that color (on at least one of its edges) cannot be placed on one of the two corresponding squares.

Example 3. Given a piece p_C with one blue edge, two pink edges and a red edge. Say we want to place it on square q_{15} and the d_{30} is one of the corresponding diamonds and $C_{center} = \{blue, cyan, green, orange, pink, red\}$. The forbidden color clauses would be:

$$(\overline{x}_{C,15} \vee \overline{y}_{30,cyan}) \wedge (\overline{x}_{C,15} \vee \overline{y}_{30,green}) \wedge (\overline{x}_{C,15} \vee \overline{y}_{30,orange}) \tag{7}$$

Notice that, provided the encoding of corner and border pieces as described in Section 4.1, these clauses only make sense for center pieces. Let $C(p_i)$ be the set of colors of piece p_i and $q_k^<$ be the smallest index of the corresponding square of diamond d_k, and $q_k^>$ the largest index of the corresponding square.

$$\bigwedge_{color \in C_{center} \setminus C(p_i)} (\overline{x}_{i,q_k^<} \vee \overline{y}_{k,color}) \wedge \bigwedge_{color \in C_{center} \setminus C(p_i)} (\overline{x}_{i,q_k^>} \vee \overline{y}_{k,color}) \text{ for } p_i \in P_{center}, d_k \in D_{center} \tag{8}$$

Several assignments that are implicitly violated by the compact translation, become explicitly violated by the forbidden color clauses. For instance, two pieces cannot be placed on neighbouring squares if they do not have at least one edge in common, because the diamond between these squares cannot be colored. In the compact translation, not all rotation clauses can be satisfied in that situation, although the SAT solver may not see it, yet. However, with the additional forbidden color clauses this directly results in a conflict.

The disadvantage of adding forbidden color clauses, as with all types of additional clauses, is that the encoding will require more resources. Especially when the number of center colors is large, the number of forbidden color clauses will be enormous.

5.2 Explicit One-on-One Mapping

Recall that diamonds are explicitly forced to have exactly one-color which in turn *implicitly* forces each piece on exactly one square. Optionally, we can extend

the translation by adding it *explicitly*. A straight-forward translation of this enforcement is:

$$(\overline{x}_{i,j} \vee \overline{x}_{i,l}) \quad \text{for } p_i \in P_{\text{corner}} \text{ and } q_j, q_l \in Q_{\text{corner}} \text{ and } j < l \quad (9)$$

$$(\overline{x}_{i,j} \vee \overline{x}_{i,l}) \quad \text{for } p_i \in P_{\text{border}} \text{ and } q_j, q_l \in Q_{\text{border}} \text{ and } j < l \quad (10)$$

$$(\overline{x}_{i,j} \vee \overline{x}_{i,l}) \quad \text{for } p_i \in P_{\text{center}} \text{ and } q_j, q_l \in Q_{\text{center}} \text{ and } j < l \quad (11)$$

Notice that the number of additional clauses by this extension is $\mathcal{O}(|P_{\text{center}}|^3)$. Recall that for unbounded edge-matching problems, all pieces are in P_{center}, so the addition is much cheaper for bounded problems. However, if the problem is large enough, say $|P_{\text{center}}| > 40$, the number of additional clauses will exceed the number of original clauses. Yet, one cannot conclude that this addition is counterproductive (in terms of solving speed).

6 Results

This section offers some results of the proposed translations of edge-matching problems to CNF on a test set of bounded unsigned edge-matching problems. Four instances arise from the clue puzzles by Tomy called `cluex`. Additionally, eight problems were generated with various sizes (`a`), number of border colors (`b`) and number of center colors (`c`) called `em-a-b-c`. The smaller five generated instances have relatively many colors yielding only few solutions, while the larger three instances have few colors and therefore many solutions. For each instance from the test set we constructed four different encodings:

– $\mathcal{F}_{\text{compact}}$: The compact translation as described in Section 3 and 4;
– $\mathcal{F}_{\text{fbcolors}}$: The forbidden color clauses (Section 5.1) added to $\mathcal{F}_{\text{compact}}$;
– $\mathcal{F}_{\text{explicit}}$: The explicit one-on-one mapping (Section 5.2) added to $\mathcal{F}_{\text{compact}}$;
– $\mathcal{F}_{\text{all}}$: All presented clauses, the union of $\mathcal{F}_{\text{fbcolors}}$ and $\mathcal{F}_{\text{explicit}}$.

Table 1 offers several properties of the test set instances. Next to the names, the second column lists the size (rows $\times$ columns) of the grid. Although, we explained the translations using square grids, they can be used for rectangular grids as well. The third column shows the number of colors in the format ($|C_{\text{border}}|$, $|C_{\text{center}}|$). The fourth column shows the number of variables used for all encodings. The number of clauses of $\mathcal{F}_{\text{compact}}$, and the number of the additional knowledge clauses are listed in the last three columns.

Only two state-of-the-art SAT solvers are used for the experiments: `picosat` [1] and `ubcsat` [10]. The former is a complete solver – it can also prove that no solution exits – while the latter is a local search solver. Initially, more solvers were used, but the results of complete solvers were strongly related, as were those of various incomplete ones. Therefore, only the strongest solver (based on earlier experiments) of each category was selected. The `picosat` solver was faster than `minisat` [4], probably due to use of rapid restarts in the former. For the `ubcsat` solver, one can select from many different stochastic local search algorithms.

Table 1. Properties of the selected benchmarks. The number of variables is denoted by $\#_{\text{variables}}$. The last columns offer the number of (additional) clauses.

name	size	colors	$\#_{\text{variables}}$	$|\mathcal{F}_{\text{compact}}|$	$|\mathcal{F}_{\text{fbcolors}}|$	$|\mathcal{F}_{\text{explicit}}|$
clue1	6×6	$(4,3)$	728	3688	+704	+3864
clue2	6×12	$(4,4)$	2904	23570	+9120	+41808
clue3	6×6	$(5,4)$	788	3836	+1792	+3864
clue4	6×12	$(5,4)$	2936	23574	+9280	+41808
em-7-3-6	7×7	$(3,6)$	1473	11563	+7500	+11324
em-7-4-8	7×7	$(4,8)$	1617	14513	+11400	+11324
em-7-4-9	7×7	$(4,9)$	1677	15063	+13800	+11324
em-8-4-5	8×8	$(4,5)$	2420	22340	+11232	+29328
em-9-3-5	9×9	$(3,5)$	3857	39932	+19796	+68232
em-11-3-4	11×11	$(3,4)$	8713	99699	+29808	+285144
em-12-2-4	12×12	$(2,4)$	12584	155712	+47200	+526224
em-14-7-3	14×14	$(7,3)$	24356	305744	+41475	+1536792

Table 2. Computational costs (in seconds) to solve the test set using picosat.

name	$\mathcal{F}_{\text{compact}}$		$\mathcal{F}_{\text{fbcolors}}$		$\mathcal{F}_{\text{explicit}}$		$\mathcal{F}_{\text{all}}$	
clue1	0.11	(0.02)	**0.10**	(0.00)	**0.10**	(0.00)	**0.10**	(0.00)
clue2	506.44	(364.19)	164.84	(49.87)	2.75	(1.68)	**0.95**	(0.37)
clue3	0.22	(0.12)	0.12	(0.04)	**0.10**	(0.00)	**0.10**	(0.00)
clue4	1527.54	(536.34)	269.77	(84.72)	1.90	(2.13)	**0.54**	(0.07)
em-7-3-6	> 3600	–	> 3600	–	140.00	(135.18)	**34.91**	(27.18)
em-7-4-8	> 3600	–	> 3600	–	1132.54	(1054.23)	**852.32**	(890.52)
em-7-4-9	> 3600	–	> 3600	–	45.94	(43.75)	**41.89**	(55.94)
em-8-4-5	> 3600	–	> 3600	–	209.35	(187.79)	**86.58**	(67.23)
em-9-3-5	> 3600	–	> 3600	–	501.81	(220.31)	**152.81**	(121.13)
em-11-3-4	> 3600	–	> 3600	–	163.48	(99.87)	**51.68**	(35.56)
em-12-2-4	> 3600	–	> 3600	–	249.66	(151.65)	**88.36**	(81.92)
em-14-7-3	> 3600	–	> 3600	–	80.24	(49.73)	**32.58**	(17.91)

Table 3. Computational costs (in seconds) to solve the test set using ubcsat.

name	$\mathcal{F}_{\text{compact}}$		$\mathcal{F}_{\text{fbcolors}}$		$\mathcal{F}_{\text{explicit}}$		$\mathcal{F}_{\text{all}}$	
clue1	0.04	(0.03)	0.04	(0.02)	**0.02**	(0.01)	**0.02**	(0.01)
clue2	1.78	(1.40)	1.37	(1.21)	**0.19**	(0.04)	0.21	(0.11)
clue3	0.06	(0.06)	0.08	(0.06)	**0.03**	(0.02)	0.04	(0.02)
clue4	1.38	(0.89)	1.69	(1.38)	**0.33**	(0.19)	0.38	(0.31)
em-7-3-6	**60.73**	(17.65)	138.23	(134.30)	670.65	(701.69)	225.70	(78.76)
em-7-4-8	2376.62	(2169.16)	**1732.65**	(1801.32)	> 3600	–	> 3600	–
em-7-4-9	1690.02	(1815.71)	**1284.47**	(1502.43)	> 3600	–	> 3600	–
em-8-4-5	155.62	(180.88)	**80.54**	(74.25)	381.91	(309.00)	128.88	(135.49)
em-9-3-5	1258.10	(1492.35)	**177.36**	(138.91)	1928.79	(2352.45)	839.40	(870.74)
em-11-3-4	82.73	(35.38)	34.16	(5.10)	**2.65**	(1.30)	3.78	(1.85)
em-12-2-4	154.99	(27.92)	32.52	(17.88)	**2.32**	(1.70)	4.41	(1.89)
em-14-7-3	145.72	(21.20)	48.97	(23.19)	**11.02**	(7.99)	44.72	(39.55)

From those algorithms ddfw [6] appeared to be the fastest one on the smaller instances of the test set. Therefore, this algorithm[6] was selected.

Each solver was run on each formula with ten seeds. The execution times differed significantly for those seeds, for both picosat and ubcsat. Therefore, the results show besides the average computational costs also the variance. In case at least five seeds took more than an hour, > 3600 is listed.

The most striking result is that the complete solver picosat is unable to solve most benchmarks in the test set when the explicit one-on-one mapping clauses are not added, see Table 2. Although these clauses are redundant, they appear crucial to solve the problem. Only the four clue puzzles can be solved without these clauses, although clue2 and clue4 are solved considerably faster with them.

The results of the incomplete solver ubcsat shown in Table 3 are much more ambivalent. In contrast to picosat, the most elaborate translation hardly seems the optimal encoding. Yet, only em-7-3-6 is solved fast using the compact encoding. The smaller generated instances (with relatively few solutions) are faster solved by adding the redundant forbidden color clauses, while for the larger ones (with many solutions) the explicit one-on-one mapping clauses appear useful. Apparently, redundant clauses can guide the search for incomplete solvers too.

Yet, despite the weakness shown on the translation without the additional clauses, the (complete) picosat appears the best overall choice. When the compact translation is extended with both sets of additional clauses, of picosat outperforms ubcsat on the harder instances. Moreover, the results suggest that extending the translation with even more additional knowledge could further improve the performance.

7 Conclusions and Future Work

This paper presented a compact translation of edge-matching problems into CNF, as well as several extensions. The compact translation rarely resulted in the fastest performance, both for complete and incomplete SAT solvers. For complete solvers, the extensions even appeared crucial to solve the harder instances. Yet, these results are not very surprising and mostly show the extend of the importance of redundant clauses.

On the other hand, it is harder to explain why redundant clauses also guide the search for incomplete SAT solvers. Arguably, any performance gain due to adding redundant clauses could be interpreted as a flaw of the local search algorithm – redundant clauses only require additional resources. More research is needed to explain these results.

The focus of this paper, both the presentation and the experiments, is on bounded unsigned edge-matching problems. Translating unbounded and / or signed problems into CNF can be done in a similar manner. Future experiments will have to show whether SAT solvers can be used to solve these problems, such as Rubik's Tangle, too.

---

[6] Using the default settings with runs = 1,000, cut-off = 1,000,000

Within the domain of edge-matching problems, there remains the enormous challenge of constructing a translation of the Eternity II puzzle that could be solved with a SAT solver. The proposed encoding is merely a first step in this direction. A possible next step is to determine which other knowledge could be added using redundant clauses.

Regarding the big picture, the challenge arises how to translate a problem into CNF in general. The presented results suggest that adding redundant clauses can significantly reduce the computational costs. Therefore, further research on the use of redundant clauses may provide insight in how to meet this challenge. Also, the results show that the optimal encoding will not only depend on properties of a given problem, but also on the preferred solver, since complete and incomplete solvers will require different translations.

Acknowledgments

References

1. Armin Biere. PicoSAT Essentials. Journal on Satisfiability, Boolean Modeling and Computation **4** (2008), pp. 75–97.
2. Erik D. Demaine and Martin L. Demaine. Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity. Special issue on Computational Geometry and Graph Theory: The Akiyama-Chvatal Festschrift. Graphs and Combinatorics **23** (Supplement), June 2007, pages 195–208.
3. M. R. Dransfield, L. Liu, V. W. Marek and M. Truszczynski. Satisfiability and Computing van der Waerden Numbers. In The Electronic Journal of Combinatorics. Vol. **11** (1), (2004).
4. Niklas Eén and Niklas Sörensson, An extensible SAT-solver, In Giunchiglia and Tacchella (eds). Theory and applications of satisfiability testing, 6th international conference, SAT 2003. Santa Margherita Ligure, Italy, may 5-8, 2003 selected revised papers, Lecture Notes in Computer Science **2919** (2004), pp. 502–518.
5. Jacques Haubrich. Compendium of Card Matching Puzzles. Self-published, May 1995. Three volumes.
6. Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, Duc Nghia Pham: Neighbourhood Clause Weight Redistribution in Local Search for SAT. CP 2005 (2005), pp. 772–776.
7. João P. Marques-Silva and Inês Lynce. Towards Robust CNF Encodings of Cardinality Constraints. CP'07, Lecture Notes in Computer Science **4741** (2007).
8. Rogier Poldner. MINIZSAT: A semi SAT-based pseudo-Boolean solver. MSc thesis TU Delft (2008).
9. Pierre Schaus and Yves Deville. Hybridization of CP and VLNS for Eternity II. Journées Francophones de Programmation par Contraintes (JFPC'08) (2008).
10. Dave A.D. Tompkins and Holger H. Hoos. UBCSAT : An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. Lecture Notes in Computer Science **3542** (2005), pp. 306–320.

SAT COMPETITION 2009 :
RB-SAT benchmarks description

Haibo Huang[1], Chu Min Li[2], Nouredine Ould Mohamedou[2], and Ke Xu[1]

[1] wavebywind@163.com, kexu@nlsde.buaa.edu.cn
[2] {chu-min.li, nouredine.ould}@u-picardie.fr

1 Description of the initial problem

Model RB is a CSP model whose threshold points can be precisely located, and the instances generated at the threshold are guaranteed to be hard. In our paper [1] we use three different methods to encode the CSP instances generated from Model RB into SAT instances.We finally choose the most natural "direct" encoding method to define a new simple SAT model called RB-SAT. For further explanations, see [1].

2 Description of the SAT encoding

Model RB [2]: A class of random CSP instances generated following Model RB is denoted by $RB(k, n, \alpha, r, p)$ where, for each instance:

- $k \geq 2$ denotes the arity of each constraint,
- $n \geq 2$ denotes the number of variables,
- $\alpha > 0$ determines the domain size $d = n^\alpha$ of each variable,
- $r > 0$ determines the number $m = r.n.\ln n$ of constraints,
- $0 < p < 1$ determines the number $t = pd^k$ of disallowed (incompatible) tuples of each relation.

The following SAT model correspond to the direct encoding of Model RB (k=2), i.e. $RB\text{-}SAT(n, \alpha, r, p)$:

Step 1. Generate n disjoint sets of boolean variables, each of which has cardinality n^α (where $\alpha > 0$ is a constant), and then for every set, generate a clause which is the disjunction of all variables in this set, and for every two variables x and y in the same set, generate a binary clause $\neg x \vee \neg y$;

Step 2. Randomly select two different disjoint sets and then generate without repetitions $pn^{2\alpha}$ clauses of the form $\neg x \vee \neg z$ where x and z are two variables selected at random from these two sets respectively (where $0 < p < 1$ is a constant);

Step 3. Run Step 2 (with repetitions) for another $rn\ln(n)$-1 times (where $r > 0$ is a constant).

The instance can be forced to be satisfiable by randomly selecting a variable in each disjoint set and assigning true to it, other variables being assigned false and then keeping each constraint satisfied when selecting incompatible tuples of values for this constraint. We submit the two categories to SAT COMPETITION 2009.

3 Expected behavior of the solvers on those benchmarks

For solvers based on DPLL method, the RB-SAT instances are very difficult (exponential time). For solvers based on local search methods, these instances are exponentially harder than the random 3-SAT instances.

4 Random category

Instances of this class are generated by a generator **with repetition** of disjoint sets selected for the corresponding CSP. These benchmarks are submitted in the random category of SAT COMPETITION 2009.

5 Crafted category

Instances of this class are generated through a generator **without repetition** of disjoint sets selected for the corresponding CSP. These benchmarks are submitted in the crafted category of SAT COMPETITION 2009.

6 The benchmark test-sets

The distribution shown in table 1 is done for both categories: Random and Crafted. Here, we vary only the parameter n of Model RB-SAT.

n	rbsat-forced	rbsat-unforced	#instances	#vars	#clauses
40	rbsat-v760c43649gyes	rbsat-v760c43649g	10	760	43649
45	rbsat-v945c61409gyes	rbsat-v945c61409g	10	945	61409
50	rbsat-v1150c84314gyes	rbsat-v1150c84314g	10	1150	84314
55	rbsat-v1375c111739gyes	rbsat-v1375c111739g	10	1375	111739
60	rbsat-v1560c133795gyes	rbsat-v1560c133795g	10	1560	133795
65	rbsat-v1820c171155gyes	rbsat-v1820c171155g	10	1820	171155
70	rbsat-v2100c215164gyes	rbsat-v2100c215164g	10	2100	215164
75	rbsat-v2400c266431gyes	rbsat-v2400c266431g	10	2400	266431
80	rbsat-v2640c305320gyes	rbsat-v2640c305320g	10	2640	305320

Table 1. The benchmark test-sets: $RB\text{-}SAT(n, \alpha = 0.8, r = 3, p = 0.23)$

References

1. H. Huang, C. Li, N. Ould Mohamedou and K. Xu. SAT Encodings of Model RB and A New SAT Model, *submitted to SAT'09*.
2. K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems, *Journal of Artificial Intelligence Research* 12 (2000) 93-103.

Generator of satisfiable SAT instances

Milan Sesum, Predrag Janicic
Faculty of Mathematics, Belgrade

This program generates satisfiable SAT instances that correspond to the problem of known-plaintext attack on the cryptographic algorithm DES. DES has 56 bits long key. Finding all 56 bits by this approach is practically impossible. Because of that, it was assumed that some of 56 key bits are known and the problem is to find remaining, unknown bits. At the beginning, the program randomly generates a plaintext (of length 64) and a key (of length 56) and then finds corresponding ciphertext. After that, it uses the plaintext, the ciphertext and some of key bits in generating the corresponding SAT formula. The number of unknown key bits is a value of the second command line argument. The first command line argument is random seed parameter.

Details about this aproach can be found in the paper `http://www.matf.bg.ac.yu/~janicic/papers/frocos2005.zip`

The number of unknown key bits can be used as a fine tuning hardness parameter – as this number is growing up, the corresponding formula's hardness is growing up too.

REMARK: Generating a formula of any number of key bits should take about 20 seconds (depends on concrete hardware).

```
Usage: gss s n
       s - random seed parameter
       n - number of unknown key bits
```

Generator of SAT instances of unknown satisfiability

Milan Sesum, Predrag Janicic
Faculty of Mathematics, Belgrade

This program generates unsatisfiable SAT instances that correspond to the problem of finding a collision in the hash function MD5, with the length of messages equal to the the first command line argument. Because of the nature of hash functions (it should be almost impossible to find collisions in hash function, especially when length of starting message is small), it is extremely likely that formulae generated by this program are unsatisfiable (for length of messages less than 128).

Details about this aproach can be found in the paper `http://www.matf.bg.ac.yu/~janicic/papers/frocos2005.zip`

The length of starting message can be used as a fine tuning hardness parameter – as the length is growing up, the corresponding formula's hardness is growing up too.

REMARK: Generating a formula of any length should take about 1 minute (depends on concrete hardware). All formulae should be of size 4MB.

```
Usage: gus n
       n - length of messages (in bits)
```

Parity games instances

Oliver Friedmann

The generator is used to find parity games of a fixed size n that forced the strategy improvement algorithm to require at least i iterations. The generator creates predicate $P(n; i)$ in propositional logic that basically simulates a run of the strategy iteration on a game of size n with at least i iterations. Using a SAT solver to solve $P(n; i)$, one can draw the following conclusions:

SAT There is a game of size n that requires at least i iterations. Such a game can be extracted using the returned variable assignment for P(n; i).

UNSAT There is no game of size n that requires at least i iterations.

Since the exact number of iterations that is required to solve games of size n in the worst case is unknown, all instances that are generated are submitted as UNKNOWN.

The generator is a linux 32-bit executable that prints the specified benchmark to stdout in the usual DIMACS-format.

Although the generator understands quite a lot of parameters, there are two configurations, both of them depending on the two natural numbers N and I, that are recommended to be used for benchmarks:

```
1) -n N -i I -pp
2) -n N -i I -pp -ci -ce
```

The solution of the second configuration is of higher theoretical interest and should be tougher to be solved.

Note that the generated benchmarks will be very though already for very small N and I. It is also recommended to choose N and I in such a way that the difference between N and I is small.

C32SAT and CBMC instances

Hendrik Post, Carsten Sinz
University of Karlsruhe (TH)
Institute for Theoretical Computer Science
Research group "Verification meets Algorithm Engineering"
Am Fasanengarten 5
76131 Karlsruhe
Germany
post,sinz@ira.uka.de

1 Answer per benchmarks

```
post-c32s-col400-16  UNSAT
post-c32s-gcdm16-22  SAT
post-c32s-gcdm16-23  UNSAT
post-c32s-ss-8  UNSAT
post-cbmc-aes-d-r1  UNSAT
post-cbmc-aes-d-r2  UNSAT
post-cbmc-aes-ee-r2  UNSAT
post-cbmc-aes-ee-r3  UNSAT
post-cbmc-aes-ele  UNSAT
post-cbmc-zfcp-2.8-u2  SAT
```

2 Bounded Software Model Checking for C

c32sat[1] version 1.4.1 is used to prove properties about small, implicitely(*) represented, programs.

Author : Hendrik Post, University of Karlsruhe, Germany, post@ira.uka.de
Date : 9th of April 2008

CNF encodings of c32sat examples

- post-c32s-ss-8.cnf: Implicit representation of a selection sort implementation. The verification condition checks for all sets of a bounded size (=8) that elements are sorted after the application of the algorithm. (unsat, 15 mins with Minisat 2.0 beta)

- post-c32s-gcdm16-22.cnf and post-c32s-gcdm16-23.cnf: Implicit representation of condition that the gcd algorithm will not use more than 22/23 iterations. Unsat. for 23 iterations, sat for 22 iterations.

- post-c32s-col400-16.cnf: Sound encoding of the Collatz conjecture on 16 bit integers. The formula is unsat -¿ no 16 bit integer will cause more than 400 iterations of the Collatz conjecture.

(*) The verification conditions are generated and not extracted from C programs.

---

[1] http://fmv.jku.at/c32sat/

3 CBMC

1. post-cbmc-aes-d-r1.cnf
2. post-cbmc-aes-d-r2.cnf
3. post-cbmc-aes-ee-r2.cnf
4. post-cbmc-aes-ee-r3.cnf
5. post-cbmc-aes-ele.cnf
6. post-cbmc-zfcp-2.8-u2.cnf

```
#####
# 1 #
#####
```

```
c Benchmark for SATRace 2008
c Bounded Software Model Checking for C
c CBMC[3] (Version 2.8.) is used to prove that two AES implementations (Reference
c implementation[1] and Mike Scotts[2] C implementation) are equivalent.
c [1] Written by Paulo Barreto and Vincent Rijmen, K.U.Leuven
c [2] Written by Mike Scott, mike@compapp.dcu.ie
c [3] http://www.cprover.org/cbmc/
c Author : Hendrik Post, University of Karlsruhe, Germany, post@ira.uka.de
c Date   : 7th of April 2008
c Subtask: Decryption with 128 bit keys and block size, 1 rounds, arbitrary text and key
c Original output of CBMC:
c  size of program expression: 15868 assignments
c  Generated 9132 claims, 168 remaining
```

```
#####
# 2 #
#####
```

```
c Benchmark for SATRace 2008
c Bounded Software Model Checking for C
c CBMC[3] (Version 2.8.) is used to prove that two AES implementations (Reference
c implementation[1] and Mike Scotts[2] C implementation) are equivalent.
c [1] Written by Paulo Barreto and Vincent Rijmen, K.U.Leuven
c [2] Written by Mike Scott, mike@compapp.dcu.ie
c [3] http://www.cprover.org/cbmc/
c Author : Hendrik Post, University of Karlsruhe, Germany, post@ira.uka.de
c Date   : 7th of April 2008
c Subtask: Decryption with 128 bit keys and block size, 2 rounds, arbitrary text and key
c Original output of CBMC:
c size of program expression: 16863 assignments
c Generated 10192 claims, 408 remaining
```

```
#####
# 3 #
#####
```

```
c Benchmark for SATRace 2008
c Bounded Software Model Checking for C
c CBMC[3] (Version 2.8.) is used to prove that two AES implementations (Reference
c implementation[1] and Mike Scotts[2] C implementation) are equivalent.
c [1] Written by Paulo Barreto and Vincent Rijmen, K.U.Leuven
```

```
c [2] Written by Mike Scott, mike@compapp.dcu.ie
c [3] http://www.cprover.org/cbmc/
c Author : Hendrik Post, University of Karlsruhe, Germany, post@ira.uka.de
c Date   : 7th of April 2008
c Subtask: Encryption with 128 bit keys and block size, 2 rounds, arbitrary text and key
c Original output of CBMC:
c size of program expression: 16471 assignments
c Generated 112 claims, 16 remaining


#####
# 4 #
#####


c Benchmark for SATRace 2008
c Bounded Software Model Checking for C
c CBMC[3] (Version 2.8.) is used to prove that two AES implementations (Reference
c implementation[1] and Mike Scotts[2] C implementation) are equivalent.
c [1] Written by Paulo Barreto and Vincent Rijmen, K.U.Leuven
c [2] Written by Mike Scott, mike@compapp.dcu.ie
c [3] http://www.cprover.org/cbmc/
c Author : Hendrik Post, University of Karlsruhe, Germany, post@ira.uka.de
c Date   : 7th of April 2008
c Subtask: Encryption with 128 bit keys and block size, 3 rounds, arbitrary text and key
c Original output of CBMC:
c size of program expression: 17365 assignments
c Generated 140 claims, 16 remaining


#####
# 5 #
#####


c Benchmark for SATRace 2008
c Bounded Software Model Checking for C
c CBMC[3] (Version 2.8.) is used to prove that two AES implementations (Reference
c implementation[1] and Mike Scotts[2] C implementation) are equivalent.
c [1] Written by Paulo Barreto and Vincent Rijmen, K.U.Leuven
c [2] Written by Mike Scott, mike@compapp.dcu.ie
c [3] http://www.cprover.org/cbmc/
c Author : Hendrik Post, University of Karlsruhe, Germany, post@ira.uka.de
c Date   : 7th of April 2008
c Subtask: Decryption with 128 bit keys and block size,
c          Both implementations are equivalent for all encryption loop-body executions,
c          ,arbitrary text and key, arbitrary round number
c Original output of CBMC:
c size of program expression: 10502 assignments
c Generated 213 claims, 1 remaining


#####
# 6 #
#####


c Benchmark for SATRace 2008
c Bounded Software Model Checking for C
```

```
c CBMC[3] (Version 2.8.) is used to find runtime errors in the Linux zfcp-Subsystem.
c However, the unwinding bound is not set high enough, hence the instance is SAT.
c [3] http://www.cprover.org/cbmc/
c Author : Hendrik Post, University of Karlsruhe, Germany, post@ira.uka.de
c Date   : 7th of April 2008
c Original output of CBMC:
c size of program expression: 60896 assignments
c removed 51161 assignments
c Generated 53 claims, 52 remaining
```

sgen1: A generator of small, difficult satisfiability benchmarks

Ivor Spence

March 2009

The release includes a proposed set of unsatisfiable benchmarks in the directory unsat, and a proposed set of satisfiable benchmarks (together with corresponding models) in the directory sat.

The benchmark generator is compiled using

```
gcc -o sgen1 sgen1.c -lm
```

and the arguments are

```
sgen1 sat|unsat -n num-of-variables -s random-seed [-m model]
```

Thus a typical unsatisfiable benchmark can be generated by

```
sgen1 -unsat -n 61 -s 100 >x.cnf
```

A satisfiable benchmark, with its model, can be generated by

```
sgen1 -sat -n 61 -s 100 -m y.model >y.cnf
```

The supplied script file makesgenbenchmarks.bash generates the proposed benchmarks for submission.

SatSgi - Satisfiable SAT instances generated from Satisfiable Random Subgraph Isomorphism Instances

Calin Anton, Lane Olson
antonc@macewan.ca

March 2009

Random SGI model -SRSGI

For integers $n \leq m$, $0 \leq q \leq \frac{m(m-1)}{2}$, and integer percent $0 \leq p \leq 100$, a satisfiable random n, m, p, q consists of a graph H and a subgraph G. H is a G(n,m) random graph of order m and size q. (q of the $\frac{m(m-1)}{2}$ possible edges are selected at random without replacement.) G is obtained by the following two steps:

1. select at random an order n induced subgraph G' of H;

2. randomly remove floor((p/100)*(number of edges of G')) distinct edges of G'.

Therefore G is a subgraph of H, and the subgraph isomorphism problem: "Is G a subgraph of H" is satisfiable. The subgraph isomorphism problem is then encoded to SAT using the direct encoding.

Preliminary experiments indicate that the hardest instances are generated for:

m	q	n	p
20	152	18	5
21	168	19	10
22	173	20	20
23	189	21	25
24	193	22	30
26	243	24	30
28	302	26	25

Complete solvers find the instances with $p = 0$ difficult for $m - 4 \leq n \leq m - 2$ and $0.7\frac{m(m-1)}{2} \leq q \leq 0.8\frac{m(m-1)}{2}$.

SCIP – Solving Constraint Integer Programs

Timo Berthold[1][*], Stefan Heinz[1][*], Marc E. Pfetsch[2], and Michael Winkler[1]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
{berthold,heinz,michael.winkler}@zib.de
[2] Technische Universität Braunschweig, Institut für Mathematische Optimierung,
Pockelsstraße 14, 38106 Braunschweig, Germany
m.pfetsch@tu-bs.de

Abstract. This paper gives a brief description of the main components and techniques used in the pseudo-Boolean and constraint integer programming solver SCIP, which has been submitted for the pseudo-Boolean evaluation 2009 in two versions; the versions differ only in the solver interfaced to solve linear programming relaxations. The main components used within SCIP are constraint handlers, domain propagation, conflict analysis, cutting planes, primal heuristics, node selection, branching rules, and presolving.

1 Introduction

SCIP (Solving Constraint Integer Programs) is a framework for constraint integer programming, a problem class described in more detail in Section 2. A subclass of constraint integer programs are pseudo-Boolean problems and parts of SCIP provide a pseudo-Boolean solver.

In this paper we briefly describe the main components and techniques of the solver SCIP which are used to solve pseudo-Boolean optimization problems. Furthermore, we give for each component references to more detailed papers.

One main technique used in SCIP is a branch-and-bound procedure, which is a very general and widely used method to solve discrete optimization problems. The idea of *branching* is to successively subdivide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best of all solutions found in the subproblems yields the global optimum. During the course of the algorithm, a *branching tree* is generated in which each node represents one of the subproblems.

The intention of *bounding* is to avoid a complete enumeration of all potential solutions of the initial problem, which usually are exponentially many. For a minimization problem, the main observation is that if a subproblem's lower (dual) bound is greater than the global upper (primal) bound, the subproblem

---

Parts of this paper are a short version of [3].

[*] Supported by the DFG Research Center Matheon *Mathematics for key technologies* in Berlin.

can be pruned. Lower bounds are calculated with the help of a relaxation which should be easy to solve; typically such bounds are obtained with the help of a linear programming relaxation. Upper bounds are obtained by feasible solutions, found, e.g., if the solution of the relaxation is also feasible for the corresponding subproblem.

Good lower and upper bounds must be available for the bounding to be effective. In order to improve a subproblem's lower bound, one can tighten its (linear) relaxation, e.g., via domain propagation or by adding cutting planes (see Sections 4 and 6, respectively). Primal heuristics, which are described in Section 7, help to improve the upper bound.

The selection of the next subproblem in the search tree and the branching decision have a major impact on how early good primal solutions can be found and how fast the lower bounds of the subproblems increase. More details on branching and node selection are given in Section 8.

SCIP provides all necessary infrastructure to implement branch-and-bound based algorithms for solving constraint integer programs. It manages the branching tree along with all subproblem data, automatically updates the linear programming (LP) relaxation, and handles all necessary transformations due to presolving problem modifications, see Section 9. Additionally, a cut pool, cut filtering, and a SAT-like conflict analysis mechanism, see Section 5, are available. Furthermore, SCIP provides its own memory management and plenty of statistical output.

Besides the infrastructure, all main algorithms are implemented as external plugins. In the remainder of this paper, we will describe the key ingredients of SCIP. First, however, we define the class of problems which are solvable with this framework.

2 Constraint Integer Programming

Mixed integer programming (MIP) and the solution of *Boolean satisfiability problems (SAT)* are special cases of the general idea of *constraint programming (CP)*. The power of CP arises from the possibility to model a given problem directly with a huge variety of different, expressive constraints. In contrast, SAT and MIP only allow for very specific constraints: Boolean clauses for SAT and linear and integrality constraints for MIP. Their advantage, however, lies in the sophisticated techniques available to exploit the structure provided by these constraint types.

An important point for the efficiency of solving algorithms is the interaction between constraints. For instance, in SAT-solving, this takes place via propagation of the variables' domains. In MIP solving there exists a second, more complex but very powerful communication interface: the LP-relaxation.

The goal of constraint integer programming is to combine the advantages and compensate the weaknesses of CP, MIP, and SAT. To support this aim, we slightly restrict the notion of a CP, in order to be able to apply MIP and

SAT-solving techniques, and especially provide an LP-relaxation without losing (too much of) the high degree of freedom in modeling.

Definition (constraint integer program). A *constraint integer program* CIP = $(\mathfrak{C}, I, c)$ consists of solving

$$\text{(CIP)} \quad c^\star = \min\{c^T x \mid \mathfrak{C}(x), \ x \in \mathbb{R}^n, \ x_j \in \mathbb{Z} \text{ for all } j \in I\}$$

with a finite set $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ of constraints $\mathcal{C}_i : \mathbb{R}^n \to \{0, 1\}$, $i = 1, \ldots, m$, a subset $I \subseteq N = \{1, \ldots, n\}$ of the variable index set, and an objective function vector $c \in \mathbb{R}^n$. A CIP has to fulfill the following additional condition:

$$\forall \hat{x}_I \in \mathbb{Z}^I \ \exists (A', b') : \{x_C \in \mathbb{R}^C \mid \mathfrak{C}(\hat{x}_I, x_C)\} = \{x_C \in \mathbb{R}^C \mid A' x_C \leq b'\} \qquad (1)$$

with $C := N \setminus I$, $A' \in \mathbb{R}^{k \times C}$, and $b' \in \mathbb{R}^k$ for some $k \in \mathbb{Z}_{\geq 0}$.

Restriction (1) ensures that the subproblem remaining after fixing all integer variables always is a linear program. Note that this does not forbid quadratic or more involved expressions – as long as the nonlinearity only refers to the integer variables.

MIPs are special cases of CIPs in which all constraints are linear, i.e., $\mathfrak{C}(x) = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ for some $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$. One can also show that every CP with finite domains for all variables can be modeled as a CIP. The variables of *Pseudo-Boolean optimization problems (PB)* take 0/1 (false/true) values. Therefore, each PB is a CP with finite domains. In [14] is shown how to transform a PB into a CIP.

The main goal when applying CIP to PB problems is to use the MIP-machinery with LP-relaxations, cutting planes, elaborated branching rules, etc., and directly propagating the (nonlinear) multiplications of nonlinear PB problems. The hope is that on the one hand the fixings derived by domain propagation reduce the size of the LP and therefore potentially the computational overhead. On the other hand, these fixings may even yield a stronger LP-bound which vice versa can lead to further variable fixings which can be propagated and so forth.

For more details on the idea of constraint integer programming, see [2]. For an introduction to the integration of pseudo-Boolean optimization into SCIP, see [13]. For a discussion of how to handle nonlinear PB optimization, we refer to [14].

3 Constraint Handlers

Since a CIP consists of constraints, the central objects of SCIP are the *constraint handlers*. Each constraint handler represents the semantic of a single class of constraints and provides algorithms to handle constraints of the corresponding type. The primary task of a constraint handler is to check a given solution for feasibility with respect to all constraints of its type existing in the problem instance. This feasibility test suffices to provide an algorithm which correctly

solves CIPs with constraints of the supported types. To improve the performance of the solving process, constraint handlers may provide additional algorithms and information about their constraints to the framework, besides others

- presolving methods to simplify the problem's representation,
- propagation methods to tighten the variables' domains,
- a linear relaxation, which can be generated in advance or on the fly, that strengthens the LP relaxation of the problem, and
- branching decisions to split the problem into smaller subproblems, using structural knowledge of the constraints in order to generate a well-balanced branching tree.

The standard distribution of SCIP already includes a constraint handler for linear constraints that is needed to solve MIPs. Additionally, some specializations of linear constraints like knapsack, set partitioning, or variable bound constraints are supported by constraint handlers, which can exploit the special structure of these constraints in order to obtain more efficient data structures and algorithms. Furthermore, SCIP provides constraint handlers for logical constraints, such as AND, OR, and XOR constraints and for nonlinear constraints, like SOS1, SOS2, and indicator constraints.

4 Domain Propagation

Constraint propagation is an essential part of every CP solver [7]. The task is to analyze the set of constraints of the current subproblem and the current domains of the variables in order to infer additional valid constraints and domain reductions, thereby restricting the search space. The special case in which only the domains of the variables are affected by the propagation process is called *domain propagation*. If the propagation only tightens the lower and upper bounds of the domains without introducing holes it is called *bound propagation*.

In mixed integer programming, the concept of bound propagation is well-known under the term *node preprocessing*. Usually, MIP solvers apply restricted versions of the preprocessing algorithms, that are used before starting the branch-and-bound process, to simplify the subproblems [16, 26].

Besides the integrality restrictions, there are only linear constraints in a MIP. In contrast, CP models can include a large variety of constraint classes with different semantics and structures. Thus, a CP solver usually provides specialized constraint propagation algorithms for every single constraint class.

Constraint based (primal) domain propagation is supported by the constraint handler concept of SCIP. In addition, it features two dual domain reduction methods that are driven by the objective function, namely the *objective propagation* and the *root reduced cost strengthening* [24].

5 Conflict Analysis

Most MIP solvers discard infeasible and bound-exceeding subproblems without paying further attention to them. Modern SAT solvers, in contrast, try to

learn from infeasible subproblems, which is an idea due to Marques-Silva and Sakallah [23]. The infeasibilities are analyzed in order to generate so-called *conflict clauses*. These are implied clauses that help to prune the search tree. They also enable the solver to apply so-called *nonchronological backtracking*. A similar idea in CP are *no-goods*, see [27].

Conflict analysis can be generalized to CIP and, as a special case, to MIP and PB. There are two main differences of CIP and SAT solving in the context of conflict analysis. First, the variables of a CIP do not need to be of binary type. Therefore, we have to extend the concept of the conflict graph: it has to represent bound changes instead of variable fixings, see [1] for details.

Furthermore, the infeasibility of a subproblem in the CIP search tree usually has its reason in the LP relaxation of the subproblem. In this case, there is no single conflict-detecting constraint as in SAT or CP solving. To cope with this situation, we have to analyze the LP in order to identify a subset of all bound changes that suffices to render the LP infeasible or bound-exceeding. Note that it is an $\mathcal{NP}$-hard problem to identify a subset of the local bounds of *minimal cardinality* such that the LP stays infeasible if all other local bounds are removed. Therefore, we use a greedy heuristic approach based on an unbounded ray of the dual LP, see [1].

After having analyzed the LP, the algorithm works in the same fashion as conflict analysis for SAT instances: it constructs a conflict graph, chooses a cut in this graph, and produces a conflict constraint which consists of the bound changes along the frontier of this cut.

6   Cutting Plane Separators

Besides splitting the current subproblem into two or more easier subproblems by branching, one can also try to tighten the subproblem's relaxation in order to rule out the current LP solution $\check{x}$ and to obtain a different one. The LP relaxation can be tightened by introducing additional linear constraints $a^T x \leq b$ that are violated by $\check{x}$ but do not cut off feasible solutions of the subproblem. Thus, the current solution $\check{x}$ is *separated* from the convex hull of the feasible solutions of the subproblem by the *cutting plane $a^T x \leq b$*.

The theory of cutting planes is very well covered in the literature. For an overview of computationally useful cutting plane techniques, see [16, 22]. A recent survey of cutting plane literature can be found in [19].

SCIP features separators for knapsack cover cuts [8], complemented mixed integer rounding cuts [21], Gomory mixed integer cuts [17], strong Chvátal-Gomory cuts [20], flow cover cuts [25], implied bound cuts [26], and clique cuts [18, 26]. Detailed descriptions of these cutting plane algorithms and an extensive analysis of their computational impact can be found in [28].

Almost as important as finding cutting planes is the selection of the cuts that actually enter the LP relaxation. Balas, Ceria, and Cornuéjols [9] and Andreello, Caprara, and Fischetti [5] proposed to base the cut selection on *efficacy* and *orthogonality*. The efficacy is the Euclidean distance of the corresponding

hyperplane to the current LP solution. An orthogonality bound ensures that the cuts added to the LP form an almost pairwise orthogonal set of hyperplanes. SCIP follows these suggestions. Furthermore, it considers the parallelism w.r.t. the objective function.

7 Primal Heuristics

Primal heuristics have a significant relevance as supplementary procedures inside a MIP and PB solver: they aim at finding good feasible solutions early in the search process, which helps to prune the search tree by bounding and allows to apply more reduced cost fixing and other dual reductions that tighten the problem formulation.

Overall, there are 24 heuristics integrated into SCIP. They can be roughly subclassified into four categories:

- *Rounding heuristics* try to iteratively round the fractional values of an LP solution in such a way that the feasibility of the constraints is maintained or recovered by further roundings.
- *Diving heuristics* iteratively round a variable with fractional LP value and resolve the LP, thereby simulating a depth first search (see Section 8) in the branch-and-bound tree.
- *Objective diving heuristics* are similar to diving heuristics, but instead of fixing the variables by changing their bounds, they perform "soft fixings" by modifying their objective coefficients.
- *Improvement heuristics* consider one or more primal feasible solutions that have been previously found and try to construct an improved solution with better objective value.

Detailed descriptions of primal heuristics for mixed integer programs and an in-depth analysis of their computational impact can be found in [10], an overview is given in [11].

8 Node Selection and Branching Rules

Two of the most important decisions in a branch-and-bound algorithm are the selection of the next subproblem to process (*node selection*) and how to split the current problem Q into smaller subproblems (*branching rule*).

The most popular branching strategy in MIP solving is to split the domain of an integer variable x_j, $j \in I$, with fractional LP value $\check{x}_j \notin \mathbb{Z}$ into two parts, thus creating two subproblems $Q_1 = Q \cap \{x_j \leq \lfloor \check{x}_j \rfloor\}$ and $Q_2 = Q \cap \{x_j \geq \lceil \check{x}_j \rceil\}$. In case of a binary variable this boils down to $Q_1 = Q \cap \{x_j = 0\}$ and $Q_2 = Q \cap \{x_j = 1\}$. Several methods to select such a fractional variable for branching are discussed in [2, 4]. SCIP implements most of the discussed branching rules, especially *reliability branching* which is a very effective general branching rule for MIPs.

Besides a good branching strategy, the selection of the next subproblem to be processed is an important step of every branch-and-bound based search algorithm. There are essentially three methods: *depth first search*, *best bound search*, and *best estimate search*.

The default node selection strategy of SCIP is a combination of these three, which is also referred to as *interleaved best bound/best estimate search with plunging*.

9   Presolving

Presolving transforms the given problem instance into an equivalent instance that is (hopefully) easier to solve. The most fundamental presolving concepts for MIP are described in [26]. For additional information, see [16].

The task of presolving is threefold: first, it reduces the size of the model by removing irrelevant information such as redundant constraints or fixed variables. Second, it strengthens the LP relaxation of the model by exploiting integrality information, e.g., to tighten the bounds of the variables or to improve coefficients in the constraints. Third, it extracts information from the model such as implications or cliques which can be used later for branching and cutting plane separation. SCIP implements a full set of *primal* and *dual* presolving reductions for MIP problems, see [2].

Restarts differ from the classical presolving methods in that they are not applied *before* the branch-and-bound search commences, but abort a running search process in order to reapply other presolving mechanisms and start the search from scratch. They are a well-known ingredient of SAT solvers, but have not been used so far for solving MIPs.

Cutting planes, primal heuristics, strong branching [6], and reduced cost strengthening in the root node often identify fixings of variables that have not been detected during presolving. These fixings can trigger additional presolving reductions after a restart, thereby simplifying the problem instance and improving its LP relaxation. The downside is that we have to solve the root LP relaxation again, which can sometimes be very expensive.

Nevertheless, the above observation leads to the idea of applying a restart directly after the root node processing if a certain fraction of the integer variables has been fixed during the processing of the root node. In our implementation, a restart is performed if at least 5 % of the integer variables have been fixed.

10   Submission

We submitted two version of SCIP 1.1.0.7. They differ only in the solver applied to the LP relaxations. Version SCIPspx uses SoPlex 1.4.1 as LP solver. The other is equipped with the linear programming solver CLP 1.8.2 (SCIPclp). All these solvers are available in source code and free for noncommercial use [12, 15].

References

1. T. Achterberg, *Conflict analysis in mixed integer programming*, Discrete Optimization **4**, no. 1 (2007), pp. 4–20. Special issue: Mixed Integer Programming.

2. T. Achterberg, *Constraint Integer Programming*, PhD thesis, Technische Universität Berlin, 2007.

3. T. Achterberg, T. Berthold, S. Heinz, T. Koch, and K. Wolter, *Constraint Integer Programming: Techniques and Applications*, ZIB-Report 08-43, Zuse Institute Berlin, 2008.

4. T. Achterberg, T. Koch, and A. Martin, *Branching rules revisited*, Operations Research Letters **33** (2005), pp. 42–54.

5. G. Andreello, A. Caprara, and M. Fischetti, *Embedding cuts in a branch&cut framework: a computational study with $\{0, \frac{1}{2}\}$-cuts*, INFORMS Journal on Computing (2007). to appear.

6. D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem*, Princeton University Press, Princeton, 2006.

7. K. R. Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003.

8. E. Balas, *Facets of the knapsack polytope*, Mathematical Programming **8** (1975), pp. 146–164.

9. E. Balas, S. Ceria, and G. Cornuéjols, *Mixed 0-1 programming by lift-and-project in a branch-and-cut framework*, Management Science **42** (1996), pp. 1229–1246.

10. T. Berthold, *Primal heuristics for mixed integer programs*, Diploma thesis, Technische Universität Berlin, 2006.

11. T. Berthold, *Heuristics of the branch-cut-and-price-framework SCIP*, in Operations Research Proceedings 2007, J. Kalcsics and S. Nickel, eds., Springer-Verlag, 2008, pp. 31–36.

12. T. Berthold, S. Heinz, T. Koch, A. M. Gleixner, and K. Wolter, *ZIB Optimization Suite, documentation*. `http://zibopt.zib.de`.

13. T. Berthold, S. Heinz, and M. E. Pfetsch, *Solving pseudo-Boolean problems with SCIP*, ZIB-Report 08-12, Zuse Institute Berlin, 2008.

14. T. Berthold, S. Heinz, and M. E. Pfetsch, *Nonlinear pseudo-Boolean optimization: relaxation or propagation?*, ZIB-Report 09-11, ZIB, 2009.

15. J. J. H. Forrest, D. de la Nuez, and R. Lougee-Heimer, *CLP user guide*. COIN-OR, `http://www.coin-or.org/Clp/userguide`.

16. A. Fügenschuh and A. Martin, *Computational integer programming and cutting planes*, in Discrete Optimization, K. Aardal, G. L. Nemhauser, and R. Weismantel, eds., Handbooks in Operations Research and Management Science 12, Elsevier, 2005, ch. 2, pp. 69–122.

17. R. E. Gomory, *Solving linear programming problems in integers*, in Combinatorial Analysis, R. Bellman and J. M. Hall, eds., Symposia in Applied Mathematics X, Providence, RI, 1960, American Mathematical Society, pp. 211–215.

18. E. L. Johnson and M. W. Padberg, *Degree-two inequalities, clique facets, and biperfect graphs*, Annals of Discrete Mathematics **16** (1982), pp. 169–187.

19. A. Klar, *Cutting planes in mixed integer programming*, Diploma thesis, Technische Universität Berlin, 2006.

20. A. N. Letchford and A. Lodi, *Strengthening Chvátal-Gomory cuts and Gomory fractional cuts*, Operations Research Letters **30**, no. 2 (2002), pp. 74–82.

H. Marchand, *A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs*, PhD thesis, Faculté des Sciences Appliquées, Université catholique de Louvain, 1998.

22. H. Marchand, A. Martin, R. Weismantel, and L. A. Wolsey, *Cutting planes in integer and mixed integer programming*, Discrete Applied Mathematics **123/124** (2002), pp. 391–440.

23. J. P. Marques-Silva and K. A. Sakallah, *GRASP: A search algorithm for propositional satisfiability*, IEEE Transactions of Computers **48** (1999), pp. 506–521.

24. G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, 1988.

25. M. W. Padberg, T. J. van Roy, and L. A. Wolsey, *Valid inequalities for fixed charge problems*, Operations Research **33**, no. 4 (1985), pp. 842–861.

26. M. W. P. Savelsbergh, *Preprocessing and probing techniques for mixed integer programming problems*, ORSA Journal on Computing **6** (1994), pp. 445–454.

27. R. M. Stallman and G. J. Sussman, *Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis*, Artificial Intelligence **9** (1977), pp. 135–196.

28. K. Wolter, *Implementation of cutting plane separators for mixed integer programs*, Diploma thesis, Technische Universität Berlin, 2006.

SAT4J at the SAT09 competitive events

Daniel Le Berre   Anne Parrain

http://www.sat4j.org/

Overview of the SAT4J project

SAT4J is an open source library of SAT solvers written in Java. The initial aim of the SAT4J project was to allow Java programmers to access SAT technology directly in Java. As a consequence, the SAT4J library started in 2004 as an implementation in Java of the Minisat specification [6] and has been developed using both Java and Open Source coding standards. From a research point of view, SAT4J is the platform we use to develop our ideas on pseudo boolean solvers. As such, the SAT engine is generic and can handle arbitrary constraints (currently clause, cardinality and pseudo boolean constraints) as in the original Minisat. Such genericity has been designed over the years, and is one of the reason why SAT4J SAT engine is currently no longer competitive with state-of-the-art SAT solvers. However, the library has been adopted by various Java based software, in the area of software engineering[1], but also in bioinformatics[5, 9] or formal verification[1] [10]. The inclusion of SAT4J into the widely used Eclipse open platform[8] make it one of the most widely used SAT solver around the world (more than 13 Million downloads of Eclipse 3.4 in one year[2]).

SAT'09 competition

The SAT solver is based on the original Minisat implementation: the generic CDCL engine and the variable activity scheme has not changed. Most of the key component of the solvers have been made con-

---

[1]See Alloy4 (http://alloy4.mit.edu and Forge http://sdg.csail.mit.edu/forge/
[2]http://www.eclipse.org/downloads/

figurable. Here are the settings used for the SAT 09 competition, in SAT4J 2.1 RC1:

restarts The solver is using the rapid restart in/out strategy proposed by Armin Biere in Picosat[2].

clause minimization The conflict clause minimization of Minisat 1.14 (so called Expensive Simplification)[12] is used at the end of conflict analysis. Such minimization procedure has been made generic to be used with any kind of constraints (e.g. cardinality or pseudo boolean constraints).

phase selection The phase selection strategy is the lightweight caching scheme of RSAT[11].

learning strategy The solver learns only clauses whose size no longer than 10% of the number of variables. In practice, it does not change a lot the behavior of the solver on application benchmarks because the number of variables is large. On random or crafted categories, it might end up preventing the solver to learn any clause.

clause deletion strategy The solver monitors the available memory and removes half of the learned clauses (ordered by decreasing activity, as in the original minisat) when it drops below 10% of the total amount of memory.

PB'09 evaluation

Two different solvers have been submitted to the PB evaluation:

SAT4J Pseudo Resolution The solver is exactly the core SAT engine with the ability to process cardinality constraints and pseudo boolean constraints. Such constraints are considered as simple clauses during conflict analysis. As such, features such as conflict clause minimization are available in such solver. The proof system of the solver is still resolution so such solver cannot solve efficiently benchmarks such as pigeon holes for instance. The Pseudo Boolean solver PBClasp is using roughly the same approach in the PB'09 evaluation.

SAT4J Pseudo CutingPlanes That solver uses exactly the same settings as SATJ Pseudo Resolution but uses cutting planes instead of resolution during conflict analysis as described in [3, 7]. The proof system of the solver is thus more powerful than resolution and it allows to solve crafted benchmarks such as pigeon hole. However, the conflict analysis procedure is much more complex to implement than plain resolution and uses arbitrary precision arithmetic to avoid overflow. Furthermore, the solver gets slower when it learns new pseudo boolean constraints as it used to be the case in the 90's with SAT solvers because we haven't found yet a lazy data structure similar to the Head-Tails or watched literals for PB constraints to prevent it.

There are a few differences in the settings of the SAT engine when dealing with pseudo boolean constraints:

heuristics it takes into account the objective function is any to set first to false the literals that appear in the objective function to minimize.

restart strategy it uses the classical one found in Minisat.

learning strategy it learns all constraints.

other the formula cannot be simplified when literals at learned at decision level 0 (it is the case for pure SAT).

The optimization part is solved by strengthening: once a solution M is found, the value of the objective function for such solution is computed ($y = f(M)$). We add a new constraint in the solver to prevent solutions with value equal or greater than y to be found: $f < y$. Since all the added constraints are of the form $f < y'$ with $y' < y$, we simply keep one strengthening constraint per problem by replacing $f < y$ by $f < y'$ while keeping all learned constraints. Once the solver cannot find a new solution, the latest one is proved optimal.

MAXSAT'09 evaluation

The solver submitted to the MAXSAT09 evaluation basically translates the Partial Weighted Max SAT (PWMS) problems into Pseudo Boolean Optimization ones. Since all the other variants (MAXSAT, Partial MAXSAT and Weighted MAXSAT) can be considered as specific cases of PWMS, such approach can be used for all categories of the MAXSAT evaluation.

The idea is to add a new variable per weighted soft clause that represents that such clause has been violated, and to translate the maximization problem on those weighted soft clauses into a minimization problem on a linear function over those variables. Formally, suppose $T = \{C_1^{w1}, C_2^{w2}, \ldots, C_n^{wn}\}$ is the original set of weighted clauses of the weighted partial max sat problem. We translate that problem into $T' = \{s_1 \vee C_1, s_2 \vee C_2, \ldots, s_n \vee C_n\}$ plus the objective function $min : \Sigma_{i=1}^n wi \times s_i$.

That approach may look unapplicable in practice because on need to add as many new selector variables as clauses in the original problem. However, we have several cases for which no new variable is necessary:

hard clauses ($wi = \infty$) there is no need for new variables for hard clauses since they must be satisfied. They can be treated "as is" by the SAT solver.

unit soft clauses those constraints are violated when its literal is falsified. Has such, it is sufficient to consider that literal only in the optimization function, so no new selector variable is needed. In that case, the optimization function

is $min : \Sigma_{i=1}^{n} wi \times \neg l_i$ where l_i is the literal in the unit clause C_i.

The second optimization was introduced in SAT4J MAXSAT for the MAXSAT08 evaluation. As a consequence, our approach is expected to perform badly on pure MAXSAT or Weighted MAXSAT problems because we need to add as many new variables as clauses in the original problem. However, on Partial [Weighted] MAXSAT, depending on the proportion of soft clauses compared to the hard clauses, the number of additional variables can be negligible compared to the original number of variables. *This is especially true for instances of binate covering problem [4], a specific case of the partial weighted MAXSAT problem whose soft clauses are all unit, because we do not need to add any new selector variable in that case.*

The pseudo boolean solver used in SAT4J MAXSAT is SAT4J Pseudo Resolution since MAXSAT08 (in the two previous editions, it was SAT4J Pseudo Cutting Planes). Since those engines are tailored to solve application benchmarks, not randomly generated benchmarks, our approach is also expected to perform poorly on randomly generated PWMS problems.

References

[1] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[2] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[3] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *ACM/IEEE Design Automation Conference (DAC'03)*, pages 830–835, Anaheim, CA, 2003.

[4] Olivier Coudert. On solving covering problems. In *Design Automation Conference*, pages 197–202, 1996.

[5] Hidde de Jong and Michel Page. Search for steady states of piecewise-linear differential equation models of genetic regulatory networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 5(2):208–222, 2008.

[6] Niklas Eén Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[7] Heidi E. Dixon Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proceedings of The Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 635–640, 2002.

[8] Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem. In *Proceedings of IWOCE2009 - Open Component Ecosystems International Workshop*, August 2009.

[9] Jost Neigenfind, Gabor Gyetvai, Rico Basekow, Svenja Diehl, Ute Achenbach, Christiane Gebhardt, Joachim Selbig, and Birgit Kersten. Haplotype inference from unphased snp data in heterozygous polyploids based on sat. *BMC Genomics*, 9(1):356, 2008.

[10] Martin Ouimet and Kristina Lundqvist. The tasm toolset: Specification, simulation, and formal verification of real-time systems. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 126–130. Springer, 2007.

[11] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In Joao Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[12] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

Solving a Telecommunications Feature Subscription Configuration Problem

David Lesaint[1], Deepak Mehta[2], Barry O'Sullivan[2], Luis Quesada[2], and Nic Wilson[2]

[1] Intelligent Systems Research Centre, British Telecom, UK
david.lesaint@bt.com
[2] Cork Constraint Computation Centre, University College Cork, Ireland
{d.mehta,b.osullivan,l.quesada,n.wilson}@4c.ucc.ie

Abstract. Call control features (e.g., call-divert, voice-mail) are primitive options to which users can subscribe off-line to personalise their service. The configuration of a feature subscription involves choosing and sequencing features from a catalogue and is subject to constraints that prevent undesirable feature interactions at run-time. When the subscription requested by a user is inconsistent, one problem is to find an optimal relaxation. In this paper, we show that this problem is NP-hard and we present a constraint programming formulation using the variable weighted constraint satisfaction problem framework. We also present simple formulations using partial weighted maximum satisfiability and integer linear programming. We experimentally compare our formulations of the different approaches; the results suggest that our constraint programming approach is the best of the three overall.

1 Introduction

Information and communication services, from news feeds to internet telephony, are playing an increasing, and potentially disruptive, role in our daily lives. As a result, providers seek to develop personalisation solutions allowing customers to control and enrich their service. In telephony, for instance, personalisation relies on the provisioning of call control features. A feature is an increment of functionality which, if activated, modifies the basic service behaviour in systematic or non-systematic ways, e.g., do-not-disturb, multi-media ring-back tones, call-divert-on-busy, credit-card-calling, find-me.

Modern service delivery platforms provide the ability to implement features as modular applications and compose them on demand when setting up live sessions, that is, consistently with the feature subscriptions preconfigured by participants. In this context, a personalisation approach consists of exposing feature catalogues to subscribers and letting them select and sequence the features of their choice.

Not all sequences of features are acceptable though due to the possible occurrence of feature interactions. A feature interaction is "some way in which a feature modifies or influences the behaviour of another feature in generating the system's overall behaviour" [1]. For instance, a do-not-disturb feature will block any incoming call and cancel the effect of any subsequent feature subscribed by the callee. This is an undesirable interaction: as shown in Figure 1, the call originating from X will never reach
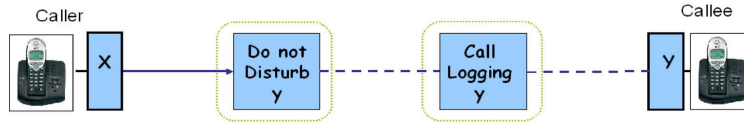
Fig. 1. An example of an undesirable feature interaction

call-logging. However, if call-logging is placed before do-not-disturb then both features will play their role.

Distributed Feature Composition (DFC) provides a method and a formal architecture model to address feature interactions [1–3]. The method consists of constraining the selection and sequencing of features by prescribing constraints that prevent undesirable interactions. These feature interaction resolution constraints are represented in a feature catalogue as precedence or exclusion constraints. A precedence constraint, $f_i \prec f_j$, means that if the features f_i and f_j are part of the same sequence then f_i must precede f_j in the sequence. An exclusion constraint between f_i and f_j means that they cannot be together in any sequence. Undesirable interactions are then avoided by rejecting any sequence that does not satisfy the catalogue constraints.

A *feature subscription* is defined by a set of features, a set of user specified precedence constraints and a set of feature interaction constraints from the catalogue. The main task is to find a sequence of features that is consistent with the constraints in the catalogue. It may not always be possible to construct a sequence of features that consists of all the user selected features and respect all user specified precedence constraints. In such cases, *the task is to find a relaxation of the feature subscription that is closest to the initial requirements of the user*.

In this paper, we shall show that checking the consistency of a feature subscription is polynomial in time, but finding an optimal relaxation of a feature subscription, when inconsistent, is NP-hard. We shall then present the formulation of finding an optimal relaxation using *constraint programming*. In particular, we shall use the variable weighted constraint satisfaction problem framework. In this framework, a branch and bound algorithm that maintains some level of consistency is usually used for finding an optimal solution. We shall investigate the impact of maintaining three different levels of consistency. The first one is Generalised Arc Consistency (GAC) [4], which is commonly used. The others are mixed consistencies. Here, mixed consistency means maintaining different levels of consistency on different sets of variables of a given problem. The first (second) mixed consistency enforces (a restricted version of) singleton GAC on some variables and GAC on the remaining variables of the problem.

We shall also consider *partial weighted maximum satisfiability*, an artificial intelligence technique, and *integer linear programming*, an operations research approach. We shall present the formulations using these approaches and shall discuss their differences with respect to the constraint programming formulation.

We have conducted experiments to compare the different approaches. The experiments are performed on a variety of random catalogues and random feature subscriptions. We shall present empirical results that demonstrate the superiority of maintaining

mixed consistency on the generalised arc consistency. For hard problems, we see a difference of up to three orders of magnitude in terms of search nodes and one order of magnitude in terms of time. Our results suggest that, when singleton generalised arc consistency is used, the constraint programming approach considerably outperforms our integer linear programming and partial weighted maximum satisfiability formulations. We highlight the factors that deteriorate the scalability of the latter approaches.

The rest of the paper is organised as follows. Section 2 provides an overview of the DFC architecture, its composition style and subscription configuration method. Section 3 presents the relevant definitions and theorems. Section 4 describes the constraint programming formulation for finding an optimal relaxation and discusses branch and bound algorithms that maintain different levels of consistency. The integer linear programming and partial weighted maximum satisfiability formulations of the problem are described in Section 5. The empirical evaluation of these approaches is shown in Section 6. Finally our conclusions are presented in Section 7.

2 Configuring Feature Subscriptions in DFC

In DFC each feature is implemented by one or more modules called *feature box types* (FBT) and each FBT has many run-time instances called *feature boxes*. We assume in this paper that each feature is implemented by a single FBT and we associate features with FBTs. As shown in Figure 2, a call session between two end-points is set up by chaining feature boxes. The routing method decomposes the connection path into a source and a target region and each region into *zones*. A source (target) zone is a sequence of feature boxes that execute for the same source (target) address.

The first source zone is associated with the source address encapsulated in the initial setup request, e.g., zone of X in Figure 2. A change of source address in the source region, caused for instance by an identification feature, triggers the creation of a new source zone [5]. If no such change occurs in a source zone and the zone cannot be expanded further, routers switch to the target region. Likewise, a change of target address in the target region, as performed

Fig. 2. DFC: Catalogues, subscriptions and sessions.

by Time-Dependent-Routing (TDR) in Figure 2, triggers the creation of a new target zone. If no such change occurs in a target zone and the zone cannot be expanded further (as for Z in Figure 2), the request is sent to the final box identified by the encapsulated target address.
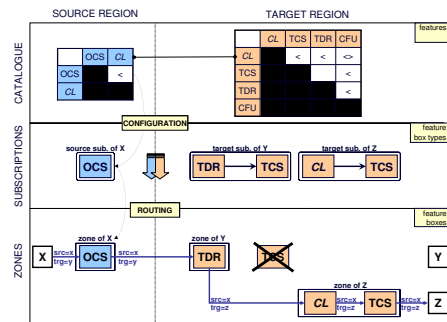
DFC routers are only concerned with locating feature boxes and assembling zones into regions. They do not make decisions as to the type of feature boxes (the FBTs)

appearing in zones or their ordering. They simply fetch this information from the *feature subscriptions* that are preconfigured for each address in each region based on the *catalogue* published by the service provider.

A catalogue is a set of features subject to precedence and exclusion constraints. Features fall into three classes: *source*, *target* and *reversible*, i.e., a subset of features that are both source and target. Constraints are formulated by designers on pairs of source features and pairs of target features to prevent undesirable feature interactions in each zone [6]. Specifically, a precedence constraint imposes a routing order between two features, as for the case of Terminating-Call-Screening (TCS) and Call-Logging (CL) in Figure 2. An exclusion constraint makes two features mutually exclusive, as for the case of CL and Call-Forwarding-Unconditional (CFU) in Figure 2.

A subscription is a subset of catalogue features and a set of user precedence constraints between features in each region. For instance, the subscription of Y in the target region includes the user precedence TDR$\prec$TCS. Configuring a subscription involves selecting, parameterising and sequencing features in each region consistently with the catalogue constraints and other integrity rules [3]. In particular, the source and target regions of a subscription must include the same reversible features in inverse order, i.e. source and target regions are not configured independently.

3   Formal Definitions

Let f_i and f_j be features, we write a precedence constraint of f_i before f_j as $\langle f_i, f_j \rangle$, or as $f_i \prec f_j$. An exclusion constraint between f_i and f_j expresses that these features cannot appear together in a sequence of features. We encode this as the pair of precedence constraints $\langle f_i, f_j \rangle$ and $\langle f_j, f_i \rangle$.

Definition 1 (Feature Catalogue). *A catalogue is a tuple $\langle F, P \rangle$, where F is a set of features that are available to users and P is a set of precedence constraints on F.*

The *transpose* of a catalogue $\langle F, P \rangle$ is the catalogue $\langle F, P^T \rangle$ such that $\forall \langle f_i, f_j \rangle \in F^2 :$ $\langle f_i, f_j \rangle \in P \Leftrightarrow \langle f_j, f_i \rangle \in P^T$. In DFC the precedence constraints between the features in the source (target) catalogue are specified with respect to the direction of the call. For the purpose of configuration, we combine the source catalogue $\langle F_s, P_s \rangle$ and the target catalogue $\langle F_t, P_t \rangle$ into a single catalogue $\langle F_c, P_c \rangle \equiv \langle F_s \cup F_t, P_s \cup P_t^T \rangle$.

Definition 2 (Feature Subscription). *A feature subscription S of catalogue $\langle F_c, P_c \rangle$ is a tuple $\langle F, C, U, W_F, W_U \rangle$, where $F \subseteq F_c$, C is the projection of P_c on F, i.e., $P_c \downarrow_F = \{ f_i \prec f_j \in P_c : \{f_i, f_j\} \subseteq F \}$, U is a set of (user defined) precedence constraints on F, $W_F : F \to \mathbb{N}$ is a function that assigns weights to features and $W_U : U \to \mathbb{N}$ is a function that assigns weights to user precedence constraints. The value of S is defined by $\mathrm{Value}(S) = \sum_{f \in F} W_F(f) + \sum_{p \in U} W_U(p)$.*

Note that a weight associated with a feature signifies its importance for the user. These weights could be elicited directly, or using data mining or analysis of user interactions.

Definition 3 (Consistency). *A feature subscription $\langle F, C, U, W_F, W_U \rangle$ of some catalogue is defined to be consistent if and only if the directed graph $\langle F, C \cup U \rangle$ is acyclic.*

Due to the composition of the source and target catalogues into a single catalogue, a feature subscription S is consistent if and only if both source and target regions are consistent in the DFC sense.

Theorem 1 (Complexity of Consistency Checking). *Determining whether a feature subscription $\langle F, C, U, W_F, W_U \rangle$ is consistent or not can be checked in $\mathcal{O}(|F| + |C| + |U|)$.*

Proof. We use *Topological Sort* [7]. In Topological Sort we are interested in ordering the nodes of a directed graph such that if the edge $\langle i, j \rangle$ is in the set of edges of the graph then node i is less than node j in the order. In order to use *Topological Sort* for detecting whether a feature subscription is consistent, we associate nodes with features and edges with precedence constraints. Then, the subscription is consistent iff for all edges $\langle i, j \rangle$ in the graph associated with the subscription we have that $i \prec j$ in the order computed by *Topological Sort*. As the complexity of *Topological Sort* is linear with respect to the size of the graph, detecting whether a feature subscription is consistent is $\mathcal{O}(|F| + |C| + |U|)$. □

If an input feature subscription is not consistent then the task is to relax the given feature subscription by dropping one or more features or user precedence constraints to generate a consistent feature subscription with maximum value.

Definition 4 (Relaxation). *A relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ is a subscription $\langle F', C', U', W'_F, W'_U \rangle$ such that $F' \subseteq F$, $C' = P_c \!\downarrow_{F'}$, $U' \subseteq U \!\downarrow_{F'}$, $W_{F'}$ is W_F restricted to F', and $W_{U'}$ is W_U restricted to U'.*

Definition 5 (Optimal Relaxation). *Let R_S be the set of all consistent relaxations of a feature subscription S. We say that $S_i \in R_S$ is an optimal relaxation of S if it has maximum value among all relaxations, i.e., if and only if there does not exist $S_j \in R_S$ such that $\mathrm{Value}(S_j) > \mathrm{Value}(S_i)$.*

Theorem 2 (Complexity of Finding an Optimal Relaxation). *Finding an optimal relaxation of a feature subscription is NP-hard.*

Proof. Given a directed graph $\langle V, E \rangle$, the *Feedback Vertex Set Problem* is to find a smallest $V' \subseteq V$ whose deletion makes the graph acyclic. This problem is known to be NP-hard [8]. We prove that finding an optimal relaxation is NP-hard by reducing the feedback vertex set problem to the latter. Given a feature subscription $S = \langle F, C, U, W_F, W_U \rangle$, the feedback vertex set problem can be reduced to our problem by associating the nodes of the directed graph V with features F, the edges E with catalogue precedence constraints C, the empty set $\emptyset$ with U, and the constant function that maps every element of its domain to 1 ($\lambda x.1$) with both W_F and W_U. Notice that, as $U = \emptyset$, the only way of finding an optimal relaxation of S is by removing a set of features from F. Assuming that an optimal relaxation is $S' = \langle F', C', U', W'_F, W'_U \rangle$, the set of features $F - F'$ corresponds to the smallest set of nodes V' whose deletion makes the directed graph acyclic. Thus, we can conclude that finding an optimal relaxation S' is NP-hard. □

4   A Constraint Programming Approach

Constraint programming has been successfully used in many applications such as planning, scheduling, resource allocation, routing, and bio-informatics [9]. Here problems are primarily stated as a Constraint Satisfaction Problems (CSP), that is a finite set of variables, together with a finite set of constraints. A solution to a CSP is an assignment of a value to each variable such that all constraints are satisfied simultaneously. The basic approach to solving a CSP instance is to use a backtracking search algorithm that interleaves two processes: *constraint propagation* and *labeling*. Constraint propagation helps in pruning values that do not lead to a solution of the problem. Labeling involves assigning values to variables that may lead to a solution.

Various generalisations of the CSP have been developed to find a solution that is optimal with respect to certain criteria such as costs, preferences or priorities. One of the most significant is the Constraint Optimisation Problem (COP). Here the goal to find an optimal solution that maximises (minimises) the objective function. The simplest COP formulation retains the CSP limitation of allowing only hard Boolean-valued constraints but adds an objective function over the variables.

4.1   Formulation

In this section we model the problem of finding an optimal relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ as a COP .

Variables and Domains.   We associate each feature $f_i \in F$ with two variables: a *Boolean variable* bf_i and an *integer variable* pf_i. A Boolean variable bf_i is instantiated to 1 or 0 depending on whether f_i is included in the subscription or not, respectively. The domain of each integer variable pf_i is $\{1, \ldots, |F|\}$. Assuming that the computed subscription is consistent, an integer variable pf_i corresponds to the position of the feature f_i in a sequence. We associate each user precedence constraint $p_{ij} \equiv (f_i \prec f_j) \in U$ with a *Boolean variable* bp_{ij}. A Boolean variable bp_{ij} is instantiated to 1 or 0 depending on whether p_{ij} is respected in the computed subscription or not respectively.

Constraints.   A catalogue precedence constraint $p_{ij} \in C$ that feature f_i should be before feature f_j can be expressed as follows:

$$bf_i \wedge bf_j \;\Rightarrow\; (pf_i < pf_j).$$

Note that the constraint is activated only if the selection variables bf_i and bf_j are instantiated to 1. A user precedence constraint $p_{ij} \in U$ that f_i should be placed before f_j in their subscription can be expressed as follows:

$$bp_{ij} \;\Rightarrow\; (bf_i \wedge bf_j \wedge (pf_i < pf_j)).$$

Note that if a user precedence constraint holds then the features f_i and f_j are included in the subscription and also the feature f_i is placed before f_j, that is, the selection variables bf_i and bf_j are instantiated to 1 and $pf_i < pf_j$ is true.

Objective Function. The objective of finding an optimal relaxation of a feature subscription can be expressed as follows:

$$\text{Maximise} \sum_{f_i \in F} bf_i \times W_F(f_i) + \sum_{p_{ij} \in U} bp_{ij} \times W_U(p_{ij}).$$

4.2 Solution Technique

A depth-first branch and bound algorithm (BB) is generally used to find an optimal solution. In case of maximisation, BB keeps the current optimal value of the solution while traversing the search tree. This value is a lower bound (lb) of the objective function. At each node of the search tree BB computes an overestimation of the global value. This value is an upper bound (ub) of the best solution that can be found as long as the current search path is maintained. If $ub \leq lb$, then a solution of a greater value than the current optimal value cannot be found below the current node, so the current branch is pruned and the algorithm backtracks.

Enforcing local consistency enables the computation of $ub_{(i,a)}$, which is a specialisation of ub for a value a of an unassigned variable i. If $ub_{(i,a)} \leq lb$, then value a can be removed because it will not be present in any solution better than the current one. Removed values are restored when BB backtracks above the node where they were eliminated. The quality of the upper bound can be improved by increasing the level of local consistency that is maintained at each node of the search tree. The different levels of local consistencies that we have considered are *generalised Arc Consistency* (GAC) [4] and *mixed consistency* [10].

A problem is said to be *generalised arc consistent* if it has non-empty domains and for any assignment of a variable each constraint in which that variable is involved can be satisfied. A problem is said to be *singleton generalised arc consistent* [11] if it has non-empty domains and for any assignment of a variable, the resulting subproblem can be made GAC. Enforcing Singleton generalised Arc Consistency (SGAC) in a SAC-1 manner [12] works by having an outer loop consisting of variable-value pairs of the form (x, a). For each a in the domain of x if there is a domain wipeout while enforcing arc consistency then a is removed from the domain of x and arc consistency is enforced. The main problem with SAC-1 is that deleting a single value triggers the outer loop again. The Restricted SAC (RSAC) algorithm avoids this triggering by considering each variable-value pair only once [13].

Mixed consistency means maintaining different levels of consistency on different variables of a problem. In [14] it has been shown that maintaining mixed consistency, in particular maintaining SAC on some variables and maintaining arc consistency on some variables, can reduce the solution time for some CSPs. In this paper we shall study the effect of maintaining different levels of consistency on different sets of variables within a branch and bound search. We shall investigate the effect of Maintaining generalised Singleton Arc Consistency (MGSAC) on the Boolean variables and Maintaining generalised Arc Consistency (MGAC) on the remaining variables of the problem. We shall also investigate the effect of Maintaining Restricted Singleton generalised Arc Consistency (MRSGAC) on the Boolean variables and MGAC on the remaining variables. The former shall be denoted by MSGAC_b and the latter by MRSGAC_b. Results presented in

Section 6 suggest that maintaining singleton generalised arc consistency on the Boolean variables of the random instances of the feature subscription configuration problem reduces the search space and time of the branch and bound algorithm significantly.

5 Other Approaches

We present a partial weighted maximum Boolean satisfiability and an integer linear programming formulation for finding an optimal relaxation of a feature subscription.

5.1 Partial Weighted Maximum Boolean Satisfiability

The Boolean Satisfiability Problem (SAT) is a decision problem whose instance is an expression in propositional logic written using only $\wedge$, $\vee$, $\neg$, variables and parentheses. The problem is to decide whether there is an assignment of *true* and *false* values to the variables that will make the expression *true*. The expression is normally written in conjunctive normal form. The Partial Weighted Maximum Boolean Satisfiability Problem (PWMSAT) is an extension of SAT that includes the notions of hard and soft clauses. Any solution should respect the hard clauses. Soft clauses are associated with weights. The goal is to find an assignment that maximises the sum of the weights of the satisfied clauses. The PWMSAT formulation of finding an optimal relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ is outlined below.

Variables. Let *PrecDom* be the set of possible precedence constraints that can be defined on F, i.e., $\{f_i \prec f_j : \{f_i, f_j\} \subseteq F \wedge f_i \neq f_j\}$). For each feature $f_i \in F$ there is a Boolean variable bf_i, which is true or false depending on whether feature f_i is included or not in the computed subscription. For each precedence constraint p_{ij} there is a Boolean variable bp_{ij}, which is true or false depending on whether the precedence constraint $f_i \prec f_j$ holds or not in the computed subscription.

Clauses. In our model, clauses are represented with a tuple $\langle w, c \rangle$, where w is the weight of clause and c is the clause itself. Note that the hard clauses are associated with weight $\top$, which represents an infinite penalty for not satisfying the clause. Each precedence constraint $p_{ij} \in C$ must be satisfied if the features f_i and f_j are included in the computed subscription. We model this by adding the following clause

$$\langle \top, (\neg bf_i \vee \neg bf_j \vee bp_{ij}) \rangle.$$

The precedence relation should be transitive and asymmetric in order to ensure that the subscription graph is acyclic. In order to ensure this, for every $\{p_{ij}, p_{jk}\} \subseteq PrecDom$, we add the following clause:

$$\langle \top, (\neg bp_{ij} \vee \neg bp_{jk} \vee bp_{ik}) \rangle. \tag{1}$$

Note that Rule (1) need only be applied to $\langle i, j, k \rangle$ such that $i \neq k$ because of Rule (2) below. In our model, both bp_{ij} and bp_{ji} can be false. However, if one of them is true

the other one is false. As this should be the case for any precedence relation, we add the following clause for every $p_{ij} \in$ *PrecDom*:

$$\langle \top, (\neg bp_{ij} \vee \neg bp_{ji}) \rangle. \tag{2}$$

We make sure that each precedence constraint $p_{ij} \in$ *PrecDom* is only satisfied when its features are included by considering the following clauses:

$$\langle \top, (bf_i \vee \neg bp_{ij}) \rangle \quad \langle \top, (bf_j \vee \neg bp_{ij}) \rangle.$$

We need to penalise any solution that does not include a feature $f_i \in F$ or a user precedence constraint $p_{ij} \in U$. This is done by adding the following clauses:

$$\langle wf_i, (bf_i) \rangle \quad \langle wp_{ij}, (bp_{ij}) \rangle,$$

where $wf_i = W_F(f_i)$ and $wp_{ij} = W_U(\langle f_i, f_j \rangle)$. The cost of violating these clauses is the weight of the feature f_i and the weight of the precedence constraint p_{ij} respectively.

The number of Boolean variables in the PWMSAT model (approximately $|F|^2$) is greater than the number of Boolean variables in the CP model ($|F| + |U|$). These extra variables are used by Rule (1) and (2) to avoid cycles in the final subscription graph. We remark that the subscription contains a cycle if and only if the transitive closure of $C \cup U$ contains a cycle. Therefore, it is sufficient to associate Boolean variables only with the precedence constraints in the transitive closure of $C \cup U$. Reducing these variables will also reduce the transitive clauses, especially when the input subscription graph is not dense. Otherwise, Rule (1) will generate $|F| \times (|F|-1) \times (|F|-2)$ transitivity clauses. For example, for the subscription $\langle F, C, U, W_F, W_U \rangle$ with $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$, $C = \{p_{12}, p_{21}, p_{34}, p_{43}, p_{56}, p_{65}\}$, and $U = \emptyset$, Rule (1) will generate 120 transitive clauses. Since any relaxation of the given subscription respecting the clauses generated by Rule (2) is acyclic, the 120 transitive clauses are useless. Thus, if *PrecDom* is instead set to be the transitive closure of $C \cup U$, then Rule (1) would not generate any clause for the mentioned example. Another way to reduce the number of transitive clauses is by not considering the ones where $\{p_{ji}, p_{kj}, p_{ik}\} \cap C \neq \emptyset$, especially when the input subscription graph is not sparse. The reason is that these transitive clauses are always entailed due to the enforcement of the catalogue precedence constraints.

Note that the two techniques described before for reducing the number of transitive clauses complement each other. This reduction in the number of clauses might have an impact on the runtime of the PWMSAT approach, since less memory might be needed. Even though it is sufficient to associate a Boolean variable with each precedence constraint in the transitive closure of $C \cup U$, it is still greater than $|F| + |U|$. Another way of reducing the number of variables is to associate a feature with a finite domain variable representing its position (as done in the CP model), log-encode the finite domain variables, and express the precedence constraints using a lexicographical comparator [15]. This approach indeed uses fewer variables than the implemented approach since only $|F| \times \log |F|$ variables are needed for encoding the position variables. However, it is not so straightforward to automatically translate the resulting Boolean formula into its corresponding conjunctive normal form.

5.2 Integer Linear Programming

In Linear Programming the goal is to optimise an objective function subject to linear equality and inequality constraints. When all the variables are forced to be integer-valued, the problem is an Integer Linear Programming (ILP) problem. The standard way of expressing these problems is by presenting the function to be optimised, the linear constraints to be respected and the domain of the variables involved. Both the CP and the PWMSAT formulations for finding an optimal relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ can be modeled in ILP. The translation of the PWMSAT formulation into ILP formulation is straightforward. For this particular model, we observed that CPLEX was not able to solve even simple problems within a time limit of 4 hours. Due to the lack of space we shall describe neither the formulation nor its corresponding results. The ILP formulation that is equivalent to the CP formulation is outlined below.

Variables. For each $f_i \in F$, we use a binary variable bf_i and an integer variable pf_i. A binary variable bf_i is equal to 1 or 0 depending on whether feature f_i is included or not. An integer variable pf_i is the position of feature f_i in the final subscription. For each user precedence constraint $p_{ij} \in U$, we use a binary variable bp_{ij}. It is instantiated to 1 or 0 depending on whether the precedence constraint $f_i \prec f_j$ holds or not.

Linear Inequalities. If the features f_i and f_j are included in the computed subscription and if $p_{ij} \in C$ then the position of feature f_i must be less than the position of feature f_j. To this effect, we need to translate the underlying implication $(bf_i \wedge bf_j \Rightarrow (pf_i < pf_j))$ into the following linear inequality:

$$pf_i - pf_j + n * bf_i + n * bf_j \leq 2n - 1 . \tag{3}$$

Here, n is a constant that is used to refer to the number of features $|F|$ selected by the user. When both bf_i and bf_j are 1, Inequality (3) will force $(pf_i < pf_j)$. Note that this is not required for any user precedence constraint $p_{ij} \in U$, since it can be violated.

A user precedence $p_{ij} \in U$ is equivalent to the implication $bp_{ij} \Rightarrow pf_i < pf_j \wedge bf_i \wedge bf_j$, which in turn is equivalent to the conjunction of the three implications $(bp_{ij} \Rightarrow (pf_i < pf_j))$, $(bp_{ij} \Rightarrow bf_i)$ and $(bp_{ij} \Rightarrow bf_j)$. These implications can be translated into the following inequalities:

$$pf_i - pf_j + n * bp_{ij} \leq n - 1 \tag{4}$$

$$bp_{ij} - bf_i \leq 0 \tag{5}$$

$$bp_{ij} - bf_j \leq 0 . \tag{6}$$

Inequality (4) means that $bp_{ij} = 1$ forces $pf_i < pf_j$ to be true. Also, if $bp_{ij} = 1$ then both bf_i and bf_j are equal to 1 from Inequalities (5) and (6) respectively.

Objective Function. The objective is to find an optimal relaxation of a feature subscription configuration problem $\langle F, C, U, W_F, W_U \rangle$ that maximises the sum of the weights of the features and the user precedence constraints that are selected:

$$\text{Maximise} \sum_{f_i \in F} wf_i \, bf_i + \sum_{p_{ij} \in U} wp_{ij} \, bp_{ij}.$$

6 Experimental Results

In this section, we shall describe the empirical evaluation of finding an optimal relaxation of randomly generated feature subscriptions using constraint programming, partial weighted maximum Boolean satisfiability and integer linear programming.

6.1 Problem Generation and Solvers

We generated and experimented with a variety of *random catalogues* and many classes of *random feature subscriptions*. All the random selections below are performed with uniform distributions. A random catalogue is defined by a tuple $\langle f_c, B_c, T_c \rangle$. Here, f_c is the number of features, B_c is the number of binary constraints and $T_c \subseteq \{<, >, <>\}$ is a set of types of constraints. Note that $f_i <> f_j$ means that in any given subscription both f_i and f_j cannot exist together. A random catalogue is generated by selecting B_c pairs of features randomly from $f_c(f_c - 1)/2$ pairs of features. Each selected pair of features is then associated with a type of constraint that is selected randomly from T_c. A random feature subscription is defined by a tuple $\langle f_u, p_u, w \rangle$. Here, f_u is the number of features that are selected randomly from f_c features, p_u is the number of user precedence constraints between the pairs of features that are selected randomly from $f_u(f_u - 1)/2$ pairs of features, and w is an integer greater than 0. Each feature and each user precedence constraint is associated with an integer weight that is selected randomly between 1 and w inclusive.

We generated catalogues of the following forms: $\langle 50, 250, \{<, >\} \rangle$, $\langle 50, 500, \{<, >, <>\} \rangle$ and $\langle 50, 750, \{<, >\} \rangle$. For each random catalogue, we generated classes of feature subscriptions of the following forms: $\langle 10, 5, 4 \rangle$, $\langle 15, 20, 4 \rangle$, $\langle 20, 10, 4 \rangle$, $\langle 25, 40, 4 \rangle$, $\langle 30, 20, 4 \rangle$, $\langle 35, 35, 4 \rangle$, $\langle 40, 40, 4 \rangle$, $\langle 45, 90, 4 \rangle$ and $\langle 50, 5, 4 \rangle$. Note that $\langle 50, 250, \{<, >\} \rangle$ is the default catalogue by and the value of w is 4 by default, unless stated otherwise. For the catalogue $\langle 50, 250, \{<, >\} \rangle$ we also generated $\langle 5, 0, 1 \rangle$, $\langle 10, 0, 1 \rangle, \ldots, \langle 50, 0, 1 \rangle$ and $\langle 5, 5, 1 \rangle$, $\langle 10, 10, 1 \rangle, \ldots, \langle 50, 50, 1 \rangle$ classes of random feature subscriptions. For each class 10 instances were generated and their mean results are reported in this paper.

The CP model was implemented and solved using CHOCO [16], a Java library for constraint programming systems. The PWMSAT model of the problem was implemented and solved using SAT4J [17], an efficient library of SAT solvers in Java. The ILP model of the problem was solved using ILOG CPLEX [18]. All the experiments were performed on a PC Pentium 4 (CPU 1.8 GHz and 768MB of RAM) processor. The performances of all the approaches are measured in terms of search nodes (#nodes) and runtime in milliseconds (time). We used the time limit of 4 hours to cut the search.

6.2 Maintaining Different Levels of Consistency in CP

For the CP model, we first investigated the effect of Maintaining generalised Arc Consistency (MGAC) during branch and bound search. We then studied the effect of maintaining different levels of consistency on different sets of variables within a problem. In particular we investigated, (1) maintaining generalised singleton arc consistency on the Boolean variables and MGAC on the remaining variables, and (2) maintaining restricted

singleton generalised arc consistency on the Boolean variables and MGAC on the remaining variables; the former is denoted by MSGAC$_b$ and the latter by MRSGAC$_b$. The results are presented in Table 1 for these three branch and bound search algorithms.

Table 1 clearly shows that maintaining (R)SGAC on the Boolean variables and GAC on the integer variables dominates maintaining GAC on all the variables. To the best of our knowledge this is the first time that such a significant improvement has been observed by maintaining a partial form of singleton arc consistency. We also

$\langle f, p \rangle$	MGAC		MRSGAC$_b$		MSGAC$_b$	
	time	#nodes	time	#nodes	time	#nodes
$\langle 10, 5 \rangle$	17	21	23	16	26	16
$\langle 15, 20 \rangle$	92	726	34	41	42	41
$\langle 20, 10 \rangle$	203	1,694	39	47	50	46
$\langle 25, 40 \rangle$	14,985	88,407	595	187	678	169
$\langle 30, 20 \rangle$	6,073	29,211	653	184	768	161
$\langle 35, 35 \rangle$	124,220	481,364	7,431	1,279	8,379	1,093
$\langle 40, 40 \rangle$	1,644,624	5,311,838	67,798	9,838	76,667	8,475

Table 1. Average results of MGAC, MRSGAC$_b$ and MSGAC$_b$.

see that there is no difference in the number of nodes visited by MRSGAC$_b$ and MSGAC$_b$ for the first two classes of feature subscriptions. However, as the problem size increases the difference in terms of the number of nodes also increases significantly. Note that in the remainder of the paper the results that correspond to the CP approach are obtained by using MSGAC$_b$ algorithm.

6.3 Comparison between the Alternative Approaches

The performances of using constraint programming (CP), partial weighted maximum satisfiability (PWMSAT) and integer linear programming (CPLEX) approaches are presented in Tables 2 and 3. If any approach failed to find and prove an optimal relaxation within a time limit of 4 hours then that time limit is used as the runtime of the algorithm and the number of nodes visited in that time limit is used as the number of nodes of the algorithm in order to compute the average runtime and average search nodes of a given problem class. In the tables, the column labelled as #us is used to denote the number of instances for which the time limit was exceeded. If this column is not present for any approach then it means that all the instances of all the problem classes were solved within the time limit. In general finding an optimal relaxation is NP-hard. Therefore, we need to investigate which approach can do it in reasonable time.

Tables 2 and 3 suggest that our CP approach performs better than our ILP and PWM-SAT approaches. Although in very few cases the CP approach is outperformed by the other two approaches, it performs significantly better in all other cases. Nevertheless,

Table 2. Catalogue $\langle 50, 250, \{<, >\} \rangle$.

$\langle f, p \rangle$	optimal value	PWMSAT			CPLEX			CP	
		#nodes	time	#us	#nodes	time	#us	#nodes	time
$\langle 10, 5 \rangle$	36	167	345	0	0	11	0	16	23
$\langle 15, 20 \rangle$	69	721	1,039	0	51	61	0	41	34
$\langle 20, 10 \rangle$	62	1,295	1,619	0	50	47	0	47	39
$\langle 25, 40 \rangle$	115	5,039	4,391	0	3,482	1,945	0	187	595
$\langle 30, 20 \rangle$	93	5,415	6,397	0	1,901	1,025	0	184	653
$\langle 35, 35 \rangle$	118	30,135	23,955	0	35,247	22,763	0	1,279	7,431
$\langle 40, 40 \rangle$	123	186,913	282,760	0	299,829	247,140	0	9,838	67,798
$\langle 45, 90 \rangle$	173	6,291,957	12,638,251	8	5,280,594	7,690,899	2	104,729	1,115,515
$\langle 50, 4 \rangle$	96	165,928	195,717	0	1,164,755	1,010,383	0	60,292	413,611

it is also true that a remarkable improvement in our CP approach is due to maintaining (restricted) singleton arc consistency on the Boolean variables. For example, for feature subscription $\langle 40, 40 \rangle$ and catalogue $\langle 50, 250, \{<, >\} \rangle$ constraint programming (with MSGAC$_b$), on average, requires approximately only 1 minute whereas MGAC requires approximately half an hour.

The CP approach solved all the instances within the time limit. CPLEX could not solve 2 instances. More precisely, it could not prove their optimality within the time limit. SAT4J exceeded the time limit for 9 instances. This could be a consequence of $\mathcal{O}(n^3)$ transitive clauses, where $n = |F|$. Figure 3 depicts a plot between the number of clauses and the runtime of SAT4J. This plot clearly suggests that the runtime of SAT4J increases as the number of clauses increases. The high number of clauses restricts the scalability of the PWM-SAT approach. For large instances SAT4J also

Fig. 3. Clauses vs Time

runs out of the default memory (64MB). For instance, for catalogue $\langle 50, 250, \{<, >\} \rangle$ and feature subscription $\langle 45, 90 \rangle$, SAT4J runs out of memory when solving one of the instances. Note that the results for SAT4J presented in this section correspond to the instances that are generated after reducing the variables and the clauses by applying the techniques described in Section 5.1. The application of these techniques reduces the runtime up to 65%. However, this only enabled one of the previously unsolvable instances to be solved.
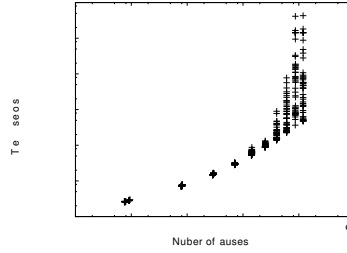
Figure 4 presents the comparison of the different approaches in terms of their runtimes for the subscriptions, when $U = \emptyset$ and the weight of each feature is 1. The runtimes of the approaches for the instances when $|F| = |U|$ are presented in Figure 5. Overall, the CP approach performs best. Although, the SAT4J solver performs best when $|F| > 35$ and $U = \emptyset$, it would be interesting to find out whether its performance will deteriorate when $|F| > 50$. In Figure 5, when $|F| = 50$, neither the ILP approach nor the PWMSAT approach managed to solve all the instances. This is the reason that their average runtimes, for the case of 50 features, are close to the timeout. If the timeout

Table 3. Results for more dense catalogues.

|	Catalogue $\langle 50, 500, \{<, >, <>\} \rangle$						Catalogue $\langle 50, 750, \{<, >\} \rangle$						
	PWMSAT		CPLEX		CP		PWMSAT			CPLEX		CP	
$\langle f, p \rangle$	#nodes	time	#nodes	time	#nodes	time	#nodes	time	#us	#nodes	time	#nodes	time
$\langle 10, 5 \rangle$	326	528	0	10	13	3	246	500	0	28	33	16	7
$\langle 15, 20 \rangle$	1,066	1,173	4	53	31	28	1,111	985	0	306	261	40	45
$\langle 20, 10 \rangle$	2,583	1,981	18	85	49	59	2,484	1,542	0	798	540	82	145
$\langle 25, 40 \rangle$	5,753	2,961	76	554	110	250	6,904	3,158	0	7,043	5,741	236	910
$\langle 30, 20 \rangle$	9,738	4,092	90	447	158	417	11,841	5,025	0	22,253	18,461	591	2,381
$\langle 35, 35 \rangle$	12,584	6,841	300	1,824	461	1,643	31,214	18,278	0	109,472	126,354	2,288	12,879
$\langle 40, 40 \rangle$	22,486	11,310	711	3,018	892	3,914	68,112	92,105	0	354,454	514,275	6,363	42,268
$\langle 45, 90 \rangle$	60,504	59,267	2,130	17,452	2,286	14,803	602,192	2,443,228	1	1,969,716	3,780,539	19,909	188,826
$\langle 50, 4 \rangle$	43,765	21,472	1,500	3,771	4,208	16,921	184,584	319,531	0	1,646,752	3,162,084	51,063	342,492
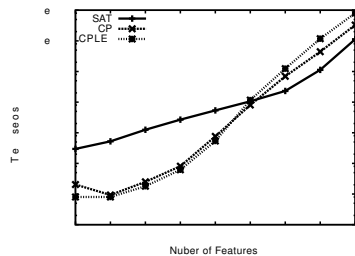
Fig. 4. Results for $\langle f_u, 0, 1 \rangle$, where f_u varies from 5 to 50 in steps of 5.
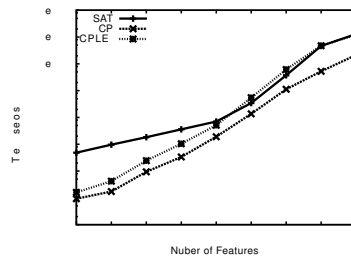
Fig. 5. Results for $\langle f_u, p_u, 1 \rangle$, where $f_u = p_u$ and f_u varies from 5 to 50 in steps of 5.

was higher, the gap between the CP approach and the other approaches, for the case of 50 features in Figure 5 would be even more significant.

7   Conclusions

We presented, and evaluated, three optimisation-based approaches to finding optimal re-configurations of call-control features when the user's preferences violate the technical constraints defined by a set of DFC rules. We proved that finding an optimal relaxation of a feature subscription is NP-hard. For the constraint programming approach, we studied the effect of maintaining generalised arc consistency and two mixed consistencies during branch and bound search. Our experimental results suggest that maintaining (restricted) generalised singleton arc consistency on the Boolean variables and generalised arc consistency on the integer variables outperforms MGAC significantly. Our results also suggest that the CP approach when applied with stronger consistency, is able to scale well compared to the other approaches. Finding an optimal relaxation for a reasonable size catalogue (e.g., [19] refers to a catalogue with up to 25 features) is feasible using constraint programming.

References

1. Bond, G.W., Cheung, E., Purdy, H., Zave, P., Ramming, C.: An Open Architecture for Next-Generation Telecommunication Services. ACM Transactions on Internet Technology **4**(1) (2004) 83–123

2. Jackson, M., Zave, P.: Distributed Feature Composition: a Virtual Architecture for Telecommunications Services. IEEE TSE **24**(10) (October 1998) 831–847
3. Jackson, M., Zave, P.: The DFC Manual. AT&T. (November 2003)
4. Bessiere, C.: Constraint propagation. Technical Report 06020, LIRMM, Montpellier,France (March 2006)
5. Zave, P., Goguen, H., Smith, T.M.: Component Coordination: a Telecommunication Case Study. Computer Networks **45**(5) (August 2004) 645–664
6. Zave, P.: An Experiment in Feature Engineering. In McIver, A., Morgan, C., eds.: Programming Methodology. Springer-Verlag (2003) 353–377
7. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. The MIT Press (1990)
8. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the The Theory of NP-Completeness. W. H. Freeman and Company (1979)
9. Wallace, M.: Practical applications of constraint programming. Constraints Journal **1**(1) (September 1996) 139–168
10. Dooms, G., Deville, Y., Dupont, P.: CP(Graph): Introducing a graph computation domain in constraint programming. In: CP 2005. (2005)
11. Bessiere, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. Artificial Intelligence (2007)
12. Debruyne, R., Bessiére, C.: Some practical filtering techniques for the constraint satifaction problem. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence, Nagoya, Japan (1997) 412–417
13. Prosser, P., Stergiou, K., Walsh, T.: Singleton Consistencies. In Dechter, R., ed.: CP-2000. (September 2000) 353–368
14. Lecoutre, C., Patrick, P.: Maintaining singleton arc consistency. In: Proceedings of the 3rd International Workshop on Constraint Propagation And Implementation (CPAI'2006) held with CP'2006, Nantes, France (September, 2006) 47–61
15. Hawkins, P., Lagoon, V., Stuckey, P.J.: Solving set constraint satisfaction problems using ROBDDs. Journal of Artificial Intelligence Research **24** (2005) 109 – 156
16. Laburthe, F., Jussien, N.: JChoco: A java library for constraint programming.
17. Le Berre, D.: SAT4J: An efficient library of SAT solvers in java.
18. ILOG: CPLEX solver 10.1 http://www.ilog.com/products/cplex/.
19. Bond, G.W., Cheung, E., Goguen, H., Hanson, K.J., Henderson, D., Karam, G.M., Purdy, K.H., Smith, T.M., Zave, P.: Experience with Component-Based Development of a Telecommunication Service. In: CBSE 2005. (2005) 298–305

PB SAT Benchmarks from Routing in Non-Optical and Optical Computer Networks

Miroslav N. Velev

Aries Design Automation, Chicago, IL, U.S.A.

miroslav.velev@aries-da.com

May 31, 2009

```
File format:    PB07
```

```
Description:    For each of 7 networks, 10 routing problems are generated
                as PB SAT formulas.
                File <network>__<start node>_<end node>.opb encodes the
                feasibility of routing between the given start node and end node
                in a non-optical variant of the network.
                Each of files <network>__<start node>_<end node>__<wavelengths>.opb
                encodes the same routing problem but for an optical network with
                the given number of wavelengths per link.
```

Combining Adaptive Noise and Promising Decreasing Variables in Local Search for SAT

Chu Min Li[1] and Wanxia Wei[2]

[1] LaRIA, Université de Picardie Jules Verne
33 Rue St. Leu, 80039 Amiens Cedex 01, France
`chu-min.li@u-picardie.fr`
[2] Faculty of Computer Science, University of New Brunswick, Fredericton, Canada, E3B 5A3
{`wanxia.wei,hzhang`}`@unb.ca`

Given a CNF formula $\mathcal{F}$ and an assignment A, the objective function that local search for SAT attempts to minimize is usually the total number of unsatisfied clauses in $\mathcal{F}$ under A. The score of a variable x with respect to A, $score_A(x)$, is the decrease of the objective function when it is flipped.

A variable x is said to be *decreasing* with respect to A if $score_A(x) > 0$. Promising decreasing variables are defined in [3] as follows:

1. Before any flip, i.e., when A is an initial random assignment, all decreasing variables with respect to A are promising.
2. Let x and y be two different variables and x be not decreasing with respect to A. If, after y is flipped, x becomes decreasing with respect to the new assignment, then x is a promising decreasing variable with respect to the new assignment.
3. A promising decreasing variable remains promising with respect to subsequent assignments in local search until it is no longer decreasing.

G^2WSAT [3] deterministically picks a promising decreasing variable to flip, if such variables exist. If there is no promising decreasing variable, G^2WSAT uses a heuristic, such as $Novelty$ [5], $Novelty+$ [1], or $Novelty++$ [3], to pick a variable to flip from a randomly selected unsatisfied clause c. G^2WSAT uses $Novelty++$.

$Novelty(p, c)$: Sort the variables in clause c by their scores, breaking ties in favor of the least recently flipped variable. Consider the best and second best variables from the sorted variables. If the best variable is not the most recently flipped one in c, then pick it. Otherwise, with probability p, pick the second best variable, and with probability 1-p, pick the best variable.

$Novelty++(p, dp, c)$: With probability dp (diversification probability), pick the least recently flipped variable in c, and with probability 1-dp, do as $Novelty$.

$AdaptG2wsat0$ [4] is G^2WSAT using the adaptive noise mechanism [2] to adjust noise during search. In addition, when there are several promising decreasing variables, $adaptG2wsat0$ flips the oldest one. $AdaptG2wsat0$ won a silver medal in the random category of the SAT 2007 competition under the purse based scoring. It would win the gold medal if the lexicographical scheme was used.

We propose two solvers based on $adaptG2wsat0$ for the SAT 2009 competition.

$adaptg2wsat2009$**:** when there is no promising decreasing variable, with probability dp (diversification probability), randomly pick a variable to flip in the chosen unsatisfied clause c, after excluding the most recently flipped variable in c, and with probability 1-dp, do as $Novelty$.

$adaptg2wsat2009$**++:** when there is no promising decreasing variable, use $Novelty$++(p, dp) to select the variable to flip,

The commande line is

```
DIR/solvername BENCHNAME RANDOMSEED
```

where solvername is adaptg2wsat2009 or adaptg2wsat2009++, and DIR is the name of the directory where adaptg2wsat2009 or adaptg2wsat2009++ files are stored.

References

1. H. Hoos. On the run-time behavior of stochastic local search algorithms for sat. In *Proceedings of AAAI-99*, pages 661–666, 1999.
2. H. Hoos. An adaptive noise mechanism for walksat. In *Proceedings of AAAI-02*, pages 655–660. AAAI Press / The MIT Press, 2002.
3. C. M. Li and W. Q. Huang. Diversification and Determinism in Local Search for Satisfiability. In *Proceedings of SAT2005, 8th International Conference on Theory and Applications of Satisfiability Testing*, pages 158–172, 2005.
4. C. M. Li, W. Wei, and H. Zhang. Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In *Proceedings of LSCS-2006*, pages 2–16, 2006.
5. D.A. McAllester, B. Selman, and H. Kautz. Evidence for invariant in local search. In *Proceedings of AAAI-97*, pages 321–326, 1997.

The iPAWS Solver

John Thornton and Duc Nghia Pham

ATOMIC Program, Queensland Research Lab, NICTA and
Institute for Integrated and Intelligent Systems, Griffith University, QLD, Australia
{john.thornton,duc-nghia.pham}@nicta.com.au

iPAWS is a local search solver based on the PAWS algorithm [1] but modified to automatically adapt its weight decay parameter to each problem instance. Details of iPAWS have been published in [2]. The submitted version follows the proposal of that paper in that iPAWS reverts back to PAWS with $MaxInc = 10$ when encountering any problem consisting entirely of three literal clauses. As iPAWS is a local search technique, it will only terminate upon successfully finding a solutions to satisfiable instance.

The $MaxThres$ Parameter: The key innovation of iPAWS is the automatic adjustment of a $MaxThres$ weight decay parameter that replaces the manually tuned PAWS $MaxInc$ parameter. The operation of $MaxThres$ is detailed in the UpdateClauseWeights function of Algorithm 1. This function is called whenever iPAWS decides it has reached a local minimum and differs from the original PAWS only at lines 5 and 6. Previously, PAWS reduced weight at line 5 if $IncCounter > MaxInc$ and omitted the while loop of line 6. Now the $MaxThres$ parameter causes weight to be reduced when the number of weighted clauses ($|\mathcal{W}|$) and the number of false clauses ($|\mathcal{F}|$) both exceed $MaxThres$ and only after at least $MinInc$ consecutive weight increase phases have been completed ($MinInc$ is fixed at 3). In addition, the while loop at line 6 ensures that each weight reduction phase reduces $|\mathcal{W}|$ to a value less than $MaxThres$ (this step becomes necessary when evaluating the performance of different $MaxThres$ values during the same run). The main advantage of $MaxThres$ is that we can (on average) obtain equivalent performance with the original PAWS while reducing the number of different parameter settings from 22 for $MaxInc$ to 8 for $MaxThres$.

Algorithm 1: UpdateClauseWeights

Input: $\mathcal{F} \leftarrow$ the set of currently false clauses; $\mathcal{W} \leftarrow$ the set of currently weighted clauses;
Output: updated membership of $\mathcal{W}$; updated clause weights for $\mathcal{F} \cup \mathcal{W}$;

1   **for** *each $c_i \in \mathcal{F}$* **do**
2   Weight(c_i) $\leftarrow$ Weight(c_i) + 1;
3   **if** $Weight(c_i) = 2$ **then** $\mathcal{W} \leftarrow \mathcal{W} \cup c_i$;

4   $IncCounter \leftarrow IncCounter + 1$;
5   **if** $|\mathcal{W}| > MaxThres$ *and* $|\mathcal{F}| > MaxThres$ *and* $IncCounter > MinInc$ **then**
6   **while** $|\mathcal{W}| > MaxThres$ **do**
7   **for** *each $c_i \in \mathcal{W}$* **do**
8   Weight(c_i) $\leftarrow$ Weight(c_i) − 1;
9   **if** $Weight(c_i) = 1$ **then** $\mathcal{W} \leftarrow \mathcal{W} - c_i$;

10      $IncCounter \leftarrow 0$;

Local Search Cost Distribution Shape: A *local search cost distribution* is the distribution of the count of false clauses recorded at each flip during a sequence of local search steps. The basis of iPAWS is that it uses the *shape* of this distribution to estimate the best $MaxThres$ parameter setting according to the following rule of thumb: select the distribution with the smallest mean, given the distribution has a roughly normal shape. As a result of extensive preliminary experimentation, we decided to use skewness and kurtosis statistics as an additional guide for parameter setting. Skewness measures the degree of symmetry of a distribution (where a zero value indicates perfect symmetry) and is calculated as follows:

$$\frac{1}{n} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{\sigma} \right)^3$$

In the case of measuring the skewness of a local search cost distribution for a particular $MaxThres$ value, n would be the number of flips taken at the selected $MaxThres$ value, x_i the number of false clauses observed at flip i, and $\overline{x}$ and σ the mean and standard deviation respectively of the distribution of x_i's. Kurtosis measures the degree of "peakedness" of a distribution, where a higher value indicates a sharper peak with flatter tails (in comparison to a standard normal distribution). We calculated kurtosis as follows:

$$\frac{1}{n} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{\sigma} \right)^4 - 3$$

Simulated Annealing: Having identified a few promising measures, we required a method to control the parameter value selection process during the lifetime of a single run. To achieve this we used two interleaved searches on the same problem, one with a high $MaxThres$ setting (750) and the other with a good low default setting (75), as follows: each search starts with its own copy of the same problem initialisation, and then pursues its own separate search trajectory; the two search procedures then compete for processor time according to a simulated annealing (SA) schedule shown in Algorithm 2.

Here, SA is used to control a decision model that begins by randomly allocating time slices to the two search procedures and then, as the temperature decreases, biases decisions more and more towards respecting

---

Algorithm 2: DecideUpperOrLowerSetting

Input: $lowerThres \leftarrow$ lower $MaxThres$ setting; $upperThres \leftarrow$ upper $MaxThres$ setting;
$temp \leftarrow 1024$; $step \leftarrow 400$; $tempStep \leftarrow$ initial steps allocated to upper setting;
$upperStep \leftarrow$ current steps allocated to upper setting;
$lowerStep \leftarrow$ current steps allocated to lower setting;

1 **if** $lowerStep < upperStep$ **then** $tempStep \leftarrow tempStep + lowerStep$;
2 **else** $tempStep \leftarrow tempStep + upperStep$;
3 **while** $tempStep \geq step$ **do**
4 $temp \leftarrow temp \div 2$;
5 $step \leftarrow step \times 2$;
6 $cost \leftarrow$ CostDifference$(lowerThres, upperThres)$;
7 $diff \leftarrow$ AbsoluteValue$(cost)$;
8 $uphillProb \leftarrow 50e^{-(\frac{diff}{temp})}$;
9 **if** $probability \leq uphillProb$ **then**
10   **if** $cost \geq 0$ **then return** $lowerThres$; **else return** $upperThres$;
11 **else**
12   **if** $cost \leq 0$ **then return** $lowerThres$; **else return** $upperThres$;

---

the CostDifference measure defined in Algorithm 3. This measure quantifies our notion of local search cost distribution shape. An important point to note here is that all statistics for each distribution (i.e. the mean, standard deviation, skewness and kurtosis) are reset each time the distribution reaches a solution that improves on the previously best minimum cost (for that distribution). This eliminates the initial high variance phase of the search and avoids the distorting effects of outlying cost values. In addition, we ignore the sign of the skewness and kurtosis measures, taking their absolute value only (see *AbsSkew* and *AbsKurt* in Algorithm 3).

Algorithm 3: CostDifference$(thres1, thres2)$

---

1 $minCostRatio \leftarrow 10 \times (MinCost(thres1) \div (MinCost(thres1) + MinCost(thres2)))$;
2 $rangeRatio \leftarrow 10 \times (Range(thres1) \div (Range(thres1) + Range(thres2)))$;
3 $skewRatio \leftarrow 10 \times (AbsSkew(thres1) \div (AbsSkew(thres1) + AbsSkew(thres2)))$;
4 $kurtRatio \leftarrow 10 \times (AbsKurt(thres1) \div (AbsKurt(thres1) + AbsKurt(thres2)))$;
5 **return** $100 - ((9 \times rangeRatio) + (7 \times minCostRatio) + (2 \times (skewRatio + kurtRatio)))$;

The DecideUpperOrLowerSetting procedure controls the PAWS $MaxThres$ setting for the first 50,000 flips of the combined search trajectories. During this phase, the iPAWS will behave much like its predecessor (with $MaxInc$ set to 10), except that it will "waste" a certain number of flips exploring the non-optimal distribution. Such exploration will help if the best setting is in the upper distribution, but otherwise it will degrade the relative performance.

Binary Search: After the 50,000 flip threshold, both the upper and lower search trajectories are allowed to explore other $MaxThres$ settings within a lower range of $\{25\ 50\ 75\ 125\}$ and an upper range of $\{250\ 500\ 750\ \infty\}$. This procedure takes the form of a binary search, such that after every search step of 100 flips (where the value of $MaxThres$ remains fixed) the DecideUpperOrLowerSetting function determines which half of the parameter space will be used next. Then we use the DecideSetting and FindBestCost functions to further subdivide the parameter space into a single setting. For example, if DecideUpperOrLowerSetting selects lower, then we will call:

$$\text{DecideSetting(FindBestCost(25, 50), FindBestCost(75, 125))}$$

Otherwise we will call:

$$\text{DecideSetting(FindBestCost(250, 500), FindBestCost(750, \infty))}$$

The DecideSetting function follows the simulated annealing approach of DecideUpperOrLowerSetting with two changes to reflect the finer grain of the decision. Firstly, the annealing function has a consistently higher probability of returning an uphill move, replacing line 8 of Algorithm 2 with:

$$uphillProb \leftarrow 30e^{-(\frac{diff}{temp})} + 20$$

Secondly, the annealing schedule is only reduced according to the number of steps taken since the last minimum cost was discovered for each distribution, replacing lines 1-2 from Algorithm 2 with:

    if $(lowerStep < upperStep)$ **then** $tempStep \leftarrow lowerStep$;
    else $tempStep \leftarrow upperStep$;

Finally, we limit the parameter search space on problems with more than 50,000 clauses to only consider 25 or 50 in the lower distribution (the upper distribution parameter range remains unchanged). This reflects empirical observations showing that larger problems tend to have smaller optimal $MaxThres$ settings.

References

1. Thornton, J.R.: Clause weighting local search for SAT. Journal of Automated Reasoning **35**(1-3) (2005) 97–142
2. Thornton, J.R., Pham, D.N.: Using cost distributions to guide weight decay in local search for SAT. In: Proceedings of PRICAI-08. (2008) 405–416

Upgradeability Problem Instances[*]

Josep Argelich[1], Inês Lynce[2], and Joao Marques-Silva[3]

[1] INESC-ID, Lisbon, Portugal
[2] IST/INESC-ID, and Technical University of Lisbon, Portugal
[3] CASL/CSI, University college Dublin, Ireland

1 Description

We have all been through a situation where the installation of a new piece of software turns out to be a nightmare. These kinds of problems may occur because there are *constraints* between the different pieces of software (called *packages*). Although these constraints are expected to be handled in a consistent and efficient way, current software distributions are developed by distinct individuals. This is opposed to traditional systems which have a centralized and closed development. Open systems also tend to be much more complex, and therefore some packages may become incompatible. In such circumstances, user preferences should be taken into account.

The constraints associated with each package can be defined by a tuple (p, D, C), where p is the package, D are the dependencies of p, and C are the conflicts of p. D is a set of dependency clauses, each dependency clause being a disjunction of packages. C is a set of packages conflicting with p. Previous work has applied SAT-based tools to ensure the consistency of both repositories and installations, as well as to upgrade consistently package installations. SAT-based tools have first been used to support distribution editors [3]. The developed tools are automatic and ensure completeness, which makes them more reliable than ad-hoc and manual tools. Recently, Max-SAT has been applied to solve the software package installation problem from the user point of view [1]. In addition, the OPIUM tool [4] uses PB constraints and optimizes a user provided *single* objective function. One modeling example could be preferring smaller packages to larger ones.

The encoding of these constraints into SAT is straightforward: for each package p_i there is a Boolean variable x_i that is assigned to true *iff* package p_i is installed, and clauses are either dependency clauses or conflict clauses (one clause for each pair of conflicting packages).

Example 1 *Given a set of package constraints* $S = \{(p_1, \{p_2, p_5 \vee p_6\}, \emptyset),$ $(p_2, \emptyset, \{p_3\}), (p_3, \{p_4\}, \{p_1\}), (p_4, \emptyset, \{p_5, p_6\})\}$, *its encoded CNF instance is the*

$$\neg x_1 \lor x_2 \qquad \neg x_3 \lor x_4$$
$$\neg x_1 \lor x_5 \lor x_6 \quad \neg x_3 \lor \neg x_1$$
$$\neg x_2 \lor \neg x_3 \quad \neg x_4 \lor \neg x_5$$
$$\neg x_4 \lor \neg x_6$$

The problem described above is called software *installability* problem. The possibility of upgrading some of the packages (or introducing new packages) poses new challenges as existing packages may eventually be deleted. The goal of the software *upgradeability* problem is to find a solution that satisfies user preferences by minimizing the impact of introducing new packages in the current system, which is a reasonable assumption. Such preferences may be distinguished establishing the following hierarchy: (1) constraints on packages cannot be violated, (2) required packages should be installed, (3) packages that have been previously installed by the user should not be deleted, (4) the number of remaining packages installed (as a result of dependencies) should be minimized.

The software upgradeability problem can be naturally encoded as a weighted partial MaxSAT problem. In weighted MaxSAT, each clause is a pair (C, w) where C is a CNF clause and w is its corresponding weight. In weighted partial MaxSAT, *hard* clauses *must* be satisfied, in contrast to the remaining *soft* clauses that *should* be satisfied. Hard clauses are associated with a weight that is greater than the sum of the weights of the soft clauses. A solution to the weighted partial MaxSAT problem maximizes the sum of the weights of the satisfied clauses.

The following example shows a weighted partial MaxSAT formula for the upgradeability problem.

Example 2 *Given a set of package constraints* $S = \{(p_1, \{p_2, p_5\}, \{p_4\}),$
$(p_2, \emptyset, \emptyset), (p_3, \{p_2 \lor p_4\}, \emptyset), (p_4, \emptyset, \emptyset), (p_5, \emptyset, \emptyset)\}$, *the set of packages the user wants to install* $I = \{p_1\}$, *and the current set of installed packages in the system* $A = \{p_2\}$, *its weighted partial MaxSAT instance is the following:*

$$
\begin{array}{lll}
(\neg x_3, 1) & (x_2, 4) & (\neg x_1 \lor x_2, 16) \\
(\neg x_4, 1) & (x_1, 8) & (\neg x_1 \lor x_5, 16) \\
(\neg x_5, 1) & & (\neg x_1 \lor \neg x_4, 16) \\
& & (\neg x_3 \lor x_2 \lor x_4, 16)
\end{array}
$$

This example uses a weight distribution that gives priority to the user preferences over all the other packages, and also gives priority to the current installation profile over the remaining packages. The minimum weight (with value 1) is assigned to clauses encoding packages being installed as a result of dependencies, whose number should be minimized. A medium weight (with value 4, resulting from the sum of the weights of the previous clauses plus 1) is assigned to clauses encoding packages currently installed in our system, in order to minimize the number of removed packages. A maximum weight (with value 8) is assigned to the packages the user wants to install. Finally, we assign a *hard* weight (with value 16) to clauses encoding the dependencies and conflicts.

For more details on how to solve Upgradeability Problem instances in an efficient way, see [2].

References

1. J. Argelich and I. Lynce. CNF instances from the software package installation problem. In *RCRA Workshop*, 2008.
2. J. Argelich, I. Lynce, and J. P. Marques-Silva. On solving boolean multilevel optimization problems. In *21st International Joint Conference on Artificial Intelligence, IJCAI-09, Pasadena, CA, USA*, 2009.
3. F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Int. Conf. Automated Soft. Engineering*, pages 199–208, 2006.
4. C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal package install/uninstall manager. In *Int. Conf. Soft. Engineering*, pages 178–188, 2007.

Partial Max-SAT Encoding for the Job Shop Scheduling Problem

Miyuki Koshimura
Kyushu University

Hidetomo Nabeshima
University of Yamanashi

1 Job Shop Scheduling Problem

A Job Shop Scheduling Problem(JSSP) consists of a set of jobs and a set of machines. Each job is a sequence of operations. Each operation requires the exclusive use of a machine for an uninterrupted duration, i.e. its processing time. A schedule is a set of start times for each operation. The time required to complete all the jobs is called the makespan. The objective of the JSSP is to determine the schedule which minimizes the makespan.

2 Partial Max-SAT Encoding

In this distribution, we follow a variant of the SAT encoding proposed by Crawford and Baker [2]. In the SAT encoding, we assume there is a schedule whose makespan is at most i and generate a SAT instance S_i. If S_i is satisfiable, then the JSSP can complete all the jobs by the makespan i. Therefore, if we find a positive integer k such that S_k is satisfiable and S_{k-1} is unsatisfiable, then the minimum makespan is k.

Before encoding, we estimate the lower bound L_{low} and the upper bound L_{up} of the minimum makespan. In this encoding, we use a job-shop solver by Brucker [1] for the estimatsion.

In order to solve the JSSP in the Max-SAT framework, we introduce a set $P_{L_{up}} = \{p_1, p_2, \ldots, p_{L_{up}}\}$ of new atoms. The intended meaning of $p_i = true$ is that we found a schedule whose makespan is i or longer than i. To realize the intention, the formulas $F_i(i = 1, \ldots, L_{up})$, which represent "if all the operations complete at i, then p_i becomes true," are introduced. Besides, we introduce a formula $T_{L_{up}} = (\neg p_{L_{up}} \vee p_{L_{up}-1}) \wedge (\neg p_{L_{up}-1} \vee p_{L_{up}-2}) \wedge \cdots \wedge (\neg p_2 \vee p_1)$ which implies that $\forall l(1 \leq l < k)(p_l = true)$ must hold if $p_k = true$ holds.

In this setting, if we obtain a model M of $G_{L_{up}}(= S_{L_{up}} \wedge F_1 \wedge \cdots \wedge F_{L_{up}} \wedge T_{L_{up}})$ and k is the maximum integer such that $p_k \in M$, that is, $\forall j(k < j \leq L_{up})(p_j \notin M)$, then we must have $\forall l(1 \leq l \leq k)(p_l \in M)$, namely, $M \cap P_{L_{up}} = \{p_1, \ldots, p_k\}$. The existence of such k is guaranteed by F_k and $T_{L_{up}}$, and indicates that there is a schedule whose makespan is k. If k is the minimum makespan, there is no model of $G_{L_{up}}$ smaller than M with respect to $P_{L_{up}}$. Thus, a minimal model of $G_{L_{up}}$ with respect to $P_{L_{up}}$ represents a schedule which minimizes the makespan.

The encoding is easily adapted for a partial Max-SAT encoding by adding some unit clauses as follows: We introduce $L_{up} - L_{low}$ unit clauses $\neg p_i(i = L_{low} + 1, \ldots, L_{up})$. Then, we solve $MAX_{L_{up}}(= G_{L_{up}} \wedge \neg p_{L_{low}+1} \wedge \ldots \wedge \neg p_{L_{up}})$

Table 1: Benchmark problems

File Name	#vars	#clauses(#soft-clauses)	Optimum
ft10-808-1090	110641	1151201(282)	930
la01-666-0671	34497	342463(5)	660
la04-567-0696	35772	359683(129)	590
orb08-894-1058	107409	1121427(164)	899

with a partial Max-SAT solver where all clauses in $G_{L_{up}}$ are treated as hard clauses and $\neg p_i (i = L_{low} + 1, \ldots, L_{up})$ are as soft clauses. A Max-SAT model of $MAX_{L_{up}}$ represents a optimum schedule.

Some experimental results are reported by Nabeshima [4] and Koshimura [3].

3   The Benchmark Test-sets

The distribution is shown in Table 1. We encode four JSSPs in OR-Library [5]. These four are not hard according to our experience.

Each problem file is named xxx-yyy-zzz. xxx is problem name in OR-Library, yyy is a lower bound of the optimal schedule, and zzz is a upper bound of the optimal schedule. A Max-SAT solution represents a optimal schedule whose length is 'yyy+uuu' where uuu is the number of soft clauses satisfied in the solution.

References

[1] P. Brucker P, B. Jurisch, and B. Sievers: A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics,* Vol.49, pp.107-127, 1994.

[2] J. M. Crawford and A. B. Baker: Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In Proc. of AAAI-94, pp.1092–1097 (1994)

[3] M. Koshimura, H. Nabeshima, H. Fujita, and R. Hasegawa: Minimal Model Generation with respect to an Atom Set. In Proc. of FTP2009, pp.49–59 (2009).

[4] H. Nabeshima, T. Soh, K. Inoue, and K. Iwanuma: Lemma Reusing for SAT based Planning and Scheduling. In Proc. of ICAPS'06, pp.103–112 (2006).

[5] OR-Library. `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`

IUT_BMB_Maxsatz

Abdorrahim Bahrami
Isfahan University of Technology
Abdorrahim@ec.iut.ac.ir

Seyyed Rasoul Mousavi
Isfahan University of Technology
srm@cc.iut.ac.ir

Kiarash Bazargan
Isfahan University of Technology
Kiarash@cc.iut.ac.ir

IUT_BMB_Maxsatz is has been derived from the well-known Maxsatz Max-sat solver [1], which was developed by J. Planes et al. This solver have many inference rules which can simplify the CNF Formula and improve the best-so-far answer called UB (Upper Bound) using the inference rules. UB is an upper bound for the best-so-far answer. In IUT_BMB_Maxsatz first an upper bound for best-so-far is estimated using UBCSAT [2]. UBCSAT is a collection of solvers including non-deterministic solvers which can be used to estimate the best result for a CNF formula.

In IUT_BMB_Maxsatz, a local search has been added to the original Maxsatz. Whenever, a leaf of the search tree is achieved and UB is updated, the local search algorithm is called which tried to improve the new UB. The local search algorithm used in the current version is a simple hill climbing. The neighbors of a candidate solution is defined as to be all the candidate solutions which are different from the current one in one bit only The better the UB is, the further the search tree is pruned.

A non-deterministic adaptive scheme has been used to decide whether or not to call the local search. Initially, it is called with the probability of 1. The probability for subsequent applications of local search, however, depends on the outcomes of its previous applications. Each successful application of local search, i.e. one which improves UB, increases this probability by 0.02 and each unsuccessful one decreases it by the same value 0.02. The reason for adopting such a non-deterministic adaptive scheme is that the likelihood of achieving an improved UB via the application of a local search varies (is usually decreased) as the Branch and Bound tree search proceeds. At the beginning of the search, UB is usually rather raw and is likely to be improved by the local search algorithm. However, by progression of the search, UB gets nearer and nearer the best possible result and the probability of unsuccessful local search is increased, in which case the local search will only be an unfruitful time-consuming task and should be avoided. This solver was particularly successful on random and crafted data.

References

[1] Li, C.M., Manya, F., Planes, J., "New Inference Rules for Max-SAT", Journal of Artificial Intelligence Research 30 page(s):321-359, 2007.

[2] Tompkins, D., Hoos, H., "UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT.", *In SAT*, 2004.

IUT_BMB_LSMaxsatz

Abdorrahim Bahrami
Isfahan University of Technology
Abdorrahim@ec.iut.ac.ir

Seyyed Rasoul Mousavi
Isfahan University of Technology
srm@cc.iut.ac.ir

Kiarash Bazargan
Isfahan University of Technology
Kiarash@cc.iut.ac.ir

IUT_BMB_Maxsatz has been derived from the well-known Maxsatz Max-sat solver [1], which was developed by J. Planes et al. This solver have many inference rules which can simplify the CNF Formula and improve the best-so-far answer called UB (Upper Bound) using the inference rules. UB is an upper bound for the best-so-far answer. In IUT_BMB_LSMaxsatz the initial value of upper bound is the number of clauses unlike the IUT_BMB_Maxsatz which an initial value for upper bound are estimated with UBCSAT [2]. As IUT_BMB_Maxsatz, it incorporates a local search scheme. Whenever, a leaf of the search tree is achieved and UB is updated, the local search algorithm is called which tried to improve the new UB. The local search algorithm used in the current version is a simple hill climbing. The neighbors of a candidate solution is defined as to be all the candidate solutions which are different from the current one in one bit only The better the UB is, the further the search tree is pruned.

A non-deterministic adaptive scheme has been used to decide whether or not to call the local search. Initially, it is called with the probability of 1. The probability for subsequent applications of local search, however, depends on the outcomes of its previous applications. Each successful application of local search, i.e. one which improves UB, increases this probability by 0.02 and each unsuccessful one decreases it by the same value 0.02. The reason for adopting such a non-deterministic adaptive scheme is that the likelihood of achieving an improved UB via the application of a local search varies (is usually decreased) as the Branch and Bound tree search proceeds. At the beginning of the search, UB is usually rather raw and is likely to be improved by the local search algorithm. However, by progression of the search, UB gets nearer and nearer the best possible result and the probability of unsuccessful local search is increased, in which case the local search will only be an unfruitful time-consuming task and should be avoided. This solver was particularly successful on industrial data.

References

[1] Li, C.M., Manya, F., Planes, J., "New Inference Rules for Max-SAT", Journal of Artificial Intelligence Research 30 page(s):321-359, 2007.

[2] Tompkins, D., Hoos, H., "UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT.", *In SAT*, 2004.

IUT_BCMB_WMaxsatz

Abdorrahim Bahrami
Isfahan University of Technology
Abdorrahim@ec.iut.ac.ir

Seyyed Rasoul Mousavi
Isfahan University of Technology
srm@cc.iut.ac.ir

Kiarash Bazargan
Isfahan University of Technology
Kiarash@cc.iut.ac.ir

IUT_BCMB_WMaxsatz has been derived from the well-known Weighted Maxsatz Weighted Max-sat solver [1], which was developed by Chu Min Li et al. This solver have many inference rules which can simplify the CNF Formula and improve the *best-so-far* answer called UB (Upper Bound) using the inference rules. UB is an upper bound for the best-so-far answer. In IUT_BCMB_WMaxsatz first an upper bound for best-so-far is estimated using UBCSAT [2]. UBCSAT is a collection of solvers including non-deterministic solvers which can be used to estimate the best result for a CNF formula.

In IUT_BCMB_WMaxsatz, a local search has been added to the original Maxsatz. Whenever, a leaf of the search tree is achieved and UB is updated, the local search algorithm is called which tried to improve the new UB. The local search algorithm used in the current version is a simple hill climbing. The neighbors of a candidate solution is defined as to be all the candidate solutions which are different from the current one in one bit only The better the UB is, the further the search tree is pruned.

A non-deterministic adaptive scheme has been used to decide whether or not to call the local search. Initially, it is called with the probability of 1. The probability for subsequent applications of local search, however, depends on the outcomes of its previous applications. Each successful application of local search, i.e. one which improves UB, increases this probability by 0.02 and each unsuccessful one decreases it by the same value 0.02. The reason for adopting such a non-deterministic adaptive scheme is that the likelihood of achieving an improved UB via the application of a local search varies (is usually decreased) as the Branch and Bound tree search proceeds. At the beginning of the search, UB is usually rather raw and is likely to be improved by the local search algorithm. However, by progression of the search, UB gets nearer and nearer the best possible result and the probability of unsuccessful local search is increased, in which case the local search will only be an unfruitful time-consuming task and should be avoided. This solver was particularly successful on random and crafted data.

References

[1] Li, C.M., Manya, F., Planes, J., "New Inference Rules for Max-SAT", Journal of Artificial Intelligence Research 30 page(s):321-359, 2007.

[2] Tompkins, D., Hoos, H., "UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT.", *In SAT*, 2004.

IUT_BCMB_LSWMaxsatz

Abdorrahim Bahrami
Isfahan University of Technology
Abdorrahim@ec.iut.ac.ir

Seyyed Rasoul Mousavi
Isfahan University of Technology
srm@cc.iut.ac.ir

Kiarash Bazargan
Isfahan University of Technology
Kiarash@cc.iut.ac.ir

IUT_BCMB_LSWMaxsatz has been derived from the well-known Weighted Maxsatz Weighted Max-sat solver [1], which was developed by Chu Min Li et al. This solver have many inference rules which can simplify the CNF Formula and improve the best-so-far answer called UB (Upper Bound) using the inference rules. UB is an upper bound for the best-so-far answer. In IUT_BCMB_LSWMaxsatz the initial value of upper bound is the number of clauses unlike the IUT_BCMB_WMaxsatz which an initial value for upper bound are estimated with UBCSAT [2]. As IUT_BCMB_WMaxsatz, it incorporates a local search scheme. Whenever, a leaf of the search tree is achieved and UB is updated, the local search algorithm is called which tried to improve the new UB. The local search algorithm used in the current version is a simple hill climbing. The neighbors of a candidate solution is defined as to be all the candidate solutions which are different from the current one in one bit only The better the UB is, the further the search tree is pruned.

A non-deterministic adaptive scheme has been used to decide whether or not to call the local search. Initially, it is called with the probability of 1. The probability for subsequent applications of local search, however, depends on the outcomes of its previous applications. Each successful application of local search, i.e. one which improves UB, increases this probability by 0.02 and each unsuccessful one decreases it by the same value 0.02. The reason for adopting such a non-deterministic adaptive scheme is that the likelihood of achieving an improved UB via the application of a local search varies (is usually decreased) as the Branch and Bound tree search proceeds. At the beginning of the search, UB is usually rather raw and is likely to be improved by the local search algorithm. However, by progression of the search, UB gets nearer and nearer the best possible result and the probability of unsuccessful local search is increased, in which case the local search will only be an unfruitful time-consuming task and should be avoided. This solver was particularly successful on industrial data.

References

[1] Li, C.M., Manya, F., Planes, J., "New Inference Rules for Max-SAT", Journal of Artificial Intelligence Research 30 page(s):321-359, 2007.

[2] Tompkins, D., Hoos, H., "UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT.", *In SAT*, 2004.

Max-SAT Evaluation 2009 Solver Description: Clone

Knot Pipatsrisawat, Akop Palyan, Mark Chavira, Arthur Choi, and Adnan Darwiche

Computer Science Department
University of California, Los Angeles
{thammakn, apalyan, chavira, aychoi, darwiche}@cs.ucla.edu

Clone is an exact Max-SAT solver that uses branch-and-bound search to find optimal solutions. Clone computes lower bounds by computing solutions to relaxed problems. Clone removes some constraints in the original CNF and turns it into a relaxed formula, which is then compiled into a d-DNNF (Deterministic Decomposable Negation Normal Form). The relaxed formula's Max-SAT solution, which can be computed very efficiently from the d-DNNF, can be used as a bound on the solution of the original problem. Once every variable involved in the relaxation is assigned a value, the solution of the conditioned relaxed formula is no longer a bound—it becomes exact. Thus, Clone only needs to perform branch-and-bound search on the search space of those variables involved in the relaxation of constraints, resulting in a smaller search space. For more information about the bound computation and other techniques used in Clone, please see [1]. Clone is also available for download from `http://reasoning.cs.ucla.edu/clone`.

References

1. PIPATSRISAWAT, K., PALYAN, A., CHAVIRA, M., CHOI, A., AND DARWICHE, A. Solving weighted Max-SAT problems in a reduced search space: A performance analysis. *Journal on Satisfiability Boolean Modeling and Computation (JSAT) 4* (2008), 191–217.

The MSUnCore MaxSAT Solver

Joao Marques-Silva
CASL/CSI, University College Dublin
jpms@ucd.ie

Abstract—This paper describes MSUnCore, a state of the art, unsatisfiability-based, MaxSAT solver. MSUnCore is built on top of PicoSAT [3], a state-of-the-art conflict-drived clause learning (CDCL) SAT solver.

I. Introduction

MSUnCore (acronym for *Maximum Satisfiability with UNsatisfiable COREs*) entails a number of unsatisfiability-based MaxSAT algorithms [10], [11], [15]–[17], implements several MaxSAT algorithms, being capable of solving plain MaxSAT [17], and (weighted) (partial) MaxSAT [10], [11], [15], [16]. In addition, MSUnCore represents an alternative to branch-and-bound MaxSAT algorithms (see for example [8], [9]), and targets large-scale practical MaxSAT problem instances. The first version of MSUnCore was implemented in late 2007 and early 2008 [15]–[17]. The most recent version of MSUnCore is from early 2009 [10], [11]. This paper provides a brief overview of the MSUnCore family of MaxSAT algorithms. Additional detail can be found in the MSUnCore publications [10], [11], [15]–[17].

II. Unsatisfiability-Based MaxSAT

An alternative to the widely used branch-and-bound algorithms for MaxSAT (see for example [8], [9]) is the iterated identification of unsatisfiable sub-formulas. Fu & Malik [6] proposed the first unsatisfiability-based MaxSAT solver. The proposed approach consists of identifying unsatisfiable sub-formulas, relaxing each clause in each unsatisfiable sub-formula, and adding a new constraint requiring exactly one relaxation variable to be relaxed in each unsatisfiable sub-formula. Key aspects of this work include the use of the quadratic pairwise [18] CNF encoding for the EqualsOne constraint, and the use of multiple relaxation variables for each clause. Several optimizations were first proposed in some of the MSUnCore algorithms [15]–[17], including the use of non-quadratic CNF encodings of AtMostOne and EqualsOne constraints, and the use of constraints involving the relaxation variables of each clause. Several new unsatisfiability-based MaxSAT algorithms were proposed in [15]–[17], several of which requiring a single relaxation variable per clause. Moreover, additional unsatisfiability-based MaxSAT algorithms have been recently proposed [2]. Finally, extensions of unsatisfiability-based MaxSAT algorithms for weighted (partial) MaxSAT were independently proposed in [2], [10], [11].

MSUnCore Contributors: Jordi Planes and Vasco Manquinho.

III. MSUnCore

MSUnCore is based on iterative identification of unsatisfiable sub-formulas (or cores), by directly interfacing a CDCL [12] SAT solver.

A. Core Extraction

MSUnCore uses a SAT solver for iterative identification of unsatisfiable sub-formulas. Albeit ideally a minimal unsatisfiable sub-formula would be preferred, any unsatisfiable can be considered. Clauses in unsatisfiable sub-formulas are then relaxed by adding a relaxation variable to each clause. MSUnCore implements a number of different algorithms. Some algorithms require multiple relaxation variables per clause [6] whereas others use a single relaxation variable [16], [17].

B. Cardinality Constraints

MSUnCore encodes AtMost, AtLeast and Equals constraints into CNF. A number of encodings can be used. Concrete examples include the pairwise and bitwise encodings [18], [19], the ladder encoding [7], sequential counters [20], sorting networks [5], and binary decision diagrams (BDDs) [5]. Albeit not yet available in MSUnCore, Reduced Boolean Circuits (RBCs) [1] can be used to enable sharing of clauses among the encodings of cardinality constraints.

C. SAT Solver Interface

Even though MSUnCore interfaces PicoSAT [3], any CDCL SAT solver can be used, as long as it computes unsatisfiable sub-formulas in unsatisfiable instances. A number of standard techniques can be used when interacting with the SAT solvers. These include variable filtering of auxiliary variables [14] and reuse of learnt clauses [13].

D. Implementation

The version of MSUnCore used in the 2009 MaxSAT Evaluation is a re-implementation of the MSUnCore prototype used in the 2008 MaxSAT Evaluation. For the 2008 MaxSAT Evaluation, the MSUnCore prototype was written in Perl and interfaced a modified version of MiniSAT 1.14. For the 2009 MaxSAT Evaluation, the new version of MSUnCore is written in C++, and interfaces PicoSAT [3].

E. Using MSUnCore

The most recent MSUnCore distribution contains the MSUnCore executable (for a number of target architectures). In addition, the MSUnCore distribution includes both a linkable library and a header file that exposes a simple API

to integrate MSUNCORE in applications. The use of the library is illustrated with a simple driver, also included in the MSUNCORE distribution.

MSUNCORE accepts the most widely used formats for representing MAXSAT problem instances, including CNF, WCNF, and PMCNF. Albeit MSUNCORE accepts any problem instance of (weighted) (partial) MAXSAT, it is currently unable to handle arbitraly large integers, being restricted to C++ long integers.

F. Availability

There are two versions of MSUNCORE, one developed in 2008 and one developed in 2009. The 2008 version is implemented in Perl and interfaces a modified version of MiniSat 1.14 [4]. The 2009 version is implemented in C++, and interfaces PicoSAT 914 [3]. Both versions of MSUNCORE can be obtained from the lead developer home page http://www.csi.ucd.ie/staff/jpms/soft/, and are available for research and educational purposes. Communication of errors or suggestions for improvements should be forwarded to the lead developer.

IV. CONCLUSIONS

This paper outlines MSUNCORE, a state-of-the-art, unsatisfiability-based, MAXSAT solver, capable of solving (weighted) (partial) MAXSAT problem instances from practical application domains. More detailed descriptions of the algorithms available in the existing versions of MSUNCORE can be found elsewhere [10], [11], [15], [16]. MSUNCORE has consistently ranked as the best performer for instances of the industrial MAXSAT category in the MAXSAT evaluation. In the industrial categories of (weighted) (partial) MAXSAT, MSUNCORE is among the top performers for concrete classes of instances.

REFERENCES

[1] P. A. Abdulla, P. Bjesse, and N. Eén, "Symbolic reachability analysis based on SAT solvers," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[2] C. Ansótegui, M. L. Bonet, and J. Levy, "Solving (weighted) partial MaxSAT through satisfiability testing," in *International Conference on Theory and Applications of Satisfiability Testing*, June 2009, pp. 427–440.

[3] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 75–97, 2008.

[4] N. Een and N. Sörensson, "An extensible SAT solver," in *International Conference on Theory and Applications of Satisfiability Testing*, May 2003, pp. 502–518.

[5] ——, "Translating pseudo-Boolean constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 3-4, pp. 1–25, March 2006.

[6] Z. Fu and S. Malik, "On solving the partial MAX-SAT problem," in *International Conference on Theory and Applications of Satisfiability Testing*, August 2006, pp. 252–265.

[7] I. P. Gent and P. Nightingale, "A new encoding of AllDifferent into SAT," in *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, September 2004, pp. 95–110.

[8] F. Heras, J. Larrosa, and A. Oliveras, "MiniMaxSat: An efficient weighted Max-SAT solver," *Journal of Artificial Intelligence Research*, vol. 31, pp. 1–32, 2008.

[9] C. M. Li, F. Manyà, and J. Planes, "New inference rules for Max-SAT," *Journal of Artificial Intelligence Research*, vol. 30, pp. 321–359, 2007.

[10] V. Manquinho, J. Marques-Silva, and J. Planes, "Algorithms for weighted Boolean optimization," *CoRR*, vol. abs/0903.0843v2, March 2009.

[11] ——, "Algorithms for weighted Boolean optimization," in *International Conference on Theory and Applications of Satisfiability Testing*, June 2009, pp. 495–508.

[12] J. Marques-Silva and K. Sakallah, "GRASP: A new search algorithm for satisfiability," in *International Conference on Computer-Aided Design*, November 1996, pp. 220–227.

[13] ——, "Robust search algorithms for test pattern generation," in *Fault-Tolerant Computinig Symposium*, June 1997, pp. 152–161.

[14] J. Marques-Silva and I. Lynce, "Towards robust CNF encodings of cardinality constraints," in *Principles and Practice of Constraint Programming*, 2007, pp. 483–497.

[15] J. Marques-Silva and V. Manquinho, "Towards more effective unsatisfiability-based maximum satisfiability algorithms," in *International Conference on Theory and Applications of Satisfiability Testing*, May 2008, pp. 225–230.

[16] J. Marques-Silva and J. Planes, "On using unsatisfiability for solving maximum satisfiability," *Computing Research Repository*, vol. abs/0712.0097, December 2007.

[17] ——, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Design, Automation and Testing in Europe Conference*, March 2008, pp. 408–413.

[18] S. Prestwich, "CNF encodings," in *SAT Handbook*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, pp. 75–98.

[19] S. D. Prestwich, "Variable dependency in local search: Prevention is better than cure," in *International Conference on Theory and Applications of Satisfiability Testing*, May 2007, pp. 107–120.

[20] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," in *International Conference on Principles and Practice of Constraint Programming*, October 2005, pp. 827–831.

Solver WBO

João Marques-Silva   Vasco Manquinho   Jordi Planes

July 22, 2009

WBO is a Weighted Boolean Optimization solver able to tackle both MaxSAT and Pseudo-Boolean problem instances. It is based on the ideas described in the paper "Algorithms for Weighted Boolean Optimization" published in SAT 2009 proceedings. WBO is based on the identification of unsatisfiable subformulas. After the identification of an unsatisfiable core, it relaxes the constraints in the core by adding a new relaxation variable to each constraint. A new constraint is added so that at most one of the relaxation variables can be assigned value 1. The algorithm ends when the resulting formula becomes satisfiable. See the referred paper for details.

References

[1] J. Marques-Silva, V. Manquinho, J. Planes. Algorithms for Weighted Boolean Optimization. In *12th International Conference on Theory and Applications of Satisfiability Testing – SAT 2009*. Volume 5584 of LNCS, pages 495–508, Springer, 2009.

Solver MaxSatz in Max-SAT Evaluation 2009

Chu Min Li[1]   Felip Manyà[2]
Nouredine Ould Mohamedou[1]   and Jordi Planes[3]

[1]Université de Picardie Jules Verne, [2]Institut d'Investigació en
Intel·ligència Artificial - CSIC, [3]Universitat de Lleida

MaxSatz applies resolution style inference rules, and incorporates a lower bound computation method that increments the lower bound by one when it detects an inconsistent subset using unit propagation and failed literal detection. It adapts to Max-SAT the technology of the SAT solver Satz. For further details see [3, 4, 2, 1].

References

[1] LI, C. M., MANÀ, F., MOHAMEDOU, N., AND PLANES, J. Exploiting cycle structures in max-sat. In *Proceedings of Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)* (Swansea, Wales, UK, June 2009), O. Kullmann, Ed., vol. 5584 of *LNCS*, Springer, pp. 467–480.

[2] LI, C. M., MANYÀ, F., MOHAMEDOU, N. O., AND PLANES, J. Transforming inconsistent subformulas in MaxSAT lower bound computation. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP 2008)* (Sydney, Australia, 2008), P. J. Stuckey, Ed., vol. 5202 of *LNCS*, Springer, pp. 582–587.

[3] LI, C. M., MANYÀ, F., AND PLANES, J. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)* (Boston/MA, USA, 2006), AAAI Press, pp. 86–91.

[4] LI, C. M., MANYÀ, F., AND PLANES, J. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research 30* (2007), 321–359.