Proceedings of
SAT COMPETITION 2013
Solver and Benchmark Descriptions

Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo *(editors)*

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2013-1

ISSN 1458-4786
ISBN 978-952-10-8991-6 (PDF)
Helsinki 2013

PREFACE

The area of Boolean satisfiability (SAT) solving has seen tremendous progress over the last years. Many problems (e.g., in hardware and software verification) that seemed to be completely out of reach a decade ago can now be handled routinely. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for this success. To keep up the driving force in improving SAT solvers, SAT solver competitions provide opportunities for solver developers to present their work to a broader audience and to objectively compare the performance of their own solvers with that of other state-of-the-art solvers.

SAT Competition 2013 (SC 2013), a open competitive event for SAT solvers, was organized as a satellite event of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT 2013, July 8-12 in Helsinki, Finland). SC 2013 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held yearly from 2002 to 2005 and biannually starting from 2007, the SAT-Races held in 2006, 2008 and 2010, and SAT Challenge 2012.

SC 2013 consisted of a total of 14 competition tracks, each track being characterized by the combination of (i) the type of solvers allowed to participate in the track, (ii) the computational resources provided to each solver, and (iii) the class of benchmarks used (Application / Hard Combinatorial / Random; SAT/UNSAT/SAT+UNSAT). In addition to nine main tracks for sequential *core* solvers and three tracks for parallel core solvers, the competition included an *Open Track* for any types of (parallel) solvers, as well as a *MiniSAT Hack Track* following the tradition set forth by previous SAT Competitions. New for 2013 was that solvers competing in the three main tracks on purely unsatisfiable formulas were required to output actual proofs as certificates for unsatisfiability.

There were two ways of contributing to SC 2013: by submitting one or more solvers for competing in one or more of the competition tracks, and by submitting interesting benchmark instances on which the submitted solvers could be evaluated on in the competition. Following SAT Challenge 2012, the rules of SC 2013 required all contributors (both solver and benchmark submitters) to submit a short, around 2-page long solver/benchmark description as part of their contribution. This book contains all these descriptions in a single volume, providing a way of consistently citing the individual descriptions. Furthermore, we have included descriptions of the selection and generation process applied in forming the benchmark instances used in the SC 2013 competition tracks. We hope this compilation is of value to the research community at large both at present and in the future, providing the reader new insights into the details of state-of-the-art SAT solver implementations and the SC 2013 benchmarks, and also as a future historical reference providing a snapshot of the SAT solver technology actively developed in 2013.

We would like to thank all those who contributed to SC 2013 by submitting either solvers or benchmarks and the related description. We thank Youssef Hamadi, Karem Sakallah, and Roberto Sebastiani for agreeing to act as judges for SC 2013. We also thank SC 2013 Technical Assistants Daniel Diepold and Simon Gerber (University of Ulm, Germany) for their active role in running SC 2013. Experiment Design and Administration for Computer Clusters (EDACC) platform, and the bwGrid computing infrastructure operated by eight Baden-Württemberg state universities, both provided critical infrastructure for successfully running the competition.

Ulm, Dublin, Austin, and Helsinki, June 18, 2013

Adrian Balint, Anton Belov, Marijn J.H. Heule, & Matti Järvisalo
SAT Competition 2013 Organizers

Contents

Benchmark Descriptions

SOLVER DESCRIPTIONS

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Balance between intensification and diversification: two sides of the same coin

Chumin LI*†
†MIS, Université de Picardie Jules Verne, France
Email: †chu-min.li@u-picardie.fr

Chong HUANG*, Ruchu XU‡
*‡Huazhong University of Science and Technology, China
Email: *jason_hch@hotmail.com, ‡xrcy0315@sina.com

***Abstract*—This document describes the SAT solver BalancedZ, a stochastic local search algorithm featuring the balance between intensification and diversification, where intensification refers to search steps improving the objective function, and diversification refers to search steps moving to different areas of the search space. BalancedZ employs different techniques to keep the balance between intensification and diversification according to the $80/20$ Rule.**

I. Introduction

In Stochastical Local Search (SLS) for SAT, intensification refers to search steps improving the objective function, and diversification refers to search steps moving to different areas of the search space. With the introduction of g2wsat [1], the intensification steps are more clearly distinguished from the diversification steps: when there are promising decreasing variables, they are deterministically flipped to improve the objective function, otherwise novelty [2] is called to diversify search, in which intensification steps can still be made, controlled by a noise.

Although very effective, promising decreasing variable is a too strong notion forbidding many useful intensification steps, because there are few promising decreasing variables in a local search solving a hard SAT instance. Recently, a Configuration Checking (CC) notion is introduced in SLS for SAT and proves to be very useful: an improving variable x is deterministically flipped if one of its neighbors has been flipped since the last time x was flipped. The CC notion combined with another intensification technique called Aspiration mechanism allows the CCA solver [3] to win the random category of the SAT challenge 2012 [4].

However, we found that the CC notion is too weak, especially for large k-SAT, where a variable often has many neighbors: it is easy to have one of its neighbors flipped since the flipping of a variable. In this case, too many search steps make intensification and the search is not sufficiently diversified.

We believed that the promising decreasing notion and the CC notion are two extremities and a compromise between them can be found to be more effective. The SLS solver BalancedZ is designed to realize this compromise. Moreover, BalancedZ employs other techniques to the balance between intensification and diversification according to the $80/20$ Rule (i.e. 80% of steps are intensification and 20% of steps are diversification).

II. Main Techniques

Given a SAT instance ϕ to solve, the weight of all clauses being initialized to 1. BalancedZ first generates a random assignment. The objective function of BalancedZ is the sum of weights of all unsatisfied clauses to be reduced to 0. The score of a variable is the decrease of the objective function if the variable is flipped. While the objective function is not 0, BalancedZ modifies the assignment as follows:

1) If there are changing decreasing variables, flip the best one (CD step);
2) Otherwise, if there are decreasing variables with very high score, flip the best one (AD step);
3) Otherwise, randomly pick an unsatisfied clause c and flip the least recently flipped one (Div step);
4) Increase the weight of unsatisfied clauses, and smooth the clause weights under some conditions

where CD refers to Changing Decreasing, AD refers to Aspiration Decreasing, and Div refers to Diversification. The framework of BalancedZ is similar to that of CCASat, where the intensification mode (CD step and AD step) is yet more clearly distinguished from the diversification mode (Div step) than g2wsat. However, all these three steps are re-considered:

A variable x is a changing decreasing variable if its score is positive and if it occurs in a clause that has been changed from satisfied to unsatisfied, or from unsatisfied to satisfied since the last time x was flipped. On the one hand, a changing variable x is necessarily a CC variable, i.e. one of its neighbors has been flipped since x was flipped. However the inverse is not true. Obviously, changing variables reflect more a changing context than the CC variables. On the other hand, a promising decreasing variable is necessarily a changing decreasing variable, but the inverse is not true. Changing variables allow more intensification steps.

Since there are fewer changing decreasing variables in the CD steps than the CC variables in CCASat, we increase the number of AD steps by introducing a parameter λ: a variable x is considered to have very high score if $score(x) > \lambda * g$, where g is the averaged clause weight (over all clauses) and $0 < \lambda < 1$. In CCASat, λ=1.

In a CD step, the best variable is defined to be the variable having the highest score, breaking tie in favor of the variable that most recently becomes a changing decreasing variable.

In a AD step, the best variable is defined to be the variable having the highest score, breaking tie in favor of the variable

that least recently becomes a changing decreasing variable.

The remaining ties of a CD step and a AD step are broken in favor of the least recently flipped variable.

Increasing the weight of unsatisfied clauses increases the score of variables in these clauses, making some steps intensifying. So clause weighting techniques are very important for the performance of a SLS solver. BalancedZ utilizes two clause weighting techniques for random k-SAT problem: SWT scheme[5] for $k < 4$; PAWS scheme[6] for $k \geq 4$. PAWS scheme is also applied to the crafted problems. In PAWS scheme, BalancedZ has an indicator called $decreasingFlag$ to record whether there are decreasing variables in the last flip or not, and if there are, the $decreasingFlag$ is $true$ and $false$ otherwise. In the PAWS scheme, BalancedZ decreases each satisfied clause's weight by one whose weight is larger than one with a smooth probability sp or when the $decreasingFlag$ is $true$, otherwise, update all the unsatisfied clauses' weight by increasing one. In order to improve the performance for solving random large k-SAT problem, there is a noise[7] in BalancedZ as well. With a diversification probability of dp[1], the weights of all unsatisfied clauses are increased by one regardless of the smooth probability sp and the $decreasingFlag$.

Last but not least, compared to the microscopic balance mentioned above, BalancedZ accomplishes the task of balancing the intensification and diversification macroscopically as well. On a more macroscopic level, BalancedZ manages the noise sp adaptively in a similar way to Hoo's adaptive noise mechanism[8]. Our experimental analysis of solving random k-SAT problems with different noises indicates that BalancedZ delivers optimal performance when the ratio of number of steps CD and AD to the number of steps Div is roughly 80% to 20%, which conforms to the $80/20$ Rule (Pareto Principle) by coincidence.

III. PARAMETER DESCRIPTION

The parameters of BalancedZ mentioned above are set as follows:

1) $\lambda = 0.9$, this parameter is not sensitive, but should be set bigger than 0.8;
2) sp is managed adaptively in a similar way to Hoo's adaptive noise mechanism. It controls the clause weighting techniques in large k-SAT problems and is performance-sensitive especially in random 7-SAT;
3) $dp = 0.001$, this parameter is designed for diversification, and should be set smaller than 0.5;

Other parameters include: -maxtries a, -seed b, allowing to run a times BalancedZ and the random seed of the first run being b.

IV. SAT COMPETITION 2013 SPECIFICS

BalancedZ is submitted to three tracks in the competition: Application track, Hard-combinatorial SAT track and Random SAT track. BalancedZ is complied by gcc with the following command:

gcc -O3 -static -m32 BalancedZ.c -o BalancedZ

BalancedZ should be called in the competition using:
BalancedZ INSTANCE -seed SEED

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China. (Grant No. 61272014 and Grant No. 61070235)

REFERENCES

[1] C. M. Li and W. Q. Huang, "Diversification and determinism in local search for satisfiability," *in Proc. of SAT-05*, pp. 158–172, 2005.

[2] D. McAllester, B. Selman, and H. Kautz, "Evidence for invariant in local search," *in Proc. of AAAI-97*, pp. 321–326, 1997.

[3] S. Cai and K. Su, "Configuration checking with aspiration in local search for sat," *in Proc. of AAAI-12*, pp. 434–440, 2012.

[4] A. Balint, A. Belov, M. Järvisalo, and C. Sinz, in *http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html*, 2012.

[5] F. Hutter, D. A. D. Tompkins, and H. H. Hoos, "Scaling and probabilistic smoothing: Efficient dynamic local search for sat," *in Proc. of CP-02*, pp. 233–248, 2002.

[6] J. Thornton, D. N. Pham, S. Bain, and V. F. Jr, "Additive versus multiplicative clause weighting for sat," *in Proc. of AAAI-04*, pp. 191–196, 2004.

[7] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," *in Proc. of AAAI-94*, pp. 337–343, 1994.

[8] H. H. Hoos, "An adaptive noise mechanism for walksat," *in Proc. of AAAI-02*, pp. 655–660, 2002.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

ShatterGlucose and BreakIDGlucose

Jo Devriendt
University of Leuven
Leuven, Belgium

Bart Bogaerts
University of Leuven
Leuven, Belgium

***Abstract*—The defining characteristic of our submitted SAT solvers is the addition of a preprocessing step in which symmetry breaking clauses are added to the CNF theory.**

I. Introduction

Many real-world problems exhibit symmetry, but the SAT competition and SAT challenge seldomly feature solvers who are able to exploit these symmetry properties. This discrepancy can only be explained by the assumption that for most of the problems in these competitions, symmetry exploitation is not worth the incurred overhead. To experimentally verify this hypothesis, we submit two symmetry breaking SAT solvers which in a preprocessing step detect symmetry and add symmetry breaking clauses to the CNF theory. Another aim is to simply compare both of these symmetry breaking approaches with eachother.

II. Main Techniques

A. ShatterGlucose

This solver represents the current state-of-the-art in symmetry detection and breaking for SAT problems. It couples Glucose [1], a former SAT competition winner, with Saucy [2], a symmetry detection tool, and Shatter [3], a symmetry breaking preprocessor for CNF theories. Since Saucy can not handle duplicate clauses in a CNF theory, we also initially run cnfdedup, which removes duplicate clauses from a CNF theory.

The ShatterGlucose workflow is as follows:

1) cnfdedup removes duplicate clauses from the CNF theory
2) Saucy 3.0 detects symmetry by converting the CNF theory to a graph
3) Using the default literal ordering, for each generator returned by Saucy, Shatter adds symmetry breaking clauses to the CNF theory
4) Glucose 2.2 solves the resulting CNF theory

Note that we run four preprocessors before actually starting the SAT solver search.: a first one to remove duplicate clauses, a second one to detect symmetry, a third one to break symmetry, and finally Glucose uses Minisat's builtin theory simplification algorithms before starting the search.

B. BreakIDGlucose

It can be shown that the generators returned by Saucy, and the default ordering used by Shatter, are suboptimal, in the sense that in certain cases, when using other generators for the same symmetry group and another literal ordering to construct symmetry breaking clauses, the resulting search space can be exponentially smaller. BreakGlucose tries to improve these points:

1) cnfdedup removes duplicate clauses from the CNF theory
2) Saucy 3.0 detects symmetry by converting the CNF theory to a graph
3) BreakID searches for special symmetry subgroups and for symmetry generators of the detected symmetry group which will result in short symmetry breaking constraints
4) BreakID constructs an order adjusted to the characteristics of the detected generators and possible subgroups
5) Symmetry breaking clauses are added to the CNF theory
6) Glucose 2.2 solves the resulting CNF theory

Note that we again run four preprocessors before actually starting Glucose's search.

III. Main Parameters

The main user-controlled parameters control when symmetry detection should be halted, and how much time can be devoted to looking for good symmetry generators. The submitted version of BreakIDGlucose put Saucy at a timeout of 200 seconds, while BreakID is allowed to construct symmetry generators for 200 seconds, or until 100.000 generators are constructed. Internally, BreakID also cuts of the size of any symmetry breaking formula constructed for one symmetry generator to 100 tseitin literals to reduce the overhead of introducing many tseitin literals.
ShatterGlucose uses no symmetry detection time limit.

IV. Special Algorithms, Data Structures, and Other Features

We will focus on the algorithms used by BreakID, since cnfdedup has a trivial algorithm, and the other preprocessors and solver of this submission are already documented fully in literature. Some terms we will use in the next part are:

Definition IV.1. A literal l *occurs* in a symmetry S if $S(l) \neq l$.

Definition IV.2. The *support* of a symmetry S is the number of literals that occur in S.

Given a symmetry group Σ represented by a (small) set of generators σ, BreakID works in three steps.
Firstly, it constructs a large set of small generators γ by recursively applying σ to any recently added members of γ.

If a resulting symmetry S has a support smaller than or equal to a floating limit n, S is added to γ, and n is decreased to the support of S. If no more symmetries can be added to γ, n is increased to infinity and σ is applied to the whole of γ. This procedure continues until γ contains all symmetries of Σ, until a timeout is reached, or until a limit to the size of γ is reached.

Given this set of symmetry generators γ, the second step consists of detecting easily-breakable symmetry subgroups generated by a subset of γ. These symmetry subgroups are known in literature, and can informally be described as pigeonhole-like symmetries, which refers to the prototypical pigeonhole problem. With the right ordering of literals to construct symmetry breaking clauses, it has been shown that breaking these groups results in an exponential speedup of the solver at hand.

Finally, given a set of pigeonhole-like symmetry groups Π and a set of generators γ, BreakID then creates an ordering O of the literals by firstly ordering all literals occurring in Π so that each member of Π is completely broken. Secondly, all literals not occurring in Π are smaller than those occurring in Π, and are subsequently ordered based on their total occurrence in γ: the lower the smaller. Given ordering O, symmetry breaking formula's are added to the CNF theory for any pigeonhole-like symmetry group in Π, and for any symmetry generator S in γ such that the symmetry breaking formula of S contains at least one relatively short constraint. This last condition, together with the fact that we never add symmetry breaking formula's longer than 100 tseitin variables, limits the total number of clauses and tseitin variables induced by static symmetry breaking.

V. Implementation details

BreakID and cnfdedup were written from scratch in C++. We refer to the webpages of the other programs for their implementation details.

VI. SAT Competition 2013 Specifics

ShatterGlucose and BreakIDGlucose were both submitted to the application and hard-combinatorial SAT+UNSAT tracks of the SAT13 competition. GCC 4.8.0 was used by the organizers, with -O3 optimization flags. The resulting binaries were 64 bit.

VII. Availability

An older version of Shatter and Saucy are publicly available at http://www.aloul.net/Tools/shatter/, where version 3.0 is available upon request. Glucose 2.2 is available at https://www.lri.fr/~simon/?page=glucose. BreakID and cnfdedup are available at https://bitbucket.org/krr/symbreaker.

VIII. Acknowledgements

We would like to thank

1) Paul T. Darga, Mark Liffiton and Hadi Katebi for the symmetry detection tool Saucy
2) Fadi A. Aloul and Paul T. Darga for the symmetry breaking tool Shatter
3) Laurent Simon and Gilles Audemard for their SAT solver Glucose

References

[1] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, C. Boutilier, Ed., 2009, pp. 399–404.

[2] H. Katebi, K. A. Sakallah, and I. L. Markov, "Symmetry and satisfiability: An update," in *SAT*, ser. Lecture Notes in Computer Science, O. Strichman and S. Szeider, Eds., vol. 6175. Springer, 2010, pp. 113–127.

[3] F. Aloul, K. Sakallah, and I. Markov, "Efficient symmetry breaking for boolean satisfiability," *Computers, IEEE Transactions on*, vol. 55, no. 5, pp. 549–558, 2006.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

CCA2013

Chengqian Li[1] Yi Fan[1,2]
[1]Department of Computer Science,
Sun Yat-sen University, Guangzhou 510006, China
[2]Institute for Integrated and Intelligent Systems,
Griffith University, Brisbane, Australia
lichengq@mail2.sysu.edu.cn

***Abstract*—This document briefly describes the stochastic local search solver CCA2013 in the configuration it has been submitted to the SAT Challenge 2013. An interesting strategy called configuration checking with aspiration (CCA) was recently proposed to deal with the 3-SAT problem. For large k-SAT it is enhanced with look ahead techniques which significantly improves the performance. The solver CCA2013 is an implementation of the CCA algorithm and look ahead, with some minor enhancements.**

I. Introduction

Stochastic Local Search (SLS) is an effective method for solving the propositional satisfiability problem (SAT). CCA2013 is a SAT solver based on the solver CCASat developed by Shaowei Cai [1], and it embed the look head technique for large k-SAT [2].

II. Main Techniques

CCA2013 is an incomplete SAT solver based on stochastic local search, and it incorporates techniques like tabu, look head, clause weighting, and some heuristics for selecting the initial assignment.

As we adapt CCASat to develop our solver, we exploit CCA techniques as the tabu mechanism. That is, we forbid flipping the variables whose configuration has not changed since it was flipped last time.

However, this strategy only applies effectively to 3-SAT. As to 5-SAT, 6-SAT and 7-SAT, it was recently observed that look head technique [2] improves the performance CCA strategy significantly. The intuition is, that clauses in which 2 literals are assigned true will be more likely to keep true after a sequence of flips, compared to those where only 1 literal is assigned true.

We adopt clause weighting scheme based on a threshold of the averaged weight. Clause weights of all unsatisfied clauses are increased by one; further, if the averaged weight $\bar{w}$ exceeds a threshold γ, all clause weights are smoothed as $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor (1-\rho)\bar{w} \rfloor$.

Unlike GSAT [3], our procedure starts with an initial assignment greedily to some extent, and it restarts after it attempts a number of flips. In this initial assignment, each variable is assigned independently according to the probability of their positive occurrences (and negative occurrences) in the CNF formula.

III. Main Parameters

Restart: our procedure restarts after it attempts a number of flips. For further details, readers can refer to [4].

For 3-SAT and non-k-SAT, we adopt a clause weight $\gamma = 200 + \frac{V(F)+250}{500}$, where $V(F)$ is the number of variables in F. For large k-SAT, we adopt a clause weighting scheme similar to PAWS [5]. With probability sp, smooth clause weights: for each satisfied clauses whose weight is bigger than 1, decrease the weight by 1. Otherwise, clause weights of all unsatisfied clauses are increased by one.

We set ρ to be 0.3.

IV. SAT Competition 2013 Specifics

We use the CCA and look head heuristic to develop a new SLS algorithm called CCA2013, which has been submitted to SAT Challenge 2013, for the Random SAT track. We have referred to CCASat, but we write codes ourselves, and we have fixed some bugs in the original source codes.

CCA2013 is implemented in C++ on the basis of the codes of CCASat [1]. It is compiled by g++ with the following command:

gcc CCA2013.cpp -O2 -static -o CCA2013.

Its running command is:

CCA2013 <instance file name> <random seed>.

V. Availability

The solver is not open source because the codes are preliminary, and the authors are implementing various optimizations on it. Readers can contact the authors to obtain the solver, but are not allowed to use it for any commercial purposes.

Acknowledgment

We thank Prof. Yongmei Liu for introducing us into such an exciting and competitive domain. We also thank Prof. Kaile Su and Dr. Shaowei Cai for sharing their state-of-the-art strategies, and their source codes with us. This work was supported by the Natural Science Foundation of China under Grant No. 61073053.

References

[1] S. Cai and K. Su, "Configuration checking with aspiration in local search for sat," in *AAAI*, 2012.
[2] S. Cai and K. Su, "Comprehensive score: Towards efficient local search for sat with long clauses," in *IJCAI*, 2013.
[3] B. Selman, H. J. Levesque, and D. G. Mitchell, "A new method for solving hard satisfiability problems," in *AAAI*, 1992, pp. 440–446.

[4] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *SAT*, 2012, pp. 16–29.
[5] J. Thornton, D. N. Pham, S. Bain, and V. F. Jr., "Additive versus multiplicative clause weighting for sat," in *AAAI*, 2004, pp. 191–196.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

CCAnr

Shaowei Cai and Kaile Su
IIIS, Griffith University
shaoweicai.cs@gmail.com; k.su@griffith.edu.au

***Abstract*—This note describes the SAT solver "CCAnr", which is a local search solver designed for non-random SAT instances.**

I. Introduction

Recently, we proposed a diversification strategy for local search, which is called configuration checking (CC) [1]. The CC strategy has been successfully used to improve SAT local search algorithms [2], [3], [4]. Especially, the CCA heuristic in [3] combines an aspiration mechanism with the CC strategy and results in the Swcca algorithm, which shows state-of-the-art performance on a large range of instances. Based on the Swcca algorithm, we have developed an SLS solver called CCASat [5], which performs very well on random instances.

CCAnr (CCA-based algorithm for non-random SAT) is an improved version of Swcca, aiming to solve non-random instances more effectively. CCAnr differs from Swcca in two small but important modifications. First, in the diversification mode, after selecting a random unsatisfied clause, while Swcca picks the oldest variable from the clause to flip, CCAnr picks the variable with the greatest $score$ from the selected clause, breaking ties by favoring the oldest one. Secondly, Swcca utilizes the formula $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor (1-\rho) \cdot \overline{w} \rfloor$ to smooth clause weights, while in CCAnr, this smoothing formula is generalized as $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor q \cdot \overline{w} \rfloor$. By setting $q = 0$, the smoothing scheme actually becomes a "forgetting" scheme, which is similar to the one used in a recent local search algorithm for Minimum Vertex Cover [6].

II. Main Techniques

CCAnr is an stochastic local search (SLS) algorithm based on the CCA search framework.

We first give some definitions for the CCA heuristic. A variable x is said configuration changed iff $confChange[x] = 1$. A *configuration changed decreasing* (CCD) variable is a variable with both $confChange[x] = 1$ and $score(x) > 0$. A *significant decreasing* (SD) variable is a variable with $score(x) > g$, where g is a positive integer large enough, and in this work g is set to the averaged clause weight (over all clauses) $\overline{w}$.

Originally proposed in [3], the CCA heuristic (outlined in Algorithm 1) can be generalized as follows: The CCA heuristic switches between the greedy mode and the diversification mode. In the greedy mode, there are two levels with descending priorities. On the first level it picks the CCD variable with the greatest score to flip. If there are no CCD variables, CCA selects the SD variable with the greatest score to flip if there is one, which corresponds to the second level. If there are neither CCD variables nor SD variables, CCA switches to the the diversification mode, where clause weights are updated, and a variable in a random unsatisfied clause is picked to flip.

Algorithm 1: *pickVar*-heuristic CCA

```
1 //greedy mode
2 if there exist CCD variables then return a CCD
  variable with the greatest score;
3 if there exist SD variables then return an SD variable
  with the greatest score;
4 //diversification mode
5 update clause weights;
6 pick a random unsatisfied clause c;
7 return a variable in c;
```

Therefore, to design an SLS algorithm based on the CCA heuristic, we need to specify three technique details: (1), the tie-breaking mechanism in the greedy mode; (2), the clause weighting scheme; and (3), the heuristic to pick a variable from an unsatsified clause in the diversification mode.

III. The CCAnr Algorithm

CCAnr is based on the CCA heuristic, and the three technique details in the CCA heuristic are specified as follows for CCAnr:

1) the tie-breaking mechanism in the greedy mode: CCAnr break ties by favoring the oldest variable in the greedy mode, as Swcca does.
2) the clause weighting scheme: CCAnr adopts a Threshold-based Smoothed Weighting (TSW) scheme. Each time TSW is called, clause weights of all unsatisfied clauses are increased by one; further, if the averaged weight $\overline{w}$ exceeds a threshold γ, all clause weights are smoothed as $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor q \cdot \overline{w} \rfloor$.
3) the pick-var heuristic in the diversification mode: CCAnr picks the variable with the greatest $score$ from an unsatisfied clause, breaking ties by favoring the oldest one.

IV. Main Parameters

There are three parameters in CCAnr: the average weight threshold parameter γ, and the two factor parameters ρ and q. All of the three parameters are for the TSW weighting scheme.

The parameters are set as follows: $\gamma = 300$; $\rho = 0.3$; q is set to 0 if $r \leq 15$, and 0.7 otherwise (r is the ratio of the instance).

V. Implementation Details

CCAnr is implemented in C++. It is developed based on the codes of Swcca solver [3], which can be downloaded from www.shaoweicai.net/research.html.

VI. SAT Competition 2013 Specifies

CCAnr is submitted to "Core solvers, Sequential, Hard-combinatorial SAT track". It is compiled by g++ with the 'O2' optimization option.

Its running command is:

CCAnr <instance file name> <random seed>.

References

[1] S. Cai, K. Su, and A. Sattar, "Local search with edge weighting and configuration checking heuristics for minimum vertex cover," *Artif. Intell.*, vol. 175, no. 9-10, pp. 1672–1696, 2011.

[2] S. Cai and K. Su, "Local search with configuration checking for SAT," in *Proc. of ICTAI-11*, 2011, pp. 59–66.

[3] ——, "Configuration checking with aspiration in local search for SAT," in *Proc. of AAAI-12*, 2012, pp. 334–340.

[4] C. Luo, K. Su, and S. Cai, "Improving local search for random 3-sat using quantitative configuration checking," in *Proc. of ECAI-12*, 2012, pp. 570–575.

[5] S. Cai, C. Luo, and K. Su, "CCASat: Solver description," in *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, 2012, pp. 13–14.

[6] S. Cai, K. Su, C. Luo, and A. Sattar, "NuMVC: An efficient local search algorithm for minimum vertex cover," *J. Artif. Intell. Res. (JAIR)*, vol. 46, pp. 687–716, 2013.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

CScoreSAT2013

Shaowei Cai
Griffith University
shaoweicai.cs@gmail.com

Chuan Luo
Peking University
chuanluosaber@gmail.com

Kaile Su
Griffith University
k.su@griffith.edu.au

***Abstract*—This note describes the SAT solver "CScoreSAT2013", which is a local search solver, especially designed for random instances.**

I. Introduction

Recently, we proposed a new variable property called $subscore$ [1], which shares the same spirit with the commonly used property $score$. While $score$ measures the increment of satisfied clauses by flipping a variable, $subscore$ does that of clauses with more than one true literal. Further, we design a scoring function called *comprehensive score* [2], which is a linear combination of $score$ and $subscore$. We also define a new type of "decreasing" variables namely *comprehensively decreasing* variables [2].

Based on the notions of comprehensive score and comprehensively decreasing variable, we develop an SLS algorithm called CScoreSAT (comprehensive score based SAT algorithm) [2]. The SAT solver CScoreSAT2013 adopts WalkSAT to solve random instances whose maximum clause length (denoted by k) is greater than 3, and adopts CScoreSAT to solve instances with $k > 3$.

II. Main Techniques

Main techniques in CScoreSAT include: configuration checking [3], [4] and comprehensive score [2].

A. Configuration Checking

To avoid blind search, we utilize the configuration checking (CC) strategy . The configuration checking (CC) strategy was proposed to handle the revisiting problem in local search [5], and has proved effective in SLS algorithms for SAT [4]. In the context of SAT, the CC strategy forbids flipping a variable if since its last flip, none of its neighboring variables has been flipped. A variable is configuration changed if since its last flip, at least one of its neighboring variables has been flipped.

B. Comprehensive Score and Comprehensively Decreasing Variables

We consider the number of true literals in a clause, which can be regarded as the degree of being satisfied of the clause. The more true literals a clause contains, the less likely it would become unsatisfied in the following flips.

Definition 1: Given a CNF formula F and an assignment α to its variables, the *satisfaction degree* of a clause C, is defined as the number of true literals in C under α. A clause with a satisfaction degree of δ is said to be a δ-satisfied clause.

Among satisfied clauses, 1-satisfied clauses are the most unstable, as they can become unsatisfied by flipping only one variable. It is beneficial for SLS algorithms to take into account the transformations between 1-satisfied and 2-satisfied clauses.

Based on the above considerations, the variable property $subscore$ is defined as follows.

Definition 2: For a variable x, $subscore(x)$ is defined as $submake(x)$ minus $subbreak(x)$, where $submake(x)$ is the number of 1-satisfied clauses that would become 2-satisfied by flipping x, and $subbreak(x)$ is the number of 2-satisfied clauses that would become 1-satisfied by flipping x.

When considering clause weights in DLS algorithms, $submake(x)$ measures the total weight of the 1-satisfied clauses that would become 2-satisfied by flipping x, and $subbreak(x)$ does that of the 2-satisfied clauses that would become 1-satisfied by flipping x.

Based on the above considerations, By combining $score$ and $subscore$, we design a scoring function named comprehensive score, which is formally defined as follows.

Definition 3: For a CNF formula F, the *comprehensive score* function, denoted by $cscore$, is a function for variables such that $cscore(x) = score(x) + \lfloor subscore(x)/d \rfloor$, where d is a positive integer parameter.

In the following, we define a new type of "deceasing" variables based on the $cscore$ function.

Definition 4: Given a CNF formula F and its $cscore$ function, a variable x is *comprehensively decreasing* if and only if $score(x) \geq 0$ and $cscore(x) > 0$.

Comprehensively decreasing variables are considered to be the flip candidates in the greedy search phases of our algorithm. We utilize the configuration checking (CC) strategy to identify the "good" comprehensively decreasing variables which are *configuration changed*. For convenience, such variables are further referred to as CDCC (Comprehensively Decreasing and Configuration Changed) variables.

III. The CScoreSAT Algorithm

This section presents the CScoreSAT algorithm, which utilizes two key notions: comprehensive score and comprehensively decreasing variable.

For the sake of diversification, CScoreSAT also employs the PAWS clause weighting scheme [6]. All clause weights are initiated as 1. When a local optimum is reached, with probability sp, for each satisfied clause whose weight is larger than one, its weight is decreased by one; with probability $(1 - sp)$, the weights of all unsatisfied clauses are increased by one.

We first introduce the two scoring functions used in CScoreSAT. For the greedy search, CScoreSAT adopts the $cscore$ function. When reaching a local optimum, CScoreSAT makes use of a hybrid scoring function (denoted by $hscore$), which combines $cscore$ with the diversification property age: $hscore(x) = cscore(x) + \lfloor age(x)/\beta \rfloor$, where β is a (relatively large) positive integer parameter.

Algorithm 1: CScoreSAT

```
Input: CNF-formula F, maxSteps
Output: A satisfying assignment α of F, or "unknown"
begin
    α := randomly generated truth assignment;
    for step := 1 to maxSteps do
        if α satisfies F then return α;
        if ∃ CDCC variables then
            v := the CDCC variable with the greatest cscore,
                 breaking ties in favor of the oldest one;
        else
            update clause weights according to PAWS;
            pick a random unsatisfied clause C;
            v := the variable in C with the greatest hscore,
                 breaking ties in favor of the oldest one;
        α := α with v flipped;
    return "unknown";
end
```

CScoreSAT works in two modes, i.e., the greedy mode or the diversification mode. If there exist CDCC variables, CScoreSAT works in the greedy mode. It picks the CDCC variable with the greatest $cscore$ value to flip, breaking ties by preferring the oldest one. If no CDCC variable is present, which means a local optimum is identified, then CScoreSAT switches to the diversification mode. It first updates clause weights according to the PAWS scheme. Then it randomly selects an unsatisfied clause C, and picks the variable from C with the greatest $hscore$ value to flip, breaking ties by favoring the oldest one.

IV. Main Parameters

We combine the WalkSAT and CScoreSAT algorithms, leading to an SLS solver also called CScoreSAT2013, which adopts WalkSAT to solve instances with $k \leq 3$, and adopts CScoreSAT to solve instances with $k > 3$.

WalkSAT has one parameter, namely the noise parameter wp. In CScoreSAT2013, wp is set to 0.567 when $r \leq 4.22$, 0.777-0.05r when $r \in (4.22, 4.23]$, 1.553-0.23r when $r \in (4.23, 4.26)$ and 2.261-0.4r when $r \geq 4.26$, where r is the clause-to-variable ratio.

CScoreSAT has three parameters, namely d, β and sp. Fortunately, d is simply defined as $13 - k$, and β is a constant (2000) for any instance. The sp parameter for PAWS is set to 0.62 for $k = 4$, $0.045r - 0.29$ for $k = 5$, 0.9 for $k = 6$, and 0.92 for $k > 6$.

V. Implementation Details

CScoreSAT2013 is implemented in C++. The CScoreSAT algorithm is implemented based on the codes of CCASat solver [7], which can be downloaded from www.shaoweicai.net/research.html, while the WalkSAT algorithm is implemented from scratch.

VI. SAT Competition 2013 Specifies

CScoreSAT2013 is submitted to "Core solvers, Sequential, Random SAT" and "Core solvers, Parallel, Random SAT" tracks. It is compiled by g++ with the 'O2' optimization option. It is a 32-bit binary.

Its running command is:

CScoreSAT2013 <instance file name> <random seed>.

References

[1] S. Cai and K. Su, "Local search for boolean satisfiability with configuration checking and subscore," *submitted to Artif. Intell.*, 2013.

[2] ——, "Comprehensive score: Towards efficient local search for sat with long clauses," in *Proc. of IJCAI-13*, 2013, p. to appear.

[3] ——, "Local search with configuration checking for SAT," in *Proc. of ICTAI-11*, 2011, pp. 59–66.

[4] ——, "Configuration checking with aspiration in local search for SAT," in *Proc. of AAAI-12*, 2012, pp. 334–340.

[5] S. Cai, K. Su, and A. Sattar, "Local search with edge weighting and configuration checking heuristics for minimum vertex cover," *Artif. Intell.*, vol. 175, no. 9-10, pp. 1672–1696, 2011.

[6] J. Thornton, D. N. Pham, S. Bain, and V. F. Jr., "Additive versus multiplicative clause weighting for SAT," in *Proc. of AAAI-04*, 2004, pp. 191–196.

[7] S. Cai, C. Luo, and K. Su, "CCASat: Solver description," in *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, 2012, pp. 13–14.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

CSCH-based Portfolio using Lingeling and CryptoMinisat

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Cork Constraint Computation Centre University College Cork, Ireland
Email: y.malitsky@4c.ucc.ie
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

`CSHC` is a portfolio solver based on cost-sensitive hierarchical clustering as described in [7]. Like most state-of-the-art portfolio solvers, it dynamically selects and schedules baseline solvers depending on the input instance. `CSHC` version 1.0 participated in the sequential industrial 2-core-solver track of the SAT Competition 2013, and is built upon exactly two complete SAT solvers as describe in a subsequent section.

I. Solving techniques

In the execution phase, `CSHC` first computes features of the given problem instance. In particular, `CSHC` uses two sets of features which are subsets of the base 125 features as provided by Xu et al. [10]:

- 115 features: Based on computing features using the parameters `'-base -sp -dia -cl -ls -lobjois'` and removing all time related features.
- 32 features: Only computes basic features (see [5]).

When feature computation of the 115 exceeds 400 seconds, we compute the 32 basic features and the portfolio switches to this feature representation. If basic feature computation exceeds 100 seconds, we fall back to a default solver.

`CSHC` generates a cost-sensitive hierarchical clustering model for subsets of features. There exist multiple models that predict the solver for a given instance. The different classification related information is aggregated based on penalized average runtime (PAR-10). Since we only have two solvers available, a static schedule as presented in [4] is not applied. Instead the solver that was *not* selected is executed for 10% of the total available time.

For more detailed information on the internals of `CSHC`, please refer to [7].

II. Implementation details

The main launcher script of `CSHC` is written in Python 2.6. This script orchestrates launching of solvers and preprocessors, conversion of solutions of the simplified formulae back to the solutions of the original formulae, etc. The solver selector/scheduler program, called 'chcsrun', is written in C++ and compiled with options "-O3 -fexpensive-optimizations -static". The preprocessor SatELite [3] was modified to not map variables numbers and to explicitly append unit clauses, when possible, for variables it would have eliminated otherwise. The individual baseline solvers scheduled by `CSHC` were themselves written mainly in C/C++, and are listed below.

2-Core Baseline Solvers: The portfolio `CSHC` submitted to the sequential industrial 2-core track is composed of the following 2 baseline solvers:

1) `Lingeling 587` [2]
2) `CryptoMinisat 3.1` [1]

We chose `CryptoMinisat 3.1` as the default solver that is invoked when we encounter issues in the algorithm selection process (e.g., when feature computation takes too long).

Training Instances: We selected $4,259$ instances from all SAT Competitions and Races during 2002 and 2012 [8], whereby we discarded all instances that cannot be solved by any of the aforementioned solvers within a time limit of $5,000$ seconds (on the hardware used for training).

III. SAT Challenge 2013 specific details

The command line used to launch `CSHC` using `Lingeling 587` and `CryptoMinisat 3.1` in the industrial 2-core track of the SAT Competition 2013 was:

python algport.py –tmpdir TMPDIR CSHCapplLG BENCHNAME

Acknowledgment

The solver presented is a portfolio of existing SAT solvers. We merely added a front-end that selects and schedules these solvers intelligently, based on the features of the given SAT instance. The solver that competes here is thus heavily based on existing work on "core" SAT solvers. The authors of these solvers have invested countless hours into their implementations. It is their dedication that transforms original ideas on efficient data structures and very sophisticated algorithmic methods into practice. These authors have effectively driven the tremendous advancements that could be achieved over the past two decades in our ability to solve SAT formulae in practice. We hereby express our sincere thanks and honest admiration for their work.

REFERENCES

[1] M. Soos. CryptoMiniSat 3.1. http://www.msoos.org/2013/04/cryptominisat-3-1-released/, 2013.

[2] A. Biere. Lingeling and Friends at the SAT Competition 2011. *Technical Report*, 2011.

[3] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.

[4] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.

[5] Christian Kroer,Y. Malitsky Feature Filtering for Instance-Specific Algorithm Configuration. *ICTAI*, 2011

[6] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction. *LION*, 2013.

[7] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. *IJCAI*, to appear, 2013.

[8] SAT Competition 2011. *http://www.satcomptition.org*.

[9] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.

[10] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. *SAT*, pp. 228-241, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

CSCH-based Portfolio using Lingeling and Glucose

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Cork Constraint Computation Centre University College Cork, Ireland
Email: y.malitsky@4c.ucc.ie
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

CSHC is a portfolio solver based on cost-sensitive hierarchical clustering as described in [7]. Like most state-of-the-art portfolio solvers, it dynamically selects and schedules baseline solvers depending on the input instance. CSHC version 1.0 participated in the sequential industrial 2-core-solver track of the SAT Competition 2013, and is built upon exactly two complete SAT solvers as describe in a subsequent section.

I. Solving techniques

In the execution phase, CSHC first computes features of the given problem instance. In particular, CSHC uses two sets of features which are subsets of the base 125 features as provided by Xu et al. [10]:

- 115 features: Based on computing features using the parameters `'-base -sp -dia -cl -ls -lobjois'` and removing all time related features.
- 32 features: Only computes basic features (see [5]).

When feature computation of the 115 exceeds 400 seconds, we compute the 32 basic features and the portfolio switches to this feature representation. If basic feature computation exceeds 100 seconds, we fall back to a default solver.

CSHC generates a cost-sensitive hierarchical clustering model for subsets of features. There exist multiple models that predict the solver for a given instance. The different classification related information is aggregated based on penalized average runtime (PAR-10). Since we only have two solvers available, a static schedule as presented in [4] is not applied. Instead the solver that was *not* selected is executed for 10% of the total available time.

For more detailed information on the internals of CSHC, please refer to [7].

II. Implementation details

The main launcher script of CSHC is written in Python 2.6. This script orchestrates launching of solvers and preprocessors, conversion of solutions of the simplified formulae back to the solutions of the original formulae, etc. The solver selector/scheduler program, called 'chcsrun', is written in C++ and compiled with options "-O3 -fexpensive-optimizations -static". The preprocessor SatELite [3] was modified to not map variables numbers and to explicitly append unit clauses, when possible, for variables it would have eliminated otherwise. The individual baseline solvers scheduled by CSHC were themselves written mainly in C/C++, and are listed below.

2-Core Baseline Solvers: The portfolio CSHC submitted to the sequential industrial 2-core track is composed of the following 2 baseline solvers:

1) `Lingeling 587` [2]
2) `Glucose 2.1` [1]

We chose `Glucose 2.1` as the default solver that is invoked when we encounter issues in the algorithm selection process (e.g., when feature computation takes too long).

Training Instances: We selected $4,259$ instances from all SAT Competitions and Races during 2002 and 2012 [8], whereby we discarded all instances that cannot be solved by any of the aforementioned solvers within a time limit of $5,000$ seconds (on the hardware used for training).

III. SAT Challenge 2013 specific details

The command line used to launch CSHC using `Lingeling 587` and `Glucose 2.1` in the industrial 2-core track of the SAT Competition 2013 was:

python algport.py –tmpdir TMPDIR CSHCapplLG BENCHNAME

Acknowledgment

The solver presented is a portfolio of existing SAT solvers. We merely added a front-end that selects and schedules these solvers intelligently, based on the features of the given SAT instance. The solver that competes here is thus heavily based on existing work on "core" SAT solvers. The authors of these solvers have invested countless hours into their implementations. It is their dedication that transforms original ideas on efficient data structures and very sophisticated algorithmic methods into practice. These authors have effectively driven the tremendous advancements that could be achieved over the past two decades in our ability to solve SAT formulae in practice. We hereby express our sincere thanks and honest admiration for their work.

References

[1] G. Audemard and L. Simon. GLUCOSE 2.1 in the SAT Challenge 2012. *Proceedings of SAT Challenge 2012*, 2012.
[2] A. Biere. Lingeling and Friends at the SAT Competition 2011. *Technical Report*, 2011.
[3] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.
[4] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.

[5] Christian Kroer,Y. Malitsky Feature Filtering for Instance-Specific Algorithm Configuration. *ICTAI*, 2011
[6] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction. *LION*, 2013.
[7] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. *IJCAI*, to appear, 2013.
[8] SAT Competition 2011. *http://www.satcomptition.org*.
[9] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.
[10] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. *SAT*, pp. 228-241, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

CSCH-based Portfolio using Clasp and Sattime

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Cork Constraint Computation Centre University College Cork, Ireland
Email: y.malitsky@4c.ucc.ie
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

CSHC is a portfolio solver based on cost-sensitive hierarchical clustering as described in [7]. Like most state-of-the-art portfolio solvers, it dynamically selects and schedules baseline solvers depending on the input instance. CSHC version 1.0 participated in the sequential industrial 2-core-solver track of the SAT Competition 2013, and is built upon exactly two complete SAT solvers as describe in a subsequent section.

I. Solving techniques

In the execution phase, CSHC first computes features of the given problem instance. In particular, CSHC uses two sets of features which are subsets of the base 125 features as provided by Xu et al. [10]:

- 115 features: Based on computing features using the parameters '`-base -sp -dia -cl -ls -lobjois`' and removing all time related features.
- 32 features: Only computes basic features (see [5]).

When feature computation of the 115 exceeds 400 seconds, we compute the 32 basic features and the portfolio switches to this feature representation. If basic feature computation exceeds 100 seconds, we fall back to a default solver.

CSHC generates a cost-sensitive hierarchical clustering model for subsets of features. There exist multiple models that predict the solver for a given instance. The different classification related information is aggregated based on penalized average runtime (PAR-10). Since we only have two solvers available, a static schedule as presented in [3] is not applied. Instead the solver that was *not* selected is executed for 10% of the total available time.

For more detailed information on the internals of CSHC, please refer to [7].

II. Implementation details

The main launcher script of CSHC is written in Python 2.6. This script orchestrates launching of solvers and preprocessors, conversion of solutions of the simplified formulae back to the solutions of the original formulae, etc. The solver selector/scheduler program, called 'chcsrun', is written in C++ and compiled with options "-O3 -fexpensive-optimizations -static". The preprocessor SatELite [2] was modified to not map variables numbers and to explicitly append unit clauses, when possible, for variables it would have eliminated otherwise. The individual baseline solvers scheduled by CSHC were themselves written mainly in C/C++, and are listed below.

2-Core Baseline Solvers: The portfolio CSHC submitted to the sequential industrial 2-core track is composed of the following 2 baseline solvers:

1) `Clasp 2.1.1` [4]
2) `Sattime 2011` [1]

We chose `Clasp 2.1.1` as the default solver that is invoked when we encounter issues in the algorithm selection process (e.g., when feature computation takes too long).

Training Instances: We selected $4,259$ instances from all SAT Competitions and Races during 2002 and 2012 [8], whereby we discarded all instances that cannot be solved by any of the aforementioned solvers within a time limit of $5,000$ seconds (on the hardware used for training).

III. SAT Challenge 2013 specific details

The command line used to launch CSHC using `Clasp 2.1.1` and `Sattime 2011` in the industrial 2-core track of the SAT Competition 2013 was:

python algport.py –tmpdir TMPDIR CSHCapplLG BENCHNAME

Acknowledgment

The solver presented is a portfolio of existing SAT solvers. We merely added a front-end that selects and schedules these solvers intelligently, based on the features of the given SAT instance. The solver that competes here is thus heavily based on existing work on "core" SAT solvers. The authors of these solvers have invested countless hours into their implementations. It is their dedication that transforms original ideas on efficient data structures and very sophisticated algorithmic methods into practice. These authors have effectively driven the tremendous advancements that could be achieved over the past two decades in our ability to solve SAT formulae in practice. We hereby express our sincere thanks and honest admiration for their work.

References

[1] C.M. Li, Y. Li. Sattime 2011. *SAT Competition 2011*, 2011.
[2] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.
[3] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.
[4] B. Kaufmann,T. Schaub,M. Schneider Clasp 2.1.1. http://www.cs.uni-potsdam.de/clasp/, 2013.
[5] Christian Kroer,Y. Malitsky Feature Filtering for Instance-Specific Algorithm Configuration. *ICTAI*, 2011

[6] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction. *LION*, 2013.
[7] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. *IJCAI*, to appear, 2013.
[8] SAT Competition 2011. *http://www.satcomptition.org*.
[9] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.
[10] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. *SAT*, pp. 228-241, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Parallel Lingeling, CCASat, and CSCH-based Portfolios

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Cork Constraint Computation Centre University College Cork, Ireland
Email: y.malitsky@4c.ucc.ie
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

For the 8-core parallel portfolio track of the SAT Competition 2013 the following combined approach was submitted:

1) `Lingeling 587` [2] on 4 cores.
2) `CCASat` [1] on 1 core.
3) `CSHC` trained on industrial/crafted/random category and executed on 1 core each.

`CSHC` is a portfolio solver based on cost-sensitive hierarchical clustering as described in [7]. Like most state-of-the-art portfolio solvers, it dynamically selects and schedules baseline solvers depending on the input instance. `CSHC` version 1.0 participated in the parallel portfolio track of the SAT Competition 2013, and is built upon 27 solvers as stated in a subsequent section.

I. `CSHC` Solving techniques

Here we describe the general methodology underlying `CSHC`. Each instance of `CSHC` is trained for the industrial/crafted/random category as mentioned in a subsequent section. This version of `CSHC` is invoked sequentially and is not composed of parallel base solvers.

In the execution phase, `CSHC` first computes features of the given problem instance. In particular, `CSHC` uses two sets of features which are subsets of the base 125 features as provided by Xu et al. [10]:

- 115 features: Based on computing features using the parameters `'-base -sp -dia -cl -ls -lobjois'` and removing all time related features.
- 32 features: Only computes basic features (see [5]).

When feature computation of the 115 exceeds 400 seconds, we compute the 32 basic features and the portfolio switches to this feature representation. If basic feature computation exceeds 100 seconds, we fall back to a default solver.

`CSHC` generates a cost-sensitive hierarchical clustering model for subsets of features. There exist multiple models that predict the solver for a given instance. The different classification related information is aggregated based on penalized average runtime (PAR-10). `CSHC` first runs a fixed schedule of solvers for 10% of the time limit and then runs the selected solver for the remaining 90% of the available time (cf. [4] for details). Similar to the motivation presented in [6], `CSHC` also employs a recourse action (e.g., more time is allocated to the solver schedule) when some measures indicate a low confidence in its own algorithm selection.

For more detailed information on the internals of `CSHC`, please refer to [7].

II. Implementation details

The main launcher script of this combined approach is written in Python 2.6. This script invokes `Lingeling 587` on 4 cores and `CCASat` on 1 core. Then it invoked each version of `CSHC` specialized on either industrial/crafted/random category on 1 core each. The first approach to first successfully solve the instance at hand caused the script to verify the solution (if satisfiable) and terminate all other approaches.

The launcher script of each `CSHC` is also written in Python 2.6. This script orchestrates launching of solvers and preprocessors, conversion of solutions of the simplified formulae back to the solutions of the original formulae, etc. The solver selector/scheduler program, called 'chcsrun', is written in C++ and compiled with options "-O3 -fexpensive-optimizations -static". The preprocessor SatELite [3] was modified to not map variables numbers and to explicitly append unit clauses, when possible, for variables it would have eliminated otherwise. The individual baseline solvers scheduled by `CSHC` were themselves written mainly in C/C++, and are listed below.

Baseline Solvers: The portfolio `CSHC` is composed of the following 27 baseline solvers with additional parameters shown if default parameters have been changed (proper references omitted due to lack of space):

1) Clasp-2.1.1_jumpy, –configuration=jumpy
2) Clasp-2.1.1_trendy, –configuration=trendy
3) Ebminisat
4) Glueminisat
5) Lrglshr
6) Picosat
7) Restartsat, -rfirst=1 -var-decay=0.95
8) Circminisat
9) Clasp1, –sat-p=20,25,150 –hParam=0,512
10) Cryptominisat_2011
11) Eagleup
12) Gnoveltyp2
13) March_rw
14) MphaseSAT
15) MphaseSATm
16) Precosat
17) Qutersat

18) Sapperlot
19) Sat4j-2.3.2
20) Sattimep
21) Sparrow
22) TNM
23) Cryptominisat295
24) MinisatPSM
25) Sattime2011
26) Glucose 2.1
27) Glucose 2.1, changed restart strategy

Note that neither `Lingeling 587` nor `CCASat` are a baseline solver for `CSHC`, since they are already executed in parallel in any case.

We chose the following default solvers in each benchmark category when we encounter issues in the algorithm selection process (e.g., when feature computation takes too long).:

- Industrial: `Glucose 2.1`
- Crafted: `Clasp 2.1.1`
- Random: `March RW`

Training Instances: We selected $4,259$ instances from all SAT Competitions and Races during 2002 and 2012 [8], whereby we discarded all instances that cannot be solved by any of the aforementioned solvers within a time limit of $5,000$ seconds (on the hardware used for training). For each category, we trained a version of `CSHC`, whereby we had the following distribution of training instances among each category:

- Industrial: 1155 instances
- Crafted: 772 instances
- Random: 2389 instances

Note that the sum of the instances used in each category exceeds the total number of instances, since some instances appear in multiple categories (e.g., random and crafted).

III. SAT Competition 2013 specific details

The command line used to launch the combined approach of `Lingeling 587 CCASat` and `CSHC` in the parallel portfolio track of the SAT Competition 2013 was:

python parlaunch.py –tmpdir TMPDIR BENCHNAME

Acknowledgment

The solver presented is a portfolio of existing SAT solvers. We merely added a front-end that selects and schedules these solvers intelligently, based on the features of the given SAT instance. The solver that competes here is thus heavily based on existing work on “core” SAT solvers. The authors of these solvers have invested countless hours into their implementations. It is their dedication that transforms original ideas on efficient data structures and very sophisticated algorithmic methods into practice. These authors have effectively driven the tremendous advancements that could be achieved over the past two decades in our ability to solve SAT formulae in practice. We hereby express our sincere thanks and honest admiration for their work.

References

[1] S. Cai, K. Su Configuration Checking with Aspiration in Local Search for SAT. *Proc. of AAAI-2012*, pp. 434-440, 2012.
[2] A. Biere. Lingeling and Friends at the SAT Competition 2011. *Technical Report*, 2011.
[3] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.
[4] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.
[5] Christian Kroer,Y. Malitsky Feature Filtering for Instance-Specific Algorithm Configuration. *ICTAI*, 2011
[6] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction. *LION*, 2013.
[7] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. *IJCAI*, to appear, 2013.
[8] SAT Competition 2011. *http://www.satcomptition.org*.
[9] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.
[10] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. *SAT*, pp. 228-241, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

CSCH-based Portfolio using CCSat and March

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Cork Constraint Computation Centre University College Cork, Ireland
Email: y.malitsky@4c.ucc.ie
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

CSHC is a portfolio solver based on cost-sensitive hierarchical clustering as described in [8]. Like most state-of-the-art portfolio solvers, it dynamically selects and schedules baseline solvers depending on the input instance. CSHC version 1.0 participated in the sequential industrial 2-core-solver track of the SAT Competition 2013, and is built upon exactly two complete SAT solvers as describe in a subsequent section.

I. Solving techniques

In the execution phase, CSHC first computes features of the given problem instance. In particular, CSHC uses two sets of features which are subsets of the base 125 features as provided by Xu et al. [11]:

- 115 features: Based on computing features using the parameters '-base -sp -dia -cl -ls -lobjois' and removing all time related features.
- 32 features: Only computes basic features (see [6]).

When feature computation of the 115 exceeds 400 seconds, we compute the 32 basic features and the portfolio switches to this feature representation. If basic feature computation exceeds 100 seconds, we fall back to a default solver.

CSHC generates a cost-sensitive hierarchical clustering model for subsets of features. There exist multiple models that predict the solver for a given instance. The different classification related information is aggregated based on penalized average runtime (PAR-10). Since we only have two solvers available, a static schedule as presented in [4] is not applied. Instead the solver that was *not* selected is executed for 10% of the total available time.

For more detailed information on the internals of CSHC, please refer to [8].

II. Implementation details

The main launcher script of CSHC is written in Python 2.6. This script orchestrates launching of solvers and preprocessors, conversion of solutions of the simplified formulae back to the solutions of the original formulae, etc. The solver selector/scheduler program, called 'chcsrun', is written in C++ and compiled with options "-O3 -fexpensive-optimizations -static". The preprocessor SatELite [2] was modified to not map variables numbers and to explicitly append unit clauses, when possible, for variables it would have eliminated otherwise. The individual baseline solvers scheduled by CSHC were themselves written mainly in C/C++, and are listed below.

2-Core Baseline Solvers: The portfolio CSHC submitted to the sequential industrial 2-core track is composed of the following 2 baseline solvers:

1) CCASat [1]
2) March_rw [3]

We chose CCASat as the default solver that is invoked when we encounter issues in the algorithm selection process (e.g., when feature computation takes too long).

Training Instances: We selected $4,259$ instances from all SAT Competitions and Races during 2002 and 2012 [9], whereby we discarded all instances that cannot be solved by any of the aforementioned solvers within a time limit of $5,000$ seconds (on the hardware used for training).

III. SAT Challenge 2013 specific details

The command line used to launch CSHC using CCASat and March_rw in the industrial 2-core track of the SAT Competition 2013 was:

python algport.py –tmpdir TMPDIR CSHCapplLG BENCHNAME

Acknowledgment

The solver presented is a portfolio of existing SAT solvers. We merely added a front-end that selects and schedules these solvers intelligently, based on the features of the given SAT instance. The solver that competes here is thus heavily based on existing work on "core" SAT solvers. The authors of these solvers have invested countless hours into their implementations. It is their dedication that transforms original ideas on efficient data structures and very sophisticated algorithmic methods into practice. These authors have effectively driven the tremendous advancements that could be achieved over the past two decades in our ability to solve SAT formulae in practice. We hereby express our sincere thanks and honest admiration for their work.

References

[1] S. Cai, K. Su Configuration Checking with Aspiration in Local Search for SAT. *Proc. of AAAI-2012*, pp. 434-440, 2012.
[2] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.
[3] M. Heule. March RW. *SAT Competition 2011*, 2011.
[4] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.
[5] B. Kaufmann,T. Schaub,M. Schneider Clasp 2.1.1. http://www.cs.uni-potsdam.de/clasp/, 2013.

[6] Christian Kroer,Y. Malitsky Feature Filtering for Instance-Specific Algorithm Configuration. *ICTAI*, 2011
[7] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction. *LION*, 2013.
[8] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. *IJCAI*, to appear, 2013.
[9] SAT Competition 2011. *http://www.satcomptition.org*.
[10] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.
[11] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. *SAT*, pp. 228-241, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

DimetheusMPS

Solver Description of Dimetheus v. 1.700 for the SAT Competition 2013

Oliver Gableske
Ulm University, Theoretical Computer Science, 89081 Ulm, Germany
oliver@gableske.net, https://www.gableske.net/dimetheus

***Abstract*—This document describes the `DimetheusMPS` SAT solver, which is an SLS-based MID solver. This solver uses the Message Passing (MP) heuristic $\rho\sigma$PMP^i, which is a newly developed and very flexible MP heuristic described in [1]. The SLS search is done by a ProbSAT SLS solver as described in [2].**

***Index Terms*—MP, SLS, $\rho\sigma$PMP^i.**

I. Introduction

The `Dimetheus` solver is a SAT solver framework[1] that allows to rapidly develop new SAT solver that may be based on several preprocessing, in-processing, and search paradigms. The current `DimetheusMPS` uses this framework to realize an SLS-based Message Passing Inspired Decimation (MID) solver that uses the MP heuristic $\rho\sigma$PMP^i (see [1] for details on MID and the MP heuristic).

The general approach of this solver is to first perform light-weight preprocessing by applying pure literal elimination (PLE), subsumption elimination (SE), and failed literal detection (FLD). After that, the solver starts its search by performing MP based on $\rho\sigma$PMP^i. This heuristic provides variable biases that are used for assigning the variables using unit propagation (UP). This realizes the standard MID approach as described in [1]. As soon as MID runs into a conflict all the variable assignments are undone but saved in the variable phases. After that, a set of assignment suggestions is available. Then, the SLS module takes over. It will initialize its starting assignment by following the variable phases. Then, it performs stochastic local search based on the ProbSAT flipping heuristic (see [2] for details). The SLS solver will flip until the timeout is reached or a satisfying assignment has been found. This approach is incomplete (meaning that it cannot detect unsatisfiability).

The general idea behind this approach is to use MID in order to provide a *good* starting assignment for the SLS. This follows the intuition that the SLS solver will be able to find a satisfying assignment faster if its starting assignment is as close as possible to the next solution in terms of the Hamming-distance.

In summary, the `DimetheusMPS` solver can be characterized as a sequential (non-parallel) SLS core solver.

II. Main Parameters

The solver uses only three main parameters in the competition. The first parameter is the formula that is to be solved (`-formula` followed by the name of the formula containing the problem in DIMACS CNF input format). The second parameter is the seed for the random number generator (`-seed` followed by a natural number). The third parameter is the guide (`-guide` followed by a natural number, in this case 5). The guide basically tells the solver what type of SAT solver is supposed to be realized. Guide number 5 tells the solver to perform SLS-based MID.

Internally, the search modules that participate in this guide are MP and SLS. The MP module uses two parameters called ρ and σ that influence the MP behavior of $\rho\sigma$PMP^i (see [1] for details). The SLS module uses one parameter for the ProbSAT break value called c_b. Depending on the instance properties (clause length k, number of variables n, clauses-to-variables ratio r), the solver will pick reasonable settings for these five parameters to ensure that the modules perform as best as possible.

Furthermore, MID requires a parameter that controls how often biases are re-computed. This parameter is called p (see [1] for details on p). Again, the solver will check on the properties of the formula to determine a reasonable setting for p.

All together, the solver uses the following external command line parameters (the order in which these parameters are provided is irrelevant, but there must be a space between the name of the parameter and the value of the parameter, e.g. `-formula test.cnf`).

- `-formula` (naming the formula to be solved)
- `-seed` (the seed for the random number generator)
- `-guide` (telling the solver what type of search is to be done)

Furthermore, the solver uses one internal parameter that is related to the MID approach.

- `p` (that controls how often biases are re-computed)

Additionally, the solver uses two internal parameters related to the $\rho\sigma$PMP^i heuristic.

- ρ (that controls the carefulness of the MP heuristic)
- σ (that controls how much the MP heuristic enforces convergence)

Finally, the solver uses one internal parameter for the ProbSAT SLS flipping heurisitc.

- c_b (that controls the probability distribution for picking a specific variable for flipping in a randomly selected unsatisfied clause)

[1] See https://www.gableske.net/dimetheus

Basically, all the internal parameters have been tuned using the EDACC/AAC framework (see [3] and [4]) using *unfiltered* uniform random k-CNF formulas. The authors did their best to find reasonable parameter settings for 3-SAT to 7-SAT, and even though the settings found for 3-SAT and 4-SAT are working well, the settings for 5-SAT to 7-SAT are mostly educated guesses. The reason for not providing detailed parameter settings for these cases was the lack of time to conduct further tuning experiments. Conducting these experiments to provide the missing parameter settings is a matter of future work.

III. Implementation Details

The `DimetheusMPS` solver was implemented from scratch in the programming language C. The author followed the C99 standard. The solver provides substantial help by calling it with `--help`.

IV. SAT Competition 2013 Specifics

The solver was submitted to the SAT Competition 2013 Random SAT track. The compiler that is used is GCC 4.4. The optimization flags for GCC that are used are as follows.

- `-std=c99`
- `-O3`
- `-static`
- `-march=native`
- `-fexpensive-optimizations`
- `-Wall`
- `-pedantic`

The solver can be compiled as 32-bit or 64-bit application (depending on the operating system used). No further modifications to the source or the make-file are necessary, because the solver was implemented with robustness in mind (it uses the operating system specific definitions for variable types like `integer` or `float`). The solver version that runs during the SAT Competition 2013 is 64-bit. The version of the solver that runs during the competition is 1.700.

V. Availability

The whole solver is open source and all sources as well as the solver description will be published on the authors website[2] as soon as the SAT Competition 2013 starts. The license is GPL 3 (as provided in the sources).

Acknowledgments

The author would like to thank Adrian Balint, Jacobo Torán, Leonhard Grünschloß, Sven Müelich, Daniel Diepold, Juri Schulte, and Armin Biere.

References

[1] O. Gableske, "On the inerpolation of product-based message passing heuristics for sat," *Proceedings of the Theory and Application of Satisfiability Testing (SAT'13)*, vol. LNCS 7962, Springer Berlin, 2013.

[2] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," *Proceedings of the Theory and Application of Satisfiability Testing (SAT'12)*, vol. LNCS 7317, Springer Berlin, pp. 16–29, 2012.

[3] A. Balint, D. Gall, G. Kapler, R. Retz, D. Diepold, and S. Gerber, "Edacc - an advanced platform for the experiment design, administration and analysis of empirical algorithms," *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION5)*, 2011.

[4] D. Diepold, "Model-based parallel automated algorithm configuration," *Master Thesis supervised by Adrian Balint, Ulm University*, 2012.

[2] https://www.gableske.net/dimetheus

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

SAT11 and SAT11k

Donald Knuth

Stanford University

INTRO

This program is part of a series of "SAT-solvers" that I'm putting together for my own education as I prepare to write Section 7.2.2.2 of *The Art of Computer Programming* [1]. My intent is to have a variety of compatible programs on which I can run experiments to learn how different approaches work in practice. This description is part of the SAT11 source code[1].

RELATION TO OTHER IMPLEMENTATIONS

Many of the previous implementations in this series—SAT0, SAT3, SAT4, SAT5, and SAT10—were based on a natural backtracking approach that has come to be known in the SAT community as the DPLL paradigm, honoring the pioneering work of Davis, Putnam, Logemann, and Loveland [2], [3]. Several decades of experience with that paradigm have led to an extremely efficient class of programs now called *lookahead solvers* [4], which devote considerable time to choosing the variables on which to branch. The extra work of making that choice might cost us a factor of a thousand, say, at every branch node; yet we might also decrease the number of nodes by a factor of a million, thus making a net thousand-fold gain. Somewhat to my surprise, this rosy prediction (contrary to what I had believed for many years) actually does work in practice: There are many SAT problems (especially those based on combinatorial tasks, as well as the academic yet appealing cases of unsatisfiable random 3SAT) for which judicious lookaheads outperform any other known method.

SAT11

Consequently SAT11 is intended to represent a modern lookahead solver. I've based it largely on Marijn Heule's MARCH [5], which has been regularly classed with the world's best lookahead solvers for the last decade or so. I expect SAT11 to be the most ambitious program of this series, because it combines many advanced ideas that I wish to understand and to explain to the readers of *TAOCP*. On the other hand, I have not included all of the bells and whistles of MARCH; in particular, I've omitted the separate treatment of clause sets that represent linear equations mod 2, as well as the "limited discrepancy search" technique by which branches of the search tree are explored in a nonstandard order.

This basic SAT11 program, like the earliest versions of MARCH, is intended for 3SAT problems only: All clauses must have size 3 or less. However, a changefile converts this program to SAT11K, which has no such restriction. A good understanding of the 3SAT version presented below will make it easier to understand the modifications by which the algorithms can be adapted to handle clauses of any length.

The running time in "mems" is also reported, together with the approximate number of bytes needed for data storage. One "mem" essentially means a memory access to a 64-bit word. (These totals don't include the time or space needed to parse the input or to format the output.)

A lookahead solver explores a binary tree of possibilities by choosing, at every decision node, a variable x for which the node's subtrees correspond to asserting x or $\bar{x}$. Several more-or-less independent activities are part of this process:

1) *Preselection*. At each decision node we choose a subset P of the unassigned variables, based on our best guess as to which of them might be good candidates for further exploration.
2) *Selection*. We look ahead at the immediate consequences of asserting the truth and falsity of each variable in P. Then we choose the variable that appears to reduce the problem most efficiently.
3) *Propagation*. We update the current state of the problem by incorporating all consequences of a new assertion.
4) *Backtracking*. When a contradiction arises in some branch, we must undo the effects of propagation and move to an unexplored branch of the tree.

More details of SAT11 and SAT11K can be found in the source code[1].

[1] SAT11 as well as several other SAT solver implementations are available at http://www-cs-faculty.stanford.edu/~knuth/programs/. To obtain the full documentation of SAT11, download `sat11.w` and compile it using `cweave sat11` and `tex sat11`.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.
[2] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.
[3] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.
[4] M. J. H. Heule and H. van Maaren, *Look-Ahead Based SAT Solvers*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 5, pp. 155–184.
[5] M. J. H. Heule, "Smart solving: Tools and techniques for satisfiability solvers," Ph.D. dissertation, TU Delft, 2008.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Hyperplane Guided MiniSAT

Doug Hains
Colorado State University
Fort Collins, CO

Darrel Whitley
Colorado State University
Fort Collins, CO

Adele Howe
Colorado State University
Fort Collins, CO

***Abstract*—We present a new preprocessing technique utilizing subspace averages to perform a reduction by fixing truth assignments. We use the Walsh transform to efficiently compute the average evaluation of solutions in subspaces of the search space that we refer to as hyperplanes. A hyperplane contains all solutions that have the same truth assignments over some subset of the n variables in the given formula. We use the hyperplane averages as a heuristic for determining the quality of solutions contained in the hyperplane. We use this heuristic to select a promising hyperplane and reduce the original formula by eliminating clauses satsified by the variables with consistent truth assignments across all solutions in the hyperplane. We then run the unmodified MiniSAT complete solver on this reduced instance.**

I. Introduction

A strategy that has performed well in previous competitions is to incorporate the use of a preprocessor, typically SatELite [2], to reduce both the number of variables and clauses of the original formula. This reduction has been shown to make the task of finding a satisfying solution easier on many problems [2]. In our submission, we use unmodified versions of MiniSAT and SatELite. For our solver, we use the standard MiniSAT [1] solver available from the MiniSAT web page[1]. We also employ the SatELite [2] preprocessor. Our contribution is the introduction of an additional preprocessing step that we call hyperplane-guided reduction.

For all k-bounded pseudo-Boolean optimization (PBO) problems, we can convert the evaluation functions into a polynomial form in $O(n)$ time. This allows us to quickly and exactly compute low order hyperplane averages. We refer to a *hyperplane* as a maximal set of solutions that share the same truth assignment over some subset of n variables. The *hyperplane average* is the average evaluation of all solutions in the hyperplane. Using our technique to efficiently compute hyperplane averages, we can then explicitly determine which combination of variable assignments will lead to the highest overall combined hyperplane average.

We find the hyperplane averages of the hyperplanes corresponding to each possible assignment of the ten variables that appear most often in the reduced formula produced by SatELite. We then reduce the formula further by first removing the clauses satisfied by the partial assignment over the ten variables that corresponds to the highest average hyperplane. We next remove the ten variables from the remaining clauses they appear in. The resulting formula is then passed to MiniSAT [1].

[1]http://minisat.se/Main.html

The SAT space is known to be deceptive [3]. In other words, we know that the hyperplane with the best average may not contain a globally optimal solution. It is therefore possible that our reduced problem may be unsatisfiable even though the original formula is satisfiable. To allow for this possibility, we limit the run-time of MiniSAT to 10 minutes on the hyperplane reduced formula. If this upper limit is reached, or MiniSAT finds the formula to be unsatisfiable before the time limit, MiniSAT runs on the SatELite reduced formula until it terminates (either by deciding the satisfiability of the problem or reaching some maximum time limit).

II. Computing Hyperplane Averages

A discrete function $f : \{0,1\}^n \mapsto \mathbb{R}$ can be decomposed into an orthogonal basis

$$f(x) = \sum_{i=0}^{2^n-1} w_i \psi_i(x)$$

where w_i is a real-valued weight known as a *Walsh coefficient* and ψ_i is a *Walsh function*. The index i and vector x can be represented as binary strings, and standard binary operations can be applied. The Walsh function

$$\psi_i(x) = -1^{i^T x}(-1)^{bitcount(i \wedge x)}$$

generates a sign: if $i^T x$ is odd $\psi_i(x) = -1$ and if $i^T x$ is even $\psi_i(x) = 1$.

The MAXSAT objective function is given by

$$f(x) = \sum_{j=1}^{m} f_j(x, mask_j)$$

where each subfunction f_j corresponds to a clause and $mask_j$ selects the bits used by f_j. Since MAXSAT is a linear combination of subfunctions, we can apply the Walsh transform to each clause:

$$w = \sum_{j=1}^{m} \mathbf{W} f_j$$

where w is a vector of polynomial coefficients and $\mathbf{W}$ is a discrete Fourier transform known as the Walsh transform. This generates the Walsh coefficients associated with each clause, and then adds them together as needed. We will use the Walsh transform without normalization, since this results in all of the Walsh coefficients being integer values. Rana et al. [3] show that we can dispense with matrix $\mathbf{W}$ and directly compute the Walsh coefficient associated with each clause.

Each subfunction f_j contributes at most 2^k nonzero Walsh coefficients to vector w.

The Walsh coefficients can be used to efficiently compute the average evaluation of solutions contained in any $(n-j)$-dimensional hyperplane [3] [4]. Let h denote a $(n-j)$ dimensional hyperplane where j variables have preassigned bit values. Let $\alpha(h)$ be a mask with 1 bits marking the locations where the j variables appear in the problem encoding, and 0 bits elsewhere. Let solution x assign values to the j variables. Let $\beta(h) = \alpha(h) \wedge x$. This means $\beta(h)$ has value 0 in all of the positions where the j bits do not appear, and has the assigned values of the relevant j bits in the appropriate bit locations. Then the average fitness of hyperplane h is

$$Avg(h) = f_{\text{avg}} + \sum_{\forall b, b \subseteq \alpha(h)} w_b \psi_b(\beta(h))$$

where $f_{\text{avg}} = w_o$ is the average over the entire SAT search space, i.e. $f_{\text{avg}} = (2^k - 1)/(2^k) * m$.

For our preprocessor, we use the $j = 10$ variables that correspond to the 10 variables that appear most frequently in the given formula. We then compute $Avg(h)$ over all possible truth assignments to the j variables. The assignment yielding the highest average is then chosen and the problem is reduced by first eliminating the clauses satisfied by the assignment and then removing the j variables from the remaining clauses. The reduction is then passed off to MiniSAT.

The major drawback of this method is that the partial assignment may not correspond to a model for the original formula. However, in our experiments we find that the best hyperplane often does contain a satisfying solution. When the hyperplane does contain a satisfying solution, it is found quickly by MiniSAT. We therefore allow MiniSAT 10 minutes on the hyperplane reduction, if it is found unsatisfiable or reaches the time limit before finding a satisfying solution, the remaining time is spent by MiniSAT running on the original problem.

We note that this may not be the ideal method of integrating the hyperplane information into a complete solver given the fact that we cannot guarantee the reduction is satisfiable if the original problem is satisfiable. Nevertheless, it does demonstrate that there are instances where this information can benefit a complete solver. Future work will lie in exploring alternative ways of guiding the solver using hyperplane information, e.g. biasing assumptions or activity scoring.

III. Main Parameters

We use unmodified MiniSAT [1] as a complete solver and use the default parameters. Our contribution is a hyperplane reduction preprocessing step. The only parameter to our reduction is the number of bits to fix. We use 10 bits as this setting gave us the most consistent results in our empirical tests. It is possible that the number of bits is instance dependant and some characteristics may be used to determine the optimal number, but we leave this for future work.

IV. Implementation details

We used C to implement the calculation of Walsh coefficients and the hyperplane averages. Our C code also performs the reduction of the problem based on the truth assignments determined by the hyperplane averages as described above. We used GCC 4.7.2 to test our code using the `O3` compiler flag to create an optimized 64-bit binary.

A BASH script wrapper is used to perform the reduction and call SatELite and MiniSAT on the reduced problem. If a satisfiable solution is found in the reduced problem, we add the truth assignments fixed by our reduction code to the satisfying solution provided by MiniSAT and report this solution to the original problem.

V. SAT Competition 2013 Specifics

We have submitted our solver to the crafted SAT and industrial SAT tracks of the competition. Although MiniSAT can determine if a hyperplane reduced instance is unsatisfiable, it does not necessarily mean that the original problem is unsatisfiable. Therefore we only submitted our entry to the SAT tracks.

We chose the crafted and industrial tracks because we conjecture that there is more variance in the hyperplane averages in structured instances than random instances. We believe that this structure allows our algorithm to be more effective on these types of problems.

VI. Availability

Our hyperplane reduction code along with the wrapper script for calling MiniSAT and SatELite can be downloaded from the following url: http://www.cs.colostate.edu/~dhains/hyperplaneminisat.tar.gz

VII. Acknowledgments

This research was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA9550-11-1-0088. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386.

References

[1] N. Een and N. Sörensson, "Minisat: A sat solver with conflict-clause minimization," *Sat*, vol. 5, 2005.

[2] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 61–75.

[3] S. Rana, R. Heckendorn, and D. Whitley, "A tractable walsh analysis of SAT and its implications for genetic algorithms," in *Proceedings of the National Conference on Artificial Intelligence*, 1998, pp. 392–397.

[4] D. Goldberg, "Genetic Algorithms and Walsh Functions: Part I, A Gentle Introduction," *Complex Systems*, vol. 3, pp. 129–152, 1989.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

forl

Mate Soos
Security Research Labs

I. Introduction

This paper presents the defining features of the conflict-driven clause-learning SAT solver *forl*. *forl* aims to be a modern SAT Solver that unifies the ideas present in SatELite [1], PrecoSat [2], glucose [3] and MiniSat [4] with some ideas of the author.

II. Primary features

A. Binary implication graphs

An implication cache mechanism is employed that stores the binary implication graph similarly to stamps [5]. Stamps are also used, as they have been found to aid along with the cache.

B. Clause cleaning

Clauses are cleaned regularly, but neither activities nor glues are used in the cleaning. Instead, the number of times a clause helped to propagate or caused a conflict is used as a measure of the effectiveness. This measure is reset after every cleaning, so clauses have to regularly prove themselves effective to stay in the database.

C. Implicit Clauses

Binary and tertiary clauses are stored and handled implicitly. This greatly eases their subsumption and strengthening. Further, it reduces the cost of creating occurrence lists out of these clauses. Implicit clauses are never cleaned.

D. Statistics

forl gathers large amounts of running statistics. Unfortunately they are not yet used to direct search. However, they can be gathered into MySQL and displayed in a web browser. Importing statistics into the database incurs setup costs and about 10% running cost and so is disabled by default.

E. Time limiting

For average problems inprocessing techniques tend to work well. However, in case of strange problems (such as problems with billions of binary clauses) they sometimes misbehave. This has been solved with more precise time measurements (measuring effort, not actual time) and sometimes complicated time-out checks.

F. Memory usage

Memory usage has been greatly improved with precise tracking of where memory is being used. Although memory leaks are not generally an issue given the programming techniques used, temporary allocation of large data structures was a problem. These issues have been fixed through algorithmic means: e.g. through the use of circular swapping for variable renumbering.

G. Hyper-binary resolution and transitive reduction

On-the-fly hyper-binary resolution [6] and transitive reduction has been implemented in both DFS and BFS probing for both irreducible and reducible binaries. This helps on instances with generally acceptable number of binary clauses. For problems with too many binary clauses, transitive reduction can take too much time. Such cases are detected and transitive reduction is turned off.

H. Certified UNSAT

The DRUP system for certified UNSAT was implemented into *forl*. The current implementation turns on all optimisations except for XOR-manipulation during certificate generation. However, for stamping and implied literal caching to work, binary clauses must never be DRUP-deleted during variable elimination. This trade-off is questionable, as it might considerably slow down proof checking. As such, there are two versions submitted, one with these options turned on, and one with these options turned off.

I. Disjoint component finding

Disjoint components are searched for on a regular basis during solving. These disjoint components are solved with a separate solver instance, renumbering the component's variables such as to minimise the startup time of the subsolver. On certain problems, *forl* can find&solve thousands of disjoint components within a matter of seconds.

III. Miscellaneous optimisations

Hand-rolled memory manager for large clauses, clause offsets instead of pointers, blocking literals, occurrence lists in watchlists, clause abstraction stored in occurrence lists, glue-based and geometric restart selection based on literal polarities, xor detection and manipulation, gate detection and manipulation, variable elimination [1], subsumption, strengthening, on-the-fly subsumption [7], recursive conflict clause minimisation [8] (and automatic disabling in case of bad performance), minimisation with stamps&cache&binary clauses (and automatic disabling in case of bad performance), blocking of long clauses [9], equivalent literal replacement, variable renumbering, literal dominator branching thanks to stamps/cache, dominator probing, polarity caching [10], vivification [11] of long and implicit clauses, watchlist sorting for quasi- prioritised implicit clause propagation, regular cleaning of false literals of all clauses, detection of long trail and consequent restart blocking in case of satisfiable problems, MiniSat-type variable activities, glue-based extra variable activity bumping,

prefetching of watchlists on literal enqueue, optional UIP conflict [12] graph generation, probing (with automatic tuning based on past performance), clause subsumption through irreducible stamps and cache, clause strengthening through reducible&irreducible stamps and cache, precise elimination cost prediction for better elimination order, gradual variable elimination, variable elimination with searching for subsumed&subsuming product clauses.

Acknowledgements

The author would like to thank in no particular order Martin Maurer, Vegard Nossum, Valentin Mayer-Eichberger, George Katsirelos, Karsten Nohl, Luca Melette, Marijn Heule, Vijay Ganesh, Trevor Hansen and Robert Aston for their help.

References

[1] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: SAT. Volume 3569 of LNCS., Springer (2005) 61–75
[2] Biere, A.: P{re,i}cosat@sc'09: a solver that predicts learnt clauses quality. In: SAT 2009 competitive events booklet. (2009) 41–42
[3] Audemard, G., Simon, L.: GLUCOSE: a solver that predicts learnt clauses quality. In: SAT 2009 competitive events booklet. (2009) 7–8
[4] Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518
[5] Heule, M., Järvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In Sakallah, K.A., Simon, L., eds.: SAT. Volume 6695 of LNCS., Springer (2011) 201–215
[6] Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 341–355
[7] Han, H., Somenzi, F.: On-the-fly clause improvement. [13] 209–222
[8] Sörensson, N., Biere, A.: Minimizing learned clauses. [13] 237–243
[9] Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In Esparza, J., Majumdar, R., eds.: TACAS. Volume 6015 of LNCS., Springer (2010) 129–144
[10] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., Sakallah, K.A., eds.: SAT. Volume 4501 of LNCS., Springer (2007) 294–299
[11] Piette, C., Hamadi, Y., Sais, L.: Vivifying propositional clausal formulae. In Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M., eds.: ECAI. Volume 178 of Frontiers in Artificial Intelligence and Applications., IOS Press (2008) 525–529
[12] Silva, J.P.M., Sakallah, K.A.: GRASP-a new search algorithm for satisfiability. In: ICCAD'96, IEEE Computer Society (1996) 220–227
[13] Kullmann, O., ed.: Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. In Kullmann, O., ed.: SAT. Volume 5584 of Lecture Notes in Computer Science., Springer (2009)

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

FrwCB2013

Chuan Luo
School of EECS, Peking University
Beijing, China
chuanluosaber@gmail.com

Kaile Su
School of EECS, Peking University
Beijing, China
kailepku@gmail.com

***Abstract*—This document describes the SAT solver "FrwCB2013", a new local search solver for the satisfiability problem (SAT).**

I. Introduction

Heuristics in SLS algorithms for SAT can be divided into three categories: GSAT, focused random walk (FRW) and dynamic local search (DLS). FRW algorithms conduct the search by always selecting a variable to flip from an unsatisfied clause chosen randomly in each step.

In this work, we propose a new heuristic for FRW, called CCBM, which combines the configuration checking (CC) strategy [1] and the break minimum (BM) strategy effectively in a subtle way. The BM strategy prefers to pick the variable which brings fewest number of clauses from satisfied to unsatisfied, and is a commonly used strategy in FRW algorithms, such as WalkSAT [2]. Originally proposed in [3], the CC strategy reduces the cycling problem by checking the circumstance information. It has been successfully used in non-FRW algorithms, leading to several state-of-the-art SLS solvers such as CCASat [4].

Based on the CCBM heuristic, we develop our solver called FrwCB2013, which cooperates CCBM with two breaking ties strategies well, for SAT.

II. Preliminaries

In this section, we introduce some basic notations and definitions used in this document. We use $V(F)$ to denote the set of all variables appearing in the formula F. Two different variables are neighbors when they appear in at least one clause, and $N(x) = \{y \mid y \in V(F),\ y \text{ and } x \text{ are neighbors}\}$ is the set of all neighbors of variable x. We also denote $CL(x) = \{c \mid c \text{ is a clause which } x \text{ appears in}\}$.

A (possibly partial) mapping $\alpha : V(F) \to \{True, False\}$ is called an *assignment*. If α maps all variables to a Boolean value, it is *complete*. For local search algorithms for SAT, a candidate solution is a complete assignment. Given a complete assignment α, each clause has two possible *states*: *satisfied* or *unsatisfied*: A clause is satisfied if at least one literal in that clause is true under α; otherwise, it is unsatisfied.

The method of selecting the flipping variable in each step is usually guided by a scoring function. In each step, the flipping variable is selected usually based on its properties, such as $make$, $break$ and $score$. For a variable x, the property $make(x)$ is defined as the number of clauses that would become satisfied if the variable is flipped; the property $break(x)$ is the number of clauses that would become unsatisfied if the variable is flipped; the property $score(x)$ is the increment in the number of satisfied clauses if the variable is flipped, and can be understood as $make(x) - break(x)$.

III. Main Techniques

The FrwCB2013 solver is abased on the Stochastic Local Search (SLS) algorithmic paradigm. Then we introduce the main techniques used in the FrwCB2013 solver.

A. The CCBM Heuristic

The CCBM heuristic is based on the concept of configuration changed *configuration changed decreasing* (CCD) variable and *break minimum* (BM) variable in a clause. The CCD variable is based on the concept of $configuration$. This work uses the clause states based $configuration$ proposed in [5]. We also employ an integer array $ConfTimes$, whose size equals the number of variables in the formula. For each variable x, $ConfTimes(x)$ measures the frequency (i.e., the number of steps) that $configuration(x)$ has been changed since x's last flip. We main $ConfTimes(x)$ as follows.

- Rule 1: In the beginning, all the variables' $ConfTimes$ are set to 1.
- Rule 2: Whenever a variable x is flipped, $ConfTimes(x)$ is reset to 0. Then we scan each clause $c \in CL(x)$ to check whether its state is changed by flipping x. If this is the case, for each variable y in c (except for x), $confTimes(y)$ is increased by 1.

A variable x is *configuration changed decreasing* (CCD) if $score(x) > 0$ and $ConfTimes(x) > 0$. For each clause c, a variable x is *break minimum* (BM) in clause c if and only if $break(x) = \min\{break(y) \mid\ y \text{ appears in } c\}$. In this work, we use *CCDVars(c)* and *BMVars(c)* to denote the sets of all CCD variables and BM variables in the clause c, respectively.

The main idea of the CCBM heuristic is the preference to flipping CCD variables and BM variables. Flipping a CCD variable brings down the amount of unsatisfied clauses, and at the same time prevents the algorithm from revisiting the scenario the algorithm recently faced with. Although previous works such as [1], [4] also prefer to flip CCD variables, they survey CCD variables globally, i.e., searching CCD variables from all the variables. In contrast, the CCBM heuristic picks a CCD variable from an unsatisfied clause. Whenever no CCD variable is present, CCBM prefers to pick a BM variable

(a variable with minimum break value) to flip, leading the algorithm to search deeply.

B. The Multilevel Score

This work also adopts the multilevel score property, inspired by [6]. Based on this concept, we design a scoring function called linear score. We give some necessary definitions used in linear score. A clause is τ-true if and only if it contains exactly τ true literals under assignment α. For a variable x, its τ-th level score, denoted by $score_\tau(x)$, is increment in the number of τ-true clauses if x is flipped. For each variable x, $lscore(x) = score(x) + score_2(x)$.

IV. The FrwCB2013 Solver

The FrwCB2013 solver is mainly based on the CCBM heuristic, and uses $ConfTimes(x)$ and $lscore(x)$ to break ties on solving random k-SAT instances. We denote the specific breaking ties mechanism as ***BTM***. For random 3-SAT and 4-SAT instances, ***BTM*** is $ConfTimes(x)$. For random 5-SAT, 6-SAT and 7-SAT instances, ***BTM*** is $lscore(x)$. The pseudo code of FrwCB2013 is outlined in Algorithm 1.

Algorithm 1: FrwCB2013

Input: CNF-formula F, $maxSteps$
Output: A satisfiable assignment α of F or $Unknown$

```
generate a random assignment α;
initialize ConfTimes(x) as 1 for each variable x;
for step ← 1 to maxSteps do
    if α satisfies F then return α;
    c ← an unsatisfied clause chosen randomly;
    if CCDVars(c) is not empty then
        v ← x with the greatest score(x) in CCDVars(c),
        breaking ties by BTM;
    else if with the fixed probability p then
        v ← x with the greatest ConfTimes(x) in
        BMVars(c), breaking ties by BTM;
    else
        v ← x with the greatest ConfTimes(x) in
        clause c, breaking ties by preferring the least
        recently flipped one;
    flip v and update ConfTimes;
return Unknown;
```

V. Main Parameters

FrwCB2013 is involved in only one parameter, i.e., the probability p. For 3-SAT instances, p is set to 0.6 (ratio$<$0.6) and 0.63 (ratio$\geq$0.63). For 4-SAT instances, p is set to 0.65 (ratio$<$9.63) and 0.7 (ratio$\geq$9.63). For 5-SAT instances, p is set to 0.53 (ratio$<$20.2) and 0.6 (ratio$\geq$20.2). For 6-SAT instances, p is set to 0.67 (ratio$<$43.0) and 0.69 (ratio$\geq$43.0). For k-SAT instances with $k \geq 7$, p is set to 0.73 (ratio$<$86.0) and 0.78 (ratio$\geq$86.0). For other instances, p is set to 0.95.

VI. Implementation Details and SAT Competition 2013 Specifics

The FrwCB2013 solver is implemented in program language C, and compiled with '-O3', '-static' and '-m64' options by gcc. The FrwCB2013 solve is open source. We submit the FrwCB2013 solver to the Core solvers, Sequential, Random SAT track and Core solvers, Parallel, Random SAT Track in SAT Competition 2013.

FrwCB2013 is a 64-bit binary. The command line of FrwCB2013 is described as follows.

```
./FrwCB2013 <instance> <seed>
```


Acknowledgment

The authors would like to thank Shaowei Cai for discussions on some novel ideas.

References

[1] S. Cai and K. Su, "Local search with configuration checking for SAT," in *Proc. of ICTAI-11*, 2011, pp. 59–66.

[2] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proc. of AAAI-94*, 1994, pp. 337–343.

[3] S. Cai, K. Su, and A. Sattar, "Local search with edge weighting and configuration checking heuristics for minimum vertex cover," *Artif. Intell.*, vol. 175, no. 9-10, pp. 1672–1696, 2011.

[4] S. Cai and K. Su, "Configuration checking with aspiration in local search for SAT," in *Proc. of AAAI-12*, 2012, pp. 434–440.

[5] C. Luo, K. Su, and S. Cai, "Improving local search for random 3-SAT using quantitative configuration checking," in *Proc. of ECAI-12*, 2012, pp. 570–575.

[6] S. Cai, K. Su, and C. Luo, "Improving WalkSAT for random k-satisfiability problem with $k > 3$," in *Proc. of AAAI-13*, 2013, p. To appear.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Glucans System Description

Xiaojuan Xu*, Yuichi Shimizu*, Shota Matsumoto*, and Kazunori Ueda*
*Department of Computer Science and Engineering
Waseda University, Tokyo, Japan
Email: {xxj, yusui, matsusho, ueda}@ueda.info.waseda.ac.jp

***Abstract*—This document describes "Glucans", a family of parallel SAT solvers based on existing CDCL solvers. Glucans run GLUCOSE and/or GLUEMINISAT in parallel, exchanging learnt clauses limited by their LBDs. The base solvers incorporate the ideas of two minisat-hack-solvers: Contrasat and CIRMinisat.**

I. Overview

Glucans are a family of parallel SAT solvers based on GLUCOSE. These solvers run GLUCOSE [1] and/or GLUEMINISAT [3] in parallel using Pthreads, letting them exchange learnt clauses [4] selected based on Literal Block Distance [2](LBD) in multiple phases. Based on experimental results, the learnt clauses whose LBDs are not greater than 5 will be sent to other threads. The base solvers also incorporate the ideas of two minisat-hack solvers: Contrasat [5], which improves the order of literals that are waiting to be propagated, and CIRMinisat [6], which changes the VSIDS scores on each restart. The base solvers can behave like these solvers by using options. Since these minisat-hack solvers were strong in the SAT instances of SAT Competition 2011, we expect our modified solvers to perform well for such instances by including them into the set of solvers.

II. Special Algorithms, Data Structures and Features

Glucans exchange learnt clauses using queues whose elements are pointers to clauses. The exchanges are made with the LBD values after each conflict or if the strict LBD[3] value of a learnt changes to two during propagation.

Our experimental results show that the exchanging time takes less than 1% of the total runtime. Different random seed values are used for each thread, and some threads run in the weak polarity mode.

Because of increasing the number of learnt clauses by exchanging, Glucans delete more useless learnt clauses evaluated by strict LBD, always keeping learnts whose strict LBDs are two or less.

III. Implementation detail

We use GLUCOSE, GLUEMINISAT, Contrasat and CIRMinisat as base solvers. The differences between our solver and the original solvers are about two hundreds lines in total.

Each thread autonomously shares the learnt clauses using queues (implemented as linked lists) by the following steps.

1) Create a copy of the learnt clause when the state of the thread is conflicting or the strict LBD of learnt is changed to two on propagation.
2) Lock the tails of the queues of the other threads and insert the pointers to the created copy.
3) Read its own queue and add received clauses to the database, if this exchange is not on propagation.

IV. SAT Competition 2013 Specifics

This solver is submitted to the open track and the track of Parallel Solvers – Application SAT+UNSAT and Hard-combinatorial SAT+UNSAT. This solver needs gcc above 4.4 and is compiled with the -O2 -march=native flag. The solver has the following command-line options.

1) -rnd-seed= : the initial seed of the first thread.
2) -nof-threads= : the number of threads to use.
3) -ex-size= : the maximum LBD for exchanging learnt clauses.
4) -sendmore= : share the learnts aggressively by sending them on propagation.
5) -mem-lim= : the memory limit to determine the best number of threads.
6) -se-lim= : the cputime limit for preprocessing.

V. Availability

Glucans will be available at our website, http://www.ueda.info.waseda.ac.jp/sat/glucans .

Acknowledgment

This solver was tested with EDACC [7]. We would like to thank the authors of the base solvers and EDACC.

References

[1] Gilles Audemard, Laurent Simon, GLUCOSE: a solver that predicts learnt clauses quality, SAT 2009 Competition Event Booklet, http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf, 2009.
[2] Gilles Audemard, Laurent Simon, Predicting Learnt Clauses Quality in Modern SAT Solver, Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09), 399-404, 2009.
[3] Hidetomo Nabeshima, Koji Iwanuma, Katsumi Inoue, GLUEMINISAT, http://GLUEMINISAT.nabelab.org/, 2011.
[4] Kei Ohmura, Kazunori Ueda, c-sat: A parallel SAT solver for clusters, SAT 2009. LNCS 5584, 524–537. Springer, Heidelberg, 2009.
[5] Allen Van Gelder, Contrasat – A Contrarian Sat Solver, Extended System Description, Journal on Satisfiability, Boolean Modeling and Computation 8, 117–122, 2012.
[6] The results of SAT Competition 2011, http://www.satcompetition.org/2011/, 2011.
[7] Balint, A., Gall, D., Kapler, G., Retz, R.: Experiment design and administration for computer clusters for SAT-solvers (EDACC). JSAT 7, 77–82, 2010.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

GlucoRed

Siert Wieringa
Aalto University School of Science
Espoo, Finland

***Abstract*—This document describes the SAT solver GlucoRed as submitted the SAT Competition 2013. GlucoRed is an implementation of the solver/reducer architecture based on Glucose 2.1.**

I. Introduction

The solver GlucoRed discussed in this document implements the solver/reducer architecture [1], on top of the solver Glucose 2.1[1] [2]. It uses two concurrently executing threads, which are called the SOLVER and the REDUCER. The SOLVER acts just like Glucose would, except for its interaction with the REDUCER. The REDUCER's sole task is to strengthen the conflict clauses derived by the SOLVER. The interaction between the SOLVER and the REDUCER is handled by passing clauses through two shared-memory data structures called the *work set* and the *result queue*.

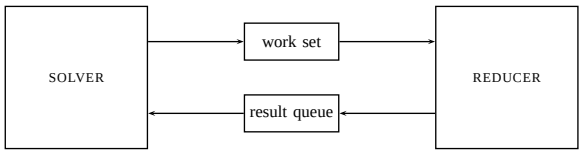

Fig. 1. The solver-reducer architecture

Whenever the SOLVER learns a clause it is pushed into the work set. The work set has a limited capacity. If a clause is pushed into the work set while it is full the new clause will replace the oldest clause in the set. The task of the REDUCER is to strengthen the clauses provided by the SOLVER through the work set. The reducer always picks the "best" clause from the work set as its next input. By setting the sorting metric for the work set the user may define the "best" clause as either the newest clause, the shortest clause or the clause with the smallest LBD [2]. The REDUCER tries to strengthen the clause by an algorithm based on unit propagation and conflict clause learning. If the REDUCER successfully reduces the length of a clause it places this new reduced clause in the result queue.

The SOLVER reads the clauses from the result queue, and adds them to its learnt clause database. The SOLVER can decide to do this at any decision level, hence the introduction of these "foreign" clauses may force the solver to backtrack.

II. Running GlucoRed

The solver GlucoRed inherits all parameters and magic constants from Glucose. Changing the value of any of these parameters affects both the SOLVER and the REDUCER, except for `-ccmin` which affects only the REDUCER. A new parameter `-solver-ccmin` controls the conflict clause minimization mode of the SOLVER. Both `-ccmin` and `-solver-ccmin` have default value "2=deep", which is the same as in MiniSAT and Glucose. There are two other new GlucoRed specific parameters. The first is the parameter `-work` which is an integer > 1, and represents the capacity of the work set. Its default value is the magic constant 1000. The second new parameter, `-rsort`, controls the sorting metric used for the work set. Legal values are "0=off" (newest first), "1=by size" (shortest first), "2=by LBD" (smallest LBD first). The default value is 2.

Although GlucoRed uses concurrency its performance remains decent when it is run on a single physical CPU core. This can be enforced for example by using the LINUX command `taskset`. We therefore submit our solver to both the sequential and parallel core solver tracks, for the benchmarks Application SAT+UNSAT, and Hard-combinatorial SAT+UNSAT.

III. Implementation details

GlucoRed is an extension of Glucose 2.1, which itself is based on MiniSAT 2.2.0[2]. All code is written in C++. The code that is unique to GlucoRed uses POSIX threads. The SOLVER and the REDUCER are both derived from Glucose's `Solver` class. GlucoRed was compiled to include MiniSAT's internal simplifier as implemented in the `SimpSolver` class. MiniSAT's original Makefile was used for compiling. Before submitting the code to the competition it has been tested after compiling it for a 64-bit architecture using `gcc` versions 4.4.7 and 4.6.3. It should also work correctly when compiled for a 32-bit architecture.

The version of GlucoRed submitted here differs from the version used for the experiments in [1] by the addition of MiniSAT's simplifier, and a minor clean-up of the source code.

IV. GlucoRed-Multi

GlucoRed-Multi is a simple multi-process portfolio of multiple instances of GlucoRed. There is no clause sharing between the different processes, but file parsing, initial iterative unit propagation, and optional simplification are only performed once. This is achieved by creating one instance of the solver, parsing the input file and performing preprocessing, and then forking the process multiple times.

[1] http://www.lri.fr/~simon

[2] http://www.minisat.se

Compared to GlucoRed the solver GlucoRed-Multi has two extra parameters, `-nc` and `-ns`. The parameter `-nc` controls the number of instances of GlucoRed to run directly on the input formula. The parameter `-ns` controls the number of instances of GlucoRed to run on the formula obtained by simplification using MiniSAT's internal simplifier. The GlucoRed-Multi solver was submitted to the same parallel core solver tracks as the basic GlucoRed solver, with parameter settings `-nc=1` and `-ns=3`. Given those settings and an input formula GlucoRed-Multi will do the following:

1) Create an instance of the 'GlucoRed' solver
2) Parse the input formula
3) Fork a copy of the process, run solver in child process.
4) Run the simplifier in the parent process.
5) Fork two copies of the parent process, run the solver in the parent process and both children.

The solver instance running in the parent process uses all the default GlucoRed settings. The solver instances running in the child processes also use the default settings, except from making 2% of their branching decisions at random and having a unique random seed based on their process id.

V. GlucoRed+March

GlucoRed+March is our submission to the *open track*. Even though the ranking in the open track is based on wall clock time this submission aims for a decent performance regarding CPU time. This is an experimental submission to see how simple heuristics compete with complex portfolios. It is not meant to be a serious contender for any awards.

GlucoRed+March runs a single copy of the solver march_rw[3] [3] if all clauses of the input formula have the same length, or if the formula contains clauses of exactly two different lengths and the diameter of the variable interaction graph (VIG) is at most 4. In all other cases GlucoRed+March runs a single copy of GlucoRed. Checking whether all clauses have the same length is a cheap way of determining that the formula is likely to be a random k-SAT formula. The use of the diameter of the VIG was inspired by [4].

VI. Availability

The source code for all submitted solvers is available from the author's web page[4]. The sources for MiniRed, a MiniSAT based solver/reducer implementation, are also provided through that same page. Both GlucoRed and MiniRed are licensed under MiniSAT's original non-restrictive license.

References

[1] S. Wieringa and K. Heljanko, "Concurrent clause strengthening," in *SAT*, ser. Lecture Notes in Computer Science, to appear 2013.

[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI*, C. Boutilier, Ed., 2009, pp. 399–404.

[3] M. J. H. Heule, "Smart solving: Tools and techniques for satisfiability solvers," Ph.D. dissertation, Delft University of Technology, The Netherlands, 2008.

[4] P. Herwig, "Decomposing satisfiability problems," Master's thesis, Delft University of Technology, The Netherlands, October 2006.

[3] http://www.st.ewi.tudelft.nl/sat

[4] http://users.ics.aalto.fi/swiering/solver_reducer

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Glucose 2.3 in the SAT 2013 Competition

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

Laurent Simon
Univ. Paris-Sud
Orsay, F-91405
simon@lri.fr

***Abstract*—Glucose is a CDCL solver heavily based on Minisat, with a special focus on removing useless clauses as soon as possible. In this new 2.3 version we proposed to increase again the aggressive nature of the deletion strategy of the previous version. This short system description is the 2013 SAT Competition companion paper for Glucose 2.3.**

I. Introduction

In the design of efficient CDCL-based SAT solvers [1], [2], a lot of effort has been put in efficient Boolean Constraint Propagation (BCP), learning mechanisms, and branching heuristics, their three main composants. In [3], a new simple measurement of learnt clause usefulness was introduced, called LBD. This measure was no more based on past clauses activities. It was proved so efficient that, since 2009, Glucose and its updated versions was always one of the award winning SAT solvers in the corresponding competitive events. This year, we proposed a minor revision of Glucose 2.2 (used in the SAT 2012 Challenge, see [4] for details) and increased its agressive database cleanup strategy once again. The solver is particularly well suited for UNSAT problems but can, thanks to the techniques developed in the 2.2 version, still be competitive on SAT problems too. This short paper (competition companion) describes the main techniques used in Glucose 2.3 from a technical point of view.

It is important to notice that, like Glucose 2.2, Glucose 2.3 is based on the version 2.2 of Minisat [2] (Glucose 1.0 was based on the previous version of Minisat). For a more comprehensive description of Glucose, please refer to [3] and our previous competition (2009,2012) reports. We focus here on the novelties brought to Glucose since the 2012 version.

II. Main Techniques

During search, each decision is often followed by a large number of unit propagations. We called the set of all literals of the same level a "blocks" of literals. Intuitively, at the semantic level, there is a chance that they are linked with each other by direct dependencies. The underlying idea developed in [3] 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 [5] and rapid restarts.

Let us just recall what is the **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. 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" (giving the name "glucose").

The LBD measure can be easily re-computed on the fly when the clause is used during unit propagation. We keep here the strategy used in Glucose 1.0: we change the LBD value of a clause only if the new value becomes smaller. However, in the 2.3 version this update is only performed during conflict analysis, and not during propagation.

III. Novelties of Glucose 2.3

Before Glucose 1.0, the state of the art was 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 deleted at most half of the clauses, depending on their score (note that binary and glue clauses are never deleted). In Glucose 1.0, we already chose a very slow increasing strategy. In this new version, we perform a more accurate management of learnt clauses.

A. Dynamic threshold, revisited

As a basis, we used the 2.2 version of the cleaning process: Every $4000 + 300 \times x$, we removed at most half of the learnt clause database , which this is much more aggressive than the version 1.0 , *i.e.* $20000 + 500 \times x$). Of course, binary clauses, glue clauses and locked clauses are always kept. A locked clause is (1) used as a reason for unit propagation in the current subtree or (2) locked (see [4]). However, in the 2.3 version, thanks to a more focused update of the interesting clauses (propagated clauses LBD scores that are not seen in any conflict are not updated), we were able to use the following more aggressive strategy : the cleaning process is fired every $2000 + 300 \times x$. This gives us clause database cleaning after

2000, 4300, 6900, 9800, ... conflicts, instead of 4000, 8300, 12900, 17800, ... used in Glucose 2.2. This choice is also related to the new technique described in the next subsection.

The dynamic nature of the threshold used to keep more or less clauses at each cleaning process is kept intact (see version 2.2 description).

B. Starts with a wider search for proofs

The `var_decay` constant is 0.95 in Minisat, which was a very good compromise for most of the benchmarks [6]. However, thanks to the work of [7], it was shown that fixing it to the same value was not always a good choice. Thus, we proposed to use it like some kind of temperature, starting from 0.8 and increasing it by 0.01 every 5000 conflicts until it reaches the 0.95 value (after 75000 conflicts). This idea arose during one of the fruitful discussions we had with George Katsirelos, Ashish Sabharwal and Horst Samulowitz and thus the credits for this idea are clearly shared with them. Adding this rule allowed us to make a small step in Glucose performances (3 additional problems solved on the previous SAT Challenge set of problems) but it may open the door for further improvements.

One may notice that, after a few ten thousands conflicts, the more agressive strategy used for clause database cleanings (using the constant 2000 instead of 4000) tends to disappear, because the strategy will be mostly dominated by the $300 \times x$ part of the increasing variable. The meaning of that is that we want to quickly drop "bad" (useless) clauses generated when `var_decay` was still not properly set.

C. Other embedded techniques

Since the first versions of Glucose, we used the stand alone simplifier "Satelite". In the 2.3 version, we used the built-in "Simp" class of Minisat that simplify the formula in place.

The laziness of the LBD update mechanism is inspired by the study of CDCL solvers proposed in the source code of the work of Long Guo in `www.cril.fr/~guo`.

IV. Support for UNSAT Proof Checking

Thanks to the effort of Marijn Heule, who implemented the support for DRUP (Delete Reverse Unit Propagation) proof checker into Minisat and Glucose 2.2, it was trivial to port his code into Glucose 2.3. Currently, there are two distinct versions of Glucose (with or without DRUP support) but the next release of Glucose will contain the support for UNSAT proof checking as an argument. More information on the work of Marijn Heule and the DRUP file format can be found at `www.cs.utexas.edu/~marijn/drup`.

V. Main Parameters

Given the fact that auto-tuning of SAT solvers is a classical technique for improving the performances and propose a more scientific approach to fix the parameters, most of the parameters are accessible via command line options. See the description above for the specific parameters we used for Glucose 2.3.

The main objective of Glucose was to target UNSAT Applications problems. However, the blocking-restart strategy introduced in Glucose 2.2 allows to keep a good score on SAT problems too (see [4] for this).

VI. Algorithm and Implementation details

Glucose uses a special data structure for binary clauses, and a very limited self-subsumption reduction with binary clauses, when the learnt clause is of interesting LBD. Glucose is targeting Application problems.

The main page of Glucose is

`http://www.lri.fr/~simon/glucose`.

References

[1] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff : Engineering an efficient SAT solver," in *DAC*, 2001, pp. 530–535.

[2] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.

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

[4] ——, "Glucose 2.1: Aggressive but reactive clause database management, dynamic restarts," in *SAT'12 Workshop on Pragmatics of SAT (POS'12)*, 2012.

[5] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *SAT*, 2007, pp. 294–299.

[6] G. Audemard and L. Simon, "Experimenting with small changes in conflict-driven clause learning algorithms," in *Proc of International Conference on Principles and Practice of Constraint Programming*, 2008.

[7] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon, "Resolution and parallelizability: Barriers to the efficient parallelization of sat solvers," in *Proceedings of the Twenty-Seventh Conference on Artificial Intelligence (AAAI-13)*, 2013.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Solvers with a Bit-Encoding Phase Selection Policy and a Decision-Depth-Sensitive Restart Policy

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

***Abstract*—To develop more efficient SAT solvers, recently we proposed two new solving policies. One is called bit-encoding phase selection policy, which aims at selecting more exactly the polarity of a decision variable. Another is called a decision-depth-sensitive restart policy, which determines when a restart begins, depending on search depths. Based on the two new policies, we developed a number of new SAT solvers, which are named glue_bit, minisat_bit, gluebit_lgl and gluebit_clasp. This paper describes briefly them.**

I. Introduction

Most of modern solvers are based on Conflict Driven Clause Learning (CDCL), which is a variant of DPLL procedure. In general, CDCL-type solvers contain some important ingredients such as variable selection, phase ((also called polarity) selection, restart, BCP (Boolean Constraint Propagation), conflict analysis, clause learning and its database maintenance etc. Changing any ingredient has an impact on the whole performance of solvers. This paper focuses on how to improve the following two ingredients: restart and phase selection. Recently, we proposed two new methods for optimizing a few ingredients. One is called decision-depth-sensitive restart [1], which is used to optimize the restart policy. Another is called bit-encoding phase selection [2], which is used to improve the quality of a polarity selection. Based on the two new policies, we developed a few SAT solvers. Below we describe briefly these new SAT solvers.

II. a Bit-Encoding Phase Selection Policy

In [2], we introduced a new phase selection policy called bit-encoding. The basic idea of this new policy is to let the phase at each decision level correspond to a bit value of the binary representation of an counter. Let n denote the value of a counter, and the binary representation of n be

$$n = b_k 2^k + b_{k-1} 2^{k-1} + \cdots + b_1 2 + b_0.$$

This phase selection policy stipulates that during the m-th search period, the phase of a variable at the k-th decision level is equal to b_k. Every time a restart begins, the counter n increases by one. Based on our experimental observation, it is better to apply bit-encoding scheme on only the first 6 levels. In the detailed implementation, we use only the first 4 bits of the counter n, and let the phase of a variable at the k-th decision level correspond to the (k modulo 4)-th bit of n, where $k < 6$. When $k \geq 6$, we use the phase selection policy of Glucose [3]. Here is the C code of this phase selection.

```
// assume current decision level is k
  if(k < 6 ) polarity[var]=(n >>(k %4))&1;
  else polarity[var]=previous[var];
```

where previous[var] is used to save the previous phase and is initially set to false. It is easy to see that the phase refresh period of our policy is 16, while that of the other existing policies are actually 1. The phase refresh period can be considered as a metric to measure the diversity of a search procedure. If the phase refresh period of a solver is two or more, it is said to be diverse. Otherwise, it is said to be non-diverse or uniform. So far, all the known phase selection policies are uniform, whereas this new phase selection policy is diverse.

III. a Decision-Depth-Sensitive Restart Policy

Here we introduece a new notion called DDD(decision depth decreasing). It is related to the Longest Decreasing Subsequence (LDS). LDS may be defined as follows. Given a sequence S, LDS(S) is the longest decreasing subsequence with the following property: (1) it contains the first term of S; (2) each term is strictly smaller than the one preceding it. For example, assuming S=$\{7,\ 11,\ 10,\ 9,\ 5,\ 6,\ 2\}$, then LDS($S$)=$\{7,\ 5,\ 2\}$. In our solver, S is seen as a sequence of conflict decision levels. The DDD of S is defined as the number of terms in LDS(S), that is, DDD(S)=|LDS(S)|. For the above example, DDD(S)=3. The larger the DDD value is, the closer the goal is likely achieved. However, in many cases, DDD=1. To get the larger DDD, we need to produce many more conflicts. This is harmful to UNSAT instances. Hence, for the restarts that are not postponed by Glucose blocking strategy, we do not apply the DDD blocking strategy. For the restart postponed by Glucose, if DDD<2, even if the restart triggering condition is true, we continue to postpone that restart. That is, our postponing interval is not smaller than that of Glucose 2.1.

Another measure related to our new blocking strategy is the average of maximal depths (denoted by AveMax_D), which may be defined as follows.

$$\text{AveMax\_D} = \frac{1}{8}\sum_{i=2}^{9} \max\{\text{conflict depths in } i\text{-th restart interval}\}$$

the reason why the above formula removes the first maximum is because the first maximum has a greater deviation from its subsequent ones in many instances. In general, on the instances with small AveMax_D, say AveMax_D < 250, we do not apply any blocking strategy. On the instances with large AveMax_D, say AveMax_D > 1500, we remove the DDD blocking strategy.

In addition to the restart triggering condition of Glucose 2.1, we embed such additional conditions as the AveMax_D test and the DDD blocking test. Here is the C++ code of the new restart triggering strategy.

```
K=AveMax_D < 250 && freeVars > 2500 ? 0.82 : 0.8;
assume learnt clause is to c;
sumLBD+= c.lbd(); conflicts++;
queueLBD.push(c.lbd());
if(queueLBD.isFull() && queueLBD.avg()*K > sumLBD/conflicts)
   if(AveMax_D < 250 || AveMax_D > 1500 || !blocked || DDD > 1) {
        queueLBD.clear();
        restart();
  }
```

To remove the postponing strategy on some instances, we add the parameter AveMax_D to our postponing algorithm. Here is C++ code for the new postponing algorithm.

```
R=AveMax_D ≥ 250 && AveMax_D ≤ 900 && conflicts < 1500000 ?
1.38 : 1.4;
if (AveMax_D ≥ 250 && freeVars > 5000 ){
    queueTrail.push(trail.size());
    if(queueLBD.isFull() && queueTrail.isFull() &&
      trail.size() > R*queueTrail.avg()) {
        queueLBD.clear();
        blocked=true;
    }
}
```

IV. System Description of SAT Solvers

Using two new technologies mentioned above, we developed a few new SAT solvers. Below we describe briefly them.

A. glue_bit

glue_bit is built on top of Glucose 2.1, but incorporates two new policies given in previous two sections. It is a sequential single-engine CDCL SAT solver, which runs SatElite as a preprocessor. In glue_bit, the bit-encoding phase selection policy is used to enhance the ability of solving UNSAT instances, whereas the decision-depth-sensitive restart policy is used to enhance the ability of solving SAT instances. For big instances, glue_bit uses still the same as the solving strategies of Glucose 2.1. This solver is submitted to the sequential, application SAT+UNSAT and SAT track of the SAT Competition 2013.

B. minisat_bit{_u}

Minisat_bit{_u} is a hack version of MiniSAT [5]. Except for the pickBranchLit procedure, Minisat_bit is the same as MiniSAT 2.2.0. In the pickBranchLit procedure, Minisat_bit{_u} adds the bit-encoding phase selection policy mentioned above. The phase selection policy of Minisat_bit{_u} is a little bit different from that of glue_bit. In glue_bit, the decision level applying the bit-encoding scheme is limited to 6, while in minisat_bit, the decision level applying the bit-encoding scheme is limited to 12. Furthermore, minisat_bit{_u} has a bit-encoding sub-scheme. Every 4 levels corresponds to a bit-encoding sub-scheme. When the decision level is greater than 12, We use the same phase selection policy as MiniSAT to select a polarity of decision variables. Minisat_bit is submitted to the sequential, MiniSAT hack-track and application of the SAT Competition 2013. Minisat_bit_u is submitted to certified UNSAT track.

C. gluebit_lgl

gluebit_lgl can be regarded as a hybrid solver or an interacting solver using two SAT solving engines. It combines glue_bit and Lingeling 587 [4] that participated in SAT Competition 2011. Its main framework is based on glue_bit. For big instances, this solver switches to glue_bit to solve them. For other instances, glue_bit and Lingeling run specified search steps in turn, and exchanges intermediate results each other. If a solver performs better than another solver, the subsequent solving will be done by that solver with the better performance. This solver is submitted to the sequential, application SAT+UNSAT and SAT track of the SAT Competition 2013.

D. gluebit_clasp

gluebit_clasp is a hybrid solver combining glue_bit and clasp 2.0-R4092 (Gold Non-portfolio in SAT Competition 2011 crafted track). At the initial stage, we use glue_bit to solve an instance. This is similar to the role of a preprocessor. Once glue_bit has found the instance not suitable for it, it aborts to solve and is switched to clasp. In some cases, clasp is switched also to glue_bit. This solver is submitted to sequential, hard-combinatorial SAT+UNSAT and SAT track of the SAT Competition 2013.

V. Conclusion

All the SAT solvers given in this paper are based on two new policies. For glue_bit, we conducted sufficient experiments, while for the other new solvers, we did only a few experiments. The results from the experiment on glue_bit show that the performance of glue_bit was surprisingly good. We believe that the other solvers containing glue_bit will perform well.

References

[1] Chen, J.C.: Decision-Depth-Sensitive Restart Policies for SAT Solvers, submitted for publication, 2013.

[2] Chen, J.C.: A Bit-Encoding Phase Selection Strategy for Satisfiability Solvers, submitted for publication, 2013.

[3] Audemard, G., Simon, L.: Refining Restarts Strategies for SAT and UNSAT, 18th International Conference on Principles and Practice of Constraint Programming (CP'12),pp. 118C-126 (2012)

[4] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010,

[5] Sörensson, N., Biere, A.: Minimizing Learned Clauses, SAT 2009, LNCS 5584, pp. 237–243 (2009)

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

GLUEMINISAT2.2.7

Hidetomo Nabeshima
University of Yamanashi, JAPAN

Koji Iwanuma
University of Yamanashi, JAPAN

Katsumi Inoue
National Institute of Informatics, JAPAN

***Abstract*—GLUEMINISAT 2.2 and the LBD-based evaluation criteria of learned clauses. The new features of the version 2.2.7 are (1) on-the-fly lazy simplification techniques based on binary resolvents, (2) probing-based preprocessing, (3) a new restart strategy based on conflict-generation speed, (4) a variant of blocked restart strategy and (5) a minor modification of the evaluation criteria of learned clauses.**

I. INTRODUCTION

GLUEMINISAT is a SAT solver based on MINISAT 2.2 [1] and the LBD-based evaluation criteria of learned clauses [2]. GLUEMINISAT shows good performance for unsatisfiable SAT instances. The previous version 2.2.5 [3] took the first and second places for UNSAT and SAT+UNSAT classes in CPU time evaluation at SAT 2011 competition, respectively.

To enhance the UNSAT performance, we have introduced some new features to GLUEMINISAT: (1) on-the-fly lazy simplification techniques based on binary resolvents, (2) probing-based preprocessing [4], [5], (2) a new restart strategy based on conflict-generation speed, (4) a variant of blocked restart strategy [6] and (5) a minor modification of the LBD-based evaluation criteria of learned clauses.

II. MAIN TECHNIQUES

Simplification of a given CNF formula is one of important techniques to decide the satisfiability of the formula efficiently. The simplification techniques are used both before and during the search process. GLUEMINISAT has the both simplification techniques. For preprocessing, we have implemented probing-based techniques which consist of false-literal probing, necessary assignment probing, equivalent variable probing [4] and binary clause probing [5], besides variable and subsumption elimination [7] which are implemented in MINISAT 2.2.

For in-processing, GLUEMINISAT executes the above probing techniques on-the-fly. To reduce the checking cost, we utilize binary resolvents extracted from unit propagation process. For example, let $\phi = \{x \to y, x \to z, y \wedge z \to v, v \wedge w \to u\}$ and w is assigned as true. If x is selected as a decision variable and assigned as true, then y, z, v, u are propagated. The cause of the propagation of y, z, v is x. This means $\phi \models (x \to y) \wedge (x \to z) \wedge (x \to v)$. However, u is not propagated from x only. It requires x and w as premise literals. The checking of whether a propagated literal has a single cause or not can be done with a constant order at the unit propagation process. Hence, we can extract a large number of binary resolvents with very low overhead. This extraction approach is similar to the dominator detection algorithm in [8]. Our algorithm detects the earliest dominator (decision literal), whereas [8] uses immediate dominators. The earliest dominator can be detected with O(1), whereas the computation of the immediate dominator sometimes requires linear search between two nodes in a implication graph.

For each literal, GLUEMINISAT holds only *one* of premise literals. We prepare an array named *premise*. Each entry of the array is indexed by each literal. The value of *premise*$[x]$ is a literal which denotes one of premise literals of x, that is, $\phi \models$ *premise*$[x] \to x$. Initially, *premise*$[x] = x$. The value of *premise*$[x]$ is updated when x is propagated and x has a single cause of the propagation.

We can execute probing techniques with a constant order by using the array *premise*. For example, the necessary assignment probing can be represented as follows: suppose that ϕ is a formula and x, y are literals. If $\phi \models x \to y$ and $\phi \models \neg x \to y$, then $\phi \models y$. This probing technique requires two premise literals of y. We can get two premise literals of y, that is, the old value of *premise*$[y]$ before updating of it and the new value of it. We denote the old and new values as *oldpremise*$_y$ and *newpremise*$_y$, respectively. Then, we can execute the necessary assignment probing as follows: if *oldpremise*$_y = \neg$*newpremise*$_y$, then $\phi \models y$ holds. The checking cost is $O(1)$. Other probing techniques can be executed in the same way. GLUEMINISAT executes these on-the-fly probing techniques when an entry of the array *premise* is changed. The array *premise* represents a set of binary resolvents. These binary resolvents are also used to shrink clauses by self-subsumption checking.

We hold only one premise literal for each literal. However, the value of *premise*$[y]$ often changes since CDCL solver execute unit propagations very frequently. This variation of premise literals contributes the realization of effective and low cost simplification techniques.

III. OTHER TECHNIQUES

GLUEMINISAT uses an aggressive restart strategy. If one of the following conditions is satisfied, then the solver restarts:

1) an average of *LBDs* over the last 50 conflicts is greater than the global average $\times$ 0.8.
2) an average of the number of decisions per a conflict from the last restart is greater than the global average $\times$ 0.95.

The former condition is same as GLUEMINISAT 2.2.5 and GLUCOSE 2.1. The latter one is a new condition which intends to generate conflicts quickly. The parameters 0.8 and 0.95 were

TABLE I
THE NUMBER OF SOLVED INSTANCES

Solver	#Solved (SAT + UNSAT)
GLUEMINISAT 2.2.5	199 (81 + 118)
GLUEMINISAT 2.2.7	**220** (**94** + **126**)
GLUCOSE 2.1	216 (**94** + 122)

determined by experiments on benchmark instances of past SAT competitions.

Even if either above restart condition is satisfied, the restart is blocked when the local trail size per a conflict is exceedingly greater than the global one [6]. This strategy helps to catch a chance of making satisfying assignment. In GLUEMINISAT, when an average of the number of propagated literals per a conflict from the last restart is greater than the global average $\times$ 2.0, the restart is blocked.

The literal blocks distance (LBD) [2] is an evaluation criteria to predict learnt clauses quality in CDCL solvers. The effectiveness of LBD was shown at past competitions by GLUCOSE and GLUEMINISAT. The LBD value of a clause is computed when the learned clause is produced from a conflict, and re-computed when the clause is used for unit propagations. In the re-computation, GLUEMINISAT 2.2.7 ignores literals whose values are fixed at the decision level 0. As the results, the LBD values may become less than the original ones. In 2.2.7, we never remove learned clauses whose updated LBD value is *one*, that is, a learned clause is never removed when every literal of the clause are assigned at the same level once.

IV. EXPERIMENTAL RESULTS

We evaluated 3 solvers for 300 instances in the application category of SAT 2011 competition. The solvers are GLUEMINISAT 2.2.5, 2.2.7 and GLUCOSE 2.1 which took the first place as a sequential solver at SAT Challenge 2012. The experiments were conducted on a Core i7 (2GHz) with 8GB memory. We set a timeout for each instance to 5000 CPU seconds. Table I is the experimental results and Fig 1 is cactus plots of the results. For SAT instances, the performance of GLUEMINISAT 2.2.7 is greatly improved from 2.2.5, and it is almost same as GLUCOSE 2.1. For UNSAT instances, GLUEMINISAT 2.2.5 solves more number of instances than 2.2.5 and GLUCOSE 2.1.

V. SAT COMPETITION 2013 SPECIFICS

GLUEMINISAT uses the option `-compe` for the competition. This option suppresses log messages. For certified UNSAT tracks, some techniques in GLUEMINISATare disabled because of the implementation issue of RUP output. The execution script for certified UNSAT tracks is `binary/glueminisat-cert-unsat.sh`, in which binary self-subsumption checking based on the `premise` array is disabled.

VI. AVAILABILITY

GLUEMINISATis developed based on MINISAT 2.2. Permissions and copyrights of GLUEMINISAT are exactly the same as MINISAT. GLUEMINISAT can be downloaded at http://glueminisat.nabelab.org/.

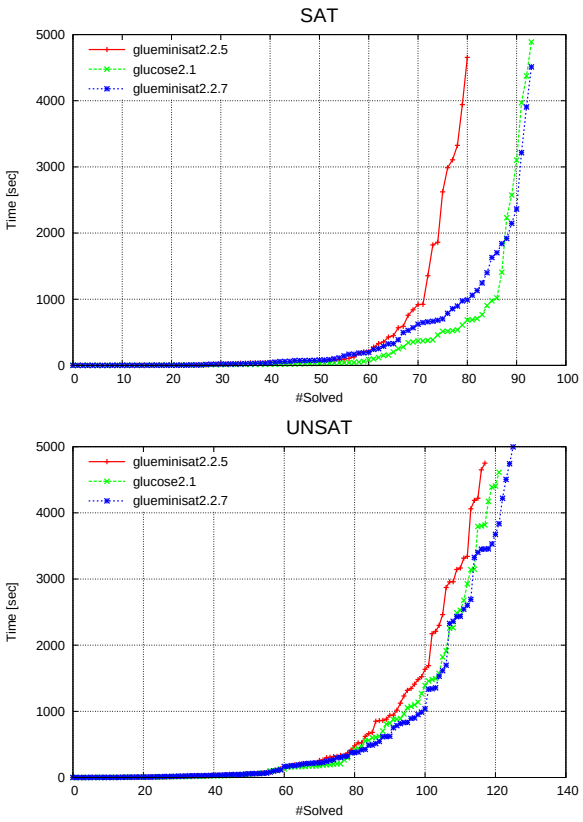

Fig. 1. A cactus plot for application category of SAT 2011 competition

ACKNOWLEDGMENT

This research is supported in part by Grant-in-Aid for Scientific Research (No. 24300007) from Japan Society for the Promotion of Science and by Artificial Intelligence Research Promotion Foundation.

REFERENCES

[1] N. Eén and N. Sörensson, "An extensible sat-solver," in *Proceedings of SAT-2003*, 2003, pp. 502–518.
[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of IJCAI-2009*, 2009, pp. 399–404.
[3] H. Nabeshima, K. Iwanuma, and K. Inoue, "GLUEMINISAT 2.2.5," 2011, SAT Competition 2011 Solver Description.
[4] D. L. Berre, "Exploiting the real power of unit propagation lookahead," *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 59–80, 2001.
[5] I. Lynce and J. P. M. Silva, "Probing-based preprocessing techniques for propositional satisfiability," in *ICTAI*. IEEE Computer Society, 2003, pp. 105–110.
[6] G. Audemard and L. Simon, "Refining restarts strategies for sat and unsat," in *CP*, ser. Lecture Notes in Computer Science, M. Milano, Ed., vol. 7514. Springer, 2012, pp. 118–126.
[7] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *SAT*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.
[8] H. Han, H. Jin, and F. Somenzi, "Clause simplification through dominator analysis," in *DATE*. IEEE, 2011, pp. 143–148.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

gluH: Modified Version of gluclose 2.1

Chanseok Oh
New York University
New York, NY, USA

***Abstract*—This document describes the SAT solver "gluH" submitted to SAT Competition 2013, a modification of glucose 2.1, whose main feature is an addition of the generational management of learnt clauses.**

I. Introduction

gluH is a modified version of glucose [1], which is in turn based on the well-known MiniSat [2], a typical CDCL solver renowned for its compactness yet decent performance. The main addition of gluH is its generational management of learnt clauses, which periodically relegates dormant learnt clauses to the second class, separating them from first-class learnts recently created or actively participating in unit propagation. The aim is to reduce the size of the database of the first-class learnts to facilitate rapid propagations, and the idea is based on the observation that often, clauses that have not been involved at all in unit propagation for a long time have a fair chance that they will be rarely useful for a while, if at all.

II. Main Techniques

The major difference between gluH and glucose is the addition of the generational management of learnt clauses, despite the implementation and its evaluation being still in a primitive stage.

Aside from the normal learnt clause management, clauses inactive for a long period of time are further classified as dormants and demoted into the second class where they receive less attention from propagation. Note that only dormant learnts can ever be demoted; gluH still discards learnts immediately and permanently during the regular reduction.

Basically, clauses classified as dormants are not always inspected for possible unit propagation or conflict but on a random basis. However, if they become involved in propagation or conflict, they are promoted back to the first class promptly.

As one can expect, the database of dormant learnts will grow over time, and gluH uses the exact same logic to reduce its size, i.e., based mainly on LBD [1] values.

III. Main Parameters

The ratio of discarding learnts when performing database reduction in original glucose, as is the case with MiniSat, is roughly half. gluH reduces this ratio to prevent removing too many useful clauses, since a considerable portion of learnts will be relegated and excluded as dormants. For the dormant learnt database, the ratio is half.

The criterion for classification of a dormant clause is whether it has ever been involved in propagation or conflict within a fixed number of last conflicts; the number is static throughout the entire execution.

IV. Implementation Details

Separate watcher lists are maintained for dormant learnts. The normal watcher lists are accessed whenever a variable is assigned, as usual, whereas the watcher lists for dormants are accessed with a relatively small and fixed probability.

The task of classifying dormants is carried out periodically, in synchronization with the regular database reduction. For the purpose of classifying dormants, each learnt clause is assigned a timestamp when created, which will be updated if the clause is involved in unit propagation or conflict; the timestamp is the accumulated number of conflicts so far.

If a clause is deemed to be dormant according to the above-mentioned criterion, it is detached from the normal watcher lists and added to the dormant lists.

V. SAT Competition 2013 Specifics

Two instances with different configurations have been submitted to SAT Competition 2013.

1) Using SatELite [3] as a front-end CNF preprocessor, ratio of normal learnt reduction: 1/4, probability of checking dormant watcher lists: 0.25, dormant learnt criterion: silent for 20000 conflicts
2) Based on "simp" version of MiniSat, ratio of normal learnt reduction: 1/4, probability of checking dormant watcher lists: 0.20, dormant learnt criterion: silent for 30000 conflicts

VI. Availability

gluH adds no additional license to that of glucose.

References

[1] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, C. Boutilier, Ed., 2009, pp. 399–404.
[2] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.
[3] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *SAT*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

gNovelty$^+$GC: Weight-Enhanced Diversification on Stochastic Local Search for SAT

Thach-Thao Duong*†, Duc-Nghia Pham*
*Institute for Integrated and Intelligent Systems, Griffith University, QLD, Australia
†Queensland Research Laboratory, NICTA
Email: {t.duong,d.pham}@griffith.edu.au

I. Introduction

Clause Weighting is an efficient scheme for stochastic local search (SLS) for Satisfiability (SAT) [3], [4], [5]. A clause weight records no-good information about how often that clause has been unsatisfied at local minima. Similarly, a variable weight stores no-good information about how often a variable has been flipped. That variable property is reported to improve SLS diversification. This work is motivated by the desire to investigate the use of weighting-based enhancement in the diversification phases. Firstly, we use clause weights to select clauses in the trap escaping phases. Secondly, we use variable weights to break ties in the scoring function.

II. Weight-Enhanced Diversification

To our best knowledge, clause-weghting SLS used clause weights in computing scoring function for selecting variables in greedy phases. Most of SLS solvers selects false clauses randomly at local minima escaping phases. Our algorithm stems from the idea of taking advantages of clause-weighting scheme at clause selection procedures [6]. Our algorithm performs a greedy selection on clauses instead of random selection as the conventional trap escaping strategy. The greedy selection is controlled by a probability noise named β. Within probability noise β, the maximum weighted clause is selected; otherwise a random unsatisfied clause is selected. At greedy phases, we employ variable weights as tie-breaking criteria.

A. Conventional clause-weighting score

A popular non-weighted scoring function is based on the simple count of currently unsatisfied clauses.

$$\mathrm{score}(v) = \sum_c (\mathrm{Cls}(c) - \mathrm{Cls}^v(c))$$

where $\mathrm{Cls}(c)$ indicates whether a clause c is unsatisfied or not under the current assignment. In order words, $\mathrm{Cls}(c) = 1$ if c is currently unsatisfied, otherwise $\mathrm{Cls}(c) = 0$. Similarly, $\mathrm{Cls}^v(c)$ shows whether clause c remains unsatisfied if variable v is flipped. Clearly, the score of flipping a variable v is defined as the decrease in the number of unsatisfied clauses after v is flipped.

Under a weighted scheme, a scoring function is defined as

$$\mathrm{wscore}(v) = \sum_c \mathrm{Wgh}(c) \times (\mathrm{Cls}(c) - \mathrm{Cls}^v(c))$$

where $\mathrm{Wgh}(c)$ is the weight of a clause c.

Algorithm 1: NoveltyGC(β, p)

```
1  if within probability β then
2      c = randomly select an unsatisfied clause;
3  else c = select an unsatisfied clause with the highest clause weight Wgh(c);
4  for all variables in clause c do
5      find the best and second best variables based on wscore, breaking ties
       based on a diversification criterion;
6  end
7  if within probability p then
8      return the second best variable;
9  else return the best variable;
```

Algorithm 2: gNovelty$^+$GC(Θ, β, sp)

```
   Input  : Formula Θ, diversification probability β, smoothing probability sp
   Output: Solution α (if found) or TIMEOUT
1  randomly generate a candidate solution α;
2  initiate all clause weights Wgh(c) = 1;
3  while not timetout do
4      if α satisfies the formula Θ then return α ;
5      if within the random walk probability wp = 0.01 then
6          perform a Random Walk;
7      else
8          if there exits promising variables then
9              select a variable v with the highest wscore(v), breaking ties
               based on a diversification criterion;
10         else
11             perform NoveltyGC(β, p) and adjust its noise p;
12             update and smooth (with the probability sp) clause weights;
13         end
14     end
15     update the candidate solution α;
16 end
17 return TIMEOUT;
```

B. Clause-Weighting Enhancement

In the diversification phases, our new heuristic greedily selects an unsatisfied clause based on the clause weights. With probability β, it selects an unsatisfied clauses randomly. Otherwise, with probability (1-β), it selects an unsatisfied clause with the maximum clause weight. The aim of selecting a clause with the highest weight is to make an attempt to satisfy the clause that is most often unsatisfied at local minima. Satisfying such a clause may help the search escape from the current trap. The positive aspect of using the diversification noise β is the flexibility in switching back and forth between greediness and randomness. This allows the solver to occasionally move away from being too greedy.

The new NoveltyGC is outlined in Algorithm 1. Once the false clause c is selected, the algorithm will compute the $\mathrm{wscore}(v)$ to find the best and second best variables w.r.t. the

score. In the next section, we will describe other diversification criteria to break such ties (Line 5 in Algorithm 1).

C. Variable-Weighting Enhancement

Novelty uses variable ages as its diversification criterion [1]. A variable age is computed as the number of flips since that variable was last flipped. In other words, the older the variable, the more likely it will be selected. Many SLS solvers use variable ages to break ties in their diversification phase. In this study, we use variable weights as an alternative diversification criterion. In this case, ties are broken by selecting the variable v with less weight, which records the number of times v has been flipped. By doing so, the algorithm prefers to direct the search to the area that involves the least frequently flipped variables. We also investigate the use of both variable weights and variable ages in a combined fashion: break ties by selecting the variable with smaller variable weight, then break further ties by preferring the older variables.

III. Implementation

The new weight-enhanced heuristics is integrated into gNovelty$^+$. The new algorithm gNovelty$^+$GC (where GC stands for 'greedy clause selection') is illustrated in Algorithm 2. It will break ties in the intensification phases using the same diversification criterion as NoveltyGC does in the diversification phases. There are two different diversification criteria: variable ages and the combination of variable ages and variable weights. In order to distinguish the two variants of gNovelty$^+$GC, we name them respectively as gNovelty$^+$GC$_a$, and gNovelty$^+$GC$_{wa}$ based on the diversification criterion being used.

For SAT 2013 Competition, parameter settings are presented in the following table.

Solvers	β	sp
gNovelty$^+$GC$_v$	15	40
gNovelty$^+$GC$_{wa}$	0	40

References

[1] Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02). pp. 635660 (2002)

[2] Pham, D.N., Thornton, J., Gretton, C., Sattar, A.: Combining adaptive and dynamic local search for satisfiability. JSAT 4(2-4), 149172 (2008)

[3] Pham, D.N., Duong, T.T., Sattar, A.: Trap avoidance in local search using pseudo-conflict learning. AAAI'12 Proceedings, 542-548, (2012)

[4] Duong, T.T. , Pham, D.N., Sattar, A.: A study of local minimum avoidance heuristics for SAT. In *ECAI*, pages 300–305, 2012.

[5] Duong, T.T. , Pham, D.N., Sattar, A.: A Method to Avoid Duplicative Flipping in Local Search for SAT. In *AI*, pages 218-229, 2012.

[6] Duong, T.T. , Pham, D.N., Sattar, A.: Weight-Enhanced Diversification in Stochastic Local Search for Satisfiability. To appear in IJCAI'13 Proceedings, 2013

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University Linz

***Abstract*—This paper serves as solver description for our SAT solver Lingeling and its two parallel variants Treengeling and Plingeling entering the SAT Competition 2013. We only list important differences to the version of these solvers used in the SAT Challenge 2012. For further information we refer to the solver description [1] of the SAT Challenge 2012 or source code.**

LINGELING

The differences on the search side, that is during the CDCL loop, are as follows. The inner-outer scheme for reduce scheduling is not enabled by default, but only enabled dynamically, if the number of remaining variables drops below 1000. It also turned out that the previous VMTF decision scheduler, though faster to compute, is less robust, and occasionally leads to time-outs on otherwise easy to satisfy instances. Thus we went back to the exponential VSIDS scheme of MiniSAT. Costly recursive clause minimization [2] is only attempted for clauses with small glucose level (LBD) [3]. Local clause minimization is tried for somewhat higher glucose levels. For even higher glucose levels and if in addition the 1^{st}-UIP clause is rather long then a decision-only clause is learned instead of the (minimized) 1^{st}-UIP clause (proposed by Donald Knuth in private communication). The decision-only clause contains the negations of all decisions required to generate the conflict, except for the last decision, which is replaced by the 1^{st}-UIP literal (if different). The variable scores are updated based on the generated but discarded 1^{st}-UIP clause.

Lingeling uses various *inprocessing* algorithms [4], which not only simplify the formula initially and in this case act as *preprocessors*, but also in regular intervals between calls to the CDCL search loop. All inprocessors are running as part of one single simplification phase. The number of conflicts is used as metric to measure the effort spent in search and if a conflict limit is reached, the solver switches to simplification.

In principle, the conflict interval for simplification is increased geometrically. However, depending on the amount of reduction achieved in the last simplification phase, measured in terms of the percentage of removed variables, the increment of the simplification interval is reduced. The more variables are removed the sooner the next simplification phase is scheduled. Furthermore, each inprocessing algorithm monitors its effectiveness individually. If an inprocessor was unsuccessful in the previous simplification phase, the inprocessor is skipped in the next simplification phase. If unsuccessful again, it is skipped twice etc. Various more advanced inprocessors are delayed until either blocked clause elimination or bounded variable elimination is completed at least once.

Supported by FWF, NFN Grant S11408-N23 (RiSE).

Regarding changes in individual inprocessors we note the following. Tree-based look-ahead already mentioned in [1] has been published [5]. Bounded variable elimination is now much more restricted. It only tries to eliminate variables with few occurrences and thus terminates resp. runs to completion much earlier than in previous versions.

We further realized that during literal probing, which occurs as part of various preprocessors, clauses satisfied during probing which contain the negation of the probed literal are under certain restrictions asymmetric tautologies [6] and can be removed. Actually, in general, short (binary or ternary) clauses determined to be redundant, such as these basic asymmetric tautologies, but also blocked clauses or covered clauses, are "moved", i.e. they are marked as redundant resp. learned clauses to preserve BCP as discussed in [4].

As also discussed in [4] we generate and add binary blocked clauses. To avoid full occurrence lists for large redundant resp. learned clauses, we only add blocked clauses with a blocking literal, which does not occur negated in large redundant clauses at all. Blocked clause addition is disabled in the parallel solvers Plingeling and Treengeling.

Finally, we added a simple form of cardinality constraint reasoning, which is similar to our previously added Gaussian elimination procedure [1]. We first extract trivially encoded at-most-one and at-most-two constraints by a simple and incomplete syntactic procedure. All clauses, which contain a literal occurring in an extracted cardinality constraint are added as cardinality constraint too, e.g. as at-most-k constraint, where $l = k + 1$ is the length of the clause. For this set of cardinality constraints we perform a simple form of variable elimination, as in the Fourier-Motzkin elimination procedure. This technique allows to derive an inconsistent constraint for large pigeon-hole formulas in a fraction of a second. Otherwise the procedure exports derived units and binary clauses.

For the certified UNSAT track we had to disable blocked clause addition, Gaussian elimination and cardinality constraint reasoning, since they can not be simulated by resolution (polynomially). Furthermore, equivalent literal substitution (ELS) turned out to be hard to map to (D)RUP and thus we had to disable all inprocessors which rely on ELS.

PLINGELING

The first version of Plingeling only shared units, while more recent versions also share equivalences. In this new version we also share short clauses with small glucose level, i.e. clauses with at most 40 literals and glucose level of at most 8. In contrast to [7] all exported clauses are imported unless they contain a "melted" literal, such as those eliminated or used as blocking literal in blocked or covered clause elimination during inprocessing. The same restriction was already required for importing equivalences.

Clauses are exported to the master and copied to a global stack. Each slave solver thread imports clauses from this global stack in regular intervals during the CDCL loop, oldest clauses first. Thus this global clause stack actually acts as a queue. The procedure for importing clauses triggers garbage collection of global clauses already imported (consumed) by all solvers in regular intervals.

TREENGELING

Treengeling is a parallel solver based on Cube & Conquer [8], [9], which tries to combine the strengths of look-ahead solving with CDCL solving and in the case of Treengeling also with inprocessing. The basic architecture of Treengeling was already described in [1].

In this new version we essentially added three improvements. First, and most important, we flipped the policy for changing the conflict limit for each search node in order to match the original motivation for Cube & Conquer. If more nodes are closed by a combination of CDCL and inprocessing we double the global conflict limit. Otherwise if the number of closed nodes is smaller than the number of added nodes then the conflict limit is decreased with a rate of 90%. This actually only happens if at least one node was closed and otherwise, if no node was closed, the conflict limit is even decreased by 50%. If the soft memory limit is hit nodes are not split and we end up in the first case (doubling the limit). Further there is a minimum conflict limit of 1,000 conflicts, an initial conflict limit of 10,000 and a maximum conflict limit of 100,000 conflicts. This limit is applied to the search phase of one node in one round (which also includes inprocessing).

The second improvement consists of disabling full tree-based look-ahead [5] if look-ahead alone removes less than 2% of the remaining variables. There is a similar penalty scheme as for inprocessors in Lingeling though. In this situation the next full look-ahead will be skipped and only a cheap to compute static heuristics is used instead.

Finally, Treengeling combines part of the infrastructure of Plingeling with Cube & Conquer by using one core for running an additional solver thread, which exports units to the worker threads in Cube & Conquer. Thus on an 8 core machine, 7 cores are allocated to Cube & Conquer worker threads, which work on 8 times more, thus 56 active nodes. On our 12 core machine with hyper threading, thus 24 virtual cores, the solver will use at most $184 = 8 * 23$ active nodes in parallel.

LICENSE

For the competition version of our solvers we use a new license scheme. It only allows the use of the software for academic and research purposes and further prohibits the use of the software in other competitions or similar events without explicit written permission. Please refer to the actual license, which comes with the source code, for more details.

REFERENCES

[1] A. Biere, "Lingeling and friends entering the SAT Challenge 2012," in *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, University of Helsinki, vol. B-2012-2, 2012, pp. 33–34.

[2] N. Sörensson and A. Biere, "Minimizing learned clauses," in *Proc. SAT'09*. Springer, 2009, pp. 237–243.

[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. IJCAI'09*. Morgan Kaufmann, 2009, pp. 399–404.

[4] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *Proc. 6th Intl. Joint Conf. on Automated Reasoning (IJCAR'12)*, ser. LNCS, vol. 7364. Springer, 2012, pp. 355–370.

[5] M. Heule, M. Järvisalo, and A. Biere, "Revisiting hyper binary resolution," in *Proc. 10th Intl. Conf. on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'13)*, ser. LNCS, vol. 7874. Springer, 2013, pp. 77–93.

[6] M. J. H. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for CNF formulas," in *Proc. LPAR-17*, ser. LNCS, vol. 6397. Springer, 2010, pp. 357–371.

[7] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette, "Revisiting clause exchange in parallel SAT solving," in *Proc. SAT'12*, ser. LNCS, vol. 7317. Springer, 2012, pp. 200–213.

[8] P. van der Tak, M. Heule, and A. Biere, "Concurrent cube-and-concur," in *Proc. 3rd Intl. Work. on Pragmatics of SAT (POS'12)*, 2012.

[9] M. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Proc. HVC'11*, ser. LNCS, vol. 7261. Springer, 2012, pp. 50–65.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

march_br

Marijn J. H. Heule

The University of Texas at Austin

INTRODUCTION

The march_br SAT solver is the latest version of the successful march solvers: march_hi, march_ks, march_dl, march_eq, and march_rw won several awards at the SAT 2004, 2005, 2007, 2009, and 2011 competitions. For the detailed descriptions, we refer to [1], [2], [3], [4]. Like its predecessors, march_br 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 change in march_br is the possibility to emit a refutation.

PROOF LOGGING

Unsatisfiability proofs of march_br are in the *branch reverse unit propagation format* (BRUP). The BRUP format was developed to express refutations of lookahead SAT solvers efficiently. The format contains three types of lines: 1) new branch decisions, 2) locally learned unit clauses (mostly failed literals) and binary clauses (mostly hyper binary resolvents), and 3) termination of a branch. A new branch is a line with a `b` prefix followed by the branch literal. Locally learned clauses are expressed in the DIMACS format (a list of integers ending with a zero). A terminated branch is expressed as a line with only a zero. Proofs in the BRUP format can easily be converted into DRUP (delete reverse unit propagation) proofs.

PARTIAL LOOKAHEAD

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

Algorithm 1 PARTIALLOOKAHEAD( )

```
1:  Let F' and F'' be two copies of F
2:  for each variable x_i in P do
3:      F' := ITERATIVEUNITPROPAGATION(F ∪ {x_i})
4:      F'' := ITERATIVEUNITPROPAGATION(F ∪ {¬x_i})
5:      if empty clause ∈ F' and empty clause ∈ F'' then
6:          return "unsatisfiable"
7:      else if empty clause ∈ F' then
8:          F := F''
9:      else if empty clause ∈ F'' then
10:         F := F'
11:     else
12:         H(x_i) = 1024 × DIFF(F, F') × DIFF(F, F'') + DIFF(F, F') +
            DIFF(F, F'')
13:     end if
14: end for
15: return x_i with greatest H(x_i) to branch on
```

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, various 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] Sid Mijnders and Boris de Wilde and Marijn J. H. Heule. Symbiosis of Search and Heuristics for Random 3-SAT. In David Mitchell and Eugenia Ternovska (Eds.), Proceedings of the Third International Workshop on Logic and Search (LaSh 2010), 2010.

[2] 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.

[3] 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.

[4] 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.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Solvers Combining Complete and Incomplete Solving Engines

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

***Abstract*—In general, SAT solvers based on a DPLL procedure are a complete, systematic depth-first search engine, and superior on application and crafted instances, while SAT solvers based on stochastic local search (SLS) process are an incomplete search engine, and superior on satisfiable random instances. Here we combines complete and incomplete search engines to build a few new SAT solvers. These new solvers are christened clasp_vflip, march_vflip and interact_open. This paper introduces briefly them.**

I. Introduction

Modern SAT solvers can be divided into two categories: complete and incomplete. Complete solvers can solve not only unsatisfiable problems, but also satisfiable problems, whereas incomplete solvers can solve only satisfiable problems, but not solve any unsatisfiable problem. Complete solvers can be divided further into Conflict Driven Clause Learning (CDCL) and look-ahead. Among the most typical CDCL solvers are Glucose [1], Lingeling [2], CryptoMiniSat [3] and clasp [4] etc. Except for clasp, these solvers are good at application instances. Clasp is very strong on combinatorial instances. Among the most typical look-ahead solvers are kcnfs, OKsolver and March [5] etc. This type of solvers is strong on unsatisfiable random and combinatorial instances. In general, most of incomplete solvers are based on stochastic local search (SLS) process. Among the most typical SLS solvers are sparrow [6], CCASat [7] and Sattime [8] etc. This type of solvers is strong on satisfiable random instances. We make a better use of the feature of various solvers to develop a number of new solvers. This paper introduces briefly the new solvers.

II. System Description of SAT Solvers

A. clasp_vflip

clasp_vflip is a hybrid solver combining clasp and vflipnum [9]. This solver is submitted to the sequential, hard-combinatorial SAT track and SAT+UNSAT track of the SAT Competition 2013. vflipnum is a new SLS solver that is built on top of Sparrow [6]. Some combinatorial instances are suited for a SLS solver. So we use vflipnum to solve a part of combinatorial satisfiable instances. The characteristic of clasp_vflip is that at the first stage clasp and vflipnum run specified search steps in turn. At the second stage, we invoke mainly clasp, but run vflipnum using very short time after a longer time interval. In clasp_vflip, the clasp version used here is 2.0-R4092. But we made a slight modification on clasp as follows. The value of parameter rlimit.maxConf used in clasp_vflip is half that of the original clasp. The other parameters are the same as clasp.

B. march_vflip

march_vflip is a hybrid solver combining March [5] and vflipnum [9]. March is good at unsatisfiable random instances. So we hope that March is dedicated to solving unsatisfiable random instances. However, the problem is how to predict whether an instance is unsatisfiable random instances or not in advance. To do this, march_vflip is to determine the property of an instance by running March a few steps and computing the search depth in it. If the search depth in March is less than 18, we continue to run March until a solution is found. Otherwise, we switch to vflipnum and let vflipnum solve that instance. This solver is submitted to the sequential, random SAT+UNSAT track of the SAT Competition 2013.

C. interact_open

interact_open is a sequential interacting solver that integrates many solving engines. It inherits many properties of interactSAT [10] that participated in SAT Challenge 2012. It is built on top of an interacting framework. The interacting framework used here is similar to that used in interactSAT, which may be described briefly as follows.

(1) Run simultaneously m solving engines. Each solving engine runs at most n conflicts to solve the instance at a time. In general, the value of n is not necessarily fixed, but in most cases initially $n = 10000$.
(2) Pass the intermediate solution of the i-th engine to the k-th ($k = (i+1) \bmod m$) engine (i.e., the next engine), where $1 \leq i \leq m$.
(3) If a solution has been found, the solving process terminates.
(4) Modify the maximum number n of conflicts for a solving engine to run. If a solving engine is better than the other solving engines, the corresponding n increases. Otherwise, the corresponding n decreases. Repeat steps (1)–(4).

In total, we here use the follow 6 solving engines: CryptoMiniSat [3], Lingeling [2], clasp [4], glue_bit, March and vflipnum. glue_bit is a solver we developed recently, which is built on top of Glucose, and use the two new policies: a

bit-encoding phase selection policy [11] and a decision-depth-densitive restart policy [12]. In the real interacting process, not all the 6 solving engines run always. Some solving engines can be regarded as a preprocessor, for example, CryptoMiniSat, March and vflipnum etc. These preprocessors play the role of a filter. That is, they solve only the instances suited for them, and give up the instances not suited for them. At the interacting phase, We have two interacting modes. The first mode uses 3 solving engines: Lingeling and two variants of glue_bit. This mode is used mainly to solve application instances. The second mode uses 2 solving engines: clasp and glue_bit. It is used mainly to solve combinatorial instances. Using a filtering technique, we let March and vflipnum solve random instances. This solver is submitted to open track of the SAT Competition 2013.

III. Conclusion

Here we presented three solvers, each containing an incomplete SLS engine, although they all are a complete, systematic search engine. Combining various solving engines is only our first attempt. Without any portfolio technique, how to combine a number of the solving engines that are good at a type of instances into a solver that are good at all the types of instances will be a challenging problem, since there are too many combination ways.

References

[1] Audemard, G., Simon, L.: Refining Restarts Strategies for SAT and UNSAT, 18th International Conference on Principles and Practice of Constraint Programming (CP'12),pp. 118C-126 (2012)

[2] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010,

[3] Soos, M.: CryptoMiniSat 2.5.0, http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_13.pdf

[4] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver, Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR07), LNAI 4483, 260 – 265 (2007)

[5] Heule, M. : March: towards a look-ahead SAT solver for general purposes, Master thesis, 2004.

[6] Tompkins, D. A. D., Hoos, H. H. : Dynamic Scoring Functions with Variable Expressions: New SLS Methods for Solving SAT, SAT 2010, 278–292 (2010)

[7] Cai, S.W., Luo, C., Su, K.L.: CCASat: Solver Description, Proceedings of SAT Challenge 2012, pp. 13 – 14, 2012.

[8] Li, C.M., Li, Y.: Description of Sattime2012, Proceedings of SAT Challenge 2012, p. 53, 2012.

[9] Chen, J.C., Huang, Y.Y: vflipnum: A Local Search with Variable Flipping Frequency Heuristics for SAT, SAT Competition 2013.

[10] Chen, J.C.: interactSAT{_c}: Interactive SAT Solvers and glue_dyphase: A Solver with a Dynamic Phase Selection Strategy, Proceedings of SAT Challenge 2012, pp. 28–30, 2012.

[11] Chen, J.C.: A Bit-Encoding Phase Selection Strategy for Satisfiability Solvers, submitted for publication, 2013.

[12] Chen, J.C.: Decision-Depth-Sensitive Restart Policies for SAT Solvers, submitted for publication, 2013.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

MiniGolf

Norbert Manthey
Knowledge Representation and Reasoning,
TU Dresden, Germany

***Abstract*—MINIGOLF is a small modification of Minisat to learn all unit clauses per conflict – if the first UIP clause would be is a unit clause.**

I. INTRODUCTION

The SAT solver MINIGOLF is a modification of the CDCL SAT solver MINISAT 2.2 [1] (SAT Competition Patch), with a slightly changed clause learning procedure.

II. MAIN TECHNIQUES

The main difference in MINIGOLF is a modification in clause learning, called *all-units-learning*: if a unit clause is learned, the procedure tests whether by continuing resolution more unit clauses can be learned as well. The process is stopped as soon as a larger (intermediate) clause would be learned. If the initially learned clause is not a unit, the solver proceeds with its usual minimization routine.

The modification is motivated by the preprocessing technique *probing* [2], where usually a decision l is made, and in case of a conflict the unit clause $\bar{l}$ (called *last UIP*) is added to the formula. However, learning a first UIP clause [3] might be more beneficial. It is interesting to see that there exists cases where neither the last UIP implies the first UIP and vice versa (by unit propagation). Therefore, adding both clauses is a first extension. Since there exists the chance of intermediate UIPs as well, all UIPs on level 1 should be learned during preprocessing.

In the search of a SAT solver, unit clauses are also learned at other levels. Still, it could be the case that there is a chain of implications, which led to this learned clause, and which contains further UIPs that correspond to a learned unit clause. As it might be beneficial for preprocessing, these additional unit clauses could also be helpful for the search process. Therefore, we collect all unit learned clauses per conflict.

III. MAIN PARAMETERS

The learning modification all-units-learning is always active.

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

To find learned unit clauses, a flag is added which indicates whether the current learned clause, or any previously learned clause generated from the current conflict is still a unit clause. The abort criterion of the clause resolution method in the analyze function is modified, so that it continues learning until a non-unit clause is learned, or until all literals of the current level are resolved.

V. IMPLEMENTATION DETAILS

The modifications are implemented directly into the source code of MINISAT – however, to meet the criterion to enter the hack track, tricks of the C++language have been used to reach a diff size of 1000 non-space symbols.

VI. SAT COMPETITION 2013 SPECIFICS

This solver was submitted to the Minisat Hack Track only. The version *prefetch* prefetches the watch list of a literal that has been enqueued to the propagation queue in this moment. All other details of this version are unchanged.

VII. AVAILABILITY

MINIGOLF will be provided at tools.computational-logic.org as source code under the GPL licence.

REFERENCES

[1] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[2] I. Lynce and J. Marques-Silva, "Probing-Based Preprocessing Techniques for Propositional Satisfiability," in *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ser. ICTAI '03. IEEE Computer Society, 2003, pp. 105–110. [Online]. Available: http://portal.acm.org/citation.cfm?id=951951.952290

[3] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 530–535. [Online]. Available: http://doi.acm.org/10.1145/378239.379017

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

MINIPURE

Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan ,Hsiao-Lun Wang
B98901096@ntu.edu.tw

***Abstract*—This document is a description of the sat solver — minipure of SAT competition 2013. The implementation embedded the pure literal detection in MINISAT which is a well known conflict based solver.**

I. INSTRUCTION OF MINIPURE

This is first version of MINIPURE. If a literal b is not in CNF formula $\mathcal{F}$ while b' is then b is called pure literal, and the clause contain b' can be eliminated for we can set b=False. Iteratively, we can eliminate all the pure literal thus reduce the number of clauses and variable of $\mathcal{F}$. But Chaff [1] and other kinds of conflict based solver sacrifice this heuristic for the efficiency of unit propagation. MINIPURE is the implementation of combining pure literal implication and unit propagation together to get better performance.

The solver is choose to embedded this idea in is MINISAT [2] for its simplicity and one of most well known conflict based solvers. The version of the MINISAT embedded is 2.2.0 . The implementation detail would be introduced later.

II. MAIN TECHNIQUES

The algorithms paradigm based on is CDCL.The further solving techniques used are: preprocessing, restart, phase saving , learning by conflict clauses. The above are common technique used in CDCL. Here are special techniques for MINIPURE: pure literal detection and pure literal learning (in part 4).

III. MAIN PARAMETERS

Beside the parameters used by MINISAT [2],there three parameters used for MINIPURE to tune.

1) additionan three paramters:

A . freq_pure : this paramters decide after how many retarts we can do pure literal detection again after restart level of stop_pure.

B. stop_pure: when the restart level is is bigger than the stop_pure *stop the pure literal detection and learning.*

C. *slowdown_pure: after which restart level should minipure slow down the pure literal detection. This number should be smaller than the stop_pure.*

For example, if slowdown_pure=2, stop_pure=3 and freq_pure=5, then at restart level 0~2 MINIPURE would do pure literal detection and learning and level 3 it would slow down the pure literal detection and after level 3 only level 8,13,...,3+5n would do pure literal detection again.

2) The paramters MINIPURE use for sat2013 competition is slowdown=0, stop_pure=1 and freq_pure=2. These three parameters are choose by the experiment result(t uned by hand) ,they have good performance on average. But for some special case these paramters should be changed.
Other parameters are the same as the default value of MINISAT.

IV. SPECIAL ALGORITHMS, DATASTRUCTURES, AND OTHER FETURES

The procedure and data structure explained below assumes the reader knowing about algorithms of Chaff and implementation of MINISAT. For more detail about the complete algorithm about Chaff and MINISAT please refer to [1] and [2].

1) Data Structure and initialize :

A. After parsing and preprocessing, MINIPURE of the original clauses. For each clauses, we assign an varible representing the clauses,e.g. $\mathcal{F}$=(a+b+..)(a'+b+..).., then c_1 represent (a+b+..), $c_2$represent(a+b+..). c_i is true iff at least one of literal is true, unassign iff c_i is not true and the literals in it are not all false. c_i is false means conflict happens need backtrack. Call them clause-variable.

Like procedure of assigning variable, if c_i is true push it into the vector clause_trail and for each decision level remember the size of clause_trail vector in another vector clause_trail_lim.

B. For each variable of positive and negative phase contruct a new vector that contains the clause-variable in positive phase if the the literal is contains in that clause,e.g.

$\mathcal{F}$=(a+b')(a'+b')(a+c)(a'+d).. then

a : | c_1 | c_3 |

a': | c_2 | c_4 |

b′: | c_1 | c_2 |

And there is a vector called lit_cla collect pointer to these kinds of vector storing the *clause-variable.*

C. For each literal MINIPURE, add one watch for it (in Chaff and MINISAT for each clause add two watches for unit propagation) ,the adding literal is clause-variable literal which is the first place of the vector descirbed in B. for literal a, c_1 is added , for literal a', c_2 is added.

Then watch(c1) returns a, b' while watch(c2) returns a' in the example of B. Here MINIPURE only do one clause-variable literal watch for clause-variable literal only has two state true

or unassigned. If c1 is assigned true check watch(c1) would get a, b' then check lit_cla[a] if c1=c3=true and a is unassigned implied by pure literal a' is true.

2) Algorithms:

A. *Pure literal detection:*

During the process of unit propagation, assume now propagating c' , after c' is propagated the original of MINISAT is to propagate the next literal in the unchecked queue if there is no conflict. Here MINIPURE change to see which clause-variable literal is assigned to be true after c' is true according to the vector lit_cla[c'].

For those clause-variable in lit_cla[c'] check whether it is unassigned or not, if unassigned set it to be true and push it into clause_trail , e.g. c': c1,c2,c3 and c1 is unassigned push it into clause_trail then check watch(c1) .Assume the propagation order is a,b,c' ,and watch(c1)=c',b,d only d is unassigned now check lit_cla[d]=(c1,c4,c5). For the clause-variable literals c4 and c5 if both are true , then by pure literal implication d' should be put into the uncheck queue, else without losing generality assume that c4 is unassigned, MINIPURE remove d from watch(c1) and push d into watch(c4).

Note that the implication of pure literal should only be done when it's unassigned otherwise some SAT problem would be UNSAT. If $\mathcal{F}$=(a+b')(a'+b) , then in decision level 1 push a in uncheck queue. a->c1->a', here c1->a' is implied by pure literal detection without checking whether variable is assigned or not which turns a SAT problem into UNSAT.

B. Pure literal clause learninig

When the conflict happens ,we need to backtrack and add a learning clause. In implication graph the clause-variable is implied to be true when one of its literal in their corresponding clause is true. This is a little different from the traditional unit propagation. However, these wouldn't affect the construction of implication just replaced the clause-variable literal with the original literal which implied it to be true. Literal b' is implied by c1,c2,c3 where b is contained by all of corresponding clauses c1,c2,c3 while c1 is put to clause_trail by literal x set true ,and c2 by y,c3 by z ,then fig.1 shows how it's reduced.

After construction the implication graph, MINIPURE can know which level to backtrack to. The remain problem is to unassign the clause-variable according to clause_trail_lim.

C. Detect Pure literal lazily

Since after restart again and again the variable order would change greatly, and MINIPURE wouldn't do pure literal detection all the time for the algorithm above should be done once every time a literal propagating. As a result, after restart for certain times(defined by user), MINIPURE would stop the pure literal detection. Also, restart the pure literal detection from time to time(defined by user) leaving the learnt clauses in the database to help us implied the pure literal while the pure literal detection mode is closed.

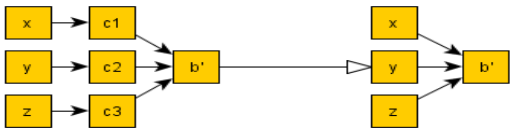

Fig 1

V. Implementation Details

1) The solver implement in language of C++.

2) The solver is modified from MINISAT 2.2.0 .

VI. SAT competition 2013 Specifics

1) The solver participate in SAT2013 competition of track 1,3,4,6,7,9,13, this solver is not joining MINISAT hack track for the difference is more than 1000 non-space characters.

2) The compiler is GCC4.5.2 with O3 optimization flag , 32bit.

3) Command line option: ./minipure <cnf file> -freq_pure=2 -stop_pure=0 -slowdown_pure=1 , and for other parameters the values are as default of MINISAT. Note that "=" must be placed.

VII. Availability

The code of MINIPURE is open and free for any who use for research and educational purpose. The license and code (including the license of MINISAT2.2.0) is available in [3].

VIII. Acknowledgement

The author thank to Jie-Hong Rolan Jiang's discussion about the implementation detail of MINIPURE and the idea of preprocessing with pure literal by Valeriy Balabanov.

IX. Refrerence

[1] M.W. Moskewicz, C.F. Madigan, Y.Zhao, L.Zhan,S. Malik **"Chaff: Engineering an Efficinet SAT Solver"** in Proc. of the 38th Design Automation Conference, 2001

[2] Niklas Sörensson, Niklas Een"**A SAT Solver with Conflict-Clause Minimzatin**",2005

[3] http://freakshare.com/files/pshpyy13/minipure.zip.html

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

MIPSAT

Sergio Núñez
Departamento de Informática
Universidad Carlos III de Madrid
28911 Leganés (Madrid), Spain
Email: sergio.nunez@uc3m.es

Daniel Borrajo
Departamento de Informática
Universidad Carlos III de Madrid
28911 Leganés (Madrid), Spain
Email: dborrajo@ia.uc3m.es

Carlos Linares López
Departamento de Informática
Universidad Carlos III de Madrid
28911 Leganés (Madrid), Spain
Email: clinares@inf.uc3m.es

***Abstract*—In this document we describe the technique used to configure the sequential portfolios submitted to the 2013 SAT Competition. We have submitted eight portfolios to the core solvers and sequential (SAT and SAT+UNSAT) tracks and one to the open track accounting for nine different portfolios in total.**

I. MIPSAT

MIPSAT portfolios have been generated using Mixed-Integer Programming (MIP), which computes the portfolio with the best achievable performance with respect to a selection of training SAT problem instances [1]. The resulting portfolio is a linear combination of candidate SAT solvers defined as a sorted set of pairs <solver, time>. Originally, our MIP model was applied to the generation of sequential portfolios for solving automated planning tasks. It considered an *objective function* that maximizes a weighted sum of different parameters including: memory usage, overall running time and coverage. In MIPSAT we only consider running time and coverage.

Given that we consider two different criteria (time and coverage), it could be viewed and solved as a multi-objective maximization problem. Instead, we solve two MIP tasks in sequence while preserving the cost of the objective function from the solution of the first MIP. Specifically, we first run the MIP task to optimize only *coverage* —i. e., total number of solved instances. If a solution exists, then a second execution of the MIP model is performed to find the combination of candidate solvers that achieves the same coverage (denoted as C) while minimizing the overall running time. To enforce a solution with the same coverage an additional constraint is added: $\sum_{i=0}^{n} coverage_i \geq C - \epsilon$, where ϵ is just any small real value used to avoid floating-point errors. Clearly, a solution is guaranteed to exist here, since a first solution was already found in the previous step. Pseudocode 1 shows the steps followed to generate all submitted portfolios where coverage was maximized first, and then running time was minimized among the combinations that achieved the optimal coverage. In our experiments, $\epsilon = 0.001$.

The MIP task used in this work does not specify any particular order to execute the solvers. It only assigns an execution time to each solver, which is either zero or a positive amount of time. The definition of the execution sequence is arbitrary and it is based just on the order in which the solvers were initially specified.

As a matter of fact, it was empirically found that the MIP solver usually distributes all the available time among the candidate solvers selected to be part of the portfolio. Running the procedure depicted in Pseudocode 1, the solution of the second MIP step could result in a sum of the times assigned to each SAT solver that is less than the available time in the competition. Thus, it is possible to have some slack time which could be distributed uniformly among the selected solvers. Besides, we can use this slack time to scripting tasks like checking if the current solver has solved the current problem or printing the solution found by the portfolio in the standard output.

Algorithm 1 Build a portfolio optimizing coverage and time

```
set weights to optimize only coverage
portfolio_1 := solve the MIP task
C := the resulting value of the objective function
if a solution exists then
    add constraint sum_{i=0}^{n} coverage_i >= C - 0.001
    set weights to optimize only overall running time
    portfolio_2 := solve the MIP task
    return portfolio_2
else
    exit with no solution
end if
```

II. CORE SOLVERS AND SEQUENTIAL TRACKS

The rules for core solvers and sequential tracks of the 2013 SAT Competition[1] only allow participants to use solvers that employ at most two different SAT solving engines for all runs and at any time in one track. To meet this requirement, we have added an extra constraint to the original MIP model so that the configuration of the generated portfolios are composed of at most two candidate solvers.

The sequential pure UNSAT tracks of the 2013 SAT Competition for core solvers require certification. Thus, the participant solvers are required to emit an unsatisfiability proof. However, we used solvers from the preceding SAT Competition[2], where this certification was not required. Hence, we have only focused on the SAT and SAT+UNSAT tracks.

Input data was generated for the MIP model to compute the configuration of each submitted portfolio. This data has been generated using the results of all single engine solvers from the preceding SAT Competition. For instance, the input data for the Application SAT track has been generated using only the SAT solutions of all the Application track problems

[1] http://www.satcompetition.org/2013/rules.shtml
[2] http://www.satcompetition.org/2011

TABLE I. SEQUENTIAL PORTFOLIOS SUBMITTED TO CORE SOLVERS AND SEQUENTIAL TRACKS

Portfolio	Solver	Allotted time	Computation time - step 1	Computation time - step 2	Performance
MIPSAT APPLICATION SAT 1	Precosat 236	3616	394.15	3446.45	104/107
	CryptoMiniSat 2.9.6	1379			
MIPSAT APPLICATION SAT 2	Precosat 576	3616	394.15	3446.54	104/107
	CryptoMiniSat 2.9.6	1379			
MIPSAT APPLICATION SAT+UNSAT	CryptoMiniSat 2.9.6	1125	29851.91	63941.28	223/227
	Glucose 2	3870			
MIPSAT HARD−COMBINATORIAL SAT	Sattime 2011	3832	12.85	229.01	125/140
	Sol 2011	1163			
MIPSAT HARD−COMBINATORIAL SAT+UNSAT 1	Sattime+ 2011	982	22056.04	1295.57	181/209
	Clasp 2.0-R4092	4013			
MIPSAT HARD−COMBINATORIAL SAT+UNSAT 2	Sattime+ 2011	982	22056.04	1295.57	181/209
	Clasp 2.1.1	4013			
MIPSAT RANDOM SAT	Sparrow 2011 ubcsat 1.2	4160	21.81	104.97	365/366
	EagleUP 1.565.350	834			
MIPSAT RANDOM SAT+UNSAT	Sparrow 2011 ubcsat 1.2	2505	47.05	680.28	473/475
	March_rw 2011	2490			

from the 2011 SAT Competition. We have not run any solver to generate the input data.

We have had some problems with the source code of CryptoMiniSat Strange Night 2 st version. Thus, we have used the newest version of CryptoMiniSat in those cases where the MIP task selected the Strange Night 2 st version as a component solver. Additionally, we have submitted an extra portfolio with the newest version of the selected solvers. These are shown in Table I with "2" appended at the end.

As we said before, the MIP task does not specify the execution sequence of the generated portfolios. However, since ties will be broken by CPU time in sequential tracks of the 2013 SAT Competition, we have sorted the execution sequence of the submitted portfolios in increasing order of the average CPU time.

Table I shows the configuration of submitted portfolios. The columns "Computation time" show the time required (in seconds) to solve the MIP task in each step: first, when maximizing coverage and then, when minimizing the overall running time. The last column shows the coverage of each portfolio in the training set and the best coverage achievable with a linear combination of solvers (without the core solvers constraint) for the same set of training instances.

III. OPEN TRACK

As mentioned above, our MIP model serves to derive sequential portfolios [1]. However, there are eight cores (of a cluster node) and 5000 seconds wall-clock time available for each participant in the open track. In spite of that, we have used the same technique to configure the submitted portfolios.

As in the previous case, input data for the MIP model has been generated using data from the preceding SAT Competition. In this track, all instances (application, crafted and random) and all participant solvers (including parallel solvers and portfolios in the set of candidate solvers) were considered.

The execution sequence of portfolios generated by the MIP task is arbitrary. However, ties are broken in ascending order of the average wall-clock time in the open track of the 2013 SAT Competition.

Table II shows the configuration of the submitted portfolio to the open track. The time required to solve the MIP tasks was 1295.7 seconds in the first step and 1809.1 seconds in the second one.

TABLE II. SEQUENTIAL PORTFOLIO SUBMITTED TO THE OPEN TRACK.

Solver	Allotted time
Sattime+ 2011	7
Sattime 2011	10
Precosat 236	22
Adaptg2wsat2011	37
Sol 2011	49
MPhaseSAT64	61
MPhaseSAT	66
Sparrow 2011 ubcsat 1.2	645
CryptoMiniSat 2.9.6	659
Ppfolio par	3399
Total Time	4955

As in the other track, the latest version of CryptoMiniSat was used instead of the version submitted to the previous competition.

IV. SAT COMPETITION 2013 SPECIFICS

We have used the optimization flags, command-line options and other parameters provided by the authors of the SAT solvers shown in Tables I and II. We also provide the random seed to be used: *12061986*. To compile and run each submitted portfolio just run *build.sh* and */binary/solve <instance file name> -tmp <temporary directory name>*.

ACKNOWLEDGMENTS

We automatically generated sequential portfolios of existing SAT solvers by means of a MIP task to be submitted to the 2013 SAT Competition. Thus, we would like to acknowledge and thank the authors of the individual SAT solvers for their contribution and hard work.

This work has been partially supported by the INNPACTO program from the Spanish government associated to the MICINN project IPT-370000-2010-8 and by the MICINN project TIN2011-27652-C03-02.

REFERENCES

[1] S. Núñez, D. Borrajo, and C. Linares López, "Performance Analysis of Planning Portfolios," in *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Ncca+: Configuration Checking and Novelty+ like heuristic

Djamal Habet Donia Toumi André Abramé

LSIS, UMR CNRS 7296
Université Aix-Marseille, Marseille, France
{Djamal.Habet,Donia.Toumi,Andre.Abrame}@lsis.org

***Abstract*—This document describes the local search SAT solver *Ncca+* based on the *CCASat* solver and Novelty+ like heuristic.**

I. Major solving techniques

Ncca+ is based on *CCASat* which is described in [1]. *CCASat* is the winner of last SAT Challenge 2012[1] on the random SAT track. The evolutions introduced by *Ncca+* are:

1) For 3-SAT random instances: when selecting a CCD (Configuration Changed Decreasing) variable [2], *CCASat* breaks ties in the favor of the oldest variable in the case of equal score of the candidate variables. In *Ncca+*, the breaking ties is done according to the number of occurrences of the CCD variables on the falsified clauses.
2) For k-SAT ($k > 3$) random instances: the number of occurrences of the variables on the falsified clauses is also considered when selecting a variable among the CCD and SD (Significant Decreasing) variables [2]. In this case, *Ncca+* works as follows:
   - If the set of CCD variables is not empty, select the one with the highest score, breaking ties in the favor of the oldest one then in the favor of the one occurring the most in the falsified clauses.
   - If the SD variable set is not empty, then select a variable which has its configuration changed, breaking ties in the favor of the oldest one then in the favor of the one occurring the most in the falsified clauses.
   - In other cases, update clause weights and with probability wp do as Novelty(p) and with probability $1 - wp$ choose a variable in a randomly selected variable as done in *CCASat* and by considering the number of occurrences of a variable in falsified clauses. The values of p and wp are adaptively adjusted during the search [3], [4].

   The rest of the solver is similar to *CCASat* regarding to the smoothing scheme of the weight clauses.

[1]http://baldur.iti.kit.edu/SAT-Challenge-2012

II. Parameter description

The parameters of *Ncca+* are similar to those of *CCASat*. The adaptive noise settings (p and wp values) are based on the ones used in [5] with $\phi = 5$ and $\theta = 2$.

III. Special Algorithms, Data Structures and Features

The code of *Ncca+* is based on *CCASat* code[2]. There is no additional data structure.

IV. Implementation detail

1) The programming language used is C++
2) The solver is based on *CCASat* with the additional features explained above.

V. SAT Challenge 2012 Specifics

1) The solver is submitted in "Core Solvers, Sequential, Random SAT" track.
2) The used compiler is *g++*
3) The optimization and compilation flags used are "-O3 -static".

VI. Availability

Our solver will be publicly available after the SAT competition 2013.

Acknowledgment

We would like to thank the authors of *CCASat* for making available the source code of their solvers.

References

[1] S. Cai, C. Luo, and K. Su, "Ccasat: Solver description." in *Proceedings of SAT Challenge 2012*, 2012, p. 13.
[2] S. Cai and K. Su, "Configuration checking with aspiration in local search for sat." in *Tewenty-sixth national conference on Artificial intelligence (AAAI-2012)*, 2012, pp. 434–440.
[3] H. H. Hoos, "An adaptive noise mechanism for walksat," in *Eighteenth national conference on Artificial intelligence (AAAI-2002)*, 2002, pp. 655–660.
[4] C. M. Li and W. Q. Huang, "Diversification and determinism in local search for satisfiability," in *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'05, 2005, pp. 158–172.
[5] C. M. Li, W. Wei, and H. Zhang, "Combining adaptive noise and promising decreasing variables in local search for sat," in *SAT Competition 2007*, 2007, pp. 158–172.

[2]Available from http://shaoweicai.net/research.html

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Nigma: A Partial Backtracking SAT Solver

Chuan Jiang
Iowa State University
Ames, Iowa, USA

Ting Zhang
Iowa State University
Ames, Iowa, USA

***Abstract*—This document describes the SAT solver *Nigma*, which is a MiniSat-based solver with a partial backtracking strategy.**

I. Introduction

An important aspect of backtracking is to decide which level the solver backtracks to after a conflict is identified and analyzed at the current level dl_{curr}. Currently, the majority of solvers choose to backtrack to the assertion level dl_{asrt}, which is the second highest decision level among the literals in the learnt clause. (We do not consider the case when the solver learns a unit clause.) As observed in [1], if restart happens frequently, similar assignments are made after each restart. We observed the same phenomenon in backtracking, especially for the solvers using *VSIDS* [2] and *phase saving* [3]. Due to the fact that the activities of most variables change little after one backtracking, similar decisions will be made and the corresponding unit propagation will be repeated.

In our SAT solver *Nigma*, we implemented a partial backtracking strategy that allows the solver to unwind to a decision level dl_{back} between dl_{asrt} and dl_{curr} so that the propagation effort made between dl_{asrt} and dl_{back} will be partially saved. (Greedily, we always choose $dl_{back} = dl_{curr} - 1$, but in fact any $dl_{back} : dl_{asrt} \leq dl_{back} < dl_{curr}$ is fine.) After backtracking to dl_{back}, the conflicting variable is still assigned at dl_{asrt}, and the variable assignments between dl_{back} and dl_{asrt} are amended for the sake of consistency.

II. Main Techniques

We highlight the following properties of Nigma's partial backtracking strategy.

1) Variables can be assigned at any existing levels. The conflicting variable is assigned at dl_{asrt}, not at the new current level dl_{back}. As a result, new implications between dl_{back} and dl_{asrt} may be induced. Nigma allows the variables to be assigned at any existing levels so that these new implications can be propagated correctly. This is a major difference between Nigma and other solvers.
2) New conflicts need to be resolved. The amendment for assignments between dl_{back} and dl_{asrt} may result in new conflicts. There are two kinds of conflicts: virtual conflicts and actual conflicts. Virtual conflicts means that the two most recently falsified literals in the conflicting clause were not assigned at the same level. Actual conflicts are just the opposite. Virtual conflicts can be simply resolved by further backtracking without learning any new clause and by generating new implications. In the worst case, a sequence of continued backtracking leads the solver back to dl_{asrt}, just as in the classic strategy. Actual conflicts are resolved by standard conflict resolving algorithm.
3) Decision levels of some assigned variables may be modified. For example, consider a clause $x_1 \vee x_2$. Initially, x_1 is assigned to TRUE at level 18. Suppose at level 20, a conflict is identified and we have $dl_{asrt} = 5$ and $dl_{back} = 19$. By (1), assignment amendment may induce new implications between dl_{asrt} and dl_{back}. Further suppose the solver assigns x_2 to FALSE at level 15. Consequently, the decision level of x_1 should be changed to 15. Nigma backtracks to level 15 to complete such a modification.
4) A variant algorithm for using watched literals is implemented. During unit propagation, if a clause becomes unit, its watched literals are then necessarily assigned at the highest decision level among all the literals. This condition may be violated during a partial backtracking. Nigma solves this problem by choosing literals with highest levels to watch.
5) A more aggressive restart strategy is adopted. Partial backtracking essentially reduces the restart frequency. For example, if dl_{asrt} is far from dl_{curr} (e.g., $dl_{asrt} = 5$ and $dl_{curr} = 100$), the classic backtracking to dl_{asrt} is almost equivalent to a restart. In contrast, Nigma only backtracks to somewhere in between, and hence it is more likely being trapped in a local search. So we adopt a more aggressive restart strategy to compensate the reduced restart frequency.

III. Main Parameters

When a conflict happens, Nigma initiates a partial backtracking only if the distance between dl_{curr} and dl_{asrt} is longer than 10; otherwise it falls back to the classic backtracking. This is because that the effort to save propagation work is worthwhile only when the solver is going to backtrack a long distance. Currently, we are investigating if the number of assigned variables between dl_{curr} and dl_{asrt} is a better triggering condition.

IV. Special Algorithms, Data Structures, and Other Features

When a partial backtracking occurs, a special propagation procedure will propagate the assignment of the conflicting

variable. Different from the standard propagation, this procedure is able to propagate the assignments that are not at the current level and deal with the issues mentioned in the Main Techniques section.

Additional features in Nigma are LBD-based clause deletion [4], dynamic restart [5], and a simple implementation of SatElite-like preprocessing [6].

V. Implementation Details

Besides the previously mentioned features, Nigma is a reimplementation of MiniSat 2.2 in C++.

VI. SAT Competition 2013 Specifics

We submit two versions of Nigma, one with partial backtracking enabled and one without. Both versions are submitted to the track for *Core solvers, Sequential, Application SAT*, the track for *Core solvers, Sequential, Application UNSAT* and the track for *Core solvers, Sequential, Application SAT+UNSAT*. Both versions are compiled by GCC in 64-bit mode with -O3 flag.

VII. Availability

Nigma is an open-source SAT solver under MIT licence. The description and source code can be found at `http://verification.cs.iastate.edu/nigma`.

References

[1] P. van der Tak, A. Ramos, and M. Heule, "Reusing the assignment trail in cdcl solvers," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 133–138, 2011.

[2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th conference on Design Automation*, New York, New York, USA, 2001, pp. 530–535.

[3] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Proceedings of the 10th international conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2007, pp. 294–299.

[4] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st international joint conference on Artificial Intelligence*, 2009, pp. 399–404.

[5] ——, "Refining Restarts Strategies for SAT and UNSAT," in *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming*. Springer-Verlag, 2012, pp. 118–126.

[6] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 61–75.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

PCASSO – a Parallel CooperAtive Sat SOlver

Ahmed Irfan and Davide Lanti and Norbert Manthey
Knowledge Representation and Reasoning,
TU Dresden, Germany

***Abstract*—The SAT solver PCASSO is a parallel SAT solver based on partitioning the search space iteratively.**

I. Introduction

PCASSO uses two main methods: creating partitions and solving partitions. Partitions are created through *partition functions*, where a partition function is a function ϕ such that, given a formula F and a natural number $n \in \mathbb{N}^+$, $\phi(F, n) := (F_1, \ldots, F_n)$, where $F \equiv F_1 \vee \ldots \vee F_n$ and each pair of partitions is disjoint: $i \neq j \in [1, n],\ F_i \wedge F_j \models \bot$. Without loss of generality we assume that partitions $F_1, \ldots, F_n$ are always of the form $F \wedge K_1, \ldots, F \wedge K_n$, where $K_1, \ldots, K_n$ are sets of clauses, called *partitioning constraints*. By iteratively applying the partition function to a formula F, a *partition tree* is produced. Nodes in the partition tree are tagged with their *positions*: the root node F is tagged with the empty position ϵ; the i-th successor (from left to right) of a node F^p at position p is the node F^{pi} (see Figure 1). Please notice that, as positions are strings, the standard *prefix* relation among strings ($<$) is defined for positions as well.

II. Main Techniques

The partition function used in PCASSO is *tabu scattering*, which is an extension of *scattering* [1]. The idea of scattering is to define each partitioning constraint as conjunctions of *cubes* [2], where a cube is a formula $Q := \{C_1, \ldots, C_n\}$ such that $|C_i| = 1$, for each $1 \leq i \leq n$. Observe that the negation of a cube $Q := \{\{l_1\}, \ldots, \{l_n\}\}$ is the clause $\{\overline{l_1}, \ldots, \overline{l_n}\}$. More precisely, given a formula F_0 and an integer n, the n partitions $F_1, \ldots, F_n$ are created by using $n-1$ cubes $Q_1, \ldots, Q_{n-1}$ and applying them according to the following schema: $F_1 := F_0 \wedge Q_1$; $F_{m+1} := F_0 \wedge (\bigwedge_{i=1}^{m} \overline{Q_i}) \wedge Q_{m+1} (1 \leq m < n-1)$; and finally $F_n := F_0 \wedge \bigwedge_{i=1}^{n-1} \overline{Q_i}$. *Tabu scattering* adds the restriction to scattering that a variable used in one cube must not be used in the cubes for creating the remaining partitions. Using tabu scattering, we diversify the search more. PCASSO uses lookahead techniques [3] for choosing the literals (in cubes). In particular it chooses variables with the maximum *mixdiff* score [3]. The score *mixdiff* of a variable is the product of the *diff* score of each polarity of the variable. We calculate the *diff* score of the polarity of a variable by applying lookahead, and use the following weighted sum: 0.3 times the number of propagated literals plus 0.7 times the number of newly created binary clauses. After choosing the variable with the maximum *mixdiff* score, we choose the polarity of the variable that has the lowest *diff* score for creating cubes. We also use the following reasoning techniques: failed literals, necessary assignments, pure literals, and add learned clauses to the partition constraints. Techniques like constraint resolvent, double lookahead, and adaptive pre-selection heuristics are also used as proposed in the literature [3].

To describe the *node-state* of a node F^p at a certain point of execution we use a triple (F^p, s, r) where $s \in \{\top, \bot, ?\}$ ($\top$ indicates that an incarnation found a model for the node, whereas $\bot$ indicates that an incarnation proved unsatisfiability of F^p; finally, ? indicates that the node has not been solved yet) and $r \in \{\blacktriangleright, \blacksquare\}$ (indicating whether an incarnation is *running* on F^p or not, respectively). Given the notion of node-state, PCASSO exploits the *overlapped solving* strategy if two incarnations are allowed to run at the same time on nodes F^p, F^q such that $p \leq q$. In order to solve an unsatisfiable node F^p, either F^p has to be directly solved by some incarnation or each child node F^{pi} has to be solved. There is no limit on the solving time for each node.

Per variable, VSIDS activity and progress saving are shared from parent to child nodes. When PCASSO starts solving, the root node and the nodes at the partition tree level one start at almost the same time. The nodes at partition tree level greater than one are usually created after some time, so we initialize their search process with the VSIDS and progress saving information of their parent, because the child node searches in the sub-search space of its parent and whatever is learned by the parent search can help the solving child node as well.

Learned clauses are shared between incarnations to intensify the search. A learned clause is considered *unsafe* if it belongs to partitioning constraints, or it is obtained by a resolution derivation involving one or more unsafe clauses. A clause that is not unsafe is called *safe* clause, and only safe clauses are shared. A learned clause is shared in a sub-partition tree if it is safe in that sub-partition tree. This information can be calculated by tagging each clause with the position of the sub-tree where the clause is valid (*position-based* tagging [4]). We propose a dynamic learned clause sharing scheme, that is based on LBD scores [5]. A learned clause is eligible for sharing by an incarnation if the LBD score of this clause is lower than a fraction δ of the global LBD average of the incarnation. In PCASSO, we use $\delta = 0.5$.

PCASSO uses different restart policy and different clause cleaning policies for the nodes, depending whether the node is root, leaf or middle (not root and not leaf).

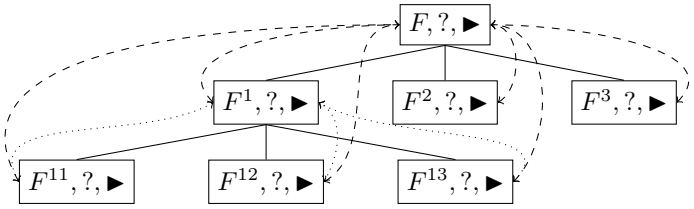

Fig. 1. Visualization of a partition tree with clause sharing and overlapped solving: The dashed lines represent the possible communication when flag based sharing is used, whereas the dotted and dashed lines together visualize the possible sharing with position based tagging.

PCASSO can have a scenario that there is only one unsolved node at some partition level. We call this scenario the *only child scenario*. Consider that if the only child scenario happens at some level of the partition tree, then there are two cases: i) the parent node is looking into the search space which has been solved by one of its children already, ii) the parent node is looking into the same search space where its unsolved children are looking. In either case, we have the risk of doing redundant work. We propose an approach to get out of this scenario by reintroducing the solving limit in a node that has only one unsolved child (AVOID). To be on safe side, we do not apply this limit for the root node. The introduced limit grows with the level of the node (level $*$ 4096 conflicts). Since in the only child scenario all learned clauses can be shared among the two participating nodes, we can also EXPLOIT this situation, by enabling this sharing. In the extreme case, this configuration is very similar to portfolio solvers, since then all clauses can be shared without restrictions. When clauses are tagged by position-based tagging [4], additional information can be obtained by performing a conflict analysis on solved unsatisfiable nodes. Consider a node $(F^p, \bot, \blacksquare)$, and let $\{\}^q$ be the empty clause derived by the incarnation that solved F^p. Then, from the main theorem in [4], we conclude that $\{\}^q$ is the semantic consequence of the node of position q in the partition tree. Observe that q is a prefix p: $q \leq p$. Consequently, not only the node at position p can be marked as unsatisfiable, but also the node F^q as well as all its child nodes. As a result, more incarnations can be terminated and start solving different partitions. We call this kind of technique *conflict driven node killing*. A similar approach is reported in [6].

III. MAIN PARAMETERS

The major parameters of the solver influence the number of threads that should be used, the number of partitions that should be created for each node, and how sharing should be performed. For the competition, we use 8 threads, and produce 8 partitions. Furthermore, we share learned clauses according to their LBD value. Finally, the treatment of the only-child scenario can be influenced.

For each of these big parts of the solver, many small parameters are provided, that control the special behavior of the system. There are only minor magic constants that control the run time of the look-ahead procedures during partitioning, which are chosen according to the literature [3].

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

Each node in the partition tree is associated a *pool of shared clauses*, where a pool is implemented as a vector of clauses. This permits to decouple the life of a shared clause from the life of the incarnation where the shared clause has been learned. Instead of tagging each clause with a position, clauses are tagged with integers representing a *level* in the partition tree (root node has level zero). Each incarnation working over a node F^p can only access the pools placed at nodes of positions $q \leq p$. Concurrent access to pools is regulated by standard `POSIX` Read-Write locks.

COPROCESSOR is used as preprocessor [7].

V. IMPLEMENTATION DETAILS

PCASSO is built on top of GLUCOSE 2.2.

VI. SAT COMPETITION 2013 SPECIFICS

PCASSO has been submitted to both the application and the crafted parallel track.

VII. AVAILABILITY

The source code of PCASSO is available at tools.computational-logic.org under the GPL license.

REFERENCES

[1] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, "A distribution method for solving SAT in grids," in *Proc. of the 9th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'06. Springer-Verlag, 2006, pp. 430–435.

[2] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Accepted for HVC 2011*, 2012.

[3] M. J. H. Heule and H. van Maaren, *Look-Ahead Based SAT Solvers*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 5, pp. 155–184.

[4] D. Lanti and N. Manthey, "Sharing information in parallel search with search space partitioning," in *Proc. of the 7th International Conference on Learning and Intelligent Optimization (LION 7)*, ser. LNCS. Springer, 2013.

[5] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proc. of the 21st international jont conference on Artifical intelligence*, ser. IJCAI'09. Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[6] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, "Grid-based SAT solving with iterative partitioning and clause learning," in *Proc. of the 17th international conference on Principles and practice of constraint programming*, ser. CP'11, 2011, pp. 385–399.

[7] N. Manthey, "Coprocessor 2.0: a flexible cnf simplifier," in *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 436–441. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31612-8_34

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

PeneLoPe in SAT Competition 2013

Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, Cédric Piette
Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{audemard,hoessen,jabbour,lagniez,piette}@cril.fr

***Abstract*—This paper provides a short system description of our updated portfolio-based solver called PeneLoPe, based on ManySat. Particularly, this solver focuses on collaboration between threads, providing different policies for exporting and importing learnt clauses between CDCL searches. Moreover, different restart strategies are also available, together with a deterministic mode.**

I. Overview

PeneLoPe [2] is a portfolio parallel SAT solver that uses the most effective techniques proposed in the sequential framework: unit propagation, lazy data structures, activity-based heuristics, progress saving for polarities, clause learning, etc. As for most of existing solvers, a first preprocessing step is achieved. For this step -which is typically sequential- we have chosen to make use of SatElite [6].

In addition, PeneLoPe includes a recent technique for its learnt clause database management. Roughly, this approach follows this schema: each learnt clause c is periodically evaluated with a so-called *psm* measure [3], which is equal to the size of the set-theoretical intersection of the current interpretation and c. Clauses that exhibit a low *psm* are considered relevant. Indeed, the lower is a *psm* value, the more likely the related clause is about to unit-propagate some literal, or to be falsified. On the opposite, a clause with a large *psm* value has a lot of chance to be satisfied by many literals, making it irrelevant for the search in progress.

Thus, only clauses that exhibit a low *psm* are selected and currently used by the solver, the other clauses being *frozen*. When a clause is frozen, it is removed from the list of the watched literals of the solver, in order to avoid the computational over-cost of maintaining the data structure of the solver for this useless clause. Nevertheless, a frozen clause is not erased but it is kept in memory, since this clause may be useful in the next future of the search. As the current interpretation evolves, the set of learnt clauses actually used by the solver evolves, too. In this respect, the *psm* value is computed periodically, and sets of clauses are frozen or unfrozen with respect to their freshly computed new value.

Let P_k be a sequence where $P_0 = 500$ and $P_{i+1} = P_i + 500 + 100 \times i$. A function "*updateDB*" is called each time the number of conflict reaches P_i conflicts (where $i \in [0..\infty[$). This function computes new *psm* values for every learnt clauses (frozen or activated). A clause that has a *psm* value less than a given limit l is activated in the next part of the search. If its *psm* does not hold this condition, then it is frozen. Moreover, a clause that is not activated after k (equal to 7 by default) time steps is deleted. Similarly, a clause remaining active more than k steps without participating to the search is also permanently deleted (see [3] for more details).

Besides the *psm* technique, PeneLoPe also makes use of the *lbd* value defined in [4]. *lbd* is used to estimate the quality of a learnt clause. This new measure is based on the number of different decision levels appearing in a learnt clause and is computed when the clause is generated. Extensive experiments demonstrates that clauses with small *lbd* values are used more often than those with higher *lbd* ones. Note also that *lbd* of clauses can be recomputed when they are used for unit propagations, and updated if it becomes smaller. This update process is important to get many good clauses.

Given these recently defined heuristic values, we present in the next Section several strategies implemented in PeneLoPe.

II. Detailled Features

PeneLoPe proposes a certain number of strategies regarding importation and exportation of learnt clauses, restarts, and the possibility of activating a deterministic mode.

Importing clause policy: When a clause is imported, we can consider different cases, depending on the moment the clause is attached for participating to the search.

- *no-freeze*: each imported clause is actually stored with the current learnt database of the thread, and will be evaluated (and possibly frozen) during the next call to *updateDB*
- *freeze-all*: each imported clause is *frozen* by default, and is only used later by the solver if it is evaluated relevant w.r.t. unfreezing conditions.
- *freeze*: each imported clause is evaluated as it would have been if locally generated. If the clause is considered relevant, it is added to the learnt clauses, otherwise it is frozen

Exporting clause policy: Since PeneLoPe can freeze clauses, each thread can import more clauses than it would with a classical management of clauses, where all of them are attached. Then, we propose different strategies, more or less restrictive, to select which clauses have to be shared:

- *unlimited*: any generated clause is exported towards the different threads.
- *size limit*: only clauses whose size is less than a given value (8 in our experiments) are exported [8].
- *lbd limit*: a given clause c is exported to other threads if its *lbd* value $lbd(c)$ is less than a given limit value d (8

by default). Let us also note that the *lbd* value can vary over time, since it is computed with respect to the current interpretation. Therefore, as soon as $lbd(c)$ is less than d, the clause is exported.

Restarts policy: Beside exchange policies, we define two restart strategies.

- *Luby*: Let l_i be the i^{th} term of the Luby serie. The i^{th} restart is achieved after $l_i \times \alpha$ conflicts (α is set to 100 by default).
- *LBD* [4]: Let LBD_g be the average value of the LBD of each learnt clause since the beginning. Let LBD_{100} be the same value computed only for the last 100 generated learnt clause. With this policy, a restart is achieved as soon as $LBD_{100} \times \alpha > LBD_g$ (α is set to 0.7 by default). In addition, the VSIDS score of variables that are unit-propagated thank for a learnt clause whose *lbd* is equal to 2 are increased, as detailled in [4].

Furthermore, we have implemented in `PeneLoPe` a deterministic mode which ensures full reproducibility of the results for both runtime and reported solutions (model or refutation proof). Large experiments show that such mecanism does not affect significantly the solving process of portfolio solvers [7]. Quite obviously, this mode can also be unactivated in `PeneLoPe`.

III. Fine Tuning Parameters of PeneLoPe

`PeneLoPe` is designed to be fine-tuned in an easy way, namely without having to modify its source code. To this end, a configuration file (called `configuration.ini`, an example is provided in Figure 1) is proposed to describe the default behavior of each thread. This file actually contains numerous parameters that can be modified by the user before running the solver. For instance, besides export, import and restart strategies, one can choose the number of threads that the solver uses, the α factor if the Luby techniques is activated for the restart strategy, etc. Each policy and/or value can obvioulsy differ from one thread to the other, in order to ensure diversification. In the next Section, we present the actual configuration file submitted at the SAT challenge.

IV. Update

A few changes were made compared to the previous version [1]. Two new restart heuristics are available. The first is the one proposed by Armin Biere in [5] and the second is the geometric serie proposed in [9]. Moreover, the glucose restart policy has been updated to match the one proposed in `Glucose2.1` [4].

Acknowledgment

`PeneLoPe` has been partially developed thank to the financial support of CNRS and OSEO, under the ISI project "Pajero".


```
ncores = 8
deterministic = false
;this is the default behavior of each
;thread, can be modified or specified
;after each [solverX] item
[default]
;if set to true, then psm is used
usePsm = true
;allowed values: avgLBD, luby
restartPolicy = avgLBD
;allowed values: lbd, unlimited, size
exportPolicy = lbd
;allowed values:
;freeze, no-freeze, freeze-all
importPolicy = freeze
;number of freeze before the clause
;is deleted
maxFreeze = 7
;initial #conflict before the first
;updateDB
initialNbConflictBeforeReduce = 500
;incremental factor for updateDB
nbConflictBeforeReduceIncrement = 100
;maximum lbd value for exchanged clauses
maxLBDExchange = 8
[solver0]
importPolicy = no-freeze
[solver1]
initialNbConflictBeforeReduce = 5000
nbConflictBeforeReduceIncrement = 1000
[solver2]
maxFreeze = 8
;solver3 is the default solver
[solver3]
[solver4]
restartPolicy = luby
lubyFactor = 100
[solver5]
exportPolicy = size
[solver6]
maxFreeze = 4
[solver7]
importPolicy = freeze-all
```

Fig. 1. Configuration.ini file

References

[1] Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Penelope, a parallel clause-freezer solver. In *proceedings of SAT Challenge 2012: Solver and Benchmarks Descriptions*, pages 43–44, Lens, may 2012.

[2] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 200–213. Springer Berlin Heidelberg, 2012.

[3] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezeing and reactivating learnt clauses. In *proceedings of SAT*, pages 147–160, 2011.

[4] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJCAI*, pages 399–404, 2009.

[5] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT*, page 2008.

[6] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *proceedings of SAT*, pages 61–75, 2005.

[7] Youssef Hamadi, Said Jabbour, Cédric Piette, and Lakhdar Saïs. Deterministic parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.

[8] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In *proceedings of IJCAI*, pages 499–504, 2009.

[9] Knot Pipatsrisawat and Adnan Darwiche. Width-based restart policies for clause-learning satisfiability solvers. In *SAT*, pages 341–355, 2009.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

pmcSAT

Ricardo Marques, Luis Guerra e Silva, Paulo Flores and L. Miguel Silveira
INESC-ID / IST-TU Lisbon
Rua Alves Redol, 9, 1000-029 Lisbon, Portugal
{rsm,lgs,pff,lms}@algos.inesc-id.pt

***Abstract*—This document describes the SAT solver PMCSAT, a conflict-driven clause learning (CDCL) portfolio solver that launches multiple instances of the same basic solver using different heuristic strategies, for search-space exploiting and problem analysis, which share information and cooperate towards the solution of a given problem.**

I. Introduction

PMCSAT is a portfolio-based multi-threaded, multi-core SAT solver, built on top of MINISAT [1]. The general strategy pursued in this solver is to launch multiple instances of the same solver, with different parameter configurations, which cooperate to a certain degree by sharing relevant information when searching for a solution. This approach has the advantage of minimizing the dependence of current SAT solvers on specific parameter configurations chosen to regulate their heuristic behavior, namely the decision process on the choice of variables, on when and how to restart, on how to backtrack, etc.

II. Main Techniques

The solver uses multiple threads (eight currently), which explore the search space independently, following different paths, due to the way each thread is configured.

In order to ensure that each thread follows divergent search paths, we defined distinct priority assignment schemes, one for each thread of PMCSAT. Note that the priority of a variable will determine its relative assignment order.

Below are described the different priority schemes that were used.

- **Thread #0/#1** - All the variables have the same priority, therefore this thread mimics the original VSIDS heuristic.
- **Thread #2** - The first half of the variables read from the file have higher priority than the second half.
- **Thread #3** - The first half of the variables read from the file have lower priority than the second half.
- **Thread #4** - The priority is sequentially decreased as the variables are read from the file.
- **Thread #5** - The priority is increased according to its number of occurences in the file.
- **Thread #6** - The priority is decreased according to its number of occurences in the file.
- **Thread #7** - The priority is decreased according to the number of variables that have the same number of common variables.

Threads #0 and #1 use the same priority-scheme, however they have different learnt clause deletion methods.

In [2] the authors show that using a more agressive clause deletion strategy could lead to good results in a CDCL SAT solver, as a result of the overhead reduced in propagation on learnt clauses. Threfore, thread #1 uses a more agressive deletion strategy, while all the other threads follow the MINISAT deletion scheme.

Although each PMCSAT thread exploits independently the search space, this is not just a purely competitive solver. All the threads cooperate by sharing the learnt clauses resulting from conflict analysis, leading to a larger pruning of the search space.

To reduce the communication overhead introduced by clause sharing, and its overall impact in performance, we designed data structures that eliminate the need for read and write locks. These structures are stored in shared memory, which is shared among all threads.

Each thread owns a queue, where the clauses to be shared are inserted. Associated to this queue is a pointer, which marks the last inserted clause, manipulated by the source thread, while every other targed thread owns a pointer that indicates last read clause from the queue. This way, this data structure eliminates the need for a locking mechanism.

A more detailed explanation of the techniques used in this solver can be found in [3].

III. Main Parameters

The internal parameters of PMCSAT are the same as in MINISAT, with the addition of the following:

1) The learnt clauses size condition to be exported. For the SAT competition the clause size limit was set to 8, i. e., only learnt clauses with less than 8 literals are exported and shared with other threads.
2) The threshold for the thread's learnt clause database to be reduced. The initial condition defined for thread #1 is 4000 learnt clauses, with a increment value of 300, as in GLUCOSE 2.1.

IV. Implementation details

1) The programming language used is C++, using pthread for parallel computing.
2) The solver was implemented on top of MINISAT v2.2.0.

V. SAT Competition 2013 Specifics

1) The solver was submitted to all Parallel Tracks: Application SAT+UNSAT, Hard-Combinatorial SAT+UNSAT, Random SAT and Open Track.

2) The used compiler is g++.
3) The optimization flag used is "-O3". The compilation options are the same as the used existing solver.
4) 64-bit binary.
5) The only command-line parameter is the input file

VI. Availability

More information about the PMCSAT solver, including its source code, can be found on the ALGOS research group publicly available website:

http://algos.inesc-id.pt/algos/software.php

Acknowledgment

The authors would like to thank the authors of MiniSat 2.2.0 for making available the source code of their solvers.

This work was partially supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under project "ParSat: Parallel Satisfiability Algorithms and its Applications" (PDTC/EIA-EIA/103532/2008) and project PEst-OE/EEI/LA0021/2011.

References

[1] N. Een and N. Sorensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solver," in *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, jul 2009, pp. 399–404.

[3] R. S. Marques, L. G. e Silva, P. Flores, and L. M. Silveira, "Improving sat solver efficiency using a cooperative multicore approach," *International FLAIRS Conference, May 22 - 24, 2013*.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

probSAT

Adrian Balint
Ulm University
Ulm,Germany

Uwe Schöning
Ulm University
Ulm,Germany

***Abstract*—We describe some details about the SLS solver probSAT, a simple and elegant SLS solver based on probability distributions, a heuristic first presented in the SLS solver Sparrow [3], which won the Random SAT track from the SAT Competition 2011.**

I. Introduction

The probSAT solver is an efficient implementation of the probSAT algorithm presented in [2] with slightly different parameterization.

II. Main Techniques

probSAT is a pure stochastic local search solver based on the following algorithm:

Algorithm 1: ProbSAT

Input : Formula F , $maxTries$, $maxFlips$
Output: satisfying assignment **a** or UNKNOWN

for $i = 1$ *to* $maxTries$ **do**
  a $\leftarrow$ randomly generated assignment
  for $j = 1$ *to* $maxFlips$ **do**
    if *(**a** is model for F)* **then**
      return **a**
    $C_u \leftarrow$ randomly selected unsat clause
    for x *in* C_u **do**
      compute $f(x, \mathbf{a})$
    $var \leftarrow$ random variable x according to probability $\frac{f(x,\mathbf{a})}{\sum_{z \in C_u} f(z,\mathbf{a})}$
    flip(var)
return UNKNOWN;

ProbSAT uses only the make and the break values of a variable in the probability functions $f(x, a)$, which can have an exponential or a polynomial shape as listed below.

$$f(x, \mathbf{a}) = \frac{(c_m)^{make(x,\mathbf{a})}}{(c_b)^{break(x,\mathbf{a})}}$$

$$f(x, \mathbf{a}) = \frac{(make(x, \mathbf{a}))^{c_m}}{(\epsilon + break(x, \mathbf{a}))^{c_b}}$$

III. Parameter settings

ProbSAT has four important parameters: (1) $fct \in \{0, 1\}$ shape of the function, (2) $cb \in \mathbb{R}$, (3) $cm \in \mathbb{R}$ which is set to 1 and (4) $epsilon \in \mathbb{R}$, which are set according to the next table:

k	fct	cb	ϵ
3	0	2.06	0.9
4	1	2.85	-
5	1	3.7	-
6	1	5.1	-
≥ 7	1	5.4	-

where k is the size of the longest clause found in the problem during parsing The parameters of probSAT have been found using automated tunning procedures included in the EDACC framework [1].

IV. Further Details

ProbSAT is implemented in C.

The solver is submitted to the Core solvers, Sequential Random SAT track. It is compiled with the Intel compiler version 12.0 using the following compiler flags: *-O3 -xhost -static -unroll-aggressive -opt-prefetch -fast*

The solver will be available online[1]

Acknowledgment

We would like to thank the BWGrid [4] project for providing the computational resources. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1.

References

[1] Balint, A. et al: EDACC - An advanced Platform for the Experiment Design, Administration and Analysis of Empirical Algorithms In: *Proceedings of LION5*, pages 586–599.
[2] Adrian Balint, Uwe Schöning: Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break Lecture Notes in Computer Science, 2012, Volume 7317, Theory and Applications of Satisfiability Testing - SAT 2012, pages 16-29
[3] Balint, A., Fröhlich, A.: Improving stochastic local search for SAT with a new probability distribution. *Proceedings of SAT 2010*, pages 10–15, 2010.
[4] bwGRiD (http://www.bw-grid.de), member of the German D-Grid initiative, funded by the Ministry for Education and Research (Bundesministerium für Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Württemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg)

[1] http://www.uni-ulm.de/en/in/institute-of-theoretical-computer-science/m/balint.html

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

The *relback* Solver: Relevant Backjumping in CDCL Solvers

Djamal Habet
LSIS, UMR CNRS 7296
Université Aix-Marseille
Av. Escadrille Normandie Niemen
13397 Marseille Cedex 20 (France)
Djamal.Habet@lsis.org

Chu Min Li
MIS
Université de Picardie Jules Verne
Rue de l'Orée du Bois
80000 Amiens (France)
chu-min.li@u-picardie.fr

***Abstract*—This document describes the SAT solver *relback* which is a CDLC-like solver with a hybrid backtracking scheme.**

I. Main Techniques

The following description concerns the submitted solver: *relback*. This solver is based on an existing implementation of a CDLC-like solver.

Indeed, *relback* is implemented under the *glucose* solver (with *SatElite* formula simplification [1]).

In *glucose* [2] and as any other Minisat-like solver, when a conflict is reached, during the propagation phase of the enqueued literals, the First UIP [3] is used in order to learn a clause and define a backjumping level.

The main purpose of our solver is to modify, under some conditions, this backjumping mechanism. Indeed, we define a new backtracking scheme based on the distance between the current empty clause and the decisions involved by this conflict.

Accordingly, the nearest one is selected and the corresponding level is defined as a backjumping one.

II. Main Parameters

We give here the use of the new backtracking scheme in the submitted solver: in *relback*, when a conflict is reached, the solver backtracks according the to nearest decision variable.

Such backtracks are applied twice at each restart achieved by the solver.

Also, each time the weight of the variables are reinitialized, the solver authorizes a new (twice) application of our backtracking scheme.

Finally, we apply the progress saving for polarity variable selection.

III. Special Algorithms, Data Structures and Features

Before handling an instance, this last one is simplified with *SatElite* [1].

The data structures used in *relback* are strictly similar to the existing ones in *glucose 2.0*. We have added the necessary ones to deal with our backtracking scheme.

IV. Implementation details

1) The programming language used is C++
2) The solver is based on *glucose 2.0* with the additional features explained above.

V. SAT Competition 2013 Specifics

1) The solver is submitted in "Core solvers, Sequential, Application SAT+UNSAT track", "Core solvers, Sequential, Hard-combinatorial SAT+UNSAT track", "Core solvers, Sequential, Application SAT track" and "Core solvers, Sequential, Hard-combinatorial SAT track".
2) The used compiler is *g++*
3) The optimization flag used is "-O3". The remaining compilation options are the same as the used ones in *glucose 2.0* and *SatElite*.

VI. Availability

Our solver will be publicly available after the SAT competition 2013.

Acknowledgment

We would like to thank the authors of *glucose*[1] for making available the source code of their solver.

References

[1] N. E. en and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *In proc. SAT?05, volume 3569 of LNCS*. Springer, 2005, pp. 61–75.

[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st international joint conference on Artificial intelligence*, ser. IJCAI'09. Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[3] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '01. IEEE Press, 2001, pp. 279–285.

[1] Available on http://www.lri.fr/∼simon/

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

The SAT Solver RISS3G at SC 2013

Norbert Manthey
Knowledge Representation and Reasoning,
TU Dresden, Germany

***Abstract*—The solver RISS3G combines the improved Minisat-style solving engine of GLUCOSE 2.2 with a state-of-the-art preprocessor COPROCESSOR and adds further modifications to the search process.**

I. INTRODUCTION

The CDCL solver RISS3G is the third generation of the SAT solver RISS. The most dramatic change is the exchange of its search engine, moving from a modular based flexible search engine that has been used for resource utilization analysis [1] to the widely used MINISAT search engine [2], more specifically to the extensions combined in GLUCOSE 2.2 [3], [4]. RISS3G is equipped with a strong preprocessor COPROCESSOR(CP3) [5], that implements most of the recently published formula simplification techniques, ready to be used as inprocessing as well.

II. MAIN TECHNIQUES

In combination with CP3, RISS3G is a CDCL solver that can also use look-ahead [6] during its search, and which can furthermore use an SLS search during preprocessing - since CP3 ships with a simple *walksat* [7].

RISS3G uses GLUCOSE 2.2 as main search engine, but enhances it with some modifications. First, if a learned clause is a unit clause, RISS3G tries to learn more unit clauses by continuing clause learning (called *all-units-learning*). Next, RISS3G provides the opportunity to update a reason clause of an implied literal, if the new reason has a better score, where score can be chosen to be the size of the clause, or the current LBD value of the clause. Then, since GLUCOSE 2.2 postpones restarts if it seems to be close to finding a model, we still trigger a restart after 20000 conflicts, and increase this limit by ten percent for the next interval. This way it is ensured that the solver still restarts sometimes – even if its internal heuristic would avoid restarts. Finally, once decision level 0 is reached, RISS3G can perform a look-ahead step based on five literals (the first five decision literals are used). From this look-ahead, the necessary assignments are added as unit clauses to the formula. Several scheduling heuristics to perform this look-ahead have been added, because always performing this procedure on decision level 0 usually slows down the search process. RISS3G is able to output proofs for unsatisfiable formulas in the DRUP format [8], [9], also when look-ahead or the all-units-learning modifications are enabled – however, CP3 does not support this format at the moment, so that the internal preprocessor should be used when a proof is necessary.

The built-in preprocessor CP3 has been ported from COPROCESSOR 2 and supports the following simplification techniques: Unit Propagation, Subsumption, Strengthening (also called self-subsuming resolution) – where for small clauses all subsuming resolvents can be produced, (Bounded) Variable Elimination (BVE) [10] combined with Blocked Clause Elimination (BCE) [11], (Bounded) Variable Addition (BVA) [12], Probing [13], Covered Clause Elimination [14], Hidden Tautology Elimination [15], Equivalent Literal Elimination [16], Unhiding (Unhide) [17], Add Binary Resolvents [18], at-most-one rewriting [19], [20], a 2SAT algorithm [21], and a walksat implementation [7]. The preprocessor furthermore supports parallel subsumption, strengthening and variable elimination, which is described in [22].

III. MAIN PARAMETERS

The main parameters control whether the preprocessor CP3 should be used as preprocessor or during search as *inprocessing*. Furthermore, the modifications to GLUCOSE 2.2 can be enabled, turning RISS3G into GLUCOSE 2.2, and vice versa.

The configuration of CP3 has been tuned for GLUCOSE 2.2 in [23] and the SAT Challenge 2012 application benchmark. The final setup for the tracks that contain both satisfiable and unsatisfiable instances uses the following techniques:

UP, SUB+STR (producing all resolvents for ternary clauses), Unhide without *hidden literal elimination* [17] and 5 iterations, BVE without on the fly BCE and BVA with a small number of 120000 steps.

Furthermore, the search is extended with the following modifications: The solver performs 4096 look-ahead steps during search as described above, and then disables this feature. Note, that for the certified tracks, CP3 has been exchanged with the internal preprocessor of GLUCOSE 2.2.

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

The implementation of the level 5 look-ahead is based on 2 bit assignments per variable. 32 of these assignments fit exactly into a 64 bit integer, so that for the look-ahead itself only an array of 64 bit integers is necessary.

The implementation of the preprocessor extends the information in the header of a clause. For the following three kinds of information, extra flags have been added to the clause: (i) being *locked*, (ii) being able to subsume other clauses, and (iii) being able to strengthen other clauses. When new clauses are added to the formula, the latter two are only an approximation. Furthermore, spin locks for each variable are introduced. The

parallelization of the preprocessor is achieved by a thread pool in the background, hidden behind a simple interface.

V. Implementation details

The solver RISS3G is build on top of MINISAT 2.2 and GLUCOSE 2.2. Furthermore, we integrated COPROCESSOR into the system, allowing inprocessing techniques to be executed during search. RISS3G has been compiled with the GCC C++compiler as a 64 bit binary.

VI. Availability

The source code of RISS3G is available at tools.computational-logic.org for research purposes.

Acknowledgment

The author would like to thank Armin Biere for many helpful discussions on formula simplification, Kilian Gebhardt for implementing the sequential and parallel algorithms of BVE and Adrian Balint for fruitful discussions on automated tuning and for tuning the preprocessor on the BWGrid [24] project. Finally, the author would like to thank TU Dresden for providing the computational resources to develop, test and evaluate RISS3G.

References

[1] S. Hölldobler, N. Manthey, and A. Saptawijaya, "Improving resource-unaware sat solvers," in *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 519–534. [Online]. Available: http://dl.acm.org/citation.cfm?id=1928380.1928417

[2] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. 21st Int. Joint Conf. on Artifical Intelligence (IJCAI '09)*. Morgan Kaufmann, 2009, pp. 399–404.

[4] ——, "Refining restarts strategies for sat and unsat," in *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming*, ser. CP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 118–126. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33558-7_11

[5] N. Manthey, "Coprocessor 2.0: a flexible cnf simplifier," in *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 436–441. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31612-8_34

[6] M. J. H. Heule and H. van Maaren, *Look-Ahead Based SAT Solvers*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 5, pp. 155–184.

[7] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *AAAI*, B. Hayes-Roth and R. E. Korf, Eds. AAAI Press / The MIT Press, 1994, pp. 337–343.

[8] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for cnf formulas," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 10 886–. [Online]. Available: http://dl.acm.org/citation.cfm?id=789083.1022836

[9] M. Heule, "Drup checker," http://www.cs.utexas.edu/~marijn/drup/.

[10] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 61–75. [Online]. Available: http://dx.doi.org/10.1007/11499107_5

[11] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 129–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_10

[12] N. Manthey, M. J. H. Heule, and A. Biere, "Automated reencoding of boolean formulas," in *Proceedings of Haifa Verification Conference 2012*, 2012.

[13] I. Lynce and J. Marques-Silva, "Probing-Based Preprocessing Techniques for Propositional Satisfiability," in *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ser. ICTAI '03. IEEE Computer Society, 2003, pp. 105–110. [Online]. Available: http://portal.acm.org/citation.cfm?id=951951.952290

[14] M. Heule, M. Järvisalo, and A. Biere, "Covered clause elimination," *CoRR*, vol. abs/1011.5202, 2010.

[15] M. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for cnf formulas," in *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 357–371. [Online]. Available: http://dl.acm.org/citation.cfm?id=1928380.1928406

[16] A. V. Gelder, "Toward leaner binary-clause reasoning in a satisfiability solver," *Ann. Math. Artif. Intell.*, vol. 43, no. 1, pp. 239–253, 2005.

[17] M. J. H. Heule, M. Järvisalo, and A. Biere, "Efficient cnf simplification based on binary implication graphs," in *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, ser. SAT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 201–215. [Online]. Available: http://dl.acm.org/citation.cfm?id=2023474.2023497

[18] W. Wei and B. Selman, "Accelerating random walks," in *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, ser. CP '02. London, UK, UK: Springer-Verlag, 2002, pp. 216–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=647489.727142

[19] N. Manthey and P. Steinke, "Quadratic Direct Encoding vs. Linear Order Encoding," in *First International Workshop on the Cross-Fertilization Between CSP and SAT(CSPSAT'11)*, 2011.

[20] M. N. V. Van Hau Nguyen and S. Hölldobler, "Application of hierarchical hybrid encoding to efficient translation of a csp to sat," Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, Tech. Rep., 2013.

[21] A. del Val, "On 2-sat and renamable horn," in *AAAI/IAAI*, H. A. Kautz and B. W. Porter, Eds. AAAI Press / The MIT Press, 2000, pp. 279–284.

[22] K. Gebhardt and N. Manthey, "Parallel Variable Elimination on CNF Formulas," in *Pragmatics of SAT*, 2013.

[23] A. Balint and N. Manthey, "Boosting the Performance of SLS and CDCL Solvers by Preprocessor Tuning," in *Pragmatics of SAT*, 2013.

[24] bwGRiD (http://www.bw grid.de/), "Member of the german d-grid initiative, funded by the ministry of education and research (bundesministerium für bildung und forschung) and the ministry for science, research and arts baden-wuerttemberg (ministerium für wissenschaft, forschung und kunst baden-württemberg)," Universities of Baden-Württemberg, Tech. Rep., 2007-2010.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

RSeq: Solver Description

Chu Min LI*†, Jiang Hua* and Djamal Habet‡
*Huazhong University of Science and Technology, China
†MIS,Universit de Picardie Jules Verne, France
‡LSIS, Univ. Aix-Marseille, France

***Abstract*—This document describes the SAT solver *RSeq*, a two-engine SAT solver based on *Sattime2013* and *Relback*.**

I. Introduction

RSeq is a two-engine SAT solver combining *Sattime2013* with *Relback*. *Sattime2013* is a Stochastic Local Search (SLS) algorithm based on *Sattime2012* [1]. In SAT challenge 2012, *Sattime2012* was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [2]. *Relback* is a CDCL-based solver due to D. Habet and C.M. LI, which is implemented by modifying the backtracking of the *Glucose 2.0* solver of G. Audemard and L. Simon [3]. In SAT challenge 2012, *Relback* was one of the best single-engine solvers in Hard Combinatorial SAT+UNSAT category [2].

We believe that each solver has its own superiority in solving different problems. *Sattime* and *Relback* should be complementary to solve different problems. In order to solve an instance, *RSeq* calls *Sattime* and *Relback* sequentially: *Sattime* is started first with a time limit. If the time limit is exceeded and a solution is not found, *Sattime* will be killed, then *Relback* is started to continue solving the instance. The starting process is controlled by a unix shell script.

II. Main Parameters

Sattime2013 is a new version of *Sattime*. Please see the description of *Sattime2013* in this book. *Sattime2013* uses the following parameters: -cutoff a, -tries b, -seed c, -nbsol d, allowing to run b times Sattime2013 for at most a steps each time, the random seed of the first run being c, to search for d solutions of the input instance. In the version submitted to the competition, a=2000000000, b=1000, and d=1.

Relback is the same version as in Sat challenge 2012. See the description of *Relback* in this book.

III. SAT Competition 2013 Specifics

RSeq is submitted to the sequential SAT and SAT+UNSAT category of Application and Hard-combinatorial instances. *Sattime2013* is compiled using the *icc* compiler using the "-O3 -static" flag. *Relback* is compiled using *g++* with the optimization flag "-O3". *RSeq* should be called in the competition using:

./Rseq.sh INSTANCE -s SEED cutofftime

where "cutofftime" is the time limit within which Sattime can run. In the competition, cutofftime is equal to 2500 seconds.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 61070235)

References

[1] C. M. LI and Y. LI, "Satisfying versus falsifying in local search for satisfiability," in *Proceedings of SAT-2012*. Springer, 2012.
[2] A. Balint, A. Belov, M. Jrvisalo, and C. Sinz, in *http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html*, 2012.
[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st international joint conference on Artificial intelligence*, ser. IJCAI'09. Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Sat4j for SAT Competition 2013

Daniel Le Berre
CRIL-CNRS UMR 8188 Université d'Artois
Lens, FRANCE

***Abstract*—The version of Sat4j submitted to the SAT Competition 2013 is a specific version of Sat4j 2.3.4 with minor modifications to meet the SAT competition 2013 requirements.**

I. Introduction

Sat4j (http://www.sat4j.org/) is an open source library of boolean satisfaction and optimization engines which aims at allowing Java programmers to access cross-platform SAT technology. Sat4j is more than a solver, it is a whole library dedicated to SAT technology: it contains SAT, Pseudo-Boolean, MAXSAT and MUS solvers and many utility classes to simplify the creation of constraints and provide efficient CNF translations for some non clausal constraints.

II. Main Techniques

The generic and flexible SAT engine available in Sat4j is based on the original Minisat 1.x implementation [1]: the generic conflict driven clause learning engine and the variable activity scheme have not changed. Most of the key components of the solver have been made configurable. See [2] for details. The default configuration of Sat4j is as follows.

The *dynamic restarts* strategy and the learned clause database management are the ones proposed by Gilles Audemard and Laurent Simon in Glucose 2.1 [3].

The *conflict clause minimization* of Minisat 1.14 (so called Expensive Simplification)[4] is used at the end of the conflict analysis. Note that our implementation is a generalized version of the original minimization procedure from minisat: it works for other data structures than clauses with watched literals. As such, it is slightly less efficient than the original one.

When the solver selects a variable to branch on, it uses a phase selection strategy implementing the *lightweight caching scheme* of RSAT[5].

Note that unlike most other SAT solvers, Sat4j does not use any preprocessor.

Sat4j allows to build ManySAT like parallel solvers with sharing of unit clauses, as in plingeling (new in Sat4j 2.3.4).

III. Main Parameters

The solver only contains two parameters:
`java [-DUNSATPROOF=filename] -jar sat4j2013.jar [Parallel] file.cnf`.

1) `-DUNSATPROOF=fileproofname` allows the solver to produce an unsat proof in a specific file.
2) `Parallel` is the name of a predefined parallel solver in the library.

IV. Implementation details

Sat4j 2.3.4 release allows solvers running in parallel to share derived unit clauses (as in plingeling). Those unit clauses are imported each time the solver restarts (not when the decision level is zero). As such, agile solvers are more likely to import new units than ones making few restarts.

Sat4j 2.3.4 also allows to produce RUP proofs, by means of a specific "listener" : most events in the CDCL engine can be hooked so a listener simply writes down in a file all clauses derived by the solver.

V. SAT Competition 2013 Specifics

Sat4j is submitted to all non random tracks of the SAT Competition 2013. Specific memory parameters for Oracle JVM are required to use the 16GB of memory. We set the JVM to use 5GB of stack size for the sequential tracks and 12GB of stack size for the parallel tracks. Note that the exact total amount of memory used by the JVM can hardly be predicted, and may change depending on the version of the JVM used, the host operating system or its architecture (32 or 64 bits).

The following configurations are used by Sat4j (Base = Expensive clause minimization from Minisat 1.14 + RSAT phase saving + Glucose 2.0 LBD based aggressive clause deletion strategy) :

- Default Glucose 2.1 like solver (Glucose 2.1 dynamic restarts).
- Luby Luby style restarts with factor 100.
- Unsat Minisat geometric restarts, no phase saving (always branch on the negative phase first).
- Unsat' Unsat + keep learnt clauses as much as possible (until memory gets low).
- LS Restarts every 1000 conflicts, heuristics with random walk of 0.1
- LS2 Restarts every 500 conflicts, pick randomly an unassigned variable and the phase to branch on.
- Luby' Luby + keep learnt clauses as much as possible (until memory gets low).
- Biere In/Out restart strategy found in Picosat [6].

The `Default` solver is used in the sequential tracks, the `Unsat` solver is used in the certified unsat tracks while the height above solvers are running in parallel with exchange of derived unit clauses in the parallel tracks.

The solver submitted to the competition is thus launched that way:

1) `java -Xms5g -Xmx5g -jar sat4j2013.jar file.cnf` for sequential tracks
2) `java -DUNSATPROOF=TEMPDIR/proof.txt -Xms5g -Xmx5g -jar sat4j2013.jar Unsat file.cnf` for certified unsat tracks
3) `java -Xms12g -Xmx12g -jar sat4j2013.jar Parallel file.cnf` for parallel tracks

VI. Availability

Sat4j is developed using both Java and open source standards: the project is supported by the OW2 consortium infrastructure and is released under both the EPL and the GNU LGPL licenses. It is available from http://www.sat4j.org/

Acknowledgment

Part of this work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013'.

References

[1] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, 2003, pp. 502–518.

[2] D. Le Berre and A. Parrain, "The sat4j library, release 2.2," *JSAT*, vol. 7, no. 2-3, pp. 59–6, 2010.

[3] G. Audemard and L. Simon, "Refining restarts strategies for sat and unsat," in *CP*, ser. Lecture Notes in Computer Science, M. Milano, Ed., vol. 7514. Springer, 2012, pp. 118–126.

[4] N. Sörensson and A. Biere, "Minimizing learned clauses," in *SAT*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 237–243.

[5] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *SAT*, ser. Lecture Notes in Computer Science, J. Marques-Silva and K. A. Sakallah, Eds., vol. 4501. Springer, 2007, pp. 294–299.

[6] A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Description of Sattime2013

Chu Min LI
Huazhong University of Science and Technology, China;
MIS, Universit de Picardie Jules Verne, France

Yu Li
Universit de Picardie Jules Verne, France

***Abstract*—This document describes the SAT solver "Sattime2013", a stochastic local search algorithm for SAT exploiting the satisfying history of the unsatisfied clauses during search to select the next variable to flip in each step.**

I. Introduction

In SAT challenge 2012, Sattime2012 was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [1]. Sattime2013 is a new version of Sattime [2] that enables more greediness than Sattime2012 by extending the notion of promising decreasing variables [3].

II. Main Techniques

During local search, clauses may frequently be satisfied or falsified. Modern SLS algorithms often exploit the falsifying history of clauses to select a variable to flip, together with variable properties such as score and age. The score of a variable x refers to the decrease in the number of unsatisfied clauses if x is flipped. The age of x refers to the number of steps done since the last time when x was flipped.

Novelty [4] and Novelty based SLS algorithms such as $Novelty+$ [5] and $Novelty++$ [3] consider the youngest variable in a randomly chosen unsatisfied clause c, which is necessarily the last falsifying variable of c whose flipping made c from satisfied to unsatisfied. If the best variable according to scores in c is not the last falsifying variable of c, it is flipped, otherwise the second best variable is flipped with probability p, and the best variable is flipped with probability 1-p. TNM [6], [7] extends Novelty by also considering the second last falsification of c, the third last falsification of c, and so on... If the best variable in c most recently and consecutively falsified c several times, TNM considerably increases the probability to flip the second best variable of c.

Another way to exploit the falsifying history of clauses is to define the weight of a clause to be the number of local minima in which the clause is unsatisfied, so that the objective function is to reduce the total weight of unsatisfied clauses.

Sattime uses a new heuristic by considering the satisfying history of clauses instead of their falsifying history, and by modifying Novelty as follows: If the best variable in c is not *the most recent satisfying variable* of c, flip it. Otherwise, flip the second best variable with probability p, and flip the best variable with probability 1-p. Here, the most recent **satisfying** variable in c is the variable whose flipping most recently made c from unsatisfied to satisfied. The intuition of the new heuristic is to avoid repeatedly satisfying c using the same variable.

Given a SAT instance ϕ to solve, Sattime2013 first generates a random assignment and while the assignment does not satisfy ϕ, it modifies the assignment as follows:

1) If there are promising decreasing variables, flip the oldest one;
2) If there are enforced decreasing variables, flip the oldest one;
3) Otherwise, randomly pick an unsatisfied clause c;
4) With probability wp, flip randomly a variable in c; With probability 1-wp, sort the variables in c according to their score and consider the best and second best variables in c (breaking tie in favor of the least recently flipped one). If the best variable is not the most recent **satisfying** variable of c, then flip it. Otherwise, with probability p, flip the second best variable, and with probability 1-p, flip the best variable.

A satisfying variable of a clause is the variable whose flipping made the clause from unsatisfied to satisfied. Probability p is adapted according to the improvement in the number of unsatisfied clauses during search according to [8], and wp=p/10.

The notion of promising decreasing variable was defined in [3], referring to those variables whose score is positive and became positive not by flipping themselves. For example, let x be a variable and score(x)<0, after flipping x, score(x) becomes positive, i.e., decreasing, then x is not promising. If score(x)$\leq$0, but after flipping another variable y, score(x) becomes positive, x is promising and will keep to be promising as long as its score is positive. Promising decreasing variables have the highest priority to be flipped.

In Sattime2013, we introduce another notion: if score(x) is positive, and its score is increased after flipping another variable, then x is called an *enforced* decreasing variable. Note that an enforced decreasing variable may or may not be promising. A decreasing variable x is enforced after another variable y is flipped, either because a clause containing it becomes falsified so that the make value of x is increased, or a clause only satisfied by x now is also satisfied by y so that the break value of y is decreased. One of enforced decreasing variables (if any) is flipped if there is not any promising decreasing variable.

III. Main Parameters

As Sattime2012, Sattime2013 uses Hoo's adaptive noise mechanism that uses two parameters, Φ and Θ. The performance of Sattime2013 is not very sensitive to the variation in the value of these parameters, In Sattime2013 as in Sattime2012, Φ=10 and Θ=5.

Other parameters include: -cutoff a, -tries b, -seed c, -nbsol d allowing to run b times Sattime2013 for at most a steps each time, the random seed of the first run being c to search for d solutions of the input instance. In the version submitted to the competition, a=2000000000, b=1000, and d=1.

IV. Special Algorithms, Data Structures, and Other Features

Sattime2013 uses the same data structures as Satz [9], [10]. It uses a preprocessing inherited from Satz to simplify the input formula by propagating all unit clauses and detecting all failed literals in the input formula, which may prove the unsatisfiability of the input instance.

V. Implementation details

Sattime2013 is implemented in C and is based on g2wsat [3].

VI. SAT Competition 2013 Specifics

Sattime2013 is an single engine solver and is submitted to all the sequential tracks on Application, Hard-combinatorial, and Random instances. Because of the preprocessing, Sattime2013 may prove the unsatisfiability of an instance.

Sattime2013 is compiled as 64-bit binary using the intel compiler as follows:

icc sattime2013.c -O3 -static -o sattime2013

Sattime2013 should be called in the competition using

sattime2013 INSTANCE -seed SEED -nbsol 1

to solve the input instance INSTANCE, where SEED can be any positive integer. If "-seed SEED" is not specified, sattime2013 also works, but it will be difficult to reproduce the same execution of sattime2013 for the input instance.

VII. Availability

The codes sources of Sattime2013 will be available for research purpose after the competition 2013 at http://home.mis.u-picardie.fr/˜cli/EnglishPage.html

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 61070235)

References

[1] A. Balint, A. Belov, M. Jrvisalo, and C. Sinz, in *http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html*, 2012.

[2] C. M. LI and Y. LI, "Satisfying versus falsifying in local search for satisfiability," in *Proceedings of SAT-2012, to appear*. Springer, 2012.

[3] C. M. Li and W. Q. Huang, "Diversification and Determinism in Local Search for Satisfiability," in *Proceedings of SAT2005*, 2005, pp. 158–172.

[4] D. McAllester, B. Selman, and H. Kautz, "Evidence for invariant in local search," in *Proceedings of AAAI-97*, 1997, pp. 321–326.

[5] H. Hoos, "On the run-time behavior of stochastic local search algorithms for sat," in *Proceedings of AAAI-99*, 1999, pp. 661–666.

[6] C. M. LI, W. Wei, and Y. LI, "Exploiting historical relationships of clauses and variables in local search for satisfiability," in *Proceedings of SAT-2012*. Springer, 2012, pp. 479–480.

[7] W. Wei and C. Li, "Switching between two adaptive noise mechanisms in local search for sat," in *TNM solver description for the SAT 2009 competition, http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf*, 2009.

[8] H. Hoos, "An adaptive noise mechanism for walksat," in *Proceedings of AAAI-02*. AAAI Press / The MIT Press, 2002, pp. 655–660.

[9] C. M. LI and Anbulagan, "Heuristics based on unit propagation for satisfiability problems," in *Proceedings of 15th International Joint Conference on Artificial Interlligence (IJCAI'97), Morgan Kaufmann Publishers, ISBN 1-55860-480-4*, 1997, pp. 366–371.

[10] C. Li and Anbulagan, "Look-Ahead Versus Look-Back for Satisfiability Problems," in *Proceedings of CP-97, Third International Conference on Principles and Practice of Constraint Programming*. Springer-Verlag, LNCS 1330, Shloss Hagenberg, Austria, 1997, pp. 342–356.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Description of SattimeClasp

Chu Min LI*†, Jiang Hua* and Xu Ruchu*
*Huazhong University of Science and Technology, China;
†MIS,Universit de Picardie Jules Verne, France

***Abstract*—This document describes the SAT solver SattimeClasp, a SAT solver based on sattime2013 and clasp2.0.**

I. Introduction

SattimeClasp is a SAT solver which combines sattime2013 with clasp2.0. Sattime2013 is a Stochastic Local Search (SLS) algorithm based on sattime2012 [1]. In SAT challenge 2012, sattime2012 was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [2]. Clasp 2.0 (R4092) is a conflict learning asp solver distributed under the GNU Public License [3]. In SAT Competition 2011, clasp has shown excellent performance in the crafted (hard combinatorial problems) category [4].

We believe that each solver has its own superiority in solving different problems. Sattime and clasp should be complementary to solve different problems. In order to solve a problem, SattimeClasp calls sattime and clasp sequentially: sattime is started first with a time limit. If the time limit is exceeded and a solution is not found, sattime will be killed and clasp is started to continue solving the instance. The starting process is controlled by a unix shell script.

II. Main Parameters

Sattime2013 is a new version of sattime. Please see the description of Sattime2013 in this book. Sattime2013 uses the following parameters: -cutoff a, -tries b, -seed c, -nbsol d, allowing to run b times Sattime2013 for at most a steps each time, the random seed of the first run being c, to search for d solutions of the input instance. In the version submitted to the competition, a=2000000000, b=1000, and d=1.

The parameters of clasp are set to it's default values.

III. SAT Competition 2013 Specifics

SattimeClasp is submitted to the sequential SAT and SAT+UNSAT category of Application, Hard-combinatorial and Random instances. Sattime is compiled as a 64-bit binary using the intel icc compiler and clasp is compiled using gcc compiler.

SattimeClasp should be called in the competition using:

./SCSeq.sh INSTANCE -seed SEED cutofftime

The parameter cutofftime specifies the time limit to sattime. In the testing stage, the time to each instance is limited to 1200 seconds, so cutofftime=600. In the competition stage, cutofftime=2500.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 61070235)

References

[1] C. M. LI and Y. LI, "Satisfying versus falsifying in local search for satisfiability," in *Proceedings of SAT-2012*. Springer, 2012.
[2] A. Balint, A. Belov, M. Jrvisalo, and C. Sinz, in *http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html*, 2012.
[3] A. N. Martin Gebser, Benjamin Kaufmann and T. Schaub, "A conflict-driven answer set solver," in *LPNMR'07*, 2007.
[4] M. J. D. L. B. O. Roussel, "The sat 2011 competition results part 2," in *http://www.cril.univ-artois.fr/SAT11/phase2.pdf*, 2011.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Description of SattimeRelbackSeq

Chu Min LI*†, Jiang Hua* and Djamal Habet‡
*Huazhong University of Science and Technology, China
†MIS,Universit de Picardie Jules Verne, France
‡LSIS, Univ. Aix-Marseille, France

***Abstract*—This document describes the SAT solver SattimeRelbackSeq, a two-engine SAT solver based on Sattime2013 and Relback.**

I. Introduction

SattimeRelbackSeq is a two-engine SAT solver combining Sattime2013 with Relback. Sattime2013 is a Stochastic Local Search (SLS) algorithm based on Sattime2012 [1]. In SAT challenge 2012, Sattime2012 was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [2]. Relback is a CDCL-based solver due to D. Habet and C.M. LI, which is implemented by modifying the backtracking of the *Glucose* solver of G. Audemard and L. Simon [3]. In SAT challenge 2012, Relback was one of the best single-engine solvers in Hard Combinatorial SAT+UNSAT category [2].

We believe that each solver has its own superiority in solving different problems. Sattime and Relback should be complementary to solve different problems. In order to solve a problem instance, SattimeRelbackSeq calls Sattime and Relback sequentially: Sattime is started first with a time limit. If the time limit is exceeded and a solution is not found, Sattime will be killed, then SatElite is called to simplify the instance before Relback is started to continue solving the simplified instance. The starting process is controlled by a unix shell script.

II. Main Parameters

Sattime2013 is a new version of Sattime. Please see the description of Sattime2013 in this book. Sattime2013 uses the following parameters: -cutoff a, -tries b, -seed c, -nbsol d, allowing to run b times Sattime2013 for at most a steps each time, the random seed of the first run being c, to search for d solutions of the input instance. In the version submitted to the competition, a=2000000000, b=1000, and d=1.

Relback is the same version as in satchallenge 2012. See the description of Relback in this book.

III. SAT Competition 2013 Specifics

SattimeRelbackSeq is submitted to the sequential SAT and SAT+UNSAT category of Application, Hard-combinatorial and Random instances. Sattime2013 is compiled using the icc compiler using the "-O3 -static" flag. Relback is compiled using *g++* with the optimization flag "-O3".

SattimeRelbackSeq should be called in the competition using:

./sattimeRelbackSeq INSTANCE -seed SEED cutofftime -tmp TMPDIR

where "cutofftime" is the time limit within which Sattime can run. "TMPDIR" is a temporary directory in which SatElite writes temporary files. In the competition, cutofftime is equal to 2500 seconds.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 61070235)

References

[1] C. M. LI and Y. LI, "Satisfying versus falsifying in local search for satisfiability," in *Proceedings of SAT-2012*. Springer, 2012.
[2] A. Balint, A. Belov, M. Jrvisalo, and C. Sinz, in *http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html*, 2012.
[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st international joint conference on Artificial intelligence*, ser. IJCAI'09. Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Description of SattimeRelbackShr

Chu Min LI*†, Jiang Hua* and Djamal Habet‡
*Huazhong University of Science and Technology, China
†MIS,Universit de Picardie Jules Verne, France
‡LSIS, Univ. Aix-Marseille, France

***Abstract*—This document describes the SAT solver SattimeRelbackShr, a SAT solver based on sattime2013 and relback.**

I. Introduction

SattimeRelbackShr is a two engine SAT solver which combines sattime2013 with relback. Sattime2013 is a Stochastic Local Search (SLS) algorithm based on sattime2012 [1]. In SAT challenge 2012, sattime2012 was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [2]. Relback is a CDLC like solver,which is implemented under the *glucoe* solver without SatElite formula simplification[3]. In SAT challenge 2012, relback was one of the best single-engine solver in Hard Combinatorial SAT+UNSAT category [2].

We believe that each solver has its own superiority in solving different problems. Sattime and relback should be complementary to solve different problems. In order to solve a problem, SattimeRelbackShr create two threads to run sattime and relback engine respectively.When one engine find a solution, the other one will be cutoff and only one solution will be reported.

II. Main Parameters

Sattime2013 is a new version of sattime. Please see the description of Sattime2013 in this book. Sattime2013 uses the following parameters: -cutoff a, -tries b, -seed c, -nbsol d, allowing to run b times Sattime2013 for at most a steps each time, the random seed of the first run being c, to search for d solutions of the input instance. In the version submitted to the competition, a=2000000000, b=1000, and d=1.

About relback ,please see the description of relback in this book.

III. SAT Competition 2013 Specifics

SattimeRelbackShr is submitted to the sequential SAT and SAT+UNSAT category of Application, Hard-combinatorial and Random instances. The used compiler is *g++* with optimization flag "-O3".

SattimeRelbackShr should be called in the competition using:

./sattimeRelbackShr INSTANCE -seed SEED

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 61070235)

References

[1] C. M. LI and Y. LI, "Satisfying versus falsifying in local search for satisfiability," in *Proceedings of SAT-2012*. Springer, 2012.
[2] A. Balint, A. Belov, M. Jrvisalo, and C. Sinz, in *http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html*, 2012.
[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st international joint conference on Artificial intelligence*, ser. IJCAI'09. Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

satUZK: Solver Description

Alexander van der Grinten*, Andreas Wotzlaw*, Ewald Speckenmeyer*, Stefan Porschen†
*Institut für Informatik
Universität zu Köln, Pohligstr. 1, D-50969 Köln, Germany
Email: {vandergrinten,wotzlaw,esp}informatik.uni-koeln.de
†Fachgruppe Mathematik, FB4
HTW-Berlin, Treskowallee 8, D-10318 Berlin, Germany
Email: porschen@htw-berlin.de

Solver Description

satUZK is a conflict-driven clause learning solver for the boolean satisfiability problem (SAT). It is written in C++ from scratch and aims to be flexible and easily extendable.

In addition to the standard DPLL [1] algorithm with clause learning the solver is able to perform various preprocessing and inprocessing techniques.

Preprocessing

We implemented SatELite-like variable elimination and self-subsumption [2], unhiding [3], a distillation technique similar to the one presented in [4], blocked clause elimination [5] and variable probing to detect failed literals, equivalent literals and literals that must be true in every model.

The preprocessing starts with unhiding, followed by self-subsumption and variable probing in order to fix some variables and increase the number of literals that can be propagated by binary constraint propagation (BCP).

After that the size of the formula is reduced by blocked clause elimination and SatELite-like variable elimination. These techniques can reduce the reasoning power of BCP and that is why they are scheduled after the previous preprocessing steps.

Preprocessing generally tries to eliminate 0.5% of the remaining variables in 1% of the available time (which is 900 seconds in this case). All preprocessing techniques are repeated until the number of variables that are affected by each simplification pass becomes too low or a limit of 10% of the time budget is reached.

Search

The data structures required for BCP are implemented in the same way as in MiniSAT 2.2 [6]. Binary clauses are stored in a separate watch list.

We are using the standard 1-UIP [1] learning scheme together with conflict clause minimization and the VSIDS decision heuristic with phase saving.

We submitted a version of our solver using a MiniSAT-like learned clause deletion strategy and a version with a more aggressive literal blocks distance based deletion strategy [7].

Both Luby restarts and glucose-like dynamic restarts are implemented [7].

Inprocessing

The DPLL procedure is interleaved with inprocessing steps that perform unhiding, variable probing and distillation. These techniques do not require literal occurrence lists and thus they can be integrated into the search without great performance overheads.

Variable probing and distillation is only applied to the most active variables and clauses.

At most 10% of the available time is used for inprocessing.

References

[1] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*, 2009.

[2] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, ser. Lecture Notes in Computer Science, vol. 3569, 2005, pp. 61–75.

[3] M. Heule, M. Järvisalo, and A. Biere, "Efficient cnf simplification based on binary implication graphs," in *Proceedings to the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT 2011)*, ser. Lecture Notes in Computer Science, vol. 6695, 2011, pp. 201–215.

[4] H. Han and F. Somenzi, "Alembic: An efficient algorithm for cnf preprocessing," in *Proceedings of the 44th Design Automation Conference (DAC 2007)*, 2007, pp. 582–587.

[5] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, ser. Lecture Notes in Computer Science, vol. 6015, 2010, pp. 129–144.

[6] N. Eén and N. Sörensson, "Minisat 2.2." [Online]. Available: http://minisat.se/downloads/minisat-2.2.0.tar.gz

[7] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 2009, pp. 399–404.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Parallel SAT Solver SatX10-GlCi 1.1

Benjamin Herta*, Ashish Sabharwal*, Horst Samulowitz*, Vijay Saraswat*, George Katsirelos†, Laurent Simon‡
*IBM Watson Research Center
New York, USA
{groved,bherta,ashish.sabharwal,samulowitz,vsaraswa}@us.ibm.com
†MIAT, INRA
Toulouse, France
george.katsirelos@toulouse.inra.fr
‡Univ. Paris-Sud, LRI/CNRS UMR8623
Orsay, F-91405, France
simon@lri.fr

`SatX10-GlCi` is an instantiation of a parallel SAT solver built using the `SatX10` framework [3] which is available at: http://x10-lang.org/x10-community/x10-in-use/applications/207-satx10.html.

The `SatX10` framework is based on the `X10` programming language designed specifically for programming on multi-core and clustered systems easily [5]. The framework provides various facilities to conveniently run algorithms (here SAT solvers) in parallel, along with a communication infrastructure.

Version 1.1 of this solver participated in the Parallel 2-core Solvers application track of the SAT Competition 2013.

I. Solving techniques

`SatX10-GlCi` is composed of 2 core `MiniSat`-based conflict directed clause learning SAT solvers:

1) `Circuit MiniSat` [4]
2) `Glucose 2.1` [1]

The `X10` framework is used to both launch multiple solvers and to enable communication of information between them. In the current configuration, every solver sends all learned clauses of a fixed maximum length to all other solvers, which incorporate these clauses either during their search or at restart points. Thus, information is shared using an implicit clique topology. Note that in general the communication amount, frequency, as well as network structure can take arbitrary form in `SatX10`.

II. Implementation details

The solver `SatX10-GlCi` is built using the `SatX10` framework [3], heavily utilizing the mechanisms it provides for incorporating new solvers and sharing information amongst solvers. The version of the `X10` language used was 2.3.1 and information sharing performed using TCP/IP sockets backend of `X10`. Each individual SAT solver is embedded in the parallel solver at the source code level, which was modified appropriately to adhere to the requirements of the `SatX10` framework. The solvers themselves were compiled into object files using GNU g++ 4.8.0 using option "-O3" and then linked into the C++ backend of `X10`. The flags used for the compiler x10c++ were "-STATIC_CHECKS -NO_CHECKS -O". The resulting *single* binary executable is then launched with environment variable X10_NTHREADS set to 1, X10_STATIC_THREADS set to true, and X10_NPLACES set to the desired number of solvers to launch. (The same executable can be used to run the solver on multiple machines as well, by specifying a list of hostnames.) The amount of clause sharing is controlled with a parameter specifying the maximum length upto which clauses are shared with other solvers.

III. SAT Competition 2013 specific details

The launcher script of `SatX10` is written in Python 2.6 and the command line used in the competition is as follows:

```
python runSatX10.py --x10nplaces=8
--x10maxlen=-10 --x10outbuf=100
--x10usepre --x10ppconfig=SC2013.ppconfig
```

The parameters define how many cores to use (8), what the maximal length of shared clauses is, the size of the clause buffer before sharing (100), to apply preprocessing, and what configuration to run (`SC2013.ppconfig`). The negative clause length indicates that the maximal shared clause length is adaptively changed with the aim of sharing a fixed percentage of the clauses learned in total (e.g., 5%).

In the used plug&play configuration `Glucose 2.1` is executed on 6 cores and `Circuit MiniSat` on 2 cores. All solvers are launched with the parameter `-verb=0`. For detailed parameter settings please refer to the `SC2013.ppconfig` file.

Preprocessing is applied to problem instances prior to execution of `SatX10`. The preprocessor SatELite [2] was modified to not map variables numbers and to explicitly append unit clauses, when possible, for variables it would have eliminated otherwise. Preprocessing is terminated if it exceeds 100 seconds. If the parallel execution fails for any reason, `Glucose 2.1` is invoked on a single core.

Acknowledgment

We express sincere thanks to the developers of the various SAT solvers whose source code served as the baseline for integration into `SatX10`.

REFERENCES

[1] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. *IJCAI*, 399–404, 2009.

[2] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.

[3] B. Bloom and D. Grove and B. Herta and A. Sabharwal and H. Samulowitz and V. Saraswat. SatX10: A Scalable Plug&Play Parallel Solver (Tool paper). *SAT*, 2012.

[4] Circuit Minisat Solver Description. SAT Competition 2011.

[5] Saraswat, Vijay and Bloom, Bard and Peshansky, Igor and Tardieu, Olivier and Grove, David Report on the Experimental Language, X10. Technical Report,*http://x10-lang.org/*, 2011.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

SINNminisat

Takeru Yasumoto
Kyushu University, Japan
yasumoto.kyushu@gmail.com

Takumi Okugawa
Kyushu University, Japan
okugawa.takumi@gmail.com

I. Introduction

SINNminisat is based on MiniSat2.2.0[1]. The SINNminisat system employs TrueLBD which is a kind of LBD and Aggressive Reduce Database.

II. MAIN TECHNIQUES

A. True LBD

True LBD is a kind of LBD[**?**]. True LBD is different from LBD in the manner of updating its value. It ignores literals assinged at level 0.

B. Aggressive Reduce Database

Aggressive Reduce Database is a learnt clause management method. This method is implemented in GlueMiniSat2.2.5[3].

Acknowledgment

I wish to express my gratitude to Mr.Hasegawa, Mr.Fujita, Mr.Koshimura for valuable advices and comments.

References

[1] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In SAT 2003, 2003.
[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In proceedings of IJCAI, 2009.
[3] Hidetomo NABESHIMA, Koji IWANUMA, KAtsumi INOUE. GLUEMINISAT2.2.5 , SAT 2011 competition System Description.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Solver43

Valeriy Balabanov
National Taiwan University
Taipei, Taiwan

I. Introduction

In the past decade SAT-solving techniques have been polished to precise excellence, and modern solvers can deal with circuits containing millions of clauses. Preprocessing techniques, such as those related to pure-literals elimination, clause subsumption and variable elimination have been known for quite a long time [1]. More recent preprocessing technique concerning blocked-clauses was introduced, and experimentally verified to be useful [2]. As available computational resources grow, more solving time could be spent for preprocessing.

"Solver43" is a complete SAT solver based on "Glucose" [3], but instrumented with new preprocessor that utilizes all aforementioned preprocessing techniques, and some new ones. In the next section we shall briefly describe all novelties that have been introduced.

II. New Preprocessing Techniques

Generally speaking, one could divide all sound preprocessing techniques into two categories: ones that do not change functionality of the input instance, and ones that preserve its satisfiability. Subsumption, and other resolution-related (we do not consider variable elimination here) techniques belong to the first category. Variable elimination, pure-literals elimination, and blocked-clauses elimination belong to the latter. "Solver43" is instrumented with all of them, but with some minor changes. First of all, during preprocessing we decided not to eliminate variables eagerly. Second, do not remove blocked 2-clauses. We also decided to look for simple definitions in preprocessing phase and simplify them on the fly, as it might enable more simplification by other techniques.

All aforementioned preprocessing techniques are quite well studied, and are devoted to elimination (variable and/or clause). Intuitively, elimination of variables and clauses shall decrease the resources used by the solver, and thus theoretically decrease the solving time. In practice, benefits from preprocessing can rather be seen for large populations of benchmarks, and in some particular cases preprocessing slows down the solver.

On the other hand, slightly puzzling is the fact that during the solving phase solvers tend to add learned clauses for completeness reasons. Whether it might be useful to add variables and/or clauses in preprocessing phase remains an opened question. Mysterious influence of freshly defined variables on the size of resolution proofs (extended resolution) is one simple example in favor of such an addition [4].

In the preprocessing phase "Solver43", after applying all standard preprocessing techniques, searches for blocked 2-clauses not present in the formula. Effectively this process is the same as one step look-ahead search for pure-literals, since we just search for a literal p, assigning which to true makes given literal q pure. It is easy to mention that resulted implication clause $(\neg p \vee q)$ has a blocked literal q, verifying that this clause is blocked and thus can be freely added to the formula without changing its satisfiability. Some of the found by "Solver43" blocked 2-clauses are added then to the original clause set.

It has been experimentally observed that preprocessing techniques interfere with each other, and the order of their application might affect the result. One of such effects is that adding blocked 2-clauses increases number of found definitions.

After preprocessing phase, "Solver43" uses "Glucose" for usual SAT-solving. If "Glucose" returns UNSAT answer, no latter postprocessing is necessary to be done. In case of SAT answer, we have to modify the SAT-assignment according to the removed clauses to obtain SAT assignment for the original instance.

III. Main Parameters

Submitted to SAT'13 competitions version of "Solver43" does not provide user-controllable parameters, except of those provided by "Glucose". Internal parameters, however include preprocessing order, and some effort parameters that decide how eagerly we should apply different preprocessing techniques.

IV. Availability

"Solver43" is open source and has exactly the same license as that of "MiniSAT" [5] and "Glucose". It is publicly available and anyone could use the solver for evaluation and research purposes.

References

[1] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *SAT*, 2005, pp. 61–75.
[2] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *TACAS*, 2010, pp. 129–144.
[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, 2009, pp. 399–404.
[4] S. A. Cook, "A short proof of the pigeon hole principle using extended resolution," *SIGACT News*, vol. 8, no. 4, pp. 28–32, Oct. 1976. [Online]. Available: http://doi.acm.org/10.1145/1008335.1008338
[5] N. Eén and N. Sörensson, "Temporal induction by incremental sat solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

SPARROW+CP3 and SPARROWTORISS

Adrian Balint
Universität Ulm, Germany

Norbert Manthey
Knowledge Representation and Reasoning,
TU Dresden, Germany

***Abstract*—SPARROW+CP3 and SPARROWTORISS are using as a first step the preprocessor CP3 to simplify the formula in a way that is beneficial for SLS solvers. SPARROW+CP3 then uses the solver SPARROW to solve the simplified problem. SPARROWTORISS is first trying to solve the problem with Sparrow, limiting its execution to $5 \cdot 10^8$ flips and then passes the assignment found to the CDCL solver RISS3G, which uses this information for initialization and then tries to solve the problem. The solver RISS3G combines the improved Minisat-style solving engine of GLUCOSE 2.2 with a state-of-the-art preprocessor COPROCESSOR and adds further modifications to the search process. The SLS solver SPARROW is an improved version of SPARROW 2011**

I. INTRODUCTION

SLS solvers showed remarkable performance on the satisfiable crafted problems in the competitions from the last years. Motivated by this results we have analyzed in [1] the utility of different preprocessing techniques for the SLS solver SPARROW. The best found technique together with SPARROW represents the basis of our solver SPARROW+CP3.

As SPARROW is not able to prove the unsatisfiability of a problem we have decided to append a CDCL solver to SPARROW+CP3, namely RISS3G after limiting the execution of SPARROW to $5 \cdot 10^8$ flips. The CDCL solver RISS3G uses the MINISAT search engine [2], more specifically the extensions added in GLUCOSE 2.2 [3], [4]. Furthermore, RISS3G is equipped with the preprocessor COPROCESSOR(CP3) [5], that implements most of the recently published formula simplification techniques, ready to be used as inprocessing as well.

II. MAIN TECHNIQUES

SPARROW is a clause weighting SLS solvers that uses promising variables and probability distribution based selection heuristics. It is described in detail in [6]. Compared to the original version, the one submitted here is updating weights of unsatisfied clauses in every step where no promising variable can be found.

The built-in preprocessor CP3 has been ported from COPROCESSOR 2 and supports the following simplification techniques: Unit Propagation, Subsumption, Strengthening (also called self-subsuming resolution) – where for small clauses all subsuming resolvents can be produced, (Bounded) Variable Elimination (BVE) [7] combined with Blocked Clause Elimination (BCE) [8], (Bounded) Variable Addition (BVA) [9], Probing [10], Covered Clause Elimination [11], Hidden Tautology Elimination [12], Equivalent Literal Elimination [13], Unhiding (Unhide) [14], Add Binary Resolvents [15], at-most-one rewriting [16], [17], a 2SAT algorithm [18], and a walksat implementation [19]. The preprocessor furthermore supports parallel subsumption, strengthening and variable elimination, which is described in [20].

RISS3G uses GLUCOSE 2.2 as main search engine – the version used in SPARROWTORISS just replaces the internal preprocessor with CP3.

The combination of the SPARROW and RISS3G, called SPARROWTORISS, does not simply execute the two solvers after each other, but also forwards information from the SLS solver to the CDCL solver: when SPARROW terminates, it outputs its last full assignment in chronological order (i.e. the oldest variable first), which is used to initialize the phase saving of RISS3G, such that the first decisions of RISS3G follow this assignment. In a brief empirical evaluation this communication turned out to be useful. The solvers are also able to forward the information about the age of the variables in the SLS search. This data could be used to initialize the activities of the variables inside RISS3G. However, this feature is not enabled in the used configuration.

III. MAIN PARAMETERS

SPARROW is using the same parameters as SPARROW 2011.

The configuration of CP3 has been tuned for SPARROW in [1] on the SAT Challenge 2012 satisfiable hard combinatorial benchmarks.

The main parameters of RISS3G control how the formula simplification of CP3 is executed. The configuration of CP3 has been tuned for GLUCOSE 2.2 in [1] on the SAT Challenge 2012 application benchmark. The final setup of the preprocessor inside RISS3G uses the following techniques: UP, SUB+STR (producing all resolvents for ternary clauses), Unhide without *hidden literal elimination* [14] and 5 iterations, BVE without on the fly BCE and BVA with a small number of 120000 steps.

For SPARROWTORISS it can be chosen whether to forward the last assignment, or the activity information.

IV. IMPLEMENTATION DETAILS

SPARROW is implemented in C. The solver RISS3G is build on top of MINISAT 2.2 and GLUCOSE 2.2. Furthermore, we integrated COPROCESSOR into the system, allowing inprocessing techniques to be executed during search – however, this feature is not used in the competition. All solvers have been compiled with the GCC C++compiler as 64 bit binaries.

V. AVAILABILITY

The source code of RISS3G (including CP3) is available at tools.computational-logic.org for research purposes.

ACKNOWLEDGMENT

The authors would like to thank Armin Biere for many helpful discussions on formula simplification and the BWGrid [21] project for providing computational resources to tune CP3. This project was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1. Finally, the authors would like to thank TU Dresden for providing the computational resources to develop, test and evaluate RISS3G.

REFERENCES

[1] A. Balint and N. Manthey, "Boosting the Performance of SLS and CDCL Solvers by Preprocessor Tuning," in *Pragmatics of SAT*, 2013.

[2] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. 21st Int. Joint Conf. on Artifical Intelligence (IJCAI '09)*. Morgan Kaufmann, 2009, pp. 399–404.

[4] ——, "Refining restarts strategies for sat and unsat," in *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming*, ser. CP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 118–126. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33558-7_11

[5] N. Manthey, "Coprocessor 2.0: a flexible cnf simplifier," in *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 436–441. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31612-8_34

[6] A. Balint and A. Fröhlich, "Improving stochastic local search for sat with a new probability distribution," in *Proceedings of the 13th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 10–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14186-7_3

[7] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 61–75. [Online]. Available: http://dx.doi.org/10.1007/11499107_5

[8] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 129–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_10

[9] N. Manthey, M. J. H. Heule, and A. Biere, "Automated reencoding of boolean formulas," in *Proceedings of Haifa Verification Conference 2012*, 2012.

[10] I. Lynce and J. Marques-Silva, "Probing-Based Preprocessing Techniques for Propositional Satisfiability," in *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ser. ICTAI '03. IEEE Computer Society, 2003, pp. 105–110. [Online]. Available: http://portal.acm.org/citation.cfm?id=951951.952290

[11] M. Heule, M. Järvisalo, and A. Biere, "Covered clause elimination," *CoRR*, vol. abs/1011.5202, 2010.

[12] M. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for cnf formulas," in *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 357–371. [Online]. Available: http://dl.acm.org/citation.cfm?id=1928380.1928406

[13] A. V. Gelder, "Toward leaner binary-clause reasoning in a satisfiability solver," *Ann. Math. Artif. Intell.*, vol. 43, no. 1, pp. 239–253, 2005.

[14] M. J. H. Heule, M. Järvisalo, and A. Biere, "Efficient cnf simplification based on binary implication graphs," in *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, ser. SAT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 201–215. [Online]. Available: http://dl.acm.org/citation.cfm?id=2023474.2023497

[15] W. Wei and B. Selman, "Accelerating random walks," in *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, ser. CP '02. London, UK, UK: Springer-Verlag, 2002, pp. 216–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=647489.727142

[16] N. Manthey and P. Steinke, "Quadratic Direct Encoding vs. Linear Order Encoding," in *First International Workshop on the Cross-Fertilization Between CSP and SAT(CSPSAT'11)*, 2011.

[17] M. N. V. Van Hau Nguyen and S. Hölldobler, "Application of hierarchical hybrid encoding to efficient translation of a csp to sat," Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, Tech. Rep., 2013.

[18] A. del Val, "On 2-sat and renamable horn," in *AAAI/IAAI*, H. A. Kautz and B. W. Porter, Eds. AAAI Press / The MIT Press, 2000, pp. 279–284.

[19] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *AAAI*, B. Hayes-Roth and R. E. Korf, Eds. AAAI Press / The MIT Press, 1994, pp. 337–343.

[20] K. Gebhardt and N. Manthey, "Parallel Variable Elimination on CNF Formulas," in *Pragmatics of SAT*, 2013.

[21] bwGRiD (http://www.bw grid.de/), "Member of the german d-grid initiative, funded by the ministry of education and research (bundesministerium für bildung und forschung) and the ministry for science, research and arts baden-wuerttemberg (ministerium für wissenschaft, forschung und kunst baden-württemberg)," Universities of Baden-Württemberg, Tech. Rep., 2007-2010.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

StrangeNight

Mate Soos
Security Research Labs

I. Introduction

In this solver description we present the feature-set of StrangeNight, a modern SAT Solver that aims to unify the advantages of SatELite [1], PrecoSat [2], Glucose [3] and MiniSat [4] with the xor-clause handling of version 1 of StrangeNight [5] to create a formula that can solve many types of different problem instances under reasonable time.

II. Features

StrangeNight is a DPLL-based SAT solver developed from MiniSat. The following list of non-exhaustive features are offered by StrangeNight relative to the original "core" MiniSat.

A. Xor clauses

XOR clauses are extracted at the beginning of the solving. They are subsequently treated differently. They have their own watchlists, their own propagation mechanism, and their own subsumption algorithm. This should mean that they are handled faster in most scenarios.

B. Binary xor clauses

Binary xor clauses are handled specially. Firstly, they are regularly searched for using a special heuristic. Secondly, a forest structure is built from them, indicating which variable is equi- or antivalent with which variable. The top of the trees are regularly replaced with those lower in the tree, reducing the number of clauses and variables in the problem, and usually leading to variable assignments (and possibly even more binary xor clauses).

C. Binary xor clause finding through regular XOR-ing of xor clauses

As per the PhD Thesis of Heule [6], xor clauses are regularly XOR-ed with one another to obtain different XOR clauses. However, contrary to that present in the paper, the smaller XOR-s are only acted upon if they are binary. In this case, they are added to the forest of equi- and antivalences, and replaced with one another at a later time, according to a heuristic.

D. Phase calculation, saving and random flipping

Default phase is calculated for each variable according to the Jeroslow and Wang [7] heuristic. The phases are saved, according to Pipatsrisawat and Darwiche [8]. The phase, however, is randomly flipped at intervals that is determined by the problem. The average branch depth is measured, and with P(1/avgBranchDepth), the current phase is flipped. According to our experience, this helps in exploring new places in the search space.

E. Automatic detection of cryptographic and industrial instances

Industrial and cryptographic instances are very different. They need different restart strategies and they need different learnt clause activity statistics. We try to detect which problem belongs to which family, and use Glucose-style learnt clause heuristics [9] or MiniSat-style learnt clause activity accordingly. We also switch the restart type from dynamic to static and vica-versa. The detection is based on the percentage of xor clauses and the stability of variable activity. Either of the two is too high, the problem is deemed to be cryptographic. The stability of variable activity is measured through saving of the top 100 variables, and comparing them with the next restart's top 100 variables. This is done for 5 restarts, and at the end, the decision is made. The detection routine is run regularly, to detect whether the problem has changed enough to switch from one type to the other.

F. Variable elimination, clause subsumption and clause strengthening

SatELite-type variable elimination, clause subsumption and clause strengthening is regularly performed. The occurrence lists are, however, not updated all the time such as the case with other solvers. Instead, occurrences are calculated on per-use basis. The number of variable elimination cycles, clause subsumption cycles and clause strengthening cycles are limited each time the simplification is done such as to avoid the routine taking overly large amounts of time.

G. On-the-fly clause improvement

Since the occurrence lists are not updated all the time, the only way to carry out subsumption is the algorithm by Han and Somenzi [10]. This lightweight subsumption-check is carried out every time a conflict analysis is done.

H. Binary clause propagation

Binary clauses are in a separate watchlist, as per Glucose [3]. They are fully propagated before other clauses are propagated. The propagation order is: binary clauses, regular clauses, xor clauses. As per PrecoSat [2], the binary clauses are always fully propagated, regardless if a conflict has been found earlier. The conflict analysis routine is then called on the last conflicting binary clause.

I. 32-bit pointers on 64-bit architectures under Linux

64-bit pointers are well-known to slow down the solving of SAT solvers, due to the extra memory and thus cache space occupied by them when going through the watchlists in the propagation phase. This limitation means that all code has to be compiled as 32-bit code, which means that

extra registers and instructions provided by modern 64-bit architectures is lost. We counter this phenomenon with small pointers. Since the memory used by SAT solvers is rarely more than 4GB, the pointers rarely contain more than 32 bit real information. We extract this information, and only store these 32 bits.

J. Binary graph treatment

Binary clauses generated by hyper-binary resolution [11] are added in an optimal manner: the binary subtree of literal a is searched and the highest-degree dominated literal c still leading to b is connected to b. This ensures maximal graph connectivity and sparsity. Binary clauses describing tautologies such as $(\neg a \vee b), (\neg b \vee c), (\neg \mathbf{a} \vee \mathbf{c})$ are regularly removed. Tautologies are also regularly and temporarily generated to subsume and strengthen other clauses.

K. Clause cleaning

Clauses are regularly removed that have at least one of their literals assigned to `true`. Contrary to "core" MiniSat, we also remove `false` literals from clauses, shortening them. Interestingly, this does not need these clauses to be re-attached, as `false` literals are not in the watchlists — or if they are, the clause is satisfied, and can be fully removed.

L. Xor clause subsumption

Xor clauses can be subsumed similarly to normal clauses. Since xor clauses represent many regular clauses, doing the subsumption natively saves significant time.

M. Dependent variable removal

Dependent variables , as per [6] are removed along with their corresponding xor clause. Dependent variables are variables that appear nowhere else but in exactly one xor clause. Since that xor clause can always be satisfied by a correct value of the dependent variable, the xor clause can be removed without further ado, and reintroduced during solution extension as per SatELite. This removes a constraint and a variable from the problem. Note that this variable could not have been removed as part of pure literal elimination. However, interestingly, blocked clause elimination (BCE) can remove these clauses and corresponding variable(s). This connection has not been noted by Biere and Jarvisalo in [12], but shows the effectiveness of their method.

N. Failed literal probing

Variables are tried to be branched both to `true` and `false` at regular intervals. If any of the branches fails (conflict is returned), that variable is assigned to the other branching. Otherwise, the assignments of both are saved and compared with one another. If they contain a common subset, that variable is assigned, as per [13]. An interesting addition to this is the method by Li [14], where binary XOR clauses are found in the same way that common subset of assignments are found. Binary xor clause $l \oplus u$ is also found when $u \in Prop(clauses, l)$ and $\neg u \in Prop(clauses, \neg l)$, following Proposition 4 of [13].

Acknowledgements

The author was supported by the RFID-AP Projet of ANR, project number `ANR-07-SESU-009`.

I would like to thank Martin Maurer, Trevor Hansen and Vijay Ganesh for their bug reports, ideas and stress-tests.

Experiments carried out to tune StrangeNight were performed using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies [15].

References

[1] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. [16] 61–75

[2] Biere, A.: P{re,i}cosat@sc'09: a solver that predicts learnt clauses quality. In: SAT 2009 competitive events booklet. (2009) 41–42

[3] Audemard, G., Simon, L.: GLUCOSE: a solver that predicts learnt clauses quality. In: SAT 2009 competitive events booklet. (2009) 7–8

[4] Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518

[5] Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. [17] 244–257

[6] Heule, M.J.: Smart solving: Tool and techniques for satisfiability solvers. Technical report, Technische Universiteit Delft (2008)

[7] Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Ann. Math. Artif. Intell. **1** (1990) 167–187

[8] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., Sakallah, K.A., eds.: SAT. Volume 4501 of Lecture Notes in Computer Science., Springer (2007) 294–299

[9] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In Boutilier, C., ed.: IJCAI. (2009) 399–404

[10] Han, H., Somenzi, F.: On-the-fly clause improvement. [17] 209–222

[11] Gershman, R., Strichman, O.: Cost-effective hyper-resolution for preprocessing CNF formulas. [16] 423–429

[12] Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In Esparza, J., Majumdar, R., eds.: TACAS. Volume 6015 of Lecture Notes in Computer Science., Springer (2010) 129–144

[13] Berre, D.L.: Exploiting the real power of unit propagation lookahead. Electronic Notes in Discrete Mathematics **9** (2001) 59–80

[14] Li, C.M.: Equivalent literal propagation in the DLL procedure. Discrete Applied Mathematics **130**(2) (2003) 251–276

[15] The Grid'5000 team: The Grid'5000 project https://www.grid5000.fr.

[16] Bacchus, F., Walsh, T., eds.: Theory and Applications of Satisfiability Testing, St. Andrews, UK. In Bacchus, F., Walsh, T., eds.: SAT. Volume 3569 of LNCS., Springer (2005)

[17] Kullmann, O., ed.: Theory and Applications of Satisfiability Testing — SAT 2009, Swansea, UK. In Kullmann, O., ed.: SAT. Volume 5584 of LNCS., Springer (2009)

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

vflipnum: A Local Search with Variable Flipping Frequency Heuristics for SAT

Jingchao Chen and Yuyang Huang
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn, sigefriedhyy@gmail.com

***Abstract*—vflipnum submitted to the SAT Competition 2013 is a new stochastic local search (SLS) solver based on variable flipping frequency heuristics. In addition to using the parameters score and age used in Sparrow to break ties, we add a new parameter – variable flipping frequency. As far as we know, the new parameter never used successfully in the known SLS solvers. This paper describes briefly vflipnum.**

I. Introduction

The simplest stochastic local search (SLS) algorithm is to pick always randomly a variable from a randomly selected unsatisfied clause, and then flip its truth assignment. This solving mechanism is used by solvers such as Walksat [1]. The main drawback of this mechanism is to be trapped easily in deep local minima. To escape local minima, ones proposed various heuristics, which result in the emergence of many variants of Walksat. The most common information used in various heuristics includes score, age and clause weighting etc. As far as we know, variable flipping frequency never used successfully in any known SLS solvers. In this paper, we use a variable flipping frequency heuristic to develop a new SLS solver called vflipnum. This new solver is based on the framework of Sparrow [2], which is the winner of the random satisfiable category of the SAT Competition 2011. Except for some magic constant settings and pickvar procedure, vflipnum is the same as Sparrow.

II. Variable flipping frequency heuristics for SAT

In vflipnum, the probability of selecting a variable depends on Sparrow evaluation and variable flipping frequency. Let C be the selected clause, and $Numflip(v_i)$ be the number of flips that occur in variable v_i. Using variable flipping frequency, for each variable v_i, the probability $pf(v_i)$ of selecting it is defined as follows.

$$MaxNumf = \max_{v_i \in C}\{Numflip(v_i)\}$$

$$\Delta(v_i) = MaxNumf + 1 - Numflip(v_i)$$

$$pf(v_i) = \frac{\Delta(v_i)}{\sum\limits_{v_j \in C} \Delta(v_j)}$$

We use $pf(v_i)$ in the following two cases. (1) If none of v_i's neighboring variables has been flipped since v_i's last flip, but its score is large enough, we pick variable v_i with probability $pf(v_i)$. This is similar to CCASat [3]. However, we break ties in favor of probability $pf(v_i)$, while CCASat break ties in favor of the oldest age. (2) If the probability selecting condition of Sparrow is reached, we replace the probability selecting formula of Sparrow:

$$\frac{sparrowScore \times sparrowAge}{\sum sparrowScore \times sparrowAge}$$

with the following formula:

$$\frac{sparrowScore \times sparrowAge \times pf}{\sum sparrowScore \times sparrowAge \times pf}$$

where the sum ranges over all variables in the given clause. For the meaning of sparrowScore and sparrowAge, see [2].

III. The vflipnum solver

The vflipnum solver is built on top of Sparrow. Sparrow can be divided into the following components: parameter setting, pickvar, flipvar, smooth and scale etc. The flipvar, smooth and scale procedure of vflipnum are the same as that of Sparrow. The parameter setting and pickvar procedure of vflipnum are different from Sparrow. Here we may describe briefly our pickvar procedure as follows.

(1) If #unsat clauses $> \theta$, like Sparrow, pick a variable with the greatest score > 0, and break ties in favor of the oldest age.
(2) Let V be set of variables whose neighboring variables have been flipped. If $V \neq \emptyset$, pick a variable in V with the greatest score > 0.
(3) If in the above two steps we fail to pick a promising variable, with probability $pf(x)$, pick variable x with the greatest score ≥ 0.
(4) If #unsat clauses ≤ 2, we always pick a variable with the oldest age.
(5) If none of the above steps has picked successfully a promising variable, we pick randomly an unsatisfied clause C, and then with probability given in our new probability formula (see the previous section), pick a variable in C to be flipped.

The parameter θ in the variable picking algorithm is set to 5 when the number of variables is greater than 3000, and 6 up to 9 otherwise.

Table 1 shows vflipnum parameter settings. The meaning of parameters c_1, c_2, c_3 and ps is in accordance with Sparrow2011. In many cases, the parameter configurations used

TABLE I
VFLIPNUM PARAMETER SETTINGS.

k-SAT	$\#var$	c_1	c_2	c_3	ps
3-SAT	$<$3000	2.15	4	100000	0.4
3-SAT	$\geq$3000	2.15	4	100000	0.347
5-SAT	$\leq$200	3	4	105000	0.9
5-SAT	$>$200	2.85	4	75000	1
7-SAT	$<$90	5.5	4	110000	0.835
7-SAT	$\geq$90	6.5	4	110000	0.83

by vflipnum are different from Sparrow2011. For example, when the number of variables is less than or equal to 200, the 5-SAT configuration in vflipnum is $\langle c_1,\ c_2,\ c_3,\ ps\rangle$ = $\langle 3, 4, 105000, 0.9\rangle$, while Sparrow2011 set $\langle c_1,\ c_2,\ c_3,\ ps\rangle$ to be $\langle 2.85, 4, 75000, 1\rangle$. Nevertheless, in any case, both vflipnum and Sparrow2011 set parameter c_2 to 4.

IV. CONCLUSION

Here we presented a new SLS sequential solver called vflipnum. Although this new solver is based on Sparrow, both performances are different since we used different mechanism for picking a variable to be flipped, and different parameter configuration. From our empirical observation, vflipnum was significantly faster than Sparrow. In designing vflipnum, many parameters are done by hand-tuning. In fact, many solvers are based on hand-tuned parameter configuration. Then, what are the optimal parameter configuration and the most efficient SLS solver? such a problem is left as an open problem.

REFERENCES

[1] Selman B., Kautz H., Cohen B.: Noise strategies for improving local search. In Proceedings of AAAI-94, 12th National Conference on Artificial Intelligence, pp. 337 – 343. Seattle, USA, 1994.

[2] Tompkins D., Balint A., Hoos H.: Captain Jack C New Variable Selection Heuristics in Local Search for SAT, SAT11, pp. 302 – 316, 2011.

[3] Cai, S.W., Luo, C., Su, K.L.: CCASat: Solver Description, Proceedings of SAT Challenge 2012, pp. 13 – 14, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

WalkSATlm2013

Shaowei Cai
Griffith University, Australia
shaoweicai.cs@gmail.com

***Abstract*—This note describes the SAT solver "WalkSATlm2013", which is a local search solver, especially designed for random instances.**

I. Introduction

Algorithms for solving SAT can be mainly categorized into two classes: complete algorithms and stochastic local search (SLS) algorithms. Among SLS algorithms for SAT, WalkSAT [1] stands out as one of the most influential algorithms.

Recently, there has been increasing interest in WalkSAT, due to the discovery of its great power on large random 3-SAT instances. However, the performance of WalkSAT on random k-SAT instances with $k > 3$ lags far behind. Indeed, there have been few works in improving SLS algorithms for such instances. We improve WalkSAT for random instances with long clauses by a simple yet very effective method, which is used to break ties in WalkSAT. The method is based on the notion of multi-level $make$ [2]. This improved algorithms is called WalkSATlm. The SAT solver WalkSATlm2013 adopts WalkSAT to solve random instances whose maximum clause length (denoted by k) is greater than 3, and adopts WalkSATlm to solve instances with $k > 3$.

II. Main Techniques

The only main new technique is a novel scoring function named $linear\ make$ [2]. We proposed the concept of τ^{th} $level\ make$ [2], denoted by $make_\tau$, which measures the number of $(\tau - 1)$-satisfied clauses that would become τ-satisfied by flipping x. Here a clause is τ-satisfied if and only if it contains exactly τ true literals. Recall that if a literal evaluates to true under the given assignment, it is a $true\ literal$; otherwise, it is a $false\ literal$. The $lmake$ function combines the $make$ property with a new property $make_2$ and is defined as $lmake(x) = w_1 * make_1(x) + w_2 * make_2(x)$.

WalkSATlm differs from WalkSAT only in the tie-breaking method (of choosing a variable from those with the equally minimum $break$ value). In detail, while WalkSAT breaks ties randomly, WalkSATlm does so by preferring the variable with the greatest $lmake$ value (further ties are broken randomly).

III. WalkSAT and WalkSATlm

WalkSAT applies the following variable selection scheme in each step. First, an unsatisfied clause C is selected randomly. If there exist variables with a $break$ value of 0 in clause C, $i.e.$, if C can be satisfied without breaking another clause, one of such variables is flipped (so-called *zero-damage* step). If no such variable exists, then with a certain probability p (the noise parameter), one of the variables from C is randomly selected; in the remaining cases, one of the variables with the minimum $break$ value from C is selected.

In WalkSAT, all ties are broken randomly, while in WalkSATlm, all ties are broken by preferring the variable with the greatest $lmake$ value (further ties are broken randomly).

IV. Main Parameters

We combine the WalkSAT and WalkSATlm algorithms, leading to an SLS solver also called WalkSATlm2013, which adopts WalkSAT to solve instances with $k \leq 3$, and adopts WalkSATlm to solve instances with $k > 3$.

WalkSAT and WalkSATlm has one same parameter, namely the noise parameter wp. In WalkSATlm2013, wp is set as follows.

For $k \leq 3$, wp is set to 0.567 when $r \leq 4.22$, 0.777-0.05r if $r \in (4.22, 4.23]$, 1.553-0.23r if $r \in (4.23, 4.26)$ and 2.261-0.4r if $r \geq 4.26$, where r is the clause-to-variable ratio.

For $k = 4$, wp is set to 0.6 if $r \leq 9$, 1.4921-0.1r if $r \in (9, 9.5]$, 1.5026-0.1r if $r \in (9.5, 9.75]$, and 1.9895-0.15r otherwise. For $k = 5$, wp is set to 0.39 if $r \leq 20$, 1.22-r/24 if $r \in (20, 20.6)$, 0.707-r/60 if $r \in [20.6, 20.8]$, and 1.231-r/24 otherwise. For $k = 6$, wp is set to 1.05-0.02r if $r < 42$, and 0.2 otherwise. For $k > 6$, wp is set to 0.12 if $r \leq 85$, and 0.232-r/750 if $r \in (85, 87]$, and 0.115 otherwise.

Besides wp, the WalkSATlm algorithm also has two other parameters namely w_1 and w_2. In WalkSATlm, $w_1 = 3, w_2 = 1$ for $k = 4$, $w_1 = 3$ and $w_2 = 2$ for $k = 5$, $w_1 = 4$ and $w_2 = 3$ for $k = 6$ and $w_1 = w_2 = 1$ for $k > 6$.

V. Implementation Details

WalkSATlm2013 is implemented in C++. It is implemented from scratch. In this implementation, for each variable, we separately record the clause numbers where positive literals appear and those where negative literals appear. By this data structure, the algorithm is significantly accelerated. To compute $break$ of a variable x, we only need to check each clause where true literals of x appear, that whether the clause has only one true literal; if this is the case, then $break(x)$ increases one, and nothing happens otherwise. The $make$ and $make_2$ properties can be calculated likewise.

VI. SAT Competition 2013 Specifies

WalkSATlm2013 is submitted to "Core solvers, Sequential, Random SAT" and "Core solvers, Parallel, Random SAT"

tracks. It is compiled by g++ with the 'O3' optimization option. It is a 32-bit binary.

Its running command is:

WalkSATlm2013 <instance file name> <random seed>.

REFERENCES

[1] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proc. of AAAI-94*, 1994, pp. 337–343.

[2] S. Cai, K. Su, and C. Luo, "Improving WalkSAT for random k-satisfiability problem with $k > 3$," in *Proc. of AAAI-13*, 2013, p. to appear.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

ZENN

Takeru Yasumoto
Kyushu University, Japan
yasumoto.kyushu@gmail.com

Takumi Okugawa
Kyushu University, Japan
okugawa.takumi@gmail.com

I. INTRODUCTION

ZENN is based on MiniSat2.2.0[1]. The ZENN system employs Phase Shift that integrates different search methods, SAFE LBD for keeping better learnt clauses, TLBD which is a kind of LBD and two restart strategies: Luby SE Restart and LBD+CDLV Restart.

II. PHASE SHIFT

Phase Shift integrates different search methods. The solver goes through two or more phases in its search. Each phase has a limited duration and the solver changes phases when the number of restarts reaches the limit. ZENN has two phases called Luby SE Phase and LBD+CDLV Phase. Luby SE Phase uses Luby SE restart as its restart strategy and RHPolicy for determining the number of learnt clauses that will be deleted. LBD+CDLV Phase uses LBD+CDLV restart for restart strategy and RQPolicy as a method to delete learnt clauses.

A. *Luby SE Phase*

1) *Luby SE Restart:* Luby SE Restart is a restart strategy based on Luby Restart. Luby Restart use a sequence that has cycles. Luby SE Restart shortens the length of each cycle, that is, skipping the initial segments of a sequence, and let the solver search more deeply.
2) *RHPolicy:* The solver deletes the first half of learnt clauses at deletion time. This policy is based on MiniSat2.2 but ZENN will not delete more than half of learnt clauses like MiniSat.
3) *VarDecayReduction:* Set var-decay (one of parameters in Minisat) to 0.990.

B. *LBD+CDLV Phase*

1) *LBD+CDLV Restart:* LBD+CDLV restart is a dynamic restart strategy used by GlueMiniSat2.2.5[2]: if one of the following conditions is satisfied, then a restart is forced.
   (a) an average of decision levels in the last 50 conflicts is greater than the global average.
   (b) an average of NTLBDs (explained later) in the last 50 conflicts is greater than the global average $\times 0.8$.
2) *RQPolicy:* The solver deletes 3 quarters of learnts clauses at deletion time. This policy is based on GlueMiniSat2.2.5.
3) *VarDecayAcceleration:* Set var-decay (one of parameters in Minisat) to 0.800.

III. TLBD

True LBD, TLBD for short, is a kind of LBD[3]. TLBD is different from LBD in the manner of updating its value. TLBD ignores literals assinged at level 0.

A. *NTLBD*
Newest TLBD, NTLBD for short, is a kind of TLBD. NTLBD takes the latest TLBD of a learnt clause.

B. *LTLBD*
Lowest TLBD, LTLBD for short, is also a kind of TLBD. LTLBD takes the best NTLBD of a learnt clause so far.

C. *HTLBD*
Highest TLBD, HTLBD for short, is also a kind of TLBD. HTLBD takes the worst NTLBD of a learnt clause so far.

IV. SAFE LBD

Safe LBD is a criterion for freezing learnt clauses. When a learnt clause is about to be deleted, if its LTLBD is lower than SAFE LOW LBD and its HTLBD is lower than SAFE HIGH LBD, it will not be deleted but be detached and kept for possible activation in the future.

ACKNOWLEDGMENT

I wish to express my gratitude to Mr.Hasegawa, Mr.Fujita, Mr.Koshimura for valuable advices and comments.

REFERENCES

[1] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In SAT 2003, 2003.
[2] Hidetomo NABESHIMA, Koji IWANUMA, Katsumi INOUE. GLUEMINISAT2.2.5, SAT2011 competition System Description.
[3] G.Audemard and L.Simon. Predicting learnt clauses quality in modern sat solvers. In proceedings of IJCAI, 2009.
[4] Gilles Audemard, Laurent Simon. Refining Restarts Strategies For SAT and UNSAT. in 18th International Conference on Principles and Practice of Constraint Programming (CP'12),october 2012.

BENCHMARK DESCRIPTIONS

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Generating the Uniform Random Benchmarks for SAT Competition 2013

Adrian Balint

Institute of Theoretical Computer Science
Ulm University, Germany

Anton Belov

School of Computer Science and Informatics,
University College Dublin, Ireland

Marijn J.H. Heule

Department of Computer Science,
The University of Texas at Austin, United States

Matti Järvisalo

HIIT & Department of Computer Science
University of Helsinki, Finland

INTRO

This description explains how the benchmarks were created of the uniform random categories of the SAT Competition 2013. These categories consists of uniform random k-SAT instances with $k \in 3, 4, 5, 6, 7$ – Boolean formulas for which all clauses have length k. For each k the same number of benchmarks have been generated. All instances have been generated with the Uniform Random k-SAT Generator which was also used in the SAT Challenge 2012 and is freely available online[1].

GENERATING THE SATISFIABLE BENCHMARKS

The satisfiable uniform random k-SAT benchmarks are generated for two different types: threshold and huge. The threshold benchmarks have a clause-to-variable ratio equal to the conjectured threshold ratio [1] [2]. New this year are the huge random benchmarks. These benchmarks have a few million clauses and are therefore as large as some of the application benchmarks. For the huge benchmarks, the ratio ranges from far from the threshold ratio to relatively close. Table I shows the details.

No filtering was applied to construct the competition suite. As a consequence, a significant fraction of the generated threshold benchmarks is unsatisfiable.

GENERATING THE UNSATISFIABLE BENCHMARKS

This section describes how the unsatisfiable benchmarks for the competition have been generated. Given a complete SAT solver, it generally holds that the computational cost to solve uniform random k-SAT formulas with n variables and m clauses is quite similar. Notice that this does not hold for satisfiable benchmarks, where a SAT solver can be "lucky". Therefore, having a set of several benchmarks with the same n and m is less useful. Hence, the generated uniform random unsatisfiable benchmarks differ all in size. Here, size refers to the number of clauses. For each $k \in \{3, 4, 5, 6, 7\}$ the formula with the smallest size can be solved in about a minute by lookahead SAT solvers — the fastest type of solvers for uniform random unsatisfiable formulas. The size is slightly increased with a constant number of clauses, such that the largest one (after 30 steps) is expected to be out of reach of today's state-of-the-art solvers. Table II shows the details.

TABLE I
PARAMETERS OF GENERATING THE SATISFIABLE BENCHMARKS

k	threshold (50)	huge (6)
3	$r = 4.267$ $n \in \{3200, 3400, \ldots, 13000\}$	$r \in \{3.7, 3.8, \ldots, 4.2\}$ $n = 1,000,000$
4	$r = 9.931$ $n \in \{830, 860, \ldots, 2300\}$	$r \in \{7.5, 8.0, \ldots, 9.5\}$ $n = 500,000$
5	$r = 21.117$ $n \in \{305, 310, \ldots, 550\}$	$r \in \{15, 16, \ldots, 20\}$ $n = 250,000$
6	$r = 43.37$ $n \in \{191, 192, \ldots, 240\}$	$r \in \{30, 32, \ldots, 40\}$ $n = 100,000$
7	$r = 87.79$ $n \in \{91, 92, \ldots, 140\}$	$r \in \{60, 65, \ldots, 85\}$ $n = 50,000$

TABLE II
PARAMETERS OF GENERATING THE UNSATISFIABLE BENCHMARKS

k	r	smallest	step	largest
3	4.267	1800	+24	2496
4	9.931	1500	+17	1993
5	21.117	1800	+30	2670
6	43.37	2800	+44	4076
7	87.79	4500	+88	7052

The number of variables for each instance is computed by dividing the number of clauses by the ratio r (rounded down to obtain a ratio slightly above the conjectured threshold ratio).

In contrast to the set of satisfiable benchmarks, we filtered the set of unsatisfiable benchmarks. For the filtering we used local search solvers: i.e, solvers that cannot determine unsatisfiability. We have used the best performing SLS solvers from the SAT Challenge 2012 from the category Random SAT, namely CCASat [2] and probSAT [3]. If a local search

[1] http://sourceforge.net/projects/ksatgenerator/

[2] The clause-to-variable ratio for which 50% of the uniform random formulas are satisfiable. For most algorithms, the closer a formula is generated near the threshold ratio, the harder it is to solve it.

solver was not able to produce a solution in 600 seconds, the instance is considered to be unsatisfiable. Given the strength of local search SAT solvers on uniform random formulas of the considered sizes, it is expected that all of them are indeed unsatisfiable.

REFERENCES

[1] S. Mertens, M. Mézard, and R. Zecchina, "Threshold values of random k-sat from the cavity method," *Random Struct. Algorithms*, vol. 28, no. 3, pp. 340–373, May 2006.

[2] S. Cai and K. Su, "Configuration checking with aspiration in local search for sat," in *Proc. AAAI*, 2012.

[3] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," *Proc. SAT 2012*, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

The Application and the Hard Combinatorial Benchmarks in SAT Competition 2013

Adrian Balint

Institute of Theoretical Computer Science
Ulm University, Germany

Anton Belov

School of Computer Science and Informatics,
University College Dublin, Ireland

Marijn J.H. Heule

Department of Computer Science,
The University of Texas at Austin, United States

Matti Järvisalo

HIIT & Department of Computer Science
University of Helsinki, Finland

The benchmarks for the Application and the Hard Combinatorial tracks of SAT Competition 2013 were drawn from a pool containing benchmarks that either (i) were used in the past six competitive SAT events (SAT Competitions 2007, 2009, 2011; SAT Races 2008, 2010; SAT Challenge 2012); (ii) were submitted to these 6 events but not used; (iii) new benchmarks submitted to SAT Competition 2013 (the descriptions for these benchmarks are provided in these proceedings). The main factor that influenced the benchmark selection process of SAT Competition 2013 is the fact that, as with the previous SAT competitions, the SAT solvers participating in the competition are ranked using the solution-count ranking system. Thus the primary requirement is that the selected set of benchmarks should contain as few as possible benchmarks that would not be solved by any submitted solver. At the same time, the set should contain as few as possible benchmarks that would be solved by all submitted solvers. In order to level out the playing field for the submitters that do not have the resources to tune their solvers on all benchmark sets used in the previous competitions, an additional requirement is that the selected set should contain as many benchmarks as possible that were not used in the previous SAT competitions. Finally, the selected set should not contain a dominating number of benchmarks from the same application domain and the same source. To accommodate this latter requirement, we assigned the benchmarks in the pool to *buckets*, where the assignment is guided by the combination of the specific application or a specific combinatorial problem the benchmark originates from and the benchmark submitter[1].

The empirical hardness of the benchmarks in the pool was evaluated using a selection of 5 well-performing SAT solvers from SAT Challenge 2012. The solvers were selected from the set of the *state-of-the-art (SOTA) contributors* [1] in the corresponding tracks of SAT Challenge 2012, with the preference given to solvers that solved a higher number of benchmarks in the Challenge uniquely. The selected solvers for each track are as follows. Application track: `glucose`, `Lingeling`, `simpsat`, `linge_dyphase`, `ZENN`. Hard Combinatorial track: `clasp-crafted`, `glucose`, `Lingeling`, `simpsat`, `sattime2012`[2]. The execution environment used for the evaluation of benchmarks' hardness is the same as the used for the Competition.

The benchmarks *rating* for the tracks was defined as follows:

`easy` — benchmarks that were solved by all 5 solvers in under 500 seconds (1/10-th of the Competition's timeout). These benchmarks are extremely unlikely to contribute to the solution-count ranking of SAT solvers in the competition, as all reasonably efficient solvers are expected to solve these instances within the 5000 seconds timeout enforced in the Competition.

`medium` — benchmarks that were solved by all 5 solvers in under 5000 seconds. Though these benchmarks are expected to be solved by the top-performers in the Competition, they can help to rank the weaker solvers.

`too-hard` — benchmarks that were not solved by any solver within 10000 seconds (2 times the timeout used in the Competition). These benchmarks are likely to be unsolved by all solvers in the Competition, and as such are also useless for the solution-count ranking, and any other ranking that takes into account the execution time of the solvers, e.g. the *careful ranking* [2].

`hard` — the remaining benchmarks, i.e. the benchmarks that were solved by at least one solver within 10000 seconds, and were not solved by at least one solver within 5000 seconds. These benchmarks are expected to be the most useful for ranking the top-performing solvers submitted to the Competition.

Once the hardness of the benchmarks in the pool was established, 300 benchmarks for each track were selected from

[1]The description files that accompany benchmark set distributions contain all information, including the assignment to buckets.

[2]`sattime2012` is an SLS-based solver, and so was only used to evaluate satisfiable benchmarks in the track

the pool. The selection process was controlled by the following constraints:

(i) the ratio of SAT to UNSAT benchmarks should be exactly 50-50;

(ii) no more than 10% of the selected set should come from the same bucket;

(iii) the ratio of new to used benchmarks should be as high as possible;

(iv) the ratio of `medium` to `hard` benchmarks should be as close to 50-50 as possible — however, in order to reduce influence of the solvers used for the rating of the benchmarks, 20% of the selected benchmarks were selected among the `medium`, `hard` and `too-hard` benchmarks in the pool without the consideration of their rating;

(v) the performance of the 5 solvers used for the evaluation of the benchmarks should be as uniform as possible — this is to avoid a potential bias towards a particular evaluation solver in the set (the potential negative effects of such bias are discussed in [3]).

The details for the selected sets are provided in Tables I and II on the following page.

REFERENCES

[1] G. Sutcliffe and C. B. Suttner, "Evaluating general purpose automated theorem proving systems," *Artif. Intell.*, vol. 131, no. 1-2, pp. 39–54, 2001.

[2] A. Van Gelder, "Careful ranking of multiple solvers with timeouts and ties," in *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, ser. SAT'11, 2011, pp. 317–328.

[3] A. Balint, A. Belov, M. Järvisalo, and C. Sinz, "Overview and analysis of the SAT Challenge 2012 solver competition," 2013, (under review).

TABLE I
DETAILED COUNTS OF THE APPLICATION BENCHMARK SET

bucket	count	SAT	UNSAT	new	old	medium	hard	too-hard
2d-strip-packing	5	3	2	0	5	3	2	0
bio	5	5	0	0	5	4	1	0
crypto-aes	11	9	2	0	11	0	11	0
crypto-des	9	9	0	0	9	3	6	0
crypto-gos	30	2	28	30	0	0	30	0
crypto-md5	11	9	2	0	11	10	1	0
crypto-sha	30	30	0	30	0	0	28	2
crypto-vmpc	8	8	0	0	8	2	5	1
diagnosis	26	17	9	0	26	17	9	0
hardware-bmc	3	0	3	0	3	3	0	0
hardware-bmc-ibm	4	0	4	0	4	0	4	0
hardware-cec	30	0	30	30	0	3	22	5
hardware-velev	21	10	11	0	21	18	3	0
planning	25	21	4	0	25	10	15	0
scheduling	30	16	14	30	0	9	21	0
scheduling-pesp	30	3	27	30	0	17	4	9
software-bit-verif	14	3	11	0	14	8	6	0
software-bmc	3	3	0	0	3	2	1	0
termination	5	2	3	0	5	5	0	0
Total	300	150	150	150	150	114	169	17

TABLE II
DETAILED COUNTS OF THE HARD COMBINATORIAL BENCHMARK SET

bucket	count	SAT	UNSAT	new	old	medium	hard	too-hard
VanderWaerden	9	4	5	0	9	3	6	0
clique-width	24	3	21	24	0	2	7	15
coloring	3	3	0	2	1	0	3	0
connm-ue-csp-sa	4	1	3	0	4	1	3	0
counting-php	13	0	13	13	0	1	0	12
edgematching	8	8	0	0	8	7	1	0
ensemble-computation	7	5	2	0	7	2	5	0
extended-resolution	3	0	3	0	3	0	3	0
factoring	12	12	0	8	4	4	8	0
fixed-shape-forced	3	3	0	0	3	0	3	0
frb	1	1	0	0	1	0	1	0
games-battleship	2	1	1	0	2	0	2	0
games-hidoku	22	1	21	21	1	20	1	1
games-pebbling	2	2	0	0	2	1	1	0
graph-isomorphism	30	0	30	30	0	3	27	0
greentao	1	1	0	0	1	0	1	0
grid-coloring	1	1	0	0	1	0	1	0
hwb	3	0	3	0	3	0	3	0
labs	30	23	7	30	0	0	30	0
lksat	1	0	1	0	1	1	0	0
markstrom	3	0	3	0	3	0	3	0
modcircuits	6	6	0	0	6	0	6	0
ordering	2	0	2	0	2	1	1	0
phnf	2	0	2	0	2	1	1	0
planning	30	29	1	30	0	0	30	0
pmg	2	0	2	0	2	0	2	0
quasigroup	5	0	5	0	5	4	1	0
ramseycube	1	1	0	0	1	1	0	0
random-mus	13	0	13	13	0	4	9	0
rbsat	14	14	0	0	14	9	5	0
satex-challenges	1	1	0	0	1	0	1	0
sgen	9	7	2	0	9	2	7	0
social-golfer-problem	2	2	0	0	2	0	2	0
software-bit-verif	30	21	9	30	0	29	1	0
xor-chain	1	0	1	0	1	0	1	0
Total	300	150	150	201	99	96	176	28

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Hard SAT instances based on factoring

Joseph Bebel
University of Southern California
Los Angeles, CA

Henry Yuen
MIT
Cambridge, MA

I. Introduction

We submit benchmark SAT instances based on the integer factorization problem. Specifically, we give a set of CNF formulas such that their satisfying assignments will encode nontrivial factors of various integers. It is widely believed that the integer factorization problem is intractable for classical (i.e. non-quantum) computers. Indeed, the security of many cryptographic protocols, such as the RSA cryptosystem, relies on this assumption [1].

We have written a SAT instance generator, called ToughSAT, that transforms instances of the integer factorization problem into SAT instances. Satisfying assignments to the SAT instances can be correspondingly transformed into solutions to the original factorization problem. Our instance generator can be accessed as a web application at http://toughsat.appspot.com. We have made the source code available as well. ToughSAT also produces instances based off of other hard problems, such as SUBSET SUM.

II. Methodology

We discuss the factoring instance generation in more detail. The input to the ToughSAT generator is an integer n, and the output is a satisfiable CNF ϕ such that any satisfying assignment will encode two integers p and q, $p, q \neq 1$, such that $n = pq$. In the instances we submit to the SAT Competition 2013, the integers n are all products of two large primes, which are observed to be the hardest instances for the integer factorization problem [2].

Formally, we can consider the decision problem FACTORING: given three positive integers $\langle n, a, b\rangle$, determine whether there exist a nontrivial factor of n between a and b. FACTORING is in the complexity class NP, and is therefore reducible in polynomial time to every NP-complete problem, including satisfiability. A corollary is that the problem of actually finding the prime factors of $n = pq$ can be polynomial-time reduced to finding a satisfying assignment of a CNF.

ToughSAT's factoring instance generator is a practical implementation of this polynomial time reduction. The reduction itself, on input n, outputs a CNF formula ϕ that encodes a boolean circuit C, of the following form: it consists of a binary multiplier and binary comparators that (1) tests for equality of the output of the multiplier on two integers p and q with the integer n, and (2) tests that both p, q are not 1 (to avoid trivial solutions). The binary encoding of these numbers p, q are left as free variables in ϕ. Therefore, satisfying assignments of ϕ must encode nontrivial factors of the number n.

It is noteworthy that there are sub-exponential time algorithms to factor integers – the state-of-the-art being the *General Number Field Sieve* [3] – so the SAT instances output by ToughSAT are likely not the hardest instances of SAT possible (under the hypothesis that SAT requires exponential time in the worst case to solve). However, these instances still seem among the most difficult SAT instances easily and consistently constructible. Furthermore, we believe that no SAT solvers to date implement any of the complex number theoretic techniques that appear in, say, the General Number Field Sieve, to specially handle SAT instances that encode the factoring problem!

In conclusion, we believe that the small but difficult instances produced by ToughSAT's factoring instance generator are a useful addition to the library of benchmarks available to SAT solver authors.

III. Our instances

In our submission, we provide 8 CNF formulas whose satisfying assignments encode non-trivial factors to the following integers:

$$N_1 = 46847546963729$$
$$N_2 = 24427471030957$$
$$N_3 = 17037614121013$$
$$N_4 = 152411913452483$$
$$N_5 = 55413665935423$$
$$N_6 = 275259516432919$$
$$N_7 = 682781751377743$$
$$N_8 = 199255464717812117.$$

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. Te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-bit rsa modulus," in *Proceedings of the 30th annual conference on Advances in cryptology*, ser. CRYPTO'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 333–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=1881412.1881436

[3] J. Buchmann, J. Loho, and J. Zayer, "An implementation of the general number field sieve," in *Advances in cryptologyCRYPTO93*. Springer, 1994, pp. 159–165.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Two Pigeons per Hole Problem

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University Linz

In the newest version of our SAT solver Lingeling we included a simple algorithm for solving large trivially encoded pigeon hole problems. The algorithm is based on cardinality reasoning. More information about the algorithm can be found in our solver description [1].

One phase of the algorithm consists of extracting *at-most-one* constraints, which we extended to extract *at-most-two* constraints too. This extension allowed us to solve the following simple extension of the pigeon hole problem.

> Given h holes, we ask whether it possible to fit $n = 2 \cdot h + 1$ pigeons into these holes, where each hole can fit at most two pigeons.

We submitted a C program gentph.c as benchmark generator, which takes the number of holes as one argument. For each hole there is an *at-most-two* constraint over n pigeons, which is encoded with $(\text{n3}) = n \cdot (n-1) \cdot (n-2)/6$ clauses of length 3. In addition, for each pigeon there is a clause of length n requiring that the pigeon is at least in one hole.

For $h = 6$ holes the problem becomes difficult for standard CDCL solvers. Glucose 2.1 needs 420 seconds, while Lingeling 587f needs 970 seconds, both on an Intel i7-3930K CPU running at 3.20GHz. Lingeling as submitted to this year's competition, but without cardinality reasoning needs 291 seconds. More holes seem to be out of reach. With cardinality constraint reasoning this problem is trivial and can be solved for up to 20 holes instantly.

We list the sizes of these new benchmarks in Table I. Compared to the well-known original pigeon hole benchmarks, with sizes listed in Table II, we observed that the benchmarks become more difficult for a smaller number of variables.

holes h	pigeons n	variables	clauses
1	3	3	4
2	5	10	25
3	7	21	112
4	9	36	345
5	11	55	836
6	13	78	1729
7	15	105	3200
8	17	136	5457
9	19	171	8740
10	21	210	13321
11	23	253	19504
12	25	300	27625
13	27	351	38052
14	29	406	51185
15	31	465	67456
16	33	528	87329
17	35	595	111300
18	37	666	139897
19	39	741	173680
20	41	820	213241

TABLE I
SUBMITTED "TWO PIGEON PER HOLES" BENCHMARKS TPH_h.

holes h	pigeons n	variables	clauses
1	2	2	3
2	3	6	9
3	4	12	22
4	5	20	45
5	6	30	81
6	7	42	133
7	8	56	204
8	9	72	297
9	10	90	415
10	11	110	561
11	12	132	738
12	13	156	949
13	14	182	1197
14	15	210	1485
15	16	240	1816
16	17	272	2193
17	18	306	2619
18	19	342	3097
19	20	380	3630
20	21	420	4221

TABLE II
WELL-KNOWN PIGEON HOLE BENCHMARKS PH_n.

REFERENCES

[1] A. Biere, "Lingeling, plingeling and treengeling entering the SAT Competition 2013," in *Proc. of SAT Competition 2013*, 2013.

Supported by FWF, NFN Grant S11408-N23 (RiSE).

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Equivalence checking of HWMCC 2012 Circuits

Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Marijn J.H. Heule

Department of Computer Science,
The University of Texas at Austin, United States

Matti Järvisalo

HIIT & Department of Computer Science
University of Helsinki, Finland

Norbert Manthey

Institute of Artificial Intelligence
Technische Universität Dresden, Germany

INTRO

A miter encodes an equivalence check of two Boolean circuits. This is encoded as a combinatorial problem searching for an input for these circuits such that their output is different. Fig 1 shows an illustration of a miter: Two circuits have the same inputs and there is an exclusive-OR (XOR) for each output of the circuits. If the output of one of these XORs can be assigned to true, a certificate is found that shows that the circuits are not equivalent. Miters are generally used as follows: one of the two circuits is an optimized variant of the other one. If the miter has no solution (unsatisfiable), it means that the circuits are equivalent and that the optimization is valid.

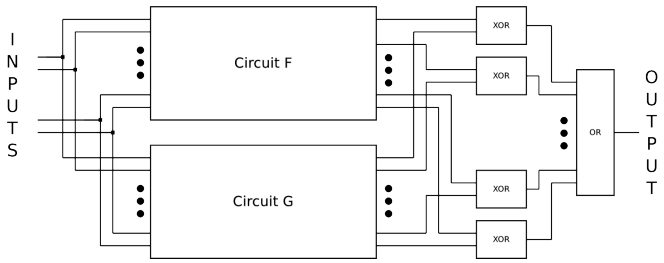

Fig. 1. Illustration of a miter.

GENERATION OF THE BENCHMARKS

We generated two types of miters using the circuits described in the AIGER benchmarks of the hardware model checking competition (HWMCC) 2012[1]. We used circuits with both single and multiple bad state properties (the latter also contain environment constraints). We used `aigmiter` for constructing *combinational* miters, e.g. next state functions of flip-flops are treated as outputs, and then translated them to CNF with `aigtocnf`.

These tools are available from http://fmv.jku.at/aiger. Note that these benchmarks are trivial on the AIG level and can simply be solved by structural hashing. Further, the benchmarks, scripts for generating these miters, as well as log files of the generation process are available from http://fmv.jku.at/miters.

[1]see http://fmv.jku.at/hwmcc12/ for details

NON-OPTIMIZED MITERS

The first type of miter was constructed using two copies of the same circuit. On the AIG level, these benchmarks are trivial. We showed that these benchmarks can also be solved on the CNF level by the preprocessing technique hyper binary resolution [1], [2] (HBR). However, some of the non-optimized miters can be hard for SAT solvers.

OPTIMIZED MITERS

The ABC tool [3] was used to construct optimized circuits (using the `dc2` command). The miters of this second type encode that the original circuit is equivalent to the optimized one. These benchmarks are much harder than the non-optimized miters.

REFERENCES

[1] F. Bacchus and J. Winter, "Effective preprocessing with hyper-resolution and equality reduction," in *Proc. SAT 2003*, ser. LNCS, vol. 2919. Springer, 2004, pp. 341–355.

[2] M. Heule, M. Järvisalo, and A. Biere, "Revisiting hyper binary resolution," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, ser. Lecture Notes in Computer Science, C. Gomes and M. Sellmann, Eds. Springer Berlin Heidelberg, 2013, vol. 7874, pp. 77–93. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38171-3_6

[3] R. K. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *CAV*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 24–40.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Minimal Unsatisfiable Cores of Random Formulas

Marijn J. H. Heule

The University of Texas at Austin, USA

Random k-SAT

A uniform random k-SAT formula consists of clauses of length k for which the literals are chosen by a uniform random distribution and literals have a 50% chance to be negated. Uniform random k-SAT formulas are particularly hard when they are generated near the phase-transition density: the clause-variable ratio for which the fraction of satisfiable / unsatisfiable formulas is 50% / 50 %. Fig. 1 and 2 shows the phase-transition phenomenon for random 3-SAT.

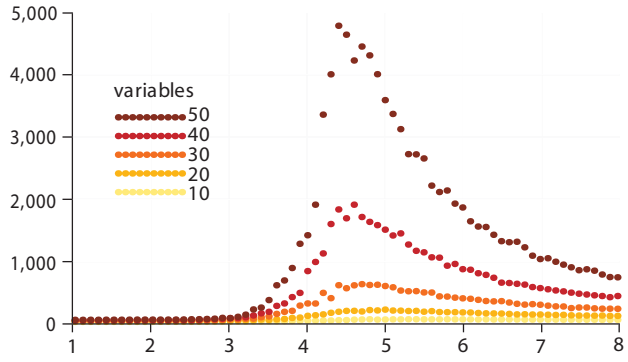

Fig. 1. Number of steps to solve formulas for a certain clause-variable ratio.

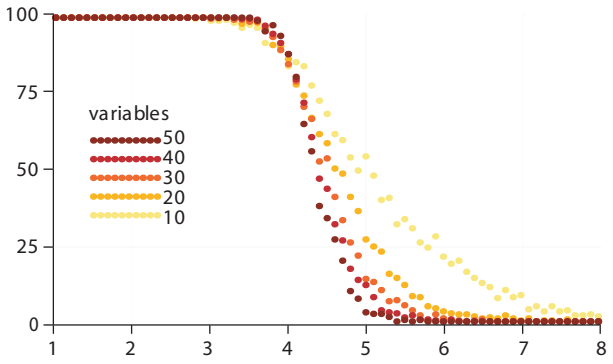

Fig. 2. Fraction of satisfiable formulas for a certain clause-variable ratio.

Minimum Unsatisfiable Cores

An minimal unsatisfiable core [1] is a formula F for which holds that F is unsatisfiable, and for any clause $C \in F$, $F \setminus \{C\}$ is satisfiable. We call a clause $C \in F$ redundant with respect to F if removing C from F preserves unsatisfiability.

Although unsatisfiable random formulas near the phase-transition are hard to solve, they contain quite some redundant clauses. The benchmarks in this suite are random 3-SAT formulas generated near the phase-transition for which the redundant clauses have been removed.

Redundant clauses have been removed by combing a delta-debugger [2] and the lookahead SAT solver march_rw [3]. The procedure works as follows: the delta-debugger obtains the random 3-SAT formula and runs march_rw on it which returns a certain result (SAT or UNSAT). Afterwards, the delta-debugger removes as many clauses as possible without changing the result. If the formula was satisfiable it immediately removes all clauses. In case it was unsatisfiable, the reduced formula is a minimum unsatisfiable core. In the latter case, the computational costs for the reduction can be large (e.g. for a formula with 300 variables about 30 minutes).

We generated random 3-SAT formulas with the number of variables between 300 and 330 and a clause-to-variable ratio of 4.26 (the phase-transition). On average, the delta-debugger was able to remove about 25% of the clauses. The reduced formulas are much harder to solve than the original random formulas. For lookahead SAT solvers, the difference is about an order of magnitude. For conflict-driven clause learning solver, the observed difference was typically even larger.

References

[1] H. Kleine Büning and O. Kullmann, *Minimal Unsatisfiability and Autarkies*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 11, pp. 339–401.

[2] R. Brummayer, F. Lonsing, and A. Biere, "Automated testing and debugging of sat and qbf solvers," in *SAT*, ser. Lecture Notes in Computer Science, O. Strichman and S. Szeider, Eds., vol. 6175. Springer, 2010, pp. 44–57.

[3] S. Mijnders, B. de Wilde, and M. J. H. Heule, "Symbiosis of search and heuristics for random 3-SAT," in *Proc. LaSh*, 2010.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

SAT Benchmarks from Clique-Width Computation

Marijn J. H. Heule

Department of Computer Sciences
The University of Texas at Austin, USA

Stefan Szeider

Institute of Information Systems
Vienna University of Technology, Vienna, Austria

Background

Clique-width is a fundamental graph invariant that has been widely studied in combinatorics and computer science. Clique-width measures in a certain sense the "complexity" of a graph. It is defined via a graph construction process involving four operations where only a limited number of vertex labels are available; vertices that share the same label at a certain point of the construction process must be treated uniformly in subsequent steps. This graph composition mechanism was first considered by Courcelle, Engelfriet, and Rozenberg [1], [2] and has since then been an important topic in combinatorics and computer science. Deciding whether the clique-width of a graph is bounded by a given number, is a very intricate combinatorial problem. More precisely, given a graph G and an integer k, deciding whether the clique-width of G is at most k is NP-complete [3].

A SAT encoding of clique-width

Recently, Heule and Szeider [4] suggested an efficient SAT encoding of the clique-width computation. It is based on a new reformulation of clique-width based on partitions, combined with a efficient encoding of cardinality constraints, called representative encoding. In particular, for an graph G and an integer k, the encoding produces a CNF formula which is satisfiable if and only if G has clique-width at most k.

We provide a benchmark set of various such CNF formulas, produced from random graphs of various density as well as from famous named graphs known from the literature.

In particular, we provide instances based on random graphs and on specific named graphs, for various values of k.

Instances based on random graphs

For edge probability in $\{0.1, 0.2, \ldots, 0.9\}$ the benchmark set contains formulas based on three random graphs with 25 vertices. We expect that most solvers will be able to solve the 0.1, 0.2, 0.8, and 0.9 instances, some the 0.3 and 0.7 instances, and that the others are interesting challenges. Hopefully some researchers will try to find good techniques for those challenging benchmarks. For each random generated graph, we made two instances: one with the clique-width (satisfiable) and one with the clique-width - 1 (unsatisfiable).

Instances based on famous named graphs

The benchmark set contains formulas corresponding to all the famous named graphs considered in [4]. Definitions of all considered graphs can be found in MathWorld [5]. We also included the *da Vinci* graph. This graph we discovered while search for small graphs with a large clique-width. The *da Vinci* graph is the smallest graph with clique-width 6.

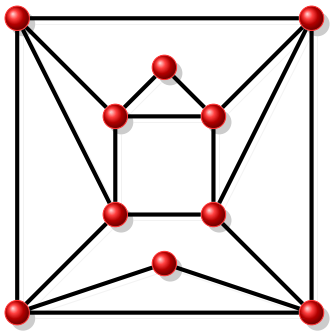

Fig. 1. the *da Vinci* graph, the smallest graph with clique-width 6.

References

[1] B. Courcelle, J. Engelfriet, and G. Rozenberg, "Context-free handle-rewriting hypergraph grammars," in *Graph-Grammars and their Application to Computer Science, 4th International Workshop, Bremen, Germany, March 5–9, 1990, Proceedings*, ser. Lecture Notes in Computer Science, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, Eds., vol. 532, 1991, pp. 253–268.

[2] ——, "Handle-rewriting hypergraph grammars," *J. of Computer and System Sciences*, vol. 46, no. 2, pp. 218–270, 1993.

[3] M. R. Fellows, F. A. Rosamond, U. Rotics, and S. Szeider, "Clique-width is NP-complete," *SIAM J. Discrete Math.*, vol. 23, no. 2, pp. 909–939, 2009.

[4] M. J. H. Heule and S. Szeider, "A sat approach to clique-width," in *Proceedings of SAT 2013, 16th International Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, M. Järvisalo and A. V. Gelder, Eds. Springer Verlag, to appear. A preprint is available from arxiv.org/abs/1304.5498.

[5] E. Weisstein, "MathWorld online maathematics resource."

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Quantifier-Free Bit-Vector Formulas with Binary Encoding: Benchmark Description

Gergely Kovásznai, Andreas Fröhlich, Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

***Abstract*—This document describes several sets of benchmarks corresponding to quantifier-free bit-vector formulas. A generation script first creates all benchmarks in SMT2 format and then uses Boolector to generate CNF instances in DIMACS format by bit-blasting.**

I. Introduction

Bit-precise reasoning over fixed-size bit-vector logics (QF_BV) is important for many practical applications of Satisfability Modulo Theories (SMT), particularly for hardware and software verifcation. In [1], we argued that a *logarithmic* (w.l.o.g. *binary*) encoding, as used e.g. in the SMT-LIB format [2], leads to NExpTime-completeness of the underlying decision problem. Bit-blasting, as used in most current SMT solvers, therefore produces exponentially larger CNF formulas on certain QF_BV formulas. We provide generation scripts for several sets of QF_BV benchmarks in SMT-LIB format where this is the case and use bit-blasting to generate SAT benchmarks out of the original SMT2 specifications. All scripts and generated benchmarks are available at `http://fmv.jku.at/smtbench`.

II. Benchmarks

Our benchmark sets can be divided into two main categories: Expressing common bit-vector operations by other operations and general properties that can be expressed by a fragment of QF_BV with a restricted set of operations.

A. Translating Bit-Vector Operations

The first category contains 13 different benchmark sets and was used for verifying correctness of various translations between bit-vector operators. Having proved that *bitwise operations*, *equality*, and *slicing* suffice to derive NExpTime-hardness theoretically, we also wanted to give concrete examples of how to replace common bit-vector operations by those *base operations*. To check correctness, we encoded all translations into SMT2 and verified that no counter-example exists. We did this for 13 different operations. All benchmarks are unsatisfiable:

addition (`bvadd`), subtraction (`bvsub`), multiplication (`bvmul`), unsigned division (`bvudiv`), signed division (`bvsdiv`), unsigned remainder (`bvurem`), signed remainder (`bvsrem`), signed modulo (`bvsmod`), logical shift right (`bvlshr`), arithmetic shift right (`bvashr`), shift left (`bvshl`), unsigned less than (`bvult`), and signed less than (`bvslt`).

To give one specific example, addition can be expressed by base operations as follows:

$t_1{}^{[n]} + t_2{}^{[n]}$ is replaced by $ts_1{}^{[n]} \oplus ts_2{}^{[n]} \oplus c_{in}{}^{[n]}$ and additional constraints

1) $ts_1{}^{[n]} = t_1{}^{[n]}$
2) $ts_2{}^{[n]} = t_2{}^{[n]}$
3) $c_{out}{}^{[n]} = (ts_1{}^{[n]} \ \& \ ts_2{}^{[n]}) \mid (ts_1{}^{[n]} \ \& \ c_{in}{}^{[n]}) \mid (ts_2{}^{[n]} \ \& \ c_{in}{}^{[n]})$
4) $c_{in}{}^{[n]} = c_{out}{}^{[n]} \ll 1^{[n]}$

are added. Now again, $c_{out}{}^{[n]} \ll 1^{[n]}$ can be replaced by $ts_3{}^{[n]}$ and additional constraints

1) $ts_3{}^{[n]}\,[n:1] = c_{out}{}^{[n]}\,[n-1:0]$
2) $ts_3{}^{[n]}\,[0:0] = 0^{[1]}$

are added.

While this is well-known for the example of addition, expressing multiplication or other operations by using only those base operations is much more complicated and cannot be detailed in the scope of this description. On the other hand, this already explains the benefit of verifying correctness by using our benchmarks.

B. Bit-Vector Properties in PSpace

The second category consists of QF_BV benchmark sets with a reduced set of operations. In [3], we showed that QF_BV becomes PSpace-complete under certain restrictions on the set of allowed operations. While bit-blasting still produces exponentially larger formulas, the original benchmarks could be solved more efficiently, e.g. by using model checkers. It will be interesting to see whether any of the SAT solvers can also profit from this fact.

The 4 benchmark sets contained in this category are the following ones:

`ndist.a`: We verify that, for two bit-vector variables $x^{[n]}$, $y^{[n]}$, it holds that $x^{[n]} < y^{[n]}$ implies $(x^{[n]} + 1^{[n]}) \leq y^{[n]}$. The instances are unsatisfiable.

`ndist.b`: We give a counter-example (due to overflow) to the claim that, for two bit-vector variables $x^{[n]}$, $y^{[n]}$, it holds that $(x^{[n]} + 1^{[n]}) \leq y^{[n]}$ implies $x^{[n]} < y^{[n]}$. The instances are satisfiable.

`power2sum`: We verify that, for two bit-vector variables $x^{[n]} = 2^j$, $y^{[n]} = 2^k$, with $j \neq k$, $x^{[n]} + y^{[n]}$ cannot be a power of 2. The instances are unsatisfiable.

`shift1add`: We verify that for an arbitrary bit-vector $x^{[n]}$, there exists no bit-vector $y^{[n]} \neq x^{[n]}$ with $(x^{[n]} + y^{[n]}) = (x^{[n]} \ll 1^{[n]})$. The instances are unsatisfiable.

III. SMT2 AND CNF GENERATION

For each of the 17 benchmark sets, an individual generation script is provided. The scripts generate several instances of the given problem set, starting from a minimal bit-width up to a maximal bit-width, incrementing the bit-width by a given step size. Given those parameters as input, they output several SMT2 formulas with bit-vector variables of corresponding bit-widths. Additionally, a `generate.sh` script is included. This script automatically calls all individual generation scripts with appropriate parameters (i.e. bit-widths that create challenging but not too-hard instances) and afterwards calls *Boolector* [4] with argument `-de` to bit-blast the SMT2 instances and create CNF formulas in DIMACS format, therefore directly providing the input benchmarks for the SAT solvers. Additional CNF instances corresponding to different bit-widths can be created manually by using the individual scripts with custom parameters and then translating the output with Boolector.

IV. PRACTICAL CONSIDERATIONS

All benchmarks were originally created to evaluate the performance of SMT solvers. While most benchmarks were challenging for all SMT solvers, some solvers turned out to perform particularly well on specific instances. So far, it is not clear whether this difference in performance is due to SMT rewriting rules, differences in bit-blasting, or because of the underlying SAT solvers. It therefore will be interesting to see how various SAT solvers perform on the bit-blasted version of our benchmarks.

ACKNOWLEDGMENT

This work is supported by FWF, NFN Grant S11408-N23 (RiSE).

REFERENCES

[1] G. Kovásznai, A. Fröhlich, and A. Biere, "On the complexity of fixed-size bit-vector logics with binary encoded bit-width," in *Proc. SMT'12*, 2012.

[2] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," in *Proc. SMT'10*, 2010.

[3] A. Fröhlich, G. Kovásznai, and A. Biere, "More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding," in *Proc. CSR'13*, 2013.

[4] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," in *Proc. TACAS'09*, 2009.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Synchronized Timetable Computation with SAT

Michael Kümmling
Faculty of Transportation and Traffic Sciences
Technische Universität Dresden
Dresden, Germany

Peter Großmann
Faculty of Transportation and Traffic Sciences
Technische Universität Dresden
Dresden, Germany
Email: peter.grossmann@tu-dresden.de

I. Introduction

Currently, railway timetables are still created by manual labor, since computer programs only assist in managing and visualizing data. New approaches cover modeling of all timetable restrictions, allowing automatically synchronized timetable computation [2], [3], [4], [1].

The whole network of all routes and their technical restrictions is modeled as periodic event network. The nodes of this network represent arrival and departure events of the lines in the stations. All time consuming processes, especially dwell time in stations, running times between stations and headways between different trains on the same track are represented by edges. They constrain the set of valid timetables. The Problem whether a valid timetable for this event network exists, is called Periodic Event Scheduling Problem (PESP) [5], which is *NP*-complete.

All departure times are encoded as propositional variables by order encoding [6], whereas all constraints exclude all infeasible pairs of departure/arrival times of the events [1].

Usually, initially generated periodic event networks for real-world problems are too restrictive [4] and hence, unsatisfiable [1]. Consequently, lots of modified, less restrictive event networks have to be probed whether they are satisfiable. In real-world timetabling, thousands of SAT instances have to be solved during one timetable computation run. Thus, short runtime is highly desirable. The networks' sizes and complexity which can be solved in reasonable time, is still restricted. However, SAT solvers already outperform all known native domain solvers despite the additionally needed encoding from PESP to SAT.

II. SAT Instances

Each SAT instance is an encoded periodic event network of a real-world timetabling problem that covers a subset of the German railway network. The "pre" instances were simplified by a native domain preprocessor.

file name	satisfiability
b04_s_2_unknown.cnf	unknown
b04_s_2_unknown_pre.cnf	unknown
b04_s_unknown.cnf	unknown
b04_s_unknown_pre.cnf	unknown
b_unsat.cnf	unsatisfiable
b_unsat_pre.cnf	unsatisfiable
ctl_3082_415_unsat.cnf	unsatisfiable
ctl_3082_415_unsat_pre.cnf	unsatisfiable
ctl_3791_556_unsat.cnf	unsatisfiable
ctl_3791_556_unsat_pre.cnf	unsatisfiable
ctl_4201_555_unsat.cnf	unsatisfiable
ctl_4201_555_unsat_pre.cnf	unsatisfiable
ctl_4291_567_1_unsat.cnf	unsatisfiable
ctl_4291_567_1_unsat_pre.cnf	unsatisfiable
ctl_4291_567_2_unsat.cnf	unsatisfiable
ctl_4291_567_2_unsat_pre.cnf	unsatisfiable
ctl_4291_567_3_unsat.cnf	unsatisfiable
ctl_4291_567_3_unsat_pre.cnf	unsatisfiable
ctl_4291_567_4_unsat.cnf	unsatisfiable
ctl_4291_567_4_unsat_pre.cnf	unsatisfiable
ctl_4291_567_5_unsat.cnf	unsatisfiable
ctl_4291_567_5_unsat_pre.cnf	unsatisfiable
ctl_4291_567_6_unsat.cnf	unsatisfiable
ctl_4291_567_6_unsat_pre.cnf	unsatisfiable
ctl_4291_567_7_unsat.cnf	unsatisfiable
ctl_4291_567_7_unsat_pre.cnf	unsatisfiable
ctl_4291_567_8_unsat.cnf	unsatisfiable
ctl_4291_567_8_unsat_pre.cnf	unsatisfiable
ctl_4291_567_9_unsat.cnf	unsatisfiable
ctl_4291_567_9_unsat_pre.cnf	unsatisfiable
ctl_4291_567_10_unsat.cnf	unsatisfiable
ctl_4291_567_10_unsat_pre.cnf	unsatisfiable
ctl_4291_567_11_unsat.cnf	unsatisfiable
ctl_4291_567_11_unsat_pre.cnf	unsatisfiable
ctl_4291_567_12_unsat.cnf	unsatisfiable
ctl_4291_567_12_unsat_pre.cnf	unsatisfiable
ctl_6280_753_unknown.cnf	unknown
ctl_6280_753_unknown_pre.cnf	unknown
ctl_s_unknown.cnf	unknown
ctl_s_unknown_pre.cnf	unknown
dhlh_25_unknown.cnf	unknown
dhlh_25_unknown_pre.cnf	unknown
dhlh_27_unknown.cnf	unknown
dhlh_27_unknown_pre.cnf	unknown
k_unsat.cnf	unsatisfiable

file name	satisfiability
k_unsat_pre.cnf	unsatisfiable
reg_s_unknown.cnf	unknown
reg_s_2_unknown.cnf	unknown
reg_s_2_unsat_pre.cnf	unsatisfiable
s_unknown.cnf	unknown
s_unknown_pre.cnf	unknown
we_unknown.cnf	unknown
we_unknown_pre.cnf	unknown

REFERENCES

[1] Peter Großmann. Polynomial reduction from PESP to SAT. Technical Report 4, Technische Universität Dresden, Germany, October 2011.

[2] Christian Liebchen and Rolf H. Möhring. The modeling power of the periodic event scheduling problem: railway timetables-and beyond. In *Proceedings of the 4th international Dagstuhl, ATMOS conference on Algorithmic approaches for transportation modeling, optimization, and systems*, ATMOS'04, pages 3–40, Berlin, Heidelberg, 2007. Springer.

[3] Karl Nachtigall. *Periodic Network Optimization and Fixed Interval Timetable*. Habilitation thesis, University Hildesheim, 1998.

[4] Jens Opitz. *Automatische Erzeugung und Optimierung von Taktfahrplänen in Schienenverkehrsnetzen*. PhD thesis, Reihe: Logistik, Mobilität und Verkehr. Gabler Verlag — GWV Fachverlage GmbH, 2009.

[5] Paolo Serafini and Walter Ukovich. A mathematical model for periodic scheduling problems. *SIAM J. Discrete Math.*, 2(4):550–581, 1989.

[6] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. A compact and efficient SAT-encoding of finite domain CSP. In *SAT*, pages 375–376, 2011.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Unsatisfiable, Almost Empty Hidokus

Norbert Manthey
Knowledge Representation and Reasoning,
TU Dresden, Germany

***Abstract*—Hidokus are number puzzles that represent the Hamiltonian Path problem on a square grid. A solution of a Hidoku is found, if in each field of the grid there is a number, and the successor of this number is placed in a neighboring field. Human solvable Hidokus usually have only a few solutions, and are easily solved by SAT solvers. However, the other way around, large empty Hidokus are challenging for SAT solvers, but can be solved easily by humans. The presented CNF generator adds another category: easily unsatisfiable Hidokus, that are easily solvable by humans again. The generated instances are crafted instances, and thus should be used in the crafted track.**

I. Introduction

Number puzzles like the Sudoku or the Hidoku are usually solved by humans. Still, there also exists research on these puzzles, for example both types of puzzles can be solved with SAT solvers [1], [2], and Sudokus have been shown to be NP-complete. Usually, these number puzzles do have at least one solution, so that the puzzles can be used to improve the performance of SAT solvers on satisfiable instances. However, providing instances to proof the unsatisfiability of a problem is a research goal as well, to tune the performance of solver further. Both parts are necessary for example if an optimization problem such as MaxSAT [3] should be solved with the help of a SAT solver.

Focussing on Hidokus, there exists types of puzzles that are easily solved by SAT solvers, but which are challenging for humans, for example because of their size. On the other hand, there exists puzzles with simple patterns that can be solved easily by humans, but are challenging for SAT solvers, like for example an empty Hidoku. Here, another pattern is presented, that forces the Hidoku to be unsatisfiable. Again, the unsatisfiability can be seen easily from the design of the instances, however, it is hard to be found for SAT solvers.

II. Hidokus

A Hidoku is a number puzzle on a $n \times n$ grid, where the numbers ranging from 1 to n^2 have to be places in the grid such that the following constraints are met:

1) Each cell contains exactly one value
2) Each number from 1 to n^2 appears exactly once on the board

3.1) If a cell contains the number i, then the number $i+1$ has to appear in a neighboring cell (except for n^2)

The third rule can also be re-formulated into:

3.2) If a cell contains the number i, then the number $i+1$ cannot appear in any non-neighboring cell

The two alternatives of the last rule allow to decode the puzzle with either the absolute encoding (direct encoding) or the support encoding.

III. Encoding

To encode a Hidoku into SAT, first the first two constraints need are encoded. The exactly-one constraint can be separated into an *at-most-one* constraint (AMO), and an *at-least-one* constraint (ALO). Then for 1), we encode per cell ALO(cell) and AMO(cell), to represent that one value has to be in this cell. Similarly, 2) can be split into ALO(board) and AMO(board). Depending on how the third constraint is encoded, the encoding can be reduced to contain less clauses, but still encoding a valid Hidoku. For a Hidoku with width n, per cell, AMO(cell) requires $3n^2$ clauses, to encode an AMO for each value, by allowing to use auxiliary variables. Per AMO of n literals, $1.5n$ auxiliary variables are introduced, since encoding AMO is done in a recursive fashion, similar to [4]. For n^2 cells the total number of clauses is $3n^4$, where all clauses are binary. For ALO(cell), one large clause (n literals) is required per cell, resulting in n^2 large clauses. For AMO(board), per value $3n^2$ binary clauses are encoded, resulting in a total of $3n^4$ clauses. Finally, ALO(board) encodes a large clause (n^2 literals) per value, adding another n^2 large clauses. Thus, encoding the full board itself, without any special constraints, results in $6n^4 + 2n^2$ clauses.

A. Support Encoding

The support encoding encodes that if a certain value is present in a cell, then one of the neighboring cells has to contain the consecutive number. Thus, for each cell and value a clause with 9 literals is added (sometimes 6 or 4, depending on the position of the cell), adding n^4 large clauses. With this constraint it is sufficient to encode ALO(cell) and AMO(cell) to obtain a valid encoding. In total, this encoding requires about $4n^4 + n^2$ clauses.

B. Absolute Encoding

With the direct encoding, constraint 3.1) is encoded. For each pair of non-neighboring cells a clause is added that disallows that the two cells contain consecutive values. Therefore, per cell up to $n^2 \times n^2$ binary clauses are added, encoding the constraint for each value. In total, the encoded number of clauses comes close to n^6. When the direct encoding is used, it is sufficient to encode ALO(board) and AMO(cell) to obtain a valid encoding. In total, this encoding requires about $n^6 + 3n^4 + n^2$ clauses.

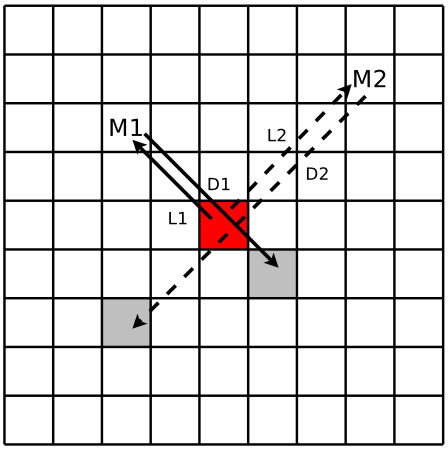

Fig. 1. The figure represents a partially filled Hidoku, which has a contradiction in the cell in the middle of the puzzle. With distance L1, a number M1 is places into a cell, and into another cell the number M1+D1 is placed, such that all the cells on the diagonal between these two cells have to be filled with consecutive values – forcing the middle cell to take a value. The same situation appears with M2, L2 and D2, such that the other diagonal is also forced to contain consecutive values. Now, the middle cell has to contain two values – however, this contradicts the rules of a valid Hidoku.

IV. Unsatisfiable Hidokus

An example of an unsatisfiable Hidoku is given in Figure 1. There, a field M1 is given, and in distance D1, there is another field. By putting the number M1+D1 into this field, the path in the solution of the Hidoku has to follow exactly the line with distance D1, thus, putting also a value into the cell in the middle. Note, since the shortest distance between these two cells is a diagonal, there does not exist another path that connects the two cells with D1 steps. The same procedure is repeated for the other diagonal. From M2 with the distance D2 the value M2+D2 is placed into the grid, such that the cell in the middle has to contain another value. It is easy to see that this Hidoku cannot be solved, since one of the constraints (for example the the first constraint) has to be violated to connect the preset fields according to the rules.

V. The Generator

The provided generator can produce Hidokus with different n, and thus takes n as a parameter. Furthermore, all the other parameters M1,D1 and M2,D2 can be specified. Additionally, it can be determined how far M1 should be away from the conflicting cell (compare Figure 1), by specifying L1. A similar parameter L2 is provided for M2. To ensure that this cell in the middle is the only conflict in the encoded Hidoku, the following constraints have to be met when the parameters are specified:

Parameter	Minimum	Maximum
n	3	inf
L1	0	$\lfloor \frac{n}{2} \rfloor$
D1	L1	$\lfloor \frac{n}{2} \rfloor +$ L1
M1	1	$\lfloor \frac{n^2}{2} \rfloor -$ D1
L2	1	$\lfloor \frac{n}{2} \rfloor$
D2	L2 + 1	$\lfloor \frac{n}{2} \rfloor +$ L2
M2	$\lfloor \frac{n^2}{2} \rfloor$ + D2 + L1	$\lfloor \frac{n^2}{2} \rfloor -$ D2

All these parameters can be specified, so that the generator creates the according Hidoku, and then encodes this Hidoku into CNF. The preset values M1, M1+D1,M2 and M2+D2 are added to the formula as unit clauses. The generator also provide an interface to create puzzles by a random seed. Therefore, only the number n has to be specified, as well as a random seed, an the remaining parameters are filled randomly by the generator. In this mode, also the encoding is chosen randomly. Finally, to control the encoding, another mode has been added. The generator also accepts parameters for n, the encoding, and to enable AMO(cell), ALO(cell), AMO(board) and ALO(board) additionally to the constraints that are required for the chosen encoding (compare Section III-A and III-B). Finally, a seed is added to determine the parameters of the unsatisfiable Hidoku randomly.

VI. Hardness of the Instances

Already empty Hidokus with a sufficiently large n can be challenging for CDCL solvers, independent of the used encoding. When increasing n, the hardness of the Hidoku, and therefore the formula can be controlled. For the unsatisfiable Hidokus it is interesting to see that the absolute encoding results in comparable easy instances for $n < 20$, whereas the support encoding already produces challenging instances when n is equal to 6. For larger n, the support encoding produces very challenging instances.

References

[1] I. Lynce and J. Ouaknine, "Sudoku as a sat problem," in *ISAIM*, 2006.
[2] S. Hölldobler, N. Manthey, V. H. Nguyen, and P. Steinke, "Solving hidokus using sat solvers," in *INFOCOM 5*, 2012, pp. 208–212.
[3] J. Marques-Silva and J. Planes, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 408–413. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403474
[4] N. Manthey, M. J. H. Heule, and A. Biere, "Automated reencoding of boolean formulas," in *Proceedings of Haifa Verification Conference 2012*, 2012.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

A Hard Satisfiable Problem with 160 Variables

Valentin Mayer-Eichberger
NICTA and
University of New South Wales
Valentin.Mayer-Eichberger@nicta.com.au

***Abstract*—In trying to solve a hard graph colouring problem we ran into an interesting SAT formula. The encoding uses just 160 variables and defines a special case of a rectangle-free coloring of a 18x18 grid using four colors. Rectangle-free means that the corners of every rectangle in the grid cannot have all the same colour. Such structured satisfiable problems pose a real challenge to SAT solvers.**

I. Introduction

In 2011 a blog post of

`blog.computationalcomplexity.org`

announced a reward of 289 $ for a solution to the problem of 4-colouring a 17×17 grid such that for each rectangle in the grid all its corners consist of at least two different colours. A solution to 16×16 was known to exist and all grids 19×19 and larger were proven to not contain such a colouring. In 2012 Steinbach and Posthoff presented a solution to 17×17 and 18×18 [1] (every solution of larger grids generates solutions to smaller). We provide the SAT competition with an interesting encoding for this problem which is similar to the approach they used. It will be valuable for the community to see if any SAT solver is able to solve this hard problem within the time-out. Our own experiments show that CDCL solvers tend to spend several hours to find a solution. By such a benchmark we might identify advantages of non-standard SAT solver techniques.

II. Encoding

Naive encodings for this problem can solve grids up to 14×14 almost instantly and do not put a challenge to a SAT solver. With some advancements and symmetry breaking one can also solve 15×15 and 16×16. However, no direct approach seems to tackle the hard cases of 17×17 and 18×18. In this section we explain the tricks that made it possible.

We identify a special case that can be extended to a full solution. If such a solution would exist then the problem is solved, but a negative result would not give much insight. Luckily, it turns out that the simplification does indeed lead to solutions.

We simplify the problem to find a two coloring. We denote the two colours as primary and secondary, and the secondary colour represents the three other colours of the original problem. A solution to this problem can be extended to a solution if

- only the primary colour needs to be rectangle-free,
- $1/4$ of all positions are filled with the primary colour,
- rotating the solution by 90,180, and 270 degrees will not map a position containing a primay colour onto another.

We can then take a solution of this problem and fill for each rotation the mapped positions of the primary colour with one of the remaining one. Since there are no collisions and rectangle-free is preserved under rotation, we generate a full solution.

A natural choice would be to define for each each position in the board a Boolean variable that is true if that position contains the primary colour. We reduce the number of variables by using the restriction that each orbit wrt. to the 90 degree rotations should have exactly one primary colour. This can be encoded in a logarithmic fashion such that two Boolean variables identify for each orbit in which of the four half section of the grid it exists. Furthermore, we break symmetries by forcing the upper left position to contain a primary colour. By these reductions we get a formula that only uses 160 variables.

III. Benchmark

The set contains 4 encodings of the same problems. They have been generated by shuffling the variables, literals and order of clauses of the encoding described above.

Acknowledgement

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

References

[1] B. Steinbach and C. Posthoff, "Extremely Complex 4-Colored Rectangle-Free Grids: Solution of Open Multiple-Valued Problems," in *ISMVL*, 2012, pp. 37–44.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

SAT Benchmark for the Car Sequencing Problem

Valentin Mayer-Eichberger
NICTA and
University of New South Wales
Valentin.Mayer-Eichberger@nicta.com.au

***Abstract*—Car sequencing occurs in the production process of the automotive industry. It addresses the problem of scheduling cars along an assembly line such that capacities of different workstations along the line are not exceeded. We provide the SAT competition with a selection of hard car sequencing problems from the CSP_{LIB} [1]. The encoding is based on a variant of the sequential counter encoding of cardinality constraints and the reuse of auxiliary variables.**

I. Introduction

Car sequencing deals with the problem of scheduling cars along an assembly line with capacity constraints for different stations (e.g. radio, sun roof, air-conditioning, etc). Cars are partitioned into classes according to their requirements. The stations are denoted as *options* and defined by a ratio u/q restricting the maximal number u of cars that can be scheduled on every subsequence of length q.

Example 1 *Given classes $C = \{1, 2, 3\}$ and options $O = \{a, b\}$. The demands (number of cars) for the classes are $3, 2, 2$, respectively. Capacity constraints on options are given by $a : 1/2$ and $b : 1/5$, respectively. Class 1 has no restrictions, class 2 requires option a and class 3 needs options $\{a, b\}$. The only legal sequence for this problem is $[3, 1, 2, 1, 2, 1, 3]$, since class 2 and 3 cannot be sequenced after another and class 3 need to be at least 5 positions apart.*

Car sequencing in the CSPlib contains a selection of benchmark problems of this form ranging from 100 to 400 cars. Over the years different approaches have been used to solve these instances, among them constraint programming, local search and integer programming [2][3][4][5][6].

Car sequencing has also been treated as an optimisation problem and several versions for the opimisation goal have been proposed. Most of the approaches use a variant of minimising the number of violated capacity constraints. However, for this benchmark we use the definition of [7] which transforms easily to sequence of decision problem and SAT solving can be directly applied: An unsatisfiable car sequencing problem can be made solvable by adding empty slots to the sequence. The goal is then to minimise the number of empty slots needed for a valid sequence. A lower bound lb is proven by unsatisfiability with $lb-1$ additional empty slots.

II. The Encoding

The car sequencing problem can be naturally modelled by Boolean cardinality constraints. Our approach is to translate cardinality constraints by a variant of the sequential counter encoding proposed by [8]. The key idea is then to integrate capacity constraint into the sequential counter of the demand constraints by reusing the auxiliary variables. This enforces a global view on the conjunction of these two constraints and facilitates propagation. Our own experiments show that this encodings is far better than naive approaches or an automatic translation from the pseudo Boolean model.

III. The Benchmark

A command line tool that generates CNF in DIMACS format from a problem description in the CSPlib is freely available at `github.com/vale1410/car-sequencing`. With this tool one can generate different encodings and compare the runtime of SAT solvers. For this benchmark we chose the best encoding according to our experiments and we are interested if the solvers from the competition are able to prove stronger bounds.

Acknowledgement

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

References

[1] I. P. Gent and T. Walsh, "CSP_{LIB}: A Benchmark Library for Constraints," in *CP*, 1999, pp. 480–481.
[2] J.-C. Régin and J.-F. Puget, "A Filtering Algorithm for Global Sequencing Constraints," in *CP*, 1997, pp. 32–46.
[3] J. Gottlieb, M. Puchta, and C. Solnon, "A Study of Greedy, Local Search, and Ant Colony Optimization Approaches for Car Sequencing Problems," in *EvoWorkshops*, 2003, pp. 246–257.
[4] M. Gravel, C. Gagné, and W. L. Price, "Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem," *The Journal of the Operational Research Society*, vol. 56, no. 11, pp. 1287–1295, 2005.
[5] B. Estellon, F. Gardi, and K. Nouioua, "Large neighborhood improvements for solving car sequencing problems," *RAIRO - Operations Research*, vol. 40, no. 4, pp. 355–379, 2006.
[6] M. Siala, E. Hebrard, and M.-J. Huguet, "An Optimal Arc Consistency Algorithm for a Chain of Atmost Constraints with Cardinality," in *CP*, 2012, pp. 55–69.
[7] L. Perron and P. Shaw, "Combining Forces to Solve the Car Sequencing Problem," in *CPAIOR*, 2004, pp. 225–239.
[8] C. Sinz, "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints," in *CP*, 2005, pp. 827–831.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

SAT encoded Graph Isomorphism (GI) Benchmark Description

Frank Mugrauer, Adrian Balint
Ulm University
Germany

***Abstract*—In this paper, we describe and implement a way to convert instances of the graph isomorphism problem into SAT instances. We then take a number of small, regular graphs and convert each into a pair of substantially bigger graphs that are almost, but not quite isomorphic, and use those pairs to create hard SAT instances.**

I. GI Description

The Graph Isomorphism (GI) problem presents itself as follows: Given two graphs G_1, G_2 with equal number of vertices n and edges m, can we find a permutation $s \in S_n$, so that after applying s to the vertex labels in G_1, the two graphs are identical. Note that the graphs considered in this paper need to fulfill a number of further requirements: We require undirected graphs without multiple edges (i.e. there can be at most one edge between two vertices) and without self-edges (i.e. a vertex cannot have an edge to itself). With the added simplification process in section III, vertices in the graphs can have colors assigned to them.

II. Transforming GI to SAT

The conversion algorithm (Algorithm 1) used in this paper is based on [1]. We take two graphs G_1 and G_2, both with n vertices and m edges, and transform them into a CNF formula with n^2 variables and $O(n)+O(n^3)+O(n^4)$ clauses. For each vertex v in G_1, we initially create n variables $var_{v,1} \dots var_{v,n}$. The semantics behind this is that if, say, $var_{i,j}$ is true, then vertex i in G_1 corresponds to vertex j in G_2, or $s(i) = j$. It is obvious that if the two graphs are isomorphic, then for each vertex v in G_1, precisely one of the variables $var_{v,1} \dots var_{v,n}$ needs to be true, and all the others need to be false. For each of the n vertices in G_1, a *type-1* clause $(var_{v,1}, \dots, var_{v,n})$ ensures that at least one of them is true; the "at most one" requirement will be fulfilled with *type-2* clauses. Type-2 clauses ensure that no two vertices in G_1 can be relabled to the same vertex G_2. So for every two vertices i, j in G_1 $(i < j)$ and every vertex k in G_2, we create a clause $(\overline{var_{i,k}}, \overline{var_{j,k}})$. Since all n vertices in G_1 are mapped to a vertex in G_2, and no two vertices can be mapped to the same vertex, any mapping the SAT-Solver can find is a permutation. There are $(\frac{n^2(n-1)}{2})$ type-2 clauses, initially, though the simplification algorithm described in section III will likely remove many of them, unless the graphs are not coloured and strictly regular. The main part of the isomorphism test resides in *type-3* clauses. These clauses will ensure that two vertices in G_1 that are connected by an edge cannot be mapped to two vertices in G_2 that have no edge between them. For each pair of connected vertices i, j in G_1 $(i < j)$, and each pair of unconnected vertices k, l in G_2, we create a clause $(\overline{var_{i,k}}, \overline{var_{j,l}})$. The number of type-3 clauses created this way is

$$2m \cdot \left(\frac{n(n-1)}{2} - m \right)$$

and depends on the number of edges m and the maximum number of possible edges $\frac{n(n-1)}{2}$ in the graphs. Graphs with very few (or very many) edges will result in fewer clauses, while graphs with $m \approx \frac{n(n-1)}{4}$ will generate the most clauses. As with type-2, the simplification process will likely remove many of these clauses.

Algorithm 1: GI to SAT converter

```
Input  : G1, G2
Output: CNF instance
C = ∅ ;
for i = 1 to n do  //Type-1
    Clause = ∅;
    for j = 1 to n do
        Clause = Clause ⋃ var_{i,j};
    C = C ⋃ Clause;
for j = 1 to n do  //Type-2
    for k = 1 to n do
        for i = 1 to j do
            C = C ⋃ (¬var_{i,k}, ¬var_{j,k});
for j = 1 to n do  //Type-3
    for i = 1 to j do
        if (i, j) ∈ G1 then
            for k = 1 to n do
                for l = 1 to n do
                    if k ≠ l and (k, l) ∉ G2 then
                        C = C ⋃ (¬var_{i,k}, ¬var_{j,l});
return C;
```

III. Simplification of SAT formulas

When solving the Graph Isomorphism problem, one often has a wealth of information available that a SAT solver can

only find through painstaking trial and error. For instance, it is immediately obvious that a vertex i in G_1 can never be mapped to a another vertex j in G_2 if $degree_{G_1}(i) \neq degree_{G_2}(j)$ or if $colour_{G_1}(i) \neq colour_{G_2}(j)$. In a CNF formula created as described above, a SAT solver will - at least for a difference in degrees - eventually discover that $var_{i,j}$ needs to be false, but only after a lengthy evaluation process. This can be sped up if the converter already takes this additional information into account, and simplifies the resulting formula by removing all variables $var_{i,j}$ and all clauses that contain $\overline{var_{i,j}}$, wherever the degrees of i and j do not match. As an extension of this, for each vertex the sum of the degrees of all adjacent vertices in either graph can be calculated. Again, if these sums don't match for two vertices (one in G_1, the other in G_2), then the corresponding variables and clauses can be removed. This can further be extended to the sum of the sums of all adjacent vertices, and so forth. In our implementation, this is done up to a depth of six.

IV. Generated SAT instances

The generated SAT instances are directly dependent on the graphs we used to create them. The original graphs are small, 6-regular graphs with 10,11 and 12 vertices. They were generated with the GENREG tool described in [2]. We created 30 graphs of each size, except for size 10, where GENREG can only find 21 graphs, and assigned each vertex in a graph random colours, with either two, three or four vertices per color. Based on these original graphs, we used the method proposed in [3] to transform each into two substantially larger and more complex graphs, that are almost isomorphic, except that for precisely four vertices i, j, k, l, one graph has edges $(i, j), (k, l)$ and the other has $(i, l), (k, j)$. We then used each pair of graphs to create an unsatisfiable SAT instance with the methods described above. The instance names adhere to the following convention: "crafted_n<numV>_d<deg>_c<cgs>_num<numG>.cnf", where deg and $numV$ are the degree and number of vertices of the original graph, cgs is the size of the color groups, and $numG$ is the graph number, identifying a graph amongst it's 30 peers with equal degree, number of vertices, and colour group size.

V. Evaluation of GI CNF

We attempted to solve the instances we generated with both the glucose 2.1 [4] and lingeling [5] solvers, with a time limit of three hours per instance. Fig. 1 shows the average runtimes of either solver, for the tree different original graph sizes, and colour cluster sizes of two, three and four. One interesting result ist that changing the colour cluster size does not seem to affect the difficulty of the instance in monotone way. It may seem intuitive that a colour group smaller size might result in an easier instance, since there are fewer vertices in the second graph that any vertice in the first graph could be mapped to, and thus fewer variables in the CNF. However, this does not appear to be the case for graphs crafted from

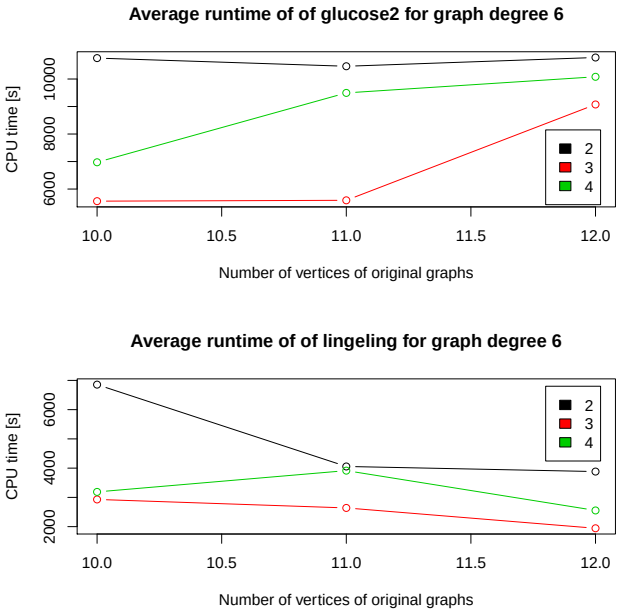

Fig. 1: Average runtime of solvers on graphs crafted from 6-regular original graphs

6-regular originals: The instances with colour groups of size two are at least as hard as the ones with larger colour groups.

Acknowledgments

The authors would like to thank the bwGRID [6] for providing the computational resources to filter the instances. The first author acknowledges funding from the Deutsche Forschungsgemeinschaft (DFG) (grant SCHO 302/9-1).

References

[1] J. Toran, "On the resolution complexity of graph non-isomorphism," in *SAT2013*, 2013.

[2] M. Meringer, "Fast generation of regular graphs and construction of cages," *Journal of Graph Theory*, vol. 30, 1999.

[3] J. yi Cai, M. Fürer, and N. Immerman, "An Optimal Lower Bound on the Number of Variables for Graph Identification," *Combinatorica*.

[4] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[5] A. Biere, "Lingeling and friends at the sat competition 2011," Institute for Formal Models and Verification, Johannes Kepler University, Tech. Rep. 11/1, 2011.

[6] bwGRiD (http://www.bw-grid.de/), "Member of the German D-Grid initiative, funded by the Ministry of Education and Research (Bundesministerium für Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Wuerttemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg)," Universities of Baden-Württemberg, Tech. Rep., 2007-2010.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

SAT encoded Low Autocorrelation Binary Sequence (LABS) Benchmark Description

Frank Mugrauer, Adrian Balint
Institute of Theoretical Computer Science
Ulm University
Germany

***Abstract*—In this paper, we describe a way to convert instances of the low autocorrelation binary sequence problem into SAT, by first converting them to instances of the pseudo boolean satisfaction problem. We describe the algorithm used for LABS to PBS conversion, and point to an already existing way of converting PBS to SAT. We then create a number of SAT instances in this fashion, and give a brief analysis of the results.**

I. LABS Description

The Low Autocorrelation Binary Sequence (LABS) problem presents itself as follows: For a given length n, find the binary sequence $x = x_1 x_2 \dots x_n$ of length n, with $x_i \in \{-1, 1\}$, where the maximum value M for the autocorrelation functions $C_k(x)$ is minimal.

$$M = \max_{k \in 1 \dots n-1} |C_k(x)|$$

$$C_k(x) = \sum_{i=1}^{n-k} x_i x_{i+k}$$

Note that this is an optimisation problem.

II. Transforming LABS to LABS PBS

The Pseudo Boolean Satisfaction problem is a constraint satisfaction problem, which allows constraints of the forms:

$$s_1 + s_2 + \dots + s_n \geq k$$

$$s_1 + s_2 + \dots + s_n = k$$

where k is a constant, and the s_i are of the form:

$$s_i = c_i * l_k * \dots * l_m$$

where the c_is are constants, and $l_k \dots l_m$ is the product of one or more Boolean literals (i.e. $l_i \in \{0, 1\}$, negated variables are allowed). The goal of PBS is to find values for the variables, which satisfy all constraints. Since LABS is an optimization problem, it cannot be directly transformed into PBS. It is, however, possible, to transform it into an (infinite) number of PBS problems, each asking "'is there a sequence with $M = k$ for some constant k. Since the values for M are typically fairly low (< 10 for $n <= 100$), creating, say, 10 PBS instances checking for target values of $M = 1 \dots 10$ for each LABS length n is a feasible approach. Algorithm 1 creates a set of PBS constraints for a given length n and target M: We need to create constraints $|C_k(x)| \leq M$ for each $k \leq n - 1$. These already look very similar to PBS constraints; we only need to correct three issues: Since PBS does not allow usage of leq and the abs() function, we need to use $\geq$ instead and split each constraint into two:

$$-C_k(x) \geq -M$$

$$C_k(x) \geq -M$$

Also, the variables in our constraints are currently in $\{-1, 1\}$, but PBS variables need to be in $\{0, 1\}$. Since the summands in our constraints currently consist of only two variables, and no constants, this can be achieved by turning a summand $x_i \cdot x_j$ into a sum:

$$1 + -2l_i\overline{l_j} + 2\overline{l_i}l_j$$

We get from an x_i to l_i by substituting a -1 with 0 and a 1 with 1, resulting in the final PBS instance having precisely n variables. The constant $+1$ here needs to be shifted to the other side of the equation, since PBS doesn't allow for summands that do not contain variables. Since we create two constraints for every C_k, and since we need to check $n - 1 C_k$s, we get a total of $2(n - 1)$ constraints.

Algorithm 1: LABS to PBS converter

Input : n, M
Output: PBS instance: set of $2(M - 1)$ PBS constraints

$C = \emptyset$;
$i = 1$;
while $i \leq n - 1$ **do**
  $C_1 = \emptyset$;
  $C_2 = \emptyset$;
  $j = 0$;
  while $j < n - i$ **do**
    $C_1 = C \bigcup (+2x_i\overline{x_j} + 2\overline{x_i}x_j)$;
    $C_2 = C \bigcup (-2x_i\overline{x_j} - 2\overline{x_i}x_j)$;
    j++;
  $C_1 = C_1 \bigcup (\leq j - M)$;
  $C_2 = C_2 \bigcup (\leq -j - M)$;
  $C = C \bigcup C_1 \bigcup C_2$;
  i++;
return C;

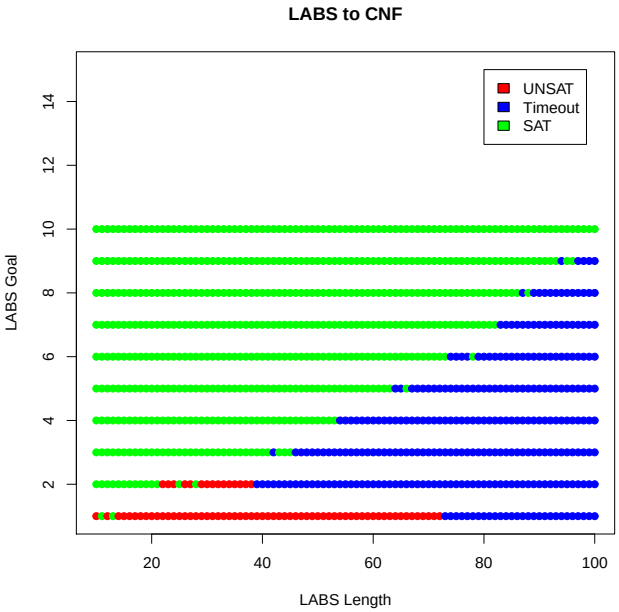

Fig. 1: Results of running glucose on the created CNF instances

III. Coding LABS PBS to LABS CNF

Transforming PBS instances to CNF form was done with the *npSolver* tool described in [1], which can solve - amongst other things - PBS problems by transforming them to SAT, then solving the SAT instances. It was modified by the authors of [1] to output the CNF formulas it creates, instead of directly solving them.

IV. Generated LABS CNF and PBS instances

To analyse the Results of the conversion process, we created PBS and CNF instances for LABS lengths of $n = 10 \dots 100$, each with targets $M = 1 \dots 10$.

V. Evaluation of LABS CNF

We attempted to solve the generated CNF instances with the glucose 2.1 solver [2] in the EDACC-System ([3],[4]), with a timeout of three hours. Glucose was able to solve instances with known optimal values (see [5]) correctly up to $n = 39$, as can be seen in fig. 1. Starting with $n = 40$ optimality of the code could not be proved any more. The runtime of the satisfiable instances is considerably lower than of the unsatisfiable ones.

Acknowledgments

The authors would like to thank the bwGRID [6] for providing the computational resources to filter the instances. The first author acknowledges funding from the Deutsche Forschungsgemeinschaft (DFG) (grant SCHO 302/9-1).

References

[1] N. Manthey and P. Steinke, "npSolver – A SAT Based Solver for Optimization Problems," in *Pragmatics of SAT(POS'12)*, 2012.

[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[3] A. Balint, D. Diepold, D. Gall, S. Gerber, G. Kapler, and R. Retz, "Edacc - an advanced platform for the experiment design, administration and analysis of empirical algorithms," in *LION'11*, 2011.

[4] A. Balint, D. Gall, G. Kappler, and R. Retz, "Experiment design and administration for computer clusters for sat-solvers (edacc), system description," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 7, 2010.

[5] J. Lindner, "Binary sequences up to length 40 with best possible autocorrelation function," *Electronics Letters*, vol. 11, no. 21, 1975.

[6] bwGRiD (http://www.bw-grid.de/), "Member of the German D-Grid initiative, funded by the Ministry of Education and Research (Bundesministerium für Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Wuerttemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg)," Universities of Baden-Württemberg, Tech. Rep., 2007-2010.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Instance generator for encoding preimage, second-preimage, and collision attacks on SHA-1

Vegard Nossum
Department of Informatics, University of Oslo
Oslo, Norway

***Abstract*—The instance generator described in this document encodes three attacks on the cryptographic hash function SHA-1. Unlike most instance generators for cryptographic hash functions, our encoding is not based purely on the Tseitin transformation. In particular, we encode modular addition using column sums represented as pseudo-boolean constraints and minimised in clausal form using the heuristic logic minimiser ESPRESSO.**

I. Introduction

SHA-1 is a cryptographic hash function that was published by the NIST in 1995 [1]. Cryptographic hash functions are functions which are "hard to invert"; in particular, this means that a given function f should satisfy the following three properties:

1) **Preimage resistance.** Given a hash H, it is infeasible to find a message M such that $f(M) = H$.
2) **Second-preimage resistance.** Given a message M, it is infeasible to find another message M' such that $f(M) = f(M')$.
3) **Collision resistance.** It is infeasible to find distinct messages M and M' such that $f(M) = f(M')$.

In this document, we describe an instance generator that encodes the corresponding attacks against the compression function of SHA-1 as SAT problems. The attack is successful if the SAT solver is able to find a solution to the instance. Given that SHA-1 was designed with these three properties in mind, we expect the resulting instances to be among the most difficult combinatorial problems for a SAT solver.

II. Parameters

A. General parameters

To seed the generator's random number generator, use `--seed` i, where i is an integer.

The type of attack to encode can be specified using one of `--attack preimage`, `--attack second-preimage`, or `--attack collision`.

B. Difficulty parameters

For preimage attacks, there are three difficulty parameters: `--rounds` t, where $16 \leq t \leq 80$, `--hash-bits` m, where $0 \leq m \leq 160$, and `--message-bits` n, where $0 \leq n \leq 512$. See section III for more information about the impact these parameters have on the expected hardness of the resulting instances.

C. Format options

The generator supports output in both CNF and OPB formats, using `--cnf` and `--opb`; exactly one of these must be given. Since the rules of the SAT Competition 2013 mandate that no comment follows the "p" line, we provide an option `--sat2011` that outputs CNF files that strictly follow the rules of the SAT Competition 2011.

III. Expected hardness

Given that SHA-1 was designed to be hard to crack, it is highly unlikely that any solver will be able to solve an instance in reasonable time; however, we have measured the mean running time for MiniSat on a series of *reduced-difficulty* instances encoding preimage attacks.

The three difficulty parameters for preimage attacks are *number of rounds*, *number of fixed hash bits*, and *number of fixed message bits*.

The full SHA-1 algorithm has 80 rounds, of which (only) the first 16 take input directly from the message to be hashed. Therefore, the possible number of rounds are between 16 and 80, where 16 is a very easy instance and 80 is a very hard instance. To see the effect of the number of rounds, we lowered the number of fixed hash bits. We observed three distinct phases; between 16 and 21 rounds, the instances are trivial to solve. Between 22 and 26 rounds, the difficulty increases extremely rapidly (an instance with 26 rounds takes approximately 2^{11} times longer to solve than an instance with 22 rounds), and from 27 rounds onwards, the difficulty increases very slowly (an instance with 80 rounds takes approximately only twice as long to solve as an instance with 27 rounds).

The number of fixed hash bits varies between 0 and 160 and effectively allows us to adjust the number of bits in the hash; with a value of 0, any message will be a solution (thus, an extremely easy instance), and with a value of 160, we require the message to hash exactly to the given hash value. The difficulty of the instance is roughly (but not quite) exponential in the number of fixed hash bits.

In the full SHA-1 algorithm, the input to the compression function is 512 bits of the message. Thus, 2^{512} is an upper limit on the size of the search space of a brute force search for a preimage. By adjusting the number of fixed message bits, we effectively give the solver parts of one known solution. By increasing this number, we effectively lower the search space of a brute force search. However, our observations indicate that by fixing a small number of bits (i.e. less than 32), the problem

becomes drastically more difficulty to solve. Only by fixing a very large number of bits (i.e. more than approximately $512 - 24$) does the problem become easier to solve.

See [2] for more detailed information about the hardness as a function of these parameters.

For the SAT Competition 2013, for instances that on average roughly take around the time limit of $5000\,\mathrm{s}$ to solve using MINISAT, we suggest the following combinations of parameters:

- 22 rounds, 128–160 hash bits, and 0 fixed message bits;
- 23 rounds, 64–96 hash bits, and 0 fixed message bits;
- 80 rounds, 8–12 hash bits, and 0 fixed message bits.

IV. ENCODING OF 5-ARY 32-BIT MODULAR ADDITION

Each round of SHA-1 includes exactly one 5-ary 32-bit adder. Expressing a constraint over 160 boolean variables (one of the inputs is an integer constant and is therefore disregarded), this actually constitutes a large part of the instance in terms of the number of clauses needed to encode it.

One very simple and frequently used way to encode addition is to use the Tseitin transformation on a standard ripple-carry adder circuit. This typically means introducing a lot of extra variables: one for each gate in the circuit. We take a different (and, we believe, novel) approach based on column sums expressed as pseudo-boolean constraints and further encoded in clausal form using the ESPRESSO heuristic logic minimiser.

Consider the following grade school addition schema for three binary numbers, x, y, and z, and their sum, w:

$$\begin{array}{cccccc} & c_3 & & & & \\ & c_2 & c_1 & & & \\ & & & c_0 & & \\ \hline & x_3 & x_2 & x_1 & x_0 \\ & y_3 & y_2 & y_1 & y_0 \\ + & z_3 & z_2 & z_1 & z_0 \\ \hline = & w_3 & w_2 & w_1 & w_0 \end{array}$$

The sum of three bits is either 0, 1, 2, or 3, and thus can be represented by a two-bit number. For the first (rightmost) column, we let $c_0 w_0$ be the sum of x_0, y_0, and z_0, and express it with the following pseudo-boolean constraint:

$$x_0 + y_0 + z_0 = 2c_0 + w_0$$

The *carry bit* c_0 is added to the next column, which is summed in a similar way. However, the sum of four bits is at most 4 and must be represented with three bits, in this case $c_2 c_1 w_1$. Thus, we obtain the constraint for the second column sum:

$$c_0 + x_1 + y_1 + z_1 = 4c_2 + 2c_1 + w_1$$

We continue in the same way for the remaining columns, with one small exception: since we are encoding *modular* addition, we will get some extraneous carry bits towards the end that should simply be discarded. These carry bits are termed *dummy* bits (as they are only ever used as placeholders for any value) and denoted with the letter d. The last two columns of this particular example are therefore encoded as follows:

$$c_1 + x_2 + y_2 + z_2 = 4d_0 + 2c_3 + w_2$$
$$c_3 + c_2 + x_3 + y_3 + z_3 = 4d_2 + 2d_1 + w_3$$

Having obtained a set of k pseudo-boolean constraints (for a k-bit adder), we now encode these constraints in CNF using ESPRESSO. Since the number of variables in each constraint is fairly small ($n + \lfloor 1 + \log_2 n \rfloor$ for an n-ary adder; at most 10 variables for 5-ary 32-bit modular addition), enumerating their truth tables (of at most 2^{10} entries) is completely feasible. The final number of clauses for each column sum depends on the constraint, but is in any case bounded by the size of its truth table.

V. COMPARISON WITH OTHER GENERATORS

We make a brief comparison with other encodings of SHA-1 preimage attacks found in the literature:

Encoding	Variables	Clauses	Ratio
Our encoding	13,408	478,476	35.69
CRYPTLOGVER [3]	44,812	248,220	5.54
Plain Tseitin [4]	$\approx$ 55,000	$\approx$ 235,000	$\approx$ 4.27

In short, our encoding has fewer variables, more clauses, and is easier to solve than the variants of the Tseitin encoding.

VI. VERIFIER

In addition to the instance generator, we also provide a *verifier* for instances encoding preimage attacks. The verifier takes the instance and a solution (as found by a SAT solver) and verifies that the solution is indeed a valid preimage for the (possibly partial) hash value encoded in the instance.

The verifier does not simply check that the solution satisfies the clauses in the instance; rather, it calculates the SHA-1 hash of the message part of the solution and checks that it matches the hash part of the solution. This ensures not only that the solver is correct, but that the encoding itself is correct. (Of course, we can only ensure that a particular solution to and encoding of a particular instance is correct, but this is good enough in practice.)

VII. AVAILABILITY

The program and scripts are available as Free Software (under the GNU General Public License version 3) from https://github.com/vegard/sha1-sat/. The program depends on the logic minimiser ESPRESSO in order to run.

REFERENCES

[1] "Secure Hash Standard," ser. FIPS, vol. 180-1. National Institute of Standards and Technology, 1995.

[2] V. Nossum, "SAT-based preimage attacks on SHA-1," Master's thesis, University of Oslo, 2012.

[3] P. Morawiecki and M. Srebrny, "A SAT-based preimage analysis of reduced KECCAK hash functions," Cryptology ePrint Archive, Report 2010/285, 2010. [Online]. Available: http://eprint.iacr.org/2010/285.pdf

[4] M. Srebrny, M. Srebrny, and L. Stepień, "SAT as a programming environment for linear algebra and cryptanalysis," in *ISAIM*, 2007.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Grain of Salt benchmarks

Mate Soos
Security Research Labs

I. Problem generator description

Grain of Salt is an advanced CNF generator for stream ciphers. It takes an input a stream cipher definition and a set of parameters how to set up the CNF and generates random problems based on the setup and its options. Grain of Salt is open-source software, available for download at https://www.gitorious.org/grainofsalt/grainofsalt/. A highly detailed description of the generator was published at the Tools for Cryptanalysis workshot at the Royal Holloway (University of London), in 2010. This 14-page description is available either from the website of the workshop, at http://www.ecrypt.eu.org/symlab/tools2010/ or from the author's website.

II. Generated problems

Two problem types were generated, one for the cipher Bivium [1] and one for HiTag2 [2]. For each, a set of key bits were given randomly as help bits, and a correct output was provided. Solution to the problem is equivelent to reversing the cipher, i.e. finding the key. Since some key bits were set randomly as help bits, this in most cases is impossible, as there is no possible key with those bits set to those values that could procude the given output. There are in fact only 2 problems that are satisfiable, both in the HiTag2 set of problems, indicated with the ending "-SAT".

Generated problems for the Bivium cipher:

- **exptime200.** 100 problems that should take about 200s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto bivium –outputs 200 –probBits 45 –num 100
- **exptime1600.** 100 problems that should take about 1600s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto bivium –outputs 200 –probBits 42 –num 100
- **exptime6400.** 100 problems that should take about 6400s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto bivium –outputs 200 –probBits 40 –num 100
- **exptime12800.** 100 problems that should take about 12800s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto bivium –outputs 200 –probBits 39 –num 100

Generated problems for the HiTag2 cipher:

- **exptime400.** 100 problems that should take about 400s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto hitag2 –outputs 60 –probBits 12 –num 100
- **exptime1200.** 100 problems that should take about 1200s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto hitag2 –outputs 60 –probBits 10 –num 100
- **exptime4800.** 100 problems that should take about 5800s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto hitag2 –outputs 60 –probBits 8 –num 100
- **exptime12800.** 100 problems that should take about 12800s on average to solve with current SAT solving technology. Used command line: ./grainofsalt –crypto hitag2 –outputs 60 –probBits 7 –num 100

III. Comments in the CNF

All CNFs generated have large amounts of information in comments, such as the definition of each and every variable and the definition of each and every set of clauses. As such, every step the SAT solver takes can be mapped back to the original problem itself.

References

[1] Raddum, H.: Cryptanalytic results on Trivium. Technical Report 2006/039, ECRYPT Stream Cipher Project (2006) www.ecrypt.eu.org/stream/papersdir/2006/039.ps.

[2] Nohl, K.: Description of HiTag2. Press release (March 2008) http://cryptolib.com/ciphers/hitag2/.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Propositional Satisfiability Benchmarks Constructed from Multi-Robot Path Planning on Graphs

Pavel Surynek

***Abstract* — A propositional satisfiability (SAT) benchmark motivated by planning paths for multiple robots on graphs is described in this short paper. It is suggested to model the question if robots can find paths in a graph to given goal vertices in the given number of time steps as propositional satisfiability. The problem, its propositional model, and benchmark generator for grid environments are described.**

I. Where the Multi-Robotic Problem Comes From

Multi-robot path planning (MRPP, also referred *as cooperative path-finding – CPF*) on graphs [5], [6] is an abstraction for centralized navigation of multiple mobile robots (distinguishable but same in other aspects). Each robot has to relocate itself from a given initial location to a given goal location while it must not collide with other robots and obstacles. Plans as sequences of movements for each robot are constructed in advance by a centralized planner which can fully observe the situation.

The problem of navigating a group of mobile robots or other movable units has many practical applications. Except the classical case with mobile robots let us mention traffic optimization, relocation of containers [5], or movement planning of units in RTS computer games.

To be able to tackle the problem a graph-based abstraction is often adopted – the environment is represented as an undirected graph with at most one robot in a vertex. Edges can be traversed by robots.

We describe a MRPP problem formally and develop SAT encoding for it in the following sections. Then an instance generator for MRPP on 4-connected grids is described.

II. From Graph Formulation to SAT Encoding

Our encoding of MRPP will be introduced through finite domain integer programming. After creating integer model, the integer variables and constraints will be replaced with vectors of propositional variables (bit-vectors) and corresponding clauses.

A. Multi-robot Path Planning on Graphs (MRPP)

Let $G = (V, E)$ be an undirected graph and let $R = \{r_1, r, \ldots, r_n\}$ be a set of robots where $|R| < |V|$. The arrangement of robots in G will be described by a uniquely invertible function $\alpha: R \rightarrow V$. The interpretation is that a robot $r \in R$ is located in a vertex $\alpha(r)$. A generalized inverse of α denoted as $\alpha^{-1}: V \rightarrow R \cup \{\perp\}$ will provide us a robot located in a given vertex or $\perp$ if the vertex is empty.

An arrangement of robots at time step $i \in \mathbb{N}_0$ will be denoted as α_i. If we formally express rules on movements in terms of location function then we have following *transition constraints*:

(i) $\forall r \in R$ either $\alpha_i(r) = \alpha_{i+1}(r)$ or $\{\alpha_i(r), \alpha_{i+1}(r)\} \in E$ holds
(*robots move along edges or do not move at all*),

(ii) $\forall r \in R \;\; \alpha_i(r) \neq \alpha_{i+1}(r) \Rightarrow \alpha_i^{-1}(r) = \perp$
(*robots move to empty vertices only*), and

(iii) $\forall r, s \in R \;\; r \neq s \Rightarrow \alpha_{i+1}(r) \neq \alpha_{i+1}(s)$
(*no two robots enter the same target vertex*).

The initial arrangement is α_0 and α^+ will denote the goal arrangement. An instance of MRPP is then given as quadruple $[G, R, \alpha_0, \alpha^+]$. The task is to transform α_0 to α^+ so that transition constraints are preserved between all the consecutive time steps.

Definition 1 (solution, makespan). Let $\Sigma = [G, R, \alpha_0, \alpha^+]$ be an instance of CPF. A *solution* of Σ is a sequence of arrangements $\alpha_0, \alpha_1, \ldots, \alpha_\mu$ where $\alpha_\mu = \alpha^+$ and transition constraints are satisfied between α_{i-1} and α_i for every $i = 1, \ldots, \mu$. The number μ is called a *makespan* of the solution. The shortest possible makespan of Σ will be denoted as $\mu^*(\Sigma)$. □

It is known that finding $\mu^*(\Sigma)$ is NP-hard [4]. If makespan sub-optimal solution is sufficient then polynomial time solving techniques from [3] can be used. An example of MRPP instance on a graph represented by a 4-connected grid is shown in Figure 1.

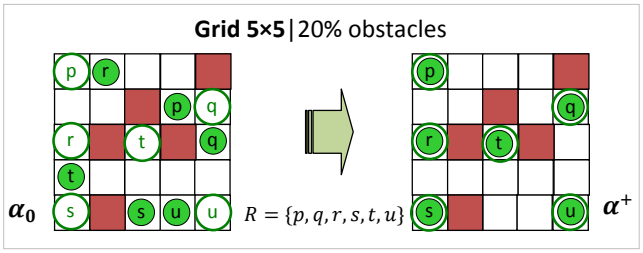

Figure 1. A typical **random MRPP instance on a grid of size 5×5** with 20% of positions occupied by obstacles.

B. k-Level MRPP Encoding as Integer Programming

An incomplete approach from domain independent planners SASE [1] and SATPlan [2] can be adopted to find makespan optimal solutions of MRPP. A question whether there exists a solution of the given MRPP of makespan k is modeled as propositional satisfiability. A solution of the optimal makespan can be found by trying larger and larger makespans in a case the MRPP instance is solvable (the unsolvability cannot be detected by this approach).

Unlike domain independent planners SASE and SATPlan we use a propositional encoding specially designed for

Pavel Surynek is with Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, Malostranské náměstí 25, 118 00 Praha 1, Czech Republic (phone: +420 221 914 245; fax: +420 221 914 323; e-mail: pavel.surynek@mff.cuni.cz).

MRPP. The employed encoding called *inverse* has been developed in [7] and is significantly smaller in terms of the number of variables and clauses than SASE and SATPlan encodings on the same MRPP instances.

Basically we need to model arrangements of robots at individual time steps and introduce transition constraints into the model. In the inverse encoding, the arrangement of robots at time step i is modeled by state variables $\mathcal{A}_i^v$ for $v \in V$ that represent inverse location function at the time step i. Next, there are state variables $\mathcal{T}_i^v$ for $v \in V$ that represent actions taken in vertices at time step i. An *outgoing action* into some of vertex neighbors or an *incoming action* from some of vertex neighbors or *noop* can be taken in each vertex. The domain of $\mathcal{T}_i^v$ consists of $2\deg_G(v)+1$ values to represent all the possible actions. It is necessary to introduce some ordering on neighbors of each vertex to be able to assign concrete actions to elements of the domain of $\mathcal{T}_i^v$. Suppose that we have a function $\sigma_v: \{u | \{v,u\} \in E\} \longrightarrow \{1,2,\ldots,\deg_G(v)\}$ and its inverse σ_v^{-1} that implements this ordering of neighbors.

Definition 2 (inverse encoding). The i-th level of *inverse encoding* consists of the following integer interval **state variables**:

- $\mathcal{A}_i^v \in \{0,1,2,\ldots,n\}$ for all $v \in V$ such that
  $\mathcal{A}_i^v = j$ iff $\alpha_i(r_j) = v$
- $\mathcal{T}_i^v \in \{0,1,2,\ldots,2\deg_G(v)\}$ for all $v \in V$ such that
  $\begin{cases} \mathcal{T}_i^v = 0 & \text{iff no-op was selected in } v; \\ \mathcal{T}_i^v = \sigma_v(u) & \text{iff an outgoing primitive action with the target } u \in V \text{ was selected in } v; \\ \mathcal{T}_i^v = \deg_G(v) + \sigma_v(u) & \text{iff an incoming primitive action with } u \in V \text{ as the source was selected in } v. \end{cases}$

and **constraints**:

- $\mathcal{T}_i^v = 0 \Rightarrow \mathcal{A}_{i+1}^v = \mathcal{A}_i^v$ for all $v \in V$ (**no-op** case);
- $0 < \mathcal{T}_i^v \leq \deg_G(v) \;\Rightarrow\; \mathcal{A}_i^u = 0 \wedge \mathcal{A}_{i+1}^u = \mathcal{A}_i^v \wedge \mathcal{T}_i^u = \sigma_u(v) + \deg_G(u)$ where $u = o_v^{-1}(\mathcal{T}_i^v)$ for all $v \in V$ (**outgoing** robot case);
- $\deg_G(v) < T_i^v \leq 2 * \deg_G(v) \Rightarrow \mathcal{T}_i^u = \sigma_u(v)$ where $u = \sigma_v^{-1}(\mathcal{T}_i^v - \deg_G(v))$ for all $v \in V$ (**incoming** robot case). □

C. Translation of IP Model of MRPP to SAT

The encoding is built upon integer finite domains variables. We eventually need propositional encoding which is obtained by translating integer state variables into bit vectors. If the state variable has N states (N elements in its domain) then we need $\lceil \log_2 N \rceil$ propositional variables to represent it.

If we are asking whether there is a solution of makespan k we need to build k levels. The initial arrangement α_0 is encoded in $\mathcal{A}_0^v$. Analogically $\mathcal{A}_k^v$ are set to the goal arrangement α^+.

III. MRPP on Grids Instance Generator

A classical MRPP benchmark introduced in [6] takes place on a 4-connected grid of certain size into which obstacles are placed randomly by excluding randomly selected nodes. Initial and goal positions for robots are random as well. In all the cases random selection is uniform from the set of remaining items. Our instance generator produces SAT encodings for these benchmarks. Several parameters are accepted by the generator:

- *size* of the grid – dimensions *height* × *width*
- *probability* of obstacles – placed randomly/uniformly
- number of *robots* – placed randomly/uniformly
- number of *levels* – corresponds to the *makespan*
- random *seed*

A. Simple Knowledge Compilation into the SAT Encoding

A simple knowledge compilation into the presented encoding is done by our instance generator. It is checked if a given robot can occur in a given vertex at a given time step. Such occurrence of a robot excludes existence of a solution if the vertex cannot be reached from the initial position in the given number of time steps or if the goal position cannot be reached in the remaining number of time steps along shortest paths.

B. Properties, Parameters and Difficulty

A property having the most significant impact on the difficulty of MRPP solving is the **intensity of interactions** among robots during their movement. It is more difficult to solve a problem when robots need to intensively avoid each other regardless of the solving method applied [7], [8]. Intensity of interaction is directly changed by the *size* of the grid, probability of *obstacles*, and the number of *robots*.

Notice also that the SAT model encodes bounded MRPP by certain number of levels. The most difficult cases appear for the number of *levels* around the optimal makespan [2]. On the other hand instances with few levels can be quickly identified as unsolvable. However, it is typically more difficult to discover solvability of instances with many levels due to increasing size of the instance.

Discussion and Future Work

Several other encodings of MRPP were investigated by the author. The presented *inverse* encoding is the most compact one if the number of robots is relatively high.

There is still room for improving encodings by compiling more sophisticated knowledge into it. Further compacting the encoding at the bit level is also planned.

References

[1] R. Huang, Y. Chen, W. Zhang. "A Novel Transition Based Encoding Scheme for Planning as Satisfiability," Proceedings of AAAI 2010, AAAI Press, 2010.

[2] H. Kautz, B. Selman, "Unifying SAT-based and Graph-based Planning," Proceedings of IJCAI 1999, pp. 318-325, Morgan Kaufmann, 1999.

[3] D. Kornhauser, G. L. Miller, P. G. Spirakis, "Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications," Proceedings of FOCS 1984, pp. 241-250, IEEE Press, 1984.

[4] D. Ratner, M. K. Warmuth, "Finding a Shortest Solution for the N × N Extension of the 15-PUZZLE Is Intractable." Proceedings of AAAI 1986, pp. 168-172, Morgan Kaufmann, 1986.

[5] M. R. K. Ryan, "Exploiting Subgraph Structure in Multi-Robot Path Planning," JAIR, Volume 31, 2008, pp. 497-542, AAAI Press, 2008.

[6] D. Silver, "Cooperative Pathfinding," Proceedings of AIIDE 2005, pp. 117-122, AAAI Press, 2005.

[7] P. Surynek, "Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving," Proceedings of PRICAI 2012, pp. 564-576, LNCS 7458, Springer, 2012.

[8] T. S. Standley, "Finding Optimal Solutions to Cooperative Pathfinding Problems," Proceedings of AAAI 2010, AAAI Press, 2010.

Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

ARCFOUR Equivalence Checking

Sean Weaver

University of Cincinnati, USA

Marijn J. H. Heule

The University of Texas at Austin, USA

Introduction

In cryptography, ARCFOUR is the most widely used software stream cipher and is used in popular protocols such as Secure Sockets Layer (SSL) (to protect Internet traffic) and WEP (to secure wireless networks). While remarkable for its simplicity and speed in software, ARCFOUR has weaknesses that argue against its use in new systems. It is especially vulnerable when the beginning of the output keystream is not discarded, or when nonrandom or related keys are used; some ways of using ARCFOUR can lead to very insecure cryptosystems such as WEP.

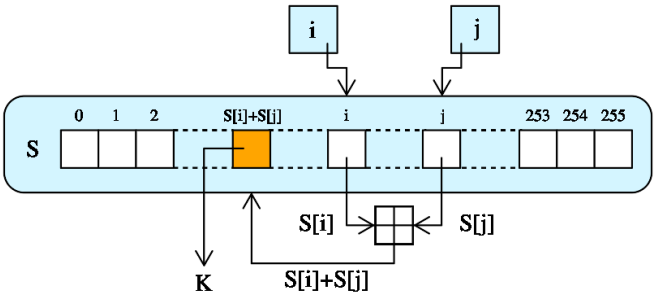

Fig. 1. The lookup stage of ARCFOUR. The output byte is selected by looking up the values of $S[i]$ and $S[j]$, adding them together modulo 256, and then looking up the sum in S; $S[S[i] + S[j]]$ is used as a byte of the key stream, K.

Key-Scheduling Algorithms

ARCFOUR generates a pseudorandom stream of bits (a keystream). As with any stream cipher, these can be used for encryption by combining it with the plaintext using bit-wise exclusive-or; decryption is performed the same way (since exclusive-or with given data is an involution). To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

- A permutation of all 256 possible bytes (denoted S below).
- Two 8-bit index-pointers (denoted i and j).

The permutation is initialized with a variable length key, typically between 40 and 256 bits, using the key-scheduling algorithm (see Fig. 1).

The key-scheduling algorithm is used to initialize the permutation in the array S. $keylength$ is defined as the number of bytes in the key and can be in the range $1 \leq keylength \leq 256$, typically between 5 and 16, corresponding to a key length of 40 to 128 bits.

keyScheduleA $(k) = S$

1 **for** $i = 0$ to 255 **do** $S[i] = i$
2 $j = 0$
3 **for** $i = 0$ to 255 **do**
4 $j := (j + S[i] + key[i \ \% \ keylength]) \ \% \ 256$
5 $tmp = S[i]$
6 $S[i] = S[j]$
7 $S[j] = tmp$

Now consider *key_schedule_B* which changes Line 4 by swapping j and $key[i \ \% \ keylength]$ in the addition.

keyScheduleB $(k) = S$

1 **for** $i = 0$ to 255 **do** $S[i] = i$
2 $j = 0$
3 **for** $i = 0$ to 255 **do**
4 $j := (key[i \ \% \ keylength] + S[i] + j) \ \% \ 256$
5 $tmp = S[i]$
6 $S[i] = S[j]$
7 $S[j] = tmp$

The benchmarks in this suite encode that

forall k, *keyScheduleA*(k) = *keyScheduleB*(k)

for various lengths of k and bit-widths of i and j.

Parts of the description are from http://en.wikipedia.org/wiki/RC4

Solver Index

Benchmark Index

Author Index