Proceedings of
SAT CHALLENGE 2012
Solver and Benchmark Descriptions

Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, & Carsten Sinz *(editors)*

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2012-2

ISSN 1458-4786
ISBN 978-952-10-8106-4 (PDF)
Helsinki 2012

PREFACE

The area of 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, we want to motivate implementors to present their work to a broader audience and to compare it with that of others.

SAT Challenge 2012 (SC 2012), a competitive event for solvers of the Boolean Satisfiability (SAT) problem, took place within 2012. It was organized as a satellite event to the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012) and stands in the tradition of the SAT Competitions held yearly from 2002 to 2005 and biannually starting from 2007, and the SAT-Races held in 2006, 2008 and 2010.

SC 2012 consisted of 5 competition tracks, including three main tracks for sequential solvers (*Application SAT+UNSAT* containing problem encodings (both SAT and UNSAT) from real-world applications, such as hardware and software verification, bio-informatics, planning, scheduling, etc; *Hard Combinatorial SAT+UNSAT* containing combinatorial problems (both SAT and UNSAT) to challenge current SAT solving algorithms, similar to the SAT Competition's category "crafted"; and *Random SAT*, containing randomly generated satisfiable instances); one track for parallel solvers (with eight computing cores, using Application SAT+UNSAT instances); and one track for sequential portfolio solvers (1/3 Application SAT+UNSAT, 1/3 Hard Combinatorial SAT+UNSAT, and 1/3 Random SAT+UNSAT instances).

There were two ways of contributing to SC 2012: in the form of submitting one or more solvers for competing in one or more of the competition tracks, and in the form of submitting interesting benchmark instances on which the submitted solvers could be evaluated on in the competition. The rules of SC 2012 required all contributors (both solver and benchmark submitters) to submit a short, around 2-page long solver/benchmark description as part of their contribution. As a result, we received high-quality descriptions that we believe to be of value to the research community at large both at present and in the future. 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 2012 competition tracks.

SC 2012 was run under the Experiment Design and Administration for Computer Clusters (EDACC) platform using the bwGrid computing infrastructure operated by eight Baden-Württemberg state universities, both providing critical infrastructure for successfully running the competition.

Last but not least, we would like to thank all those who contributed to SC 2012 by submitting either solvers or benchmarks, and for contributing the solver and benchmark descriptions that form the core of this proceedings volume. We hope this compilation provides the reader new insights into the details of state-of-the-art SAT solver implementations and the SC 2012 benchmarks.

Dublin, Helsinki, Karlsruhe, and Ulm, June 12, 2012
SAT Challenge 2012 Organizers
Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, & Carsten Sinz

Contents

Benchmark Descriptions

SOLVER DESCRIPTIONS

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

BossLS
Preprocessing and Stochastic Local Search

Oliver Gableske
Institute of Theoretical Computer Science
Faculty for Engineering and Computer Science
Ulm University
Baden-Württemberg, Germany
Contact: https://www.gableske.net/oliver

***Abstract*—This paper briefly describes the `BossLS` SAT Solver by explaining its basic functionality and outlining its features.**

I. The k-SAT Problem

In the following, let F be a Boolean formula in conjunctive normal form (CNF), containing the n Boolean variables $\mathcal{V} = \{x_1, \ldots, x_n\}$ in the form of $2n$ literals $\mathcal{L} = \{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$. A formula in CNF is a conjunction (and) of disjunctions (or) of literals. The disjunctions of literals are called clauses.

An assignment $\alpha : \mathcal{V} \rightarrow \{0, 1\}$ satisfies a formula in CNF, if and only if it assigns values to the variables such that in each clause there is at least one literal evaluating to true. Then, all clauses evaluate to true, and the formula's conjunction evaluates to true as well ($\alpha(F) = 1$). Such assignments are called satisfying assignments or solutions (for F). We call F satisfiable, if and only if there is at least one satisfying for it.

SAT is a language that consists of all satisfiable Boolean formulas. The restriction to formulas in CNF, where a clause has at most k literals, is called k-SAT. The k-SAT problem is the problem to decide whether a given formula F is in k-SAT, that is, decide if the given formula is satisfiable.

II. General Idea behind BossLS

A SAT Solver is an algorithm that might solve the (k-)SAT problem for a given (CNF-)formula F. We call such a SAT Solver complete, if it can decide $F \in$ SAT and $F \notin$ SAT, and incomplete if it can decide $F \in$ SAT but not $F \notin$ SAT.

The `BossLS` SAT solver is an incomplete SAT solver, following the approach of stochastic local search (SLS). Its basic functionality is equal to solvers like `Sparrow` [1] or `gNovelty+` [8].

The general approach of these solvers is to assume that F is satisfiable by some (randomly created) assignment, and perform search by making local modifications to the assignment in order to increase the number of satisfied clauses. If a solution is found, it is used as a proof to support the claim that F is indeed satisfiable. If F is not satisfiable, the algorithm will not terminate.

To be more precise: given F with n variables, SLS solvers will first create a random starting assignment α. They then check which of the clauses from F are not satisfied under α, that is $\alpha(C_i) = 0$. Let this set of clauses be $\mathcal{U}$. If $\mathcal{U} = \emptyset$ then all clauses are evaluating to true, and then α is a solution. In this case, the search is over.

If $\mathcal{U} \neq \emptyset$, α can be no solution. SLS solvers then typically pick one of the clauses $C_i \in \mathcal{U}$ at random and try to fix the assignment α, such that $\alpha(C_i) = 1$. This is done by selecting exactly one of the literals in α in order to invert its assignment. After flipping the assignment to the corresponding variable, and thereby modifying α into α', the solvers will again check if all clauses are satisfied under this modified α'. The major difference between the solvers named above is how exactly they pick the literal from C_i in order to flip the corresponding variable assignment.

III. Picking Literals for Flipping the corresponding Variable Assignment

Let $C_i \in F$ be a clause and let $\alpha(C_i) = 0$. Assume, that C_i has been selected for *fixing* as explained in the previous section. Let w.l.o.g. $C_i = (l_1 \vee \ldots \vee l_k)$. The `BossLS` solver will now investigate each l_i and count the number of clauses that would become unsatisfied if it would indeed flip the assignment to the corresponding variable. This is called the break count of the variable and is denoted with $b_{l_j} := b(l_j, \alpha, F) = b(\neg l_j, \alpha, F)$. Using these break counts, and a parameter called c_b (the break base parameter value), the solver computes a function f of values for each $l_j \in C_i$:

$$f(l_j, \alpha, F) = (c_b)^{-b_{l_j}} \tag{1}$$

The probability for each literal in the clause to be selected for flipping then follows by computing [2]:

$$P(\text{select } l_j \in C_i) = \frac{f(l_j, \alpha, F)}{\sum_{l_w \in C_i} f(l_w, \alpha, F)}. \tag{2}$$

According to this distribution, it picks a literal at random and then flips the corresponding variable assignment. According to [2], this scheme gives superior performance for random 3-SAT formulae. We refer the reader to this paper for additional information.

IV. Handling Crafted Instances

A. Preprocessing

The `BossLS` solver was supposed to give good performance on crafted formulae, too. This is why a preprocessor

was implemented in order to simplify the formulas before they are being handed over to the local search component within the solver. The preprocessing consists of unit propagation, failed literal detection, and asymmetric covered clause elimination (ACCE). ACCE is an extension to asymmetric blocked clause elimination (ABCE), additionally using covered literal addition (CLA). See [4], [5], [6] for an overview of clause elimination techniques.

All the techniques named above have in common that they cannot increase the clause count m of the formula. In contrast to unit propagation and failed literal detection, ACCE does not even decrease the size of the clauses but drops them completely, if possible. Furthermore, ACCE is preserving satisfiability equivalence, but not logical equivalence. This means, that for a given unsatisfiable formula F the resulting formula ACCE(F) will stay unsatisfiable, but if F is satisfiable, the set of satisfying assignments for F might change for ACCE(F). Since ACCE can only drop clauses, it can only *increase* the amount of satisfying assignments to a satisfiable formula F.

Preliminary tests suggested, that these techniques are beneficial for SLS on crafted formulas as they seem to increase the success rate of `BossLS` where ACCE does indeed succeed in removing a significant amount of clauses. We assume that the ability of ACCE to increase the set of satisfying assignments for a satisfiable formula is the main reason for this increased success rate. On formulae where ACCE does not remove a significant amount of clauses the preprocessing at least did not seem to degrade the solver's performance.

After a closer investigation, however, we found that CE does not necessarily improve the performance. We were able to find formulae (for example the em* from the SAT 2011 Competition crafted set), where a large number of clauses can be dropped by CE, but afterwards, the performance of the SLS is much worse then before. A paper dealing with this is currently in preparation.

B. Restarts

Fixed interval restarts have also been implemented, and are in use on crafted formulae. This type of restart is the most simple one possible: it picks the number of variables n and multiplies it with a fixed constant (currently 640). If the solver did this amount of flips, it performs a flip on all variables in a row with probability 50% for each variable.

C. Tabu

The solver could use a very simple tabu scheme (tabu-1) that penalizes the variable flipped last. The penalty is realized by increasing the variables break count b_{l_j} by 1, which will impact the probability of the variable being flipped depending on the parameter c_b chosen for equation (1).

Restarts and tabu are optional compile features. For the SAT 2012 Challenge, `BossLS` was compiled *with* restarts but *without* tabu.

V. Miscellaneous

The solver has been implemented in a way that is supposed to make it CPU cache friendly. That is, break scores for literals are stored along with the literal's occurrence list pointer (as you usually need both, if you access the literal). The clause data, like the current number of satisfied literals and a representative satisfied literal of a clause under the current assignment, are separated from the list of literals of the clause (as the solver usually does not need them). It is disadvantageous to store both in one array, as the infrequently used literal information supersedes the relevant information from the CPU cache as soon as the clause data is touched.

ACCE, and ABCE alike, are preprocessing techniques that may increase the number of satisfying assignments to a satisfiable formula. This means, that after the preprocessed formula is solved, a solution to the original formula must be reconstructed. The ABCE, the ABCE solution reconstruction algorithm, and the ACCE algorithm have been published in [4], [6]. The ACCE solution reconstruction algorithm is, however, not yet published. Therefore, the ACCE itself is available in the `BossLS`, but we commented out the CLA (that lifts ABCE to ACCE), and deleted the solution reconstruction algorithm from the sources in order to not leak the details. The source-code available from the website [7] currently performs ABCE and not ACCE. Even though the CLA/ACCE itself is still in the sources, without the proper reconstruction algorithm it does not make sense to activate CLA.

You can get the latest sources of the solver at [7]. The solver and its sources are published under the GNU General Public License version 3.

Acknowledgments

The author would like to thank Marijn Heule for advices and technical help. Additionally, the author would like to thank Armin Biere and Matti Järvisalo for providing not yet published information. Further thanks go to Timo Beller for fruitful discussions regarding the compressed assignments implementation.

References

[1] Balint, A., Fröhlich, A.: Improving Stochastic Local Search for SAT with a New Probability Distribution. Proc. of the 13th international Conference of the Theory and Application of Satisfiability Testing, Springer LNCS 6175, p. 10-16, 2010.

[2] A. Balint and U. Schöning, *Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break*, To be published in Theory and Applications of Satisfiability Testing – SAT 2012, Lecture Notes in Computer Science (LNCS), Springer, 2012.

[3] A. Biere and M. J. H. Heule and H. v. Maaren and T. Walsh (Eds.). *Handbook of Satisfiability*, IOS Press, 2009, ISBN 978-1-58603-929-5.

[4] M. J. H. Heule and M. Järvisalo and A. Biere, *Clause Elimination Procedures for CNF Formulas*, In Christian Fermller and Andrei Voronkov (Eds.) Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-17), LNCS 6397, pages 357–371. Springer, 2010.

[5] M. J. H. Heule and M. Järvisalo and A. Biere, *Clause Elimination Procedures*, a manuscript in preparation, 2012.

[6] M. J. H. Heule and M. Järvisalo and A. Biere, *Covered Clause Elimination*, In Christian Fermüller and Andrei Voronkov (Eds.) Short Paper Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, 2011 (to appear)

[7] O. Gableske, *The BossLS project homepage*, Get the latest sources here: https://www.gableske.net/bossls.

[8] D. N. Pham and C. Gretton, *gNovelty+*, SAT solver description from the SAT 2007 Competition. See http://www.satcompetition.org/2007/gNovelty+.pdf.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

CaGlue: Particular Clause Analysis in a CDCL Solver

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

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

I. Major solving techniques

The following description concerns the submitted solver *caglue*. This solver is based on an existing implementation of a CDLC like solver. Indeed, *caglue* is implemented under the *glucoe* solver [1] (without SatElite formula simplification [2]).

CDLC solvers start the conflict analysis on the basis of the first falsified clause (which we will note by c) reached during the propagation phase of the enqueued literals. Classically, this analysis is done according to the first implication point [3] by applying a sequence of resolutions between the clauses involved by the conflict. The clause c is the first to be used in this sequence. Also, for combinatorial reasons, keeping all learnt clauses during the search is shown to be unsuccessful. Hence, some learnt clauses are kept, and the other dropped, according to some parameters (clause activities, LBD values ...). However, what about the relevance of learnt clauses regarding to conflict analysis? Is it relevant to accomplish the analysis on the first reached empty clause? Is there any difference in the behavior of a CDCL solver if we restrict the analysis on the basis of a particular empty clause?

Accordingly, the main purpose of our solver *caglue* is to continue the propagation even if a conflict is reached and to stop it under given criteria. Such implementation modify the behavior of the *glucose* solver.

II. Parameter description

In *caglue*, the only difference with *glucose* is that the propagation of the enqueued literals is stopped, in this case of a conflict, if the falsified clause c is:

1) among the original clauses, or
2) For one case on two, c is a learnt clause such that its LBD value ≤ 2 or its size ≤ 3,
3) For the other case, stop at the first falsified learnt clause.

These criteria are near to those used in *glucose* to reduce the size of the database of the learnt clauses.

III. Special Algorithms, Data Structures and Features

In*caglue*, there is no preprocessing step. The data structures are strictly similar to the existing ones in *glucose*.

IV. Implementation detail

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

V. SAT Challenge 2012 Specifics

1) The solver is submitted in "Solver Testing Track" including : Hard Combinatorial SAT+UNSAT and Application SAT+UNSAT.
2) The used compiler is *g++*
3) The optimization flag used is "-O3". The compilation options are the same as the used existing solver.

VI. Availability

Our solver is not yet publicly available.

Acknowledgment

We would like to thank the authors of *glucose*[1] for making available the source code of their solver.

References

[1] 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.

[2] 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.

[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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

CCASat: Solver Description

Shaowei Cai and Chuan Luo
School of Electronics Engineering and Computer Science
Peking University, Beijing, China
shaowei_cai@126.com, chuanluosaber@gmail.com

Kaile Su
Institute for Integrated and Intelligent Systems
Griffith University, Brisbane, Australia
k.su@griffith.edu.au

***Abstract*—An interesting strategy called configuration checking (CC) was recently proposed to deal with the cycling problem in local search for Minimum Vertex Cover. A natural question is whether this CC strategy also works for SAT. The direct application of CC did not result in stochastic local search (SLS) algorithms that can compete with the current best SLS algorithms for SAT. We propose a new heuristic based on CC for SLS algorithms for SAT, which is called configuration checking with aspiration (CCA). In CCA, there are two levels with different priorities in the greedy mode. Those variables with great scores have a chance to be selected on the the second level, even if they do not satisfy the CC criterion. The CCA heuristic is used to develop a new SLS algorithm called CCASat.**

I. Introduction

The basic schema for an SLS algorithm for SAT is as follows: Beginning with a random complete assignment of truth values to variables, in each subsequent search step a variable is chosen and flipped. We use *pickVar* to denote the function for choosing the variable to be flipped.

SLS algorithms for SAT usually work in two different modes, i.e., the greedy (intensification) mode and the diversification mode. In the greedy mode, they prefer variables whose flips can decrease the number of unsatisfied clauses; in the diversification mode, they tend to better explore the search space, usually using randomized strategies and exploiting diversification properties to pick a variable.

Recently, a diversification strategy called configuration checking (CC) was proposed, which may help deal with the cycling problem, i.e., revisiting a candidate solution that has been visited recently [1]. It was recently proposed to deal with this issue, and was used to improve a state-of-the-art Minimum Vertex Cover (MVC) local search algorithm called EWLS [2], which leads to the much more efficient SLS solver EWCC for MVC [3]. A natural question is whether this CC strategy also works for SAT.

According to the CC strategy for MVC in [3], it is easy to develop a CC strategy for SAT, which forbids a variable to flip if since its last flip all its neighboring variables have not changed their truth values. Actually, this has been used in an SLS algorithm called Swcc. However, Swcc cannot compete with the current best SLS solvers such as Sparrow2011 [4]. In our opinion, the CC strategy is too strict for SLS algorithms for SAT, as it forbids all variables whose circumstance has not changed since its last flip to be flipped, regardless of its score. We believe this lack of differentiation is a big disadvantage.

We propose a new heuristic based on CC for SLS algorithms for SAT. We name it Configuration Checking with Aspiration (CCA), as this new heuristic utilizes a mechanism which is inspired by the aspiration mechanism in tabu search. According to CCA, there are two levels with different priorities in the greedy mode. Those variables whose flips can bring a big benefit have a chance to be selected on the the second level, even if they do not satisfy the CC criterion.

II. Configuration Checking

Originally introduced in [3], configuration checking (CC) is a diversification strategy aiming to reduce the cycling problem in local search. The CC strategy is based on the concept *configuration*. In the context of SAT, the configuration of a variable refers to truth values of all its neighboring variables.

Let $V(F)$ denote the set of all variables appear in the formula F, and $N(x) = \{y | y \in V(F)$ and y occurs in at least one clause with $x\}$ is the set of all *neighboring variables* of variable x. The formal definition of the configuration of a variable is given as follows:

Definition 1: Given a CNF formula F and s the current assignment to $V(F)$, the $configuration$ of a variable $x \in V(F)$ is a vector C_x consisting of truth values of all variables in $N(x)$ under s (i.e., $C_x = s|_{N(x)}$, which is the assignment restricted to $N(x)$).

Given a CNF formula F, the CC strategy can be described as follows: When selecting a variable to flip, for a variable $x \in V(F)$, if the configuration of x has not been changed since x's last flip, which means the circumstance of x never changes, then it is forbidden to be flipped. To implement the CC strategy, we employ an array $confChange$, whose element is an indicator for a variable — $confChange[x] = 1$ means the configuration of variable x has been changed since x's last flip; and $confChange[x] = 0$ on the contrary. During the search procedure, the variables with $confChange[x] = 0$ are forbidden to be flipped.

III. Configuration Checking with Aspiration

As we have pointed out, the CC strategy is too strict in picking a variable to flip in the greedy mode. Any variable whose configuration has not been changed since its last flip is forbidden to be flipped in the greedy mode, regardless of its score. To overcome this drawback, we propose a new *pick-var* heuristic based on CC, which is called configuration checking with aspiration (CCA) [5].

We first give some definitions. 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) $\overline{w}$.

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 the oldest variable in a random unsatisfied clause is picked to flip.

IV. The CCASat Solver

We use the CCA heuristic to develop a new SLS algorithm called CCASat, which has been submitted to SAT Challenge 2012, for the Random SAT track.

A. Clause Weighting in CCASat

CCAsat uses two different clause weighting schemes, one for 3-SAT and structured SAT (i.e., not k-SAT), and the other for large k-SAT ($k > 3$).

Clause Weighting for 3-SAT and non-k-SAT: The details of the CCA heuristic and the clause weighting scheme for 3-SAT and structured SAT can be found in [5], as described briefly as follows.

We adopt a 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 $\overline{w}$ exceeds a threshold γ, all clause weights are smoothed as $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor (1-\rho)\overline{w} \rfloor$.

For SAT Challenge 2012, we set $\gamma = 300$ and $\rho = 200 + \frac{|V(F)|+250}{500}$.

Clause Weighting for large k-SAT: We adopt a clause weighting scheme similar to PAWS. 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.

For SAT Challenge 2012, sp is set to 0.75 for k-SAT with $3 < k \leq 5$, and 0.92 for k-SAT with $k > 5$.

B. Implementation

CCASat is implemented in C++ on the basis of the codes of Swcca [5]. It is compiled by g++ with the following command:

g++ cca.cpp -m32 -O2 -static -o CCASat.

Its running command is:

CCASat <instance file name> <random seed>.

V. Conclusions

Inspired by the success of the configuration checking (CC) strategy on the Minimum Vertex Cover problem, we proposed a new variable selection heuristic called configuration checking with aspiration (CCA) for SLS algorithms for SAT. The CCA heuristic works on two levels in the greedy mode, which is more flexible compared to the CC strategy. We utilized the CCA heuristic to develop a new SLS algorithm called CCASat. We would like to note CCA can be seen as a local search framework, which can be combined with various techniques such as different clause weighting schemes.

VI. Acknowledgements

We would like to thank Zhong Jie for testing our solver on the EDACC platform.

References

[1] W. Michiels, E. H. L. Aarts, and J. H. M. Korst, *Theoretical aspects of local search*. Springer, 2007.

[2] S. Cai, K. Su, and Q. Chen, "EWLS: A new local search for minimum vertex cover," in *Proc. of AAAI-10*, 2010, pp. 45–50.

[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] A. Balint and A. Fröhlich, "Improving stochastic local search for SAT with a new probability distribution," in *Proc. of SAT-10*, 2010, pp. 10–15.

[5] S. Cai and K. Su, "Configuration checking with aspiration in local search for SAT," in *Proc. of AAAI-12*, 2012, p. to appear.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Concurrent Cube-and-Conquer

Peter van der Tak
Delft University of Technology,
The Netherlands

Marijn J.H. Heule
Delft University of Technology,
The Netherlands

Armin Biere
Johannes Kepler University Linz,
Austria

I. Introduction

The concurrent cube-and-conquer (CCC) solver implements the ideas in the paper we submitted to the PoS 2012 workshop [1]. This system description describes the main concepts, a more detailed explanation is in the paper.

Recent work has introduced the cube-and-conquer (CC) technique [2], which first partitions the search space into disjunctive sets of assumptions (cubes) using a lookahead (LA) solver (the cube phase) and then solves each cube using a CDCL solver (the conquer phase). It uses a *cutoff heuristic* to control after what number of decisions the lookahead solver should be cut off and store its decision variables (its current cube) for the CDCL solver to solve in the conquer phase. However, this heuristic is not ideal particularly because no information about the performance of CDCL on the cubes is present in the cube phase. Concurrent cube-and-conquer uses a synchronized LA and CDCL solver concurrently in the cube phase to improve the cutoff heuristic.

II. Motivation

Cube-and-conquer shows strong performance on several hard application benchmarks [2], beating both the lookahead and CDCL solvers that were used for the cube and conquer phases respectively. However, on many other instances, either lookahead or CDCL outperforms CC. We observed that for benchmarks for which CC has relatively weak performance, two important assumptions regarding the foundations of CC do not hold in general.

First, in order for CC to perform well, lookahead heuristics must be able to split the search space into cubes that, combined, take less time for the conquer solver (CDCL) to solve. Otherwise, cube-and-conquer techniques are ineffective and CDCL would be the preferred solving technique. Second, if lookahead can refute a cube, then this must mean that nearby cubes can be efficiently solved using CDCL. When this assumption fails, the cube phase either generates too few cubes and leaves a potential performance gain unused, or generates too many cubes because cubes with fewer decisions are also easy for CDCL to solve.

CCC solves these problems separately. The first by predicting on which instances cube-and-conquer techniques are ineffective and aborting in favor of a classical CDCL search. The second by also using a CDCL solver in the cube phase to better estimate the performance of CDCL on nearby cubes. This naturally cuts off easy cubes. We first discuss CCC_∞, a simplified version of CCC with no cut off heuristic and prediction in the next section, and add these two features in sections IV and V respectively. The submitted solver includes all features.

III. Concurrent cube-and-conquer

CCC_∞ is implemented by sending messages between the CDCL and the lookahead solvers using two queues: the decision queue Q_{decision} and the result queue Q_{solved}. Whenever the lookahead solver assigns a decision variable, it pushes the tuple $\langle$cube c_{id}, literal l_{dec}, *backtrackLevel*$\rangle$ comprising a uniquely allocated id, the decision literal, and the number of previously assigned decision variables (*backtrackLevel*). When the CDCL solver reads the new decision from the queue, it already knows all previous decision literals, and only needs to backtrack to the *backtrackLevel* and add l_{dec} as an assumption to start solving c_{id}. The id is used to identify the newly created cube.

If the CDCL solver proves unsatisfiability of a cube before it receives another decision, it pushes the c_{id} of the refuted cube to Q_{solved}. The solver then continues with the parent cube, by backtracking to the level where all but the last decision literal were assigned. When the lookahead solver reads the c_{id} from Q_{solved}, it backtracks to the level just above this cube's last decision variable and continues its search as if it proved unsatisfiability of the cube by itself.

To keep track of the cubes that are pending to be solved, both solvers keep the trail of decision literals (or assumptions for the CDCL solver) and the id's of the cubes up to and including each decision literal (or assumption). Whenever either solver proves unsatisfiability of the empty cube, or when it finds a satisfying assignment, the other solver is aborted.

The submitted version of CCC first simplifies the instance using Lingeling, and then uses march_rw [3] (LA) and MiniSAT 2.2 [4] (CDCL) concurrently. The CCCeq version runs march_rw with *equivalence reasoning* [3] enabled, CCCneq with *equivalence reasoning* disabled, as this has shown to affect the performance of CCC.

IV. Cutoff heuristic

One advantage of CC was that the conquer phase can be parallelized efficiently by using multiple CDCL solvers in parallel, each solving a single cube. With CCC_∞ this is no longer possible, since the lookahead solver will continue with a single branch until it is solved by either CDCL or lookahead. Additionally, CCC_∞ always uses twice as much CPU time as

wall clock time, because the lookahead and CDCL solver run in parallel.

To reduce this wasted resource utilization and allow for parallelization of the CDCL solver, we reintroduce the conquer phase by applying a suitable cutoff heuristic. As with CC, we pass cubes from the cube phase to the conquer phase using the iCNF[1] format (via the filesystem, unlike the shared memory queues in the cube phase), which is basically a concatenation of the original formula F and the generated cubes as assumptions. An incremental SAT solver iterates over each cube c_{id} in the file, and solves $F \wedge c_{id}$ until a solution is found or all cubes have been refuted.

The cutoff heuristic of CC is based on a rough prediction of the performance of CDCL on a cube. Given a cube c_{id}, it computes its difficulty[2][3] $d(c_{id}) := |\varphi_{\mathrm{dec}}|^2 \cdot (|\varphi_{\mathrm{dec}}| + |\varphi_{\mathrm{imp}}|)/n$, where $|\varphi_{\mathrm{dec}}|$ and $|\varphi_{\mathrm{imp}}|$ are the number of decision and implied variables respectively, and n is the total number of free variables. If $d(c_{id})$ is high, the CDCL solver is expected to solve c_{id} fast.

The cutoff heuristic in CC focuses on identifying cubes that are easy for CDCL to solve. It cuts off a branch if $d(c_{id})$ exceeds a dynamic threshold value t_{cc}. Initially $t_{cc} = 1000$, and it is multiplied by 0.7 whenever lookahead solves a cube (because it assumes that CDCL would have solved this cube faster) or when the number of decisions becomes too high (to avoid generating too many cubes). It is incremented by 5% at every decision to avoid the value from dropping too low.

For CCC, the same heuristic does not work because easy cubes are solved quickly by the CDCL solver. This makes the threshold very unstable so that it quickly converges to 0 or infinity depending on the instance. We therefore use a different heuristic, but using the same difficulty metric $d(c_{id})$.

Easy cubes can be detected better by CCC than by CC, because CCC can detect for which cubes CDCL finds a solution before the lookahead solver does. CCC would ideally cut off these cubes so that they can be solved in parallel. The contrary goes for when the lookahead solver solves a cube: it then seems that lookahead contributes to the search, which means that it is not desirable to cut off.

CCC uses the same difficulty metric $d(c_{id})$ as CC, but a different heuristic for determining the threshold value t_{ccc}. If a cube c_{id} is solved by CDCL, the value is updated towards $s := 0.4 \cdot d(c_{id})$, whereas it is updated towards $s := 3 \cdot d(c_{id})$ if c_{id} was solved by lookahead. To avoid too sudden changes, t_{ccc} is not changed to s directly but is filtered by $t'_{ccc} := 0.4 \cdot s + 0.6 \cdot t_{ccc}$. To furthermore avoid the threshold from dropping too low, it is incremented for every cube that is cut off.

The submitted implementation of CCC uses iLingeling to solve the cubes that were cut off by the heuristic. iLingeling basically submits these cubes to a number of independent incremental Lingeling solvers in parallel.

[1] http://users.ics.tkk.fi/swiering/icnf

[2] CC's heuristic has been improved slightly since it was initially published [2]; it now uses $|\varphi_{\mathrm{dec}}|^2$ instead of $|\varphi_{\mathrm{dec}}|$.

[3] The notation is ours.

V. Prediction

Since cube-and-conquer techniques do not work well on all instances, CCC aims to detect quickly if an instance is unsuitable. It does this based on two measurements.

First, lookahead techniques appear effective if they can solve some cubes faster than CDCL. While running the lookahead and CDCL solver in parallel, we count the number of times that lookahead is faster than CDCL. For benchmarks for which this count is increased very slowly, say less than once per second, we observed that CC was generally not an effective solving strategy.

Second, if the variable heuristics are effective then each discrepancy should result in a large reduction of the formula. Hence after a certain number of discrepancies the solver should be able to refute that branch. Preliminary experiments suggest that if CCC finds a leaf with over 20 discrepancies early in the search-tree, then lookahead variable heuristics should be considered as ineffective.

These metrics are combined as follows. CCC runs the LA and CDCL solver for a few seconds concurrently. If the LA solver enters a branch with more than 20 discrepancies terminate the solvers and use fallback solver pLingeling instead. If after 5 seconds the solvers are still running and less than 10 cubes were solved by lookahead, the solvers should also be terminated in favor of pLingeling. Otherwise, CCC is the preferred solving method and the solvers can continue. For CC, the same instances usually work well, but they cannot be detected as early because CDCL is only used in the conquer phase.

VI. Conclusion

Without performance prediction, cube-and-conquer techniques are not competitive with current state-of-the-art solvers. CCC's predictor is able to efficiently select instances for which cube-and-conquer techniques are suitable and fall back to pLingeling if not. This allows cube-and-conquer to compete with other solvers. In addition, CCC improves over CC's performance by using concurrency and improved heuristics in the cube phase.

CCC uses march_rw (LA) and the same versions of Lingeling (simplification and CDCL), iLingeling (conquer), and pLingeling (fallback) submitted to this SAT challenge. All sources are compiled into a single binary with -O3. Threading is implemented using pthreads, and communication in the cube phase using lockless queues. Communication between the simplification, cube, conquer, and fallback solvers is done via temporary CNF and iCNF files.

References

[1] P. van der Tak, M. J. H. Heule, and A. Biere, "Concurrent cube-and-conquer," 2012, submitted to PoS 2012.

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

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

[4] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT'03*, ser. LNCS, vol. 2919. Springer, 2004, pp. 502–518.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

clasp, *claspfolio*, *aspeed*: Three Solvers from the Answer Set Solving Collection *Potassco*

Benjamin Kaufmann
University of Potsdam,
kaufmann@cs.uni-potsdam.de

Torsten Schaub
University of Potsdam,
torsten@cs.uni-potsdam.de

Marius Schneider
University of Potsdam,
manju@cs.uni-potsdam.de

I. *clasp* (2.0.6)

Authors:
M. Gebser (University of Potsdam),
B. Kaufmann, and T. Schaub

clasp[1] combines the high-level modeling capacities of Answer Set Programming (ASP; [1]) with state-of-the-art techniques from the area of Boolean constraint solving. It is originally designed and optimized for conflict-driven ASP solving [2], [3], [4]. Most of its innovative algorithms and data structures, like e.g. ASP-oriented pre-processing [5] or native support of aggregates [6], are thus outside the scope of SAT solving. However, given the proximity of ASP to SAT, *clasp* can also deal with formulas in CNF via an additional DIMACS frontend. As such, it can be viewed as a chaff-type Boolean constraint solver [7] featuring a number of techniques found in SAT solvers based on Conflict-Driven Clause Learning. For example, *clasp* supports pre-processing [8], [9], phase caching [10], on-the-fly subsumption [11], and aggressive deletion [12].

Starting with version 2.0, *clasp* also supports parallel (multi-threaded) solving either by search space splitting and/or competing strategies. While the former involves dynamic load balancing in view of highly irregular search spaces, both modes aim at running searches as independently as possible in order to take advantage of enhanced sequential algorithms. Furthermore, *clasp* supports the exchange and physical sharing of (recorded) nogoods. While unary, binary, and ternary nogoods are always shared among all threads, sharing of longer ones is mainly controlled by their respective number of distinct decision levels associated with the contained literals, called the *Literal Block Distance* [12].

clasp is implemented in C++ using Intel's Threading Building Blocks library for platform-independent threads, atomics, and concurrent containers. All major routines of *clasp* are lock-free and optimized representations of constraints based on a clear distinction between read-only, shared, and thread-local data further promote the scalability of parallel search. *clasp* currently supports up to 64 freely configurable (non-hierarchic) threads.

The following configurations of *clasp* participated in the respective tracks of SAT Challenge 2012:

[1]http://potassco.sourceforge.net/#clasp

- **Application**:
  `--sat-p=20,25,240,-1,1`
  `--heuristic=Vsids`
  `--dynamic-restarts=100,0.7`
  `--dfrac=0.5 --del=3.0,1.1,20.0`
  `--dgrowS=100,1.5 --dinit=500,20000`
  `--dsched=+,10000,2000 --dglue=2`
  `--update-lbd --save-p=75`
  `--recursive-str --otfs=2`
  `--reverse-arcs=2 --cir=3`
  `--cir-bump=1023`
- **Combinatorial**:
  `--sat-p=10,25,240,-1,1`
  `--heuristic=Vsids --restarts=128,1.5`
  `--del=10.0,1.1,20.0 --dinit=1000,10000`
  `--dsched=+,10000,1000 --dglue=2`
  `--otfs=2 --reverse-arcs=1 --cir=3`
- **Parallel**:
  `--sat-p=20,25,240,-1,1`
  `--threads=8 --port=sat12-port.txt`
  `--distribute=all,4 --integrate=gp,512`

The main difference between the application and the combinatorial configuration lies in the selected restart strategy. While the application configuration uses an aggressive dynamic strategy, the combinatorial uses a geometric policy restarting every 128×1.5^i conflicts. The meaning of the individual parameters is as follows:

- `sat-p`: Enables *SatELite*-like preprocessing with (optional) blocked clause elimination. The first three parameters control number of iterations, maximal occurrence cost, timeout in seconds, respectively. The last parameter controls blocked clause elimination.
- `heuristic`: Both configurations use a *MiniSAT*-like version of the *VSIDS* heuristic.
- `dynamic-restarts`: Enables a dynamic restart strategy similar to the one of *glucose* [13]. It maintains the running average of LBDs R over the last x conflicts and restarts if $R > y\times$ global average. In contrast to other strategies, our version does not use a fixed threshold. Instead, it monitors the current restart-frequency and adapts the threshold dynamically in order to avoid either very slow or overly aggressive restarts.
- `dfrac`: Sets the fraction of clauses removed on clause

deletion. The default is 0.75.

- `del=F,G,Z, dinit, dgrowS`: Configure the primary deletion schedule based on number of lerant clauses. Given P, the number of problem clauses, the initial limit X is set to $\frac{1}{F} \times P$ clamped to the interval given by `dinit`. Whenever the grow schedule fires, X is multiplied by G but growth stops once X exceeds $Z \times P$. If `dgrowS` is not given, the selected restart strategy is used.
- `dsched`: Configures the secondary deletion schedule based on number of conflicts. The current threshold of this schedule is reset, whenever the primary schedule fires. Both configurations use an arithmetic policy firing every $X + Y \times i$ conflicts.
- `dglue`: Enables *glucose*-like *glue* clauses. Clauses with an lbd $\leq X$ are not deleted.
- `update-lbd`: Enables updates of LBD values of learnt clauses. In contrast to other solvers, *clasp* updates LBD values only for clauses participating in the resolution of new conflict clauses.
- `save-p=X`: Enables *Rsat*-like phase caching on backjumps of length $\geq X$. By default, phase caching is disabled.
- `recursive-str`: Enables *MiniSAT*-like expensive conflict clause minimization.
- `otfs`: Enables on-the-fly subsumption.
- `reverse-arcs`: Enables *ManySAT*-like *inverse-arc* learning [14].
- `cir=X, cir-bump=Y`: Enables *counter implication restarts* (see Pragmatics of SAT 2011) every Xth restart. The heuristic value Y is used to compute the amount added to the activity of variables.

Finally, the parallel configuration uses a portfolio of eight threads including the aforementioned application configuration. Individual threads distribute learnt conflict clauses with an lbd ≤ 4. Furthermore, the 512 most recently received clauses are excluded from clause deletion.

II. *claspfolio* (1.1.0)

Authors:
C. Schulz-Hanke (University of Potsdam),
T. Schaub, and M. Schneider

Inspired by *satzilla* [15], we address the high sensitivity of ASP and SAT solving to search configuration by exploring a portfolio-based approach, named *claspfolio*[2] [16]. To this end, we concentrate on the solver *clasp* and map a collection of numeric instance features onto an element of a portfolio of distinct *clasp* configurations (based on a Support Vector Regression [17]), in contrast to *satzilla*, which maps to a portfolio of different solvers.

In detail, *claspfolio* is based on 60 static and 28 dynamic features for SAT problems. The features are mainly inspired by *satzilla* [15] which are based on the results of Nudelman et al. [18]. The static features include the number of variables, number of clauses, the variable per clause ratio, balance between positive and negative occurrences of variables, the fraction of horn clauses and statistics about a randomly sampled part of the variable graph, clause graph, and variable-clause graph. The dynamic features are recorded after each restart of a pre-solving phase with at most three restarts. After each restart, these dynamic features include the number of deleted clauses, free variables, choices, conflicts, restarts and backjumps. The features are normalized with a z-score and used to evaluate the Support Vector Regression models of each configuration in the portfolio. Hence, *claspfolio* selects the configuration with the best predicted performance to solve the given instance.

The portfolio of *claspfolio* consists of complementary *clasp* configurations which have been found by manual tuning and using the automatic algorithm configuration tool *paramils* [19]. *paramils* tuned *clasp* on instances of the 2008 SAT Race, 2010 SAT Race, and 2011 SAT Competition; both on the entire instance set and on individual subclasses. In the end, *claspfolio* uses 30 configurations of *clasp* (2.0.6).

claspfolio is a branch of *clasp* and therefore, the algorithm selection is directly integrated in *clasp*. Hence, *claspfolio* is also implemented in C++.

III. *aspeed* (1.0.0)

Authors:
R. Kaminski (University of Potsdam),
H. Hoos (University of British Columbia),
T. Schaub, and M. Schneider

Inspired by the simple, yet successful portfolio-based SAT-solver *ppfolio* [20] (in the 2011 SAT Competition), our approach, dubbed *aspeed*[3] [21], computes timeout-minimal time slices for a portfolio of solvers or solver configurations and sequences these to minimize average runtime. *aspeed* performs these calculations by means of a declarative specification in ASP; its execution relies on ASP tools from the Potassco collection [22], allowing for a flexible and compact encoding of the problem constraints. In addition, *aspeed* is able to compute parallel schedules for execution on multi-core architectures. In contrast to powerful portfolio-based approaches, such as *satzilla* [15] and *3S* [23], *aspeed* does not rely on instance features and is therefore more easily applicable to problems for which features are not (yet) available.

In the 2012 SAT Challenge, *aspeed* uses a portfolio of *clasp* configurations (2.0.6). The portfolio and the corresponding runtime data are the same as used for training *claspfolio* (see above). Since the portfolio consists of *clasp* configurations and *clasp* is used to compute the schedules, *aspeed* can be seen as a self-optimizing solver.

aspeed^c uses the same portfolio as *claspfolio*. Furthermore, *aspeed*^{m} includes the medal-winning solvers of the 2011 SAT Competition, i.e. *glueminisat* [24], *lingeling* [25], *march_rw* [26], *qutersat* [27], *sattime*, and *sparrow* [28], in addition to the *claspfolio* portfolio.

[2]http://potassco.sourceforge.net/#claspfolio

[3]http://potassco.sourceforge.net/labs.html#aspeed

The *aspeed* framework is implemented in Python-2.7 and uses the ASP Potassco collection to compute the optimal schedules.

ACKNOWLEDGMENTS

This work was funded by the German Science Foundation (DFG) under grant SCHA 550/8-1/2.

REFERENCES

[1] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[2] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set solving," in *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI Press/The MIT Press, 2007, pp. 386–392.

[3] ——, "clasp: A conflict-driven answer set solver," in *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, ser. Lecture Notes in Artificial Intelligence, C. Baral, G. Brewka, and J. Schlipf, Eds., vol. 4483. Springer-Verlag, 2007, pp. 260–265.

[4] ——, "Conflict-driven answer set enumeration," in *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, ser. Lecture Notes in Artificial Intelligence, C. Baral, G. Brewka, and J. Schlipf, Eds., vol. 4483. Springer-Verlag, 2007, pp. 136–148.

[5] ——, "Advanced preprocessing for answer set solving," in *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, Eds. IOS Press, 2008, pp. 15–19.

[6] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "On the implementation of weight constraint rules in conflict-driven ASP solvers," in *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, ser. Lecture Notes in Computer Science, P. Hill and D. Warren, Eds., vol. 5649. Springer-Verlag, 2009, pp. 250–264.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*. ACM Press, 2001, pp. 530–535.

[8] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer-Verlag, 2005, pp. 61–75.

[9] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Proceedings of the Sixteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, ser. Lecture Notes in Computer Science, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer-Verlag, 2010, pp. 129–144.

[10] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, ser. Lecture Notes in Computer Science, J. Marques-Silva and K. Sakallah, Eds., vol. 4501. Springer-Verlag, 2007, pp. 294–299.

[11] H. Han and F. Somenzi, "On-the-fly clause improvement," in *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer-Verlag, 2009, pp. 209–222.

[12] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, C. Boutilier, Ed. AAAI Press/The MIT Press, 2009, pp. 399–404.

[13] ——, "GLUCOSE: A solver that predicts learnt clauses quality," in *SAT 2009 competitive events booklet: preliminary version*, D. Le Berre, O. Roussel, L. Simon, V. Manquinho, J. Argelich, C. Li, F. Manyà, and J. Planes, Eds., 2009, pp. 7–8, available at http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf.

[14] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais, "A generalized framework for conflict analysis," in *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, ser. Lecture Notes in Computer Science, H. Kleine Büning and X. Zhao, Eds., vol. 4996. Springer-Verlag, 2008, pp. 21–27.

[15] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, 2008.

[16] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller, "A portfolio solver for answer set programming: Preliminary report," in *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, ser. Lecture Notes in Artificial Intelligence, J. Delgrande and W. Faber, Eds., vol. 6645. Springer-Verlag, 2011, pp. 352–357.

[17] C. Chang and C. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[18] E. Nudelman, K. Leyton-Brown, H. Hoos, A. Devkar, and Y. Shoham, "Understanding random SAT: Beyond the clauses-to-variables ratio," in *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04)*, ser. Lecture Notes in Computer Science, M. Wallace, Ed., vol. 3258. Springer-Verlag, 2004, pp. 438–452.

[19] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stutzle, "ParamILS: An Automatic Algorithm Configuration Framework," *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, 2009.

[20] O. Roussel, "Description of ppfolio," Centre de Recherche en Informatique de Lens, Tech. Rep., 2011.

[21] H. Hoos, R. Kaminski, T. Schaub, and M. Schneider, "aspeed: ASP-based solver scheduling," 2012, to appear in Proceedings of ICLP'12.

[22] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider, "Potassco: The Potsdam answer set solving collection," *AI Communications*, vol. 24, no. 2, pp. 105–124, 2011.

[23] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm Selection and Scheduling," in *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'10)*, ser. Lecture Notes in Computer Science, J. Lee, Ed., vol. 6876. Springer-Verlag, 2011, pp. 454–469.

[24] H. Nabeshima, K. Iwanuma, and K. Inoue, "Glueminisat2.2.5," University of Yamashima and National Institute of Informatics, Japan, Tech. Rep., 2011.

[25] A. Biere, "Lingeling and friends at the SAT competition 2011," Institute for Formal Models and Verification, Johannes Kepler University, Technical Report FMV 11/1, 2011.

[26] M. Heule and H. van Maaren, "March_dl: Adding adaptive heuristics and a new branching strategy," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 47–59, 2006.

[27] C. Wu, T. Lin, C. Lee, and C. Huang, "Qutesat: a robust circuit-based sat solver for complex circuit structure," in *Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE'07)*, R. Lauwereins and J. Madsen, Eds. ACM, 2007, pp. 1313–1318.

[28] D. Tompkins, A. Balint, and H. Hoos, "Captain Jack – New Variable Selection Heuristics in Local Search for SAT," in *Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11)*, ser. Lecture Notes in Computer Science, K. Sakallah and L. Simon, Eds., vol. 6695. Springer-Verlag, 2011, pp. 302–316.

[29] C. Baral, G. Brewka, and J. Schlipf, Eds., *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, ser. Lecture Notes in Artificial Intelligence, vol. 4483. Springer-Verlag, 2007.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Contrasat12

Allen Van Gelder
University of California
Santa Cruz, CA 95064, U.S.A.

***Abstract*—This note is the Contrasat12 Description required for SAT Challenge 2012.**

I. Major solving techniques

`Contrasat` is a CDCL solver based on Minisat 2.2.0. `Contrasat` has been described in a JSAT System Description entitled "Contrasat – A Contrarian Sat Solver," which appeared early in 2012. That paper refers to this URL for the code:

`http://www.cse.ucsc.edu/~avg/Contrasat/`.

The version submitted to SAT Challenge 2012 is the same, except that there is a new command-line option "-contra-early" that permits the parameter to be varied at run time from its default value of 40. The public code requires recompilation to vary the parameter. The paper shows a table with various parameter values, including 40. The program is expected to compete using the default of 40.

The idea is to use a priority queue instead of a FIFO queue for literals that have been implied and are awaiting unit-clause propagation. Changing the order can also change the antecedent clauses and can change when a conflict is discovered. The learned clause can change due to the antecedent changes.

II. Parameter description

1) `contra-early` is the only new parameter; it is under user control. It limits the number of literals in an antecedent clause that are examined to compute a priority.
2) There are no "magic numbers".
3) `contra-early` is 40 for the competition.
4) The value of `contra-early` is not dependent on instance properties.

III. Special Algorithms, Data Structures and Features

`Contrasat` uses a priority queue implemented as a binary heap.

IV. Implementation detail

1) The programming language is `C++`.
2) The code basis is `Minisat 2.2.0`.

V. SAT Challenge 2012 Specifics

1) The solver was submitted in all tracks for sequential solvers.
2) The compiler was `g++`.
3) Optimization flags were `-O3` and -static.
4) 64-bit binary.
5) The only command-line parameter is the input file.

VI. Availability

1) `Contrasat12` is not publicly available, but it is functionally the same as Contrasat-2.2.0.B.tar.gz.
2) `http://www.cse.ucsc.edu/~avg/Contrasat` is the URL. Download anything you want in the directory listing.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

GLUCOSE 2.1 in the SAT Challenge 2012

Gilles Audemard
Univ Lille-Nord de France CRIL / CNRS UMR8188,
Lens, F-62307

Laurent Simon
Univ Paris-Sud, LRI / CNRS UMR8623
Orsay, F-91405

***Abstract*—This document shortly describes the novelties embedded in GLUCOSE 2.1, the new GLUCOSE version that participated to the SAT Challenge 2012. It is a CDCL solver, based on MINISAT 2.2, that incorporates aggressive clause database cleaning and a new restart strategy.**

I. GLUCOSE METABOLISM

GLUCOSE is a typical CDCL (Conflict Driven Clause Learning) solver [6], [4], [2] built on top of MINISAT 2.2 [2]. The version 1.0 of GLUCOSE was extensively described in [1]. It was ranked first in the SAT competition 2009 [3], category Application, UNSAT. It was also ranked first in the Competition 2011, category Application, SAT+UNSAT.

II. NOVELTIES OF THE GLUCOSE FAMILY

The fundamental novelty of the family of GLUCOSE solvers is based on a static measure of learnt clause usefulness, called the Literal Block Distance (LBD). It corresponds to the number of distinct decision levels that a learnt clause contains. As we have previously shown, this measure can be also dynamically updated when a clause is used for propagation. It was shown how important are clauses of LBD score of 2. We called those clauses "glue clauses" because they intuitively allows to stick a new literal to a block of propagation literals. The name of GLUCOSE comes from the importance of those clauses.

Based on this measure, the solvers uses a very aggressive clause deletion strategy, that is independent of the size of the initial formula. In GLUCOSE 1.0, the clause database cleaning was triggered every 20000+500*x conflicts (x being the number of previous cleaning). In GLUCOSE 2.0/2.1, the cleaning is triggered every 4000+300*x conflicts, leading to an impressive cleaning of the clause database. If we look at the traces of the SAT 2011 competition, second phase, GLUCOSE deleted more than 90% of the learnt clauses during search.

Our solver also has a dynamic restart strategy based on an estimation of whether the Solver is currently producing good clauses or not. This estimation is done thanks to an observation window of the last X conflicts, taking into account the average of clauses LBD. If Y times this average is above the average of this value over all the conflicts (Y is 0.7 in GLUCOSE 1.0 /2.0 and 0.8 in GLUCOSE 2.1, like in GLUEMINISAT [5]), then the solver estimates that the production is below the average, and thus a restart is triggered. In GLUCOSE 1.0 and 2.0, the window size X was 100. In GLUCOSE 2.1, the window size X is 50 (again, the same constant as GLUEMINISAT), but an additional component is embedded which allows to postpone (possibly many times) the next restart(s). Thus GLUCOSE has this very particular property: no guarantee is given on the restart window size (there can be a restart every 50 conflicts, or none).

In GLUCOSE 2.0, a dynamic adjustment of the clause database size was added to take into account the cases were the LBD was not discriminant enough.

III. PARAMETER AND OTHER DESCRIPTIONS

Like all the solvers, GLUCOSE is full of magic numbers. Those constant were fixed according to a few tests of classical benchmarks [3]. GLUCOSE 2.0 was tuned to improve its overall score of solved instances on the current bank of benchmarks. GLUCOSE 2.1 was tuned to first target UNSAT instances, then a refinement of our restart policy was added to gain a substantial number of new SAT instances without loosing performances on UNSAT problems.

GLUCOSE also 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 open source, and thus constants are easily accessible. Most constants were described in [1] and above. The additional constants regarding the postponing of restarts are described in a paper under submission.

GLUCOSE is targeting Application problems. It is compiled in 32 bits. The main page of GLUCOSE is

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

REFERENCES

[1] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, 2009.
[2] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
[3] D. Le Berre, M. Jarvisalo, O. Roussel, and L. Simon. SAT competition, 2002–2011. http://www.satcompetition.org/.
[4] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.
[5] Hidetomo Nabeshima, Koji Iwanuma, and Katsumi Inoue. Glueminisat2.2.5. System Desciption, available on glueminisat.nabelab.org.
[6] Joao Marques Silva and Karem Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Glucose with Implied Literals (Glucose_IL 1.0)

Arie Matsliah*, Ashish Sabharwal†, and Horst Samulowitz†
*IBM Research, Haifa, Israel
Email: ariem@il.ibm.com
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz}@us.ibm.com

Glucose_IL (also referred to as Glucose++ in SAT Challenge 2012) is an extension of the Glucose SAT solver [1] that uses *implied literals* to strengthen inference and learning during search. Like Glucose, it is a conflict directed clause learning SAT solver. Glucose_IL version 1.0 participated in the Application category of SAT Challenge 2012.

I. Solving techniques

The main idea behind Glucose_IL is to strengthen clause learning by dynamically inferring implied literals that a newly learned conflict clause entails. We refer the reader to [5] for a more in-depth discussion of the internals of Glucose_IL. The technique is related to (but not the same as) some methods known in the literature. For example, *probing* [4] simply applies unit propagation of literals at the root node in order to detect failed literals [2] or to populate literal implication lists. The latter information can then for instance be used to shrink clauses by *hidden literal elimination* (e.g., if $a \mapsto b$ then $(a \vee b \vee c)$ can be reduced to $(b \vee c)$; see e.g., [3]).

At a high level, the technique employed is as follows. We say that a literal l is an *implied literal* if all literals in a clause entail l. For instance, if $a \mapsto d$, $b \mapsto d$, and $c \mapsto d$, then $(a \vee b \vee c)$ entails d. While this insight has already been exploited in several methods (e.g., variations of *hyper binary resolution* and *hidden literal elimination*), we apply it to clause learning: when the SAT solver derives a new conflict clause c, we check if the literals in c imply a single or multiple literals which can then be propagated as new unit literals.

In order to employ this technique we first need to generate implication lists $L(l) = UnitPropagation(l)$ for each literal l. This is done at the root node of the search tree before the solving process starts, and periodically during search. During this computation we also add the corresponding not yet existing binary clauses of the following formulas: $\forall l \in L(p) : \neg l \mapsto \neg p$. Some other clauses are also added and appropriate optimizations are performed to make the technique practical. The reader is referred to [5] for details.

II. Implementation details

Glucose_IL is built upon version 2.0 of Glucose [1] and compiled on x86-64 linux using GNU g++ 4.4.5 using options "-O3 –static". The main parameters controlling the proposed technique are as follows: a maximum size limit (in terms of number of clauses and number of literals) on the CNF formula beyond which we turn off implied literal computation, constants for the geometrically increasing frequency of computation of lists of implied literals, the maximum number of implied literals and binary clauses, and the maximum size of learned clauses beyond which we do not analyze clauses for implied literals detection.

III. SAT Challenge 2012 specific details

The parameters used for the SAT Challenge 2012 submission were as follows: (a) the method is used only if the input formula has no more than 2×10^6 clauses, it has no more than 216×10^3 literals appearing in binary clauses, and the product of the number of clauses and the number of literals appearing in binary clauses is smaller than 30×10^9; (b) lists of implied literals are computed at the very beginning, after 16 restarts, then after 100 restarts, and from then on in a geometrically increasing fashion with a factor of 1.2; (c) no more than 8.5×10^6 new binary clauses are added and no more than 20×10^6 implied literal are maintained; and (d) the maximum learned clause size for implied literals inference is 7.

Acknowledgment

We express our sincere thanks to Gilles Audemard and Laurent Simon for making available their SAT solver Glucose.

References

[1] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. *IJCAI*, 399¡96¿-404, 2009.
[2] Jon Freeman. Improvements to Propositional Satisfiability Search Algorithms. *PhD. Thesis*, University of Pennsylvania, 1995.
[3] Marijn J. H. Heule, Matti Järvisalo, Armin Biere. Efficient CNF simplification based on binary implication graphs. *SAT*, 201–215, 2011.
[4] Ines Lynce, Joao Marques-Silva. Probing-Based Preprocessing Techniques for Propositional Satisfiability. *ICTAI*, 105, 2003.
[5] A. Matsliah, A. Sabharwal, H. Samulowitz. Augmenting Clause Learning with Implied Literals. *SAT*, 2012 (to appear).

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Glucans System Description

Xiaojuan Xu*, Yuichi Shimizu*, and Kazunori Ueda*
*Department of Computer Science and Engineering
Waseda University, Tokyo, Japan
Email: {xxj, yusui, 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). 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. 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.

III. Variations

Glucans consist of a basic version and two variations. The description of the three versions is as follows.

A. Glycogen

In this basic version, all threads run GLUCOSE.

B. Cellulose

Cellulose is a variation of Glycogen. The difference between Glycogen and Cellulose is that one of the threads incorporates CIRMinisat's hack and another thread incorporates Contrasat's hack. This version performs the preprocessing of GLUCOSE also.

C. Sucrose

Sucrose is another variation. In this version, half of the threads run GLUCOSE and the other half run GLUEMINISAT. Each group incorporates Contrast's and CIRMinisat's hacks as in Cellulose.

IV. 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.
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.

V. SAT Challenge 2012 Specifics

This solver is submitted to the track of Parallel Solvers – Application SAT+UNSAT. It is a 64-bit binary and compiled by gcc 4.4.4 with the -O3 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, and
3) -ex-size= : the maximum LBD for exchanging learnt clauses.

VI. Availability

Glucans will be available soon after the competition at our website, http://www.ueda.info.waseda.ac.jp/ .

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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Trap Avoidance heuristics using pseudo-conflict learning applied to gNovelty$^+$ and sparrow2011

Thach-Thao Duong*†, Duc-Nghia Pham*†
*Institute for Integrated and Intelligent Systems, Griffith University, QLD, Australia
†Queensland Research Laboratory, NICTA
Email: {thao.duong,duc-nghia.pham}@nicta.com.au

I. Introduction

Stochastic Local Search (SLS) for Satisfiability (SAT) problems is an effective method solving real world problems. However, the approach has a limitation of local minima stagnation, which can degrade the searching quality. Our proposed heuristics [4] stems from the idea of preventing stagnation instead of treating it. The stagnation heuristics consists of two phases: a pseudo-conflict learning (PCL) phase and a prevention phase. Integrations of gNovelty$^+$ [3] and sparrow2011 [1] with the heuristics, named gNovelty$^+$PCL and sparrow2011-PCL, are presented in this report.

II. PCL Heuristics : Trap Prevention Strategy

The heuristics derived from the hypothesis that local minima may cause from conflicts between the assignment of variables and constraints between clauses can be established. Assuming that conflicts occur strongly in the around area of stagnation points, the proposed heuristic focus to learn the information at local areas of stagnation points. A simple way counting the frequency a variable appearing in the warning stagnation areas was implemented. The high value of frequency indicates the high likelihood of leading to stagnation if that variable is flipped. For that reason, variables having low $stagnation\_weight$ value are preferred to be selected. The learning mechanism is described under pseudo-code in Algorithm 1 and the prevention behavior will be explained more in the integrated versions with specific algorithms.

Algorithm 1: Pseudo-conflict learning strategy PCL(k,s,T,H,W)

Input : tenure $k > 0$, "Static"or "Dynamic"option s, window size $T > 0$, flipped variable history stack H, window Queue W

```
1  stagntion_zone = ∅ ;
2  recent_flipped_var ← pop_stack(H) ;
3  stagntion_zone ← recent_flipped_var ;
4  for i ← 1 to k − 1 do
5  |  var ← pop_stack(H) ;
6  |  if s== "Dynamic"and var == recent_flipped_var then break;
7  |  stagntion_zone ← var ;
8
9  for all var in stagntion_zone do
10 |  stagnation_weight[var] + +;
11
12 if Size(P) == T then
13 |  far_most_stagntion_zone = pop_queue(W);
14 |  remove far_most_stagntion_zone from W;
15 |  for all var in stagntion_zone do
16 |  |  stagnation_weight[var] − −;
17 |
18 W ← push_queue(W, stagntion_zone)
```

There are two options of stagnation zone: "Static"and "Dynamic". Static stagnation zones have fix lengths of pathways whereas dynamic stagnation areas have adaptive lengths according to the stagnation environment. For more clarification, static zone is the backtracking path to the flipped variable history within a fixed length of tenure. The dynamic stagnation zone is the maximum backtracking pathway within tenure k so that there is no duplication of the most recent flipped variable. After identifying affected stagnation zones, the $stagnation\_weight$ of the variables in these areas are increased by one. The time window smoothing is resembled a mechanism in which a window flows along side with the searching progress. The aim of this smoothing techniques is to restrict the effects of the stagnation learning experience into limited recent stagnation times. As the window is floating along with the searching, once the far most stagnation zone is out of the window view, the experience learning from that stagnation zone will be elapsed. The far most stagnation zone is popped out and removed from the window and a new encountered stagnation path is pushed in window queue.

III. gNovelty$^+$PCL: integrated stagnation prevention strategy into gNovelty$^+$

The algorithm gNovelty$^+$PCL is illustrated in Algorithm 2. In the initialization phase, the $stagnation\_weight$ of all variables are set to zero. The variable history H and stagnation window W are initialized empty. When there is no promising variable to improve the score value, the learning strategy PCL is performed. In selecting variable procedure, breaking tie by the least $stagnation\_weight$ variable instead of breaking tie by least recent flipped variable like gNovelty$^+$.

IV. sparrow2011$^+$PCL: integrated stagnation prevention strategy into gNovelty$^+$

The sparrow2011 [1], the winner of SAT competition 2011, is based on gNovelty$^+$ [3] framework. But instead of using the Novelty$^+$ jump [2], sparrow2011 uses its own dynamic scoring function at stagnation phase. That dynamic scoring function in sparrow2011-PCL was justified by using function $p_f(v_i)$ of $stagnation\_weight(x_i)$ instead of using $p_a(v_i)$ of $age(x_i)$ of sparrow2011. The new scoring function is described as following.

$$p(v_i) = \frac{p_s(v_i) * p_f(v_i)}{\sum_{v_i} p_s(v_i) * p_f(v_i)}$$

Algorithm 2: $gNovelty^{+}PCL(k, s, T, sp)$

Input : A formula Θ, smooth probability sp, tenure k, window size T,"Static"or "Dynamic"option s
Output: Solution α (if found) or TIMEOUT

```
randomly generate a candidate solution α;
set up window queue W = ∅; set up flipping history stack H = ∅;
initialized stagnation_weight of all variables to 0;
while not timetout do
    if α satisfied the formula Θ then return α ;
    if in the random walk probability wp then
        randomly pick up a variable in a false clause;
    else
        if there exist promising variables then
            select most promising variable, breaking tie by least
            stagnation_weight;
        else Stagnation happens: perform pseudo-conflict learning
            update (and smooth in probability sp) clause_frequency;
            PCL(k,s,T,H,W);
            Novelty Jump : select the most improving variable in a random
            unsatisfied clause, breaking tie by least stagnation_weight;
    update candidate solution α with the flipped variable var;
    H ← push_stack(H, var);
return TIMEOUT;
```

Algorithm 3: $sparrow2011 - PCL(k, s, T, sp)$

Input : A formula Θ,tenure k, window size T,"Static"or "Dynamic"option s
Output: Solution α (if found) or TIMEOUT

```
randomly generate a candidate solution α;
set up window queue W = ∅; set up flipping history stack H = ∅;
initialized stagnation_weight of all variables to 0;
while not timetout do
    if α satisfied the formula Θ then return α ;
    if in the random walk probability wp then
        randomly pick up a variable in a false clause;
    else
        if there exist promising variables then
            select most promising variable, breaking tie by least
            stagnation_weight;
        else Stagnation happens: perform pseudo-conflict learning
            update (and smooth in probability sp) clause_frequency;
            PCL(k,s,T,H,W);
            Select variable in an random unsatisfied clause: prefer higher new
            score breaking tie by least stagnation_weight variable;
    update candidate solution α with the flipped variable var;
    H ← push_stack(H, var);
return TIMEOUT;
```

As reported in the work [1], component $p(v_i)$ has parameters $c1$ wheres $p_a(v_i)$ has two parameters of $c2$ and $c3$. The new component $p_f(v_i)$ is based on two parameters $c4$ and $c5$ according to the following formula. According to this scoring function, the variable whose stagnation_weight is low is preferred to be selected.

$$p_f(v_i) = (\frac{c5}{stagnation\_weight(v_i)+1})^{c4} + 1$$

Algorithm 3 illustrates integration of sparrow2011 and PCL heuristics. Similarly to gNovelty^{+}PCL, the PCL heuristics was invoked at the stagnation point. Additionally, instead of breaking tie by least recent flipped variable, the algorithm prefers least stagnation frequent variable.

V. Implementation Environments and Settings for the Competition

gNovelty^{+}PCL and sparrow2011-PCL were written on C language and developed respectively from gNovelty^{+} and sparrow2011 (which was published on SAT2011 competition website [1]). Both solvers were complied by gcc using optimization flag as the following

CFLAGS= -static-libgcc -O3 -fno-strict-aliasing -fomit-frame-pointer -funroll-all-loops -fexpensive-optimizations -malign-double -Wall -march=native -pipe -msse4.2 -ffast-math

The two solvers registered for 3 tracks: Application SAT+UNSAT, Hard Combinatorial SAT+UNSAT and Random SAT. Unfortunately, source codes are not allowed to be published but the binary executive files. For the SAT 2012 Challenge, parameter setting for gNovelty^{+}PCL is k = 20, s = "Static", T = 200, sp = 0. Meanwhile, sparrow2011-PCL in the competition employs the dynamic stagnation zone s = "Dynamic"and other parameters presented as the following table.

Problems	k	T	c4	c5
3-SAT	10	300	1	1.5
5-SAT	25	200	2	2.0
7-SAT	30	250	5	16.0

References

[1] Balint, A., Fröhlich, A.: Improving stochastic local search for sat with a new probability distribution. In: SAT. pp. 1015 (2010)
[2] Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02). pp. 635660 (2002)
[3] Pham, D.N., Thornton, J., Gretton, C., Sattar, A.: Combining adaptive and dynamic local search for satisfiability. JSAT 4(2-4), 149172 (2008)
[4] Pham, D.N., Duong, T.T., Sattar, A.: Trap avoidance in local search using pseudo-conflict learning. To appear in AAAI'12 Proceedings, 2012

[1] http://www.satcompetition.org

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Industrial Satisfiability Solver (ISS)

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Brown University, Dept. of Computer Science, Providence, RI 02912, USA
Email: ynm@cs.brown.edu
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

The Industrial Satisfiability Solver is a systematic SAT solver with the capability of proving unsatisfiability. ISS version 2.1 participated in the Application track of SAT Challenge 2012.

I. Solving techniques

ISS is based on a number of existing solvers, including systematic and local search solvers. The program hybridizes these solvers by running meta-restarts.

First, we simplify the given instance using SatELite [1]. We then continue in standard restarted clause learning manner. When a fail-limit is reached, the solver used may continue with a new ordinary restart, by setting a new fail-limit and restarting the search. However, it may also launch a meta-restart: Rather than continuing the search by itself, it may forward information learnt (conflict clauses, variable activities, polarity information etc) to another systematic solver – potentially after invoking one or more local search methods first. Moreover, before the new solver is called, the SAT instance, augmented by newly learnt clauses, is simplified again by SatELite.

Then, the new systematic solver takes over the search using its own specific way of slecting variables, forgetting clauses, adjusting variable activity levels, setting fail limits, etc. When the new solver decides to launch a meta-restart itself, it may again invoke solvers that were used already earlier.

Given an instance, the schedule of solvers to execute in this fashion is generated by analyzing the instance and choosing among a set of baseline solvers. Similar to the solver 3S [3], ISS makes its selection from 38 sequential baseline solvers.

II. Implementation details

Several of the conflict directed clause learning solvers were enhanced in an effort to support the concept of meta-restart. E.g., the solver code was modified to allow forwarding of learnt clauses to other solvers scheduled subsequently. With regard to managing launching of different solvers, even though ISS is developed with a very different mindset than the portfolio solver 3S, for convenience we used a similar Python-based setup to launch individual solvers [3]. The launcher, however, was enhanced to work with meta-restarts. E.g., it has the capability to parse learnt clause generated by baseline solvers and add them to the formula provided as input to subsequent solvers. ISS selects amongst the same 38 sequential solvers as used by 3S.

III. SAT Challenge 2012 specific details

The command line for ISS in SAT Challenge 2012 was:

python ISS-2.1.py –scale 1.3 -l 2
–tmpdir TEMPDIR INSTANCE

Please refer to the solver description of 3S [3] for an explanation of the scaling parameter used by the launcher.

The solvers exchanging information are tts-4-0, ebminisat, Glucose, Minisat, and Precosat570. A solver from the list in [3] may be launched depending on the given SAT instance. ISS is optimized for the competition timeout of 900 seconds on the competition machines. Execution with different timeouts or on other machines will likely result in reduced performance.

The solver presented is not a portfolio, although it uses existing SAT solvers that we augmented to exchange information in a non-trivial way. The resulting solver is able to solve instances that none of the existing SAT solvers it uses could solve by itself. Moreover, the source code of solvers needed to be changed to enable the exchange of information between consecutively scheduled solvers.

Acknowledgment

With the previous statement in mind, the solver that competed is nevertheless based heavily on existing work on “pure” SAT solvers. A complete list of these solvers is given in [3]. 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] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.
[2] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.
[3] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Satisfiability Solver Selector (3S). *SAT Challenge 2012 solver descriptions*, 2012.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

interactSAT{_c}: Interactive SAT Solvers and glue_dyphase: A Solver with a Dynamic Phase Selection Strategy

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

***Abstract*—The SAT solvers submitted to the SAT Challenge 2012 are interactSAT, inteactSAT_c and glue_dyphase, which are sequential solvers. The former two solvers are based on an interactive framework. the core solver of interactSAT is Glucose, while that of interactSAT_c is clasp. So they are suitable for solving the application and crafted category respectively. glue_dyphase is the improved version of Glucose. With respect to variable phase selection, glue_dyphase adopts a dynamic phase selection strategy, while Glucose does a static RSAT strategy.**

I. Introduction

The main solving mechanisms used in the state-of-the-art solvers are conflict-driven, look-ahead and local search. For some SAT problems, these mechanisms are still failure. For this reason, in our solvers interactSAT and interactSAT_c, we introduce a new solving mechanisms called interactive technique.

For conflict-driven solvers, a lot of effort has been made in designing smart restart policies, conflict analysis and learnt clause database management. Only a few work is to study the phase selection of a variable. In [3], we present a dynamic phase selection strategy that can dramatically improve the performance of Glucose [1], [2] on which glue_dyphase is based. glue_dyphase is based on the latest publicly available version of glucose.

II. An Interactive Solving Technique

Due to the diversification feature of SAT problems, different problems needs different solving strategies. If the same problem consists of multiple different structures, it needs multiple solving strategies. So far, for a SAT problem, all the state-of-the-art sequential solvers use a solving strategy to solve it. Even if the solver is based on portfolio methods, it use one solving strategy one time. If the prediction is not correct, the portfolio solver fails often to find a solution. Our interactive solver does not perform any prediction computation, but runs multiple solving strategies independently and exchanges their intermediate solutions. The interactive solving strategy used in our solver may be outlined briefly as follows.

(1) Run simultaneously m solvers with different solving strategies. For each solving strategy, each time there is a limit of at most n conflicts to solve the instance. Initially $n = 10000$.
(2) Pass the intermediate solution of the i-th solver to the $(i+1)$-th solver, where $1 \leq i \leq m$. When $i = m$, $i+1$ corresponds to 1.
(3) If a solution has been found, the solving process terminates .
(4) Modify the maximum number n of conflicts. If a solving strategy is better than the other solving strategy, the corresponding n increases. Otherwise, the corresponding n decreases. Goto step (1).

Measuring the quality of a solving strategy can be done by computing the number of fixed variables or other parameters. The above is only a framework. The specific implementation must be fine-tuned further.

III. A Dynamic Phase Selection Strategy

In modern conflict-driven SAT solvers, How to select the phase of a variable is an inseparable step that follows the decision variable selection, because we must assign each decision variable to a value. The simplest phase selection policy is that each decision variable is always assigned to false, which is used as a default heuristic of MiniSAT. No evidence shows that such a policy is always efficient. Therefore, other policies are adopted in some solvers. For example, PrecoSAT [4] used Jeroslow-Wang heuristic. Here we use a new dynamic phase selection policy. Its weight is based on a static weight, but computed dynamically. Let $\mathcal{F}$ define an input formula in CNF (Conjunctive Normal Form) clauses. The static weight of a literal x on $\mathcal{F}$ is defined as

$$W(x, \mathcal{F}) = \sum_{c \in \mathcal{F}(x)} 5^{2-\mathrm{size}(c)}$$

where $\mathcal{F}(x)$ is the set of clauses in which x occurs, and $\mathrm{size}(c)$ is the size of clause c. This is very similar to the definition of a weight in Jeroslow-Wang heuristic [5]. The main difference between them is that the base is different. Our base is 5, while the base Jeroslow-Wang heuristic is 2. Selecting 5 is because the March solver uses 5 also [8]. In our solvers, the dynamic weight of a literal x is defined as the sum of the static weight of literals implied by it. This definition can be formulated as follows.

$$DW(x, \mathcal{F}, \mathcal{F}') = \sum_{x \wedge \mathcal{F}' \vdash y} W(y, \mathcal{F})$$

where $\mathcal{F}$ and $\mathcal{F}'$ are an input formula and a formula at a search state, respectively. Usually, $\mathcal{F}$ is constant, and $\mathcal{F}'$ varies with the search state. $x \wedge \mathcal{F}' \vdash y$ means that using the fact that x is true, applying unit resolution on formula $\mathcal{F}'$ can derive an implication y. That is, y is an implied literal of x under $\mathcal{F}'$. Computing implied literals is simple. This can be done by a unit propagation, i.e. so-called BCP. The dynamic strategy here need not any additional data structure such as a full watched-literals scheme, and can apply directly a two watched-literals scheme. Therefore, our dynamic strategy is very efficient. Once a variable is decided, the dynamic strategy elects the branch with the highest dynamic weight DW. Let x be the decision variable. A search on x, including the computation of dynamic weights, may be described as follows.

search (x,W, $\mathcal{F}'$)
  $\langle Y_+$, Ret$\rangle \leftarrow$ BCP(x, $\mathcal{F}'$)
  if Ret=UNSAT then return UNSAT
  backtrack to current level
  $\langle Y_-$, Ret $\rangle \leftarrow$ BCP($\neg x$, $\mathcal{F}'$)
  if Ret=UNSAT then return UNSAT
  if $DW(W, Y_-) > DW(W, Y_+)$ then $y = \neg x$, goto next
  $y = x$
  backtrack to current level
  BCP(x, $\mathcal{F}'$)
next:
  find next literal on branch y

In the above procedure, parameters W is used to store the static weights of all the literals. $\mathcal{F}'$ is the current formula, which can be maintained usually by a trail tack. Y_+ and Y_- are the set of literals implied by x and $\neg x$, respectively. Like the usual BCP, BCP(x,$\mathcal{F}'$) fulfills two tasks of compute the implied literals and determine whether it reaches a conflict. The usual search runs only one time, but our search need to run at most three times. If the dynamic weight of $\neg x$ is large than that of x, we run BCP just two times, since in such a case, the last BCP is consistent with the search direction. Clearly, in the worst case, the cost of our search is at most triple the cost of the usual search if each BCP has the same cost.

IV. System Description of SAT Solvers

A. interactSAT

interactSAT is a kind of CDCL solver based on an interactive framework. It is dedicated to solving application instances. This solver consists of two stages. The first stage is a preprocessing developed by us, which simplifies a CNF formula. The second stage is to solve the simlified CNF formula. The solving process in this solver is based on an interactive solving strategy. Its main framework is Glucose. It consists of four solving strategies, three of which are Glucose-style solvers with different dynamic phase selection parameters. Another solving strategy is CryptoMiniSat [6]. For very large instances, we use directly Glucose to solve them.

B. interactSAT_c

interactSAT_c is a new kind of hybrid solver combining local search, CDCL and look-ahead. Its basic framework is the same as that of interactSAT mentioned above. It is a interactive solver. However, the solving strategy is different from interactSAT, since this solver is mainly used to solve crafted instances. The solving strategis used in interactSAT_c have clasp [7], march [8], sparrow2011 [9]. Like interactSAT, its framework is based on Glucose. The interactive feature of many crafted instances is weak. Therefore, in many cases, interactSAT_c uses mainly clasp to solve crafted instances.

C. glue_dyphase

glue_dyphase is an improved version of Glucose, a kind of CDCL solver. Except for the variable polarity selection strategy, glue_dyphase is the same as Glucose. glue_dyphase uses a dynamic phase selection strategy given above, while Glucose does a phase selection policy based on the RSAT heuristic[10]: it always assigns a decision variable to false if that variable was never visited, and the previous value otherwise. However, for very large instances, the phase selection strategy of glue_dyphase is the same as that of Glucose.

V. The performance sensitive parameters

The performance sensitive parameters used in interactSAT are the number of conflicts, the total number of literals, the number of clauses ($c\#$), the number of variables ($v\#$), the number of binary clauses and the number of fixed variables. When $c\# > 6000000$ or $v\# > 2000000$, interactSAT switches to the original Glucose. Otherwise, it solves the SAT problem in the following way: In the first 150000 conflicts, we run an improved Glucose, using one of three dynamics phase selection policies. If no solution was found in this stage, we proceed to the interactive solving stage. In this stage, we prepare four sub-solvers: one is CryptoMiniSat and the other three ones are Glucose-style solvers with different dynamic phase selection policies. Initially, each sub-solver runs 8 search periods (A search period refers to the search process between two restarts). In the subsequent search process, the running length of a sub-solver depends on its current performance. The performance of a solver is measured by the number of fixed variables. In the final stage, at each time, the solver with the best performance runs until at least a new variable is fixed, and the other solvers runs only one search period with the maximal number limit 1000 of conflicts. In the interactive solving stage, if the total number of literals in the input formula is very large, say 3000000, we do not run CryptoMiniSat. In the first stage, selecting which dynamic phase policies depends on the number of binary clauses.

The performance sensitive parameters used in interactSAT_c are almost the same as those used in interactSAT. In addition to the parameters given above, interactSAT_c uses the average search depth ($D\#$) to select sub-solvers. For large instances, say $c\# > 6000000$ or $v\# > 2000000$, interactSAT_c switches also to the original Glucose. In the interactive solving stage, if $D\# < 29$, we run march first and then run clasp. Otherwise,

we run directly clasp. Before entering the interactive solving stage, if $D\# > 11$ and $D\# < 100$ and $v\# < 20000$, we try to run sparrow2011 for at most three times. The maximal number of flips each time is limited to 4000000.

The performance sensitive parameters used in glue_dyphase are the number of conflicts, the total number of literals and the number of fixed variables. When the total number of literals is large, say 1600000, glue_dyphase switches also to the original Glucose. Initially, the phase saving policy is set to 2. However, in the first 1000000 conflicts, if the number of fixed variables has increased, the phase saving policy is switched to 1. In general, we adopt the first dynamic phase selection policy. When the number of conflicts reaches 5000000, we switch to the second dynamic phase selection policy.

VI. Conclusion

All the SAT solvers we submitted to the SAT Challenge 2012 are sequential. Nevertheless, the interactive solving technique described above is of parallel characteristic by itself. Therefore, parallelizing these SAT solvers should be very easy. Furthermore, the parallel solver with the interactive solving technique should be more efficient. This will be our future further work.

References

[1] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers, IJCAI 2009, 399–404 (2009)

[2] Audemard, G., Lagniez, J.M., Mazure, B., Saïs, L.: On Freezing and Reactivating Learnt Clauses, SAT 2011, LNCS 6695, 188–200 (2011)

[3] Chen, J.C.: A Dynamic Phase Selection Strategy for Satisfiability Solvers, submitted for publication, 2012.

[4] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010, http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_1+2+3+6.pdf

[5] Jeroslow, R., Wang, J.: Solving propositional satisfiability problems, Annals of Mathematics and Artificial Intelligence, 1, 167–187 (1990)

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

[7] 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)

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

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

[10] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers, SAT 2007, LNCS 4501, 294–299 (2007)

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Linge_dyphase

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

***Abstract*—Linge_dyphase is an improved version of the sequential solver Lingeline 587f [2]. In addition to adopting a static Jeroslow-Wang strategy [3] + RSAT strategy [4], Linge_dyphase adopts a dynamic phase selection strategy, while Lingeline does only a static Jeroslow-Wang strategy + RSAT strategy.**

I. A Dynamic Phase Selection Strategy

In modern conflict-driven SAT solvers, How to select the phase of a variable is an inseparable step that follows the decision variable selection, because we must assign each decision variable to a value. The simplest phase selection policy is that each decision variable is always assigned to false, which is used as a default heuristic of MiniSAT. No evidence shows that such a policy is always efficient. Therefore, other policies are adopted in some solvers. For example, PrecoSAT [2] used Jeroslow-Wang heuristic. Here we use a new dynamic phase selection policy. Its weight is based on a static weight, but computed dynamically. Let $\mathcal{F}$ define an input formula in CNF (Conjunctive Normal Form) clauses. The static weight of a literal x on $\mathcal{F}$ is defined as

$$W(x,\mathcal{F}) = \sum_{c\in\mathcal{F}(x)} 5^{2-\mathrm{size}(c)}$$

where $\mathcal{F}(x)$ is the set of clauses in which x occurs, and $\mathrm{size}(c)$ is the size of clause c. This is very similar to the definition of a weight in Jeroslow-Wang heuristic [3]. The main difference between them is that the base is different. Our base is 5, while the base Jeroslow-Wang heuristic is 2. Selecting 5 is based on the fact that the March solver uses also 5 [5]. In our solvers, the dynamic weight of a literal x is defined as the sum of the static weight of literals implied by it. This definition can be formulated as follows.

$$DW(x,\mathcal{F},\mathcal{F}') = \sum_{x\wedge\mathcal{F}'\vdash y} W(y,\mathcal{F})$$

where $\mathcal{F}$ and $\mathcal{F}'$ are an input formula and a formula at a search state, respectively. Usually, $\mathcal{F}$ is constant, and $\mathcal{F}'$ varies with the search state. $x\wedge\mathcal{F}'\vdash y$ means that using the fact that x is true, applying unit resolution on formula $\mathcal{F}'$ can derive an implication y. That is, y is an implied literal of x under $\mathcal{F}'$. Computing implied literals is simple. This can be done by a unit propagation, i.e. so-called BCP. The dynamic strategy here need not any additional data structure such as a full watched-literals scheme, and can apply directly a two watched-literals scheme. Therefore, our dynamic strategy is very efficient. Once a variable is decided, the dynamic strategy elects the branch with the highest dynamic weight DW. Let x be the decision variable. A search on x, including the computation of dynamic weights, may be described as follows.

search (x,W, $\mathcal{F}'$)
  $\langle Y_+$, Ret$\rangle \leftarrow$ BCP(x, $\mathcal{F}'$)
  if Ret=UNSAT then return UNSAT
  backtrack to current level
  $\langle Y_-$, Ret $\rangle \leftarrow$ BCP($\neg x$, $\mathcal{F}'$)
  if Ret=UNSAT then return UNSAT
  if $DW(W,Y_-) > DW(W,Y_+)$ then $y=\neg x$, goto next
  $y = x$
  backtrack to current level
  BCP(x, $\mathcal{F}'$)
next:
  find next literal on branch y

In the above procedure, parameters W is used to store the static weights of all the literals. $\mathcal{F}'$ is the current formula, which can be maintained usually by a trail tack. Y_+ and Y_- are the set of literals implied by x and $\neg x$, respectively. Like the usual BCP, BCP(x,$\mathcal{F}'$) fulfills two tasks of compute the implied literals and determine whether it reaches a conflict. The usual search runs only one time, but our search need to run at most three times. If the dynamic weight of $\neg x$ is large than that of x, we run BCP just two times, since in such a case, the last BCP is consistent with the search direction. Clearly, in the worst case, the cost of our search is at most triple the cost of the usual search if each BCP has the same cost.

II. System Description of Linge_dyphase

Linge_dyphase is an improved version of lingeling, a kind of CDCL solver. Except for the phase selection strategy, linge_dyphase is the same as lingeling. In general, in even depths of even search periods (A search period refers to the search process between two restarts), linge_dyphase uses a dynamic phase selection strategy, while in odd depths of even search periods, linge_dyphase uses almost the same as lingeline, i.e., a static Jeroslow-Wang+RAST strategy [3]. However, in odd search periods, conversely, in even depths linge_dyphase uses almost the same as lingeline, while in odd depths it uses a dynamic phase selection strategy. The Jeroslow-Wang strategy here is a bit different from that of lingeline. When linge_dyphase applies initially the Jeroslow-Wang strategy, its weight is based on the dynamic weight given above.

III. Parameter description

The performance sensitive parameters used in linge_dyphase are the number of conflicts. Within the first 10000 conflicts, we adopt the full dynamic phase

selection policy. This dynamic policy means that each decision variable need to be computed its dynamic weight. When the number of conflicts exceeds 10000, we adopt the half dynamic phase selection policy. In this stage, only half decision variables need to be computed their dynamic weights.

REFERENCES

[1] Chen, J.C.: A Dynamic Phase Selection Strategy for Satisfiability Solvers, submitted for publication, 2012.

[2] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010, http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_1+2+3+6.pdf

[3] Jeroslow, R., Wang, J.: Solving propositional satisfiability problems, Annals of Mathematics and Artificial Intelligence, 1, 167–187 (1990)

[4] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers, SAT 2007, LNCS 4501, 294–299 (2007)

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

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

LINGELING and Friends Entering the SAT Challenge 2012

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

This note describes our SAT solvers submitted to the SAT Challenge 2012, all based on the same LINGELING backend.

I. LINGELING

Compared to the version submitted to the SAT competition 2011 and described in [1], we removed complicated algorithms and features, which did not really have any observable impact on the run-time for those benchmarks we tried. In particular, various versions of distillation inprocessors were removed.

Regarding inprocessing [4], there are two new probing variants. One is called *simple probing* and tries to learn hyper binary resolutions eagerly. The other variant is based on *tree-based look-ahead*, which is a simplified version of the implementation in March [2]. These two techniques are complemented by *gaussian elimination* and a new *congruence closure algorithm*, which both use extracted gates to generate and propagate equivalences.

We also switched to one merged inprocessing phase, called *simplification*, where all inprocessors run one after each other, instead of allowing each inprocessor to be scheduled and interleaved with search individually.

Furthermore, for most inprocessors we have now a way to save the part of the formula on which the inprocessor did not run until completion (actually currently only "untried variables"). In the next simplification phase, the algorithm can be resumed on that part, such that eventually we achieve the same effect as really running the various algorithms until completion. Previously we used randomization to achieve a similar effect. This technique also allowed us to remove certain limits, such as the maximum number of occurrences or the maximum resolvent size in variable elimination.

We moved to an inner-outer scheme for the size of kept learned clauses, also called *reduce schedule*. The inner scheme follows the previously implemented LBD resp. glue based scheme as in Glucose. As in the previous version the solver might switch to activities dynamically, if the glue distribution is skewed. The outer schedule is Luby controlled and resets the learned clause data based limit to its initial size. This idea is particularly useful for crafted instances.

Another new feature is to occasionally use the opposite of the saved phase for picking the value for the decision variable. These *flipping* intervals start at the top-level and while flipping is enabled the phases of assigned variables are not saved (as in probing) in order not to counteract the effect of the phase saving mechanism.

The exponential VSIDS scheme of MiniSAT has been replaced by a new variant of a *variable-move-to-front* strategy with multiple queues ordered by priority. This seems to be at least as effective as the previous scheme, but updating and querying the queue turns out to be substantially faster.

In general we simplified internal data structures with the hope to make the code a little bit more accessible.

II. PLINGELING

The parallel version of LINGELING has not changed much. Still only units and equivalences are shared among multiple solver instances. Actually, we even removed most options previously set differently for each instance of the LINGELING core library, except of course for the seed of the random number generator and in addition kept the different choices for the default phase. Last but not least we only use one instance during parsing. This first solver instance is cloned after preprocessing. This reduces the memory usage on certain instances considerably, particularly, since we can stop cloning as soon too much memory is already in use.

III. CLINGELING

CLINGELING is based on our new *Concurrent Cube and Conquer* (CCC) approach [5]. This is an extension of our previous *Cube and Conquer* (CC) technique [3]. The new idea is to run a CDCL solver and a look-ahead solver *concurrently*.

CLINGELING uses the new `lglfork` API call, provided by the LINGELING library for copying a solver instance. It additionally uses a global LINGELING instance with assumptions to partially simulate what ILINGELING does in the original CC approach (but in an interleaving fashion and with only one worker thread). The look-ahead literal is computed by the tree-based probing algorithm discussed already above.

In the current version of CLINGELING, the CDCL part and the look-ahead are interleaved, and thus not really run in parallel. Which also means that there is no benefit from multi-core machines as in the original CC (and CCC) approach. But compared to CC we can find better cut-off limits for switching from look-ahead to pure CDCL with inprocessing this way, which is one of the motivations behind CCC.

Another drawback of this online approach is that the global solver, can not determine up-front the set of variables that can be eliminated in pre- and inprocessing.

IV. FLEGEL

FLEGEL can be seen as a poor man's version of CLINGELING. It uses the `fork` system call for backtracking and in principle such a front-end should be easy to build for any SAT library, which can produce a look-ahead decision. It just runs preprocessing and a limited amount of search of the CDCL solver (with inprocessing) at each node before calculating the next look-ahead literal. Then the process is forked. The child process adds the look-ahead literal as unit and continues the same procedure recursively. Currently parents wait for their child to terminate, before adding the negation of the original look-ahead literal as unit. So even FLEGEL uses as many processes as active search nodes, i.e. the height of the search tree, no parallelism is used.

V. TREENGELING

TREENGELING is our latest SAT solver with LINGELING backend and tries to capture the positive aspects of PLINGELING, FLEGEL and CLINGELING and actually to some extend also ILINGELING [3]. To simulate *forking* in FLEGEL we implemented a *clone* function `lglclone` as part of LINGELING. This function in contrast to `lglfork`, which is used in CLINGELING, generates an identical behaving solver instance, instead of just copying clauses and assumptions. In the context of TREENGELING this allows to additionally copy saved phases, variable queue, etc., so all the state, from the original solver instance to the clone.

The clone has all information for reconstructing a solution. So there is no need to propagate the solution back by merging a forked copy with the `lgljoin` API call, as it is necessary if the copy was generated by `lglfork`, e.g., as in CLINGELING.

Up to this point TREENGELING is very similar to FLEGEL. However, since we have all the cloned solver instances in one address space (as in CLINGELING) we can easily use multiple threads to run the updated clones in parallel. TREENGELING is a parallel solver and uses the infra-structure for parallel execution also used in PLINGELING and ILINGELING.

Solver instances are stored in nodes and we start with one single solver instance with the original formula, which is then first simplified in a *simplification* phase. If there are less open nodes than a predefined limit, a decision literal is selected by tree-based look-ahead from the smallest solver instance in a *look-ahead* phase. This instance is cloned and saved in a new node during the *splitting* phase. The decision is added as unit to the clone and negated to the original solver instance.

Lookahead and then splitting existing solvers this way is actually performed in parallel. After splitting, the solvers of open nodes are run for a certain conflict limit in a *search* phase. If a solver instance finds a solution it is printed and the whole search terminates. If the solver instance of one node proves unsatisfiability, it is closed. After all solver instances terminated their limited search, closed nodes are flushed. If no more nodes remain, the search terminates with proving unsatisfiability.

Then the conflict limit is updated in an *update* phase. If a node was closed in this round the limit is decreased and otherwise increased, both in a geometrical way. Due to the potential exponential increase of the conflict limit over multiple rounds, TREENGELING with one worker behaves very similar to the base LINGELING solver. It does not behave identically though, as it is the case for PLINGELING with one worker. The pseudo-code of this procedure looks as follows:

```
search(lim);
while (!flush) {
  simp; lookahead; split; upd(lim); search;
}
```

Sub-procedures work in parallel, e.g. simplification (`simp`) is run in parallel for the minimum of still open nodes and number of cores. The default is to use both the number of cores as maximum limit on the number of open nodes and worker (threads). For multiple workers units are added and tree-based lookahead has to be performed, but otherwise, since the workers run in parallel independently, the (wall-clock time) performance is not expected to be much worse than for plain LINGELING. Preliminary experiments justify this claim. This does not seem to hold for CLINGELING nor FLEGEL.

TREENGELING is deterministic, e.g. always traverses the same search space and produces the same number of conflicts (actually only for unsatisfiable instances) etc., as long the maximum number of active nodes stays the same and the same memory limit is used. The number of threads available to work in parallel during simplification, search or lookahead does not influence the search. With more available cores, more threads can be run in parallel, without run-time penalty.

However, in order to use more threads, more active nodes have to exist in parallel. In preliminary experiments we unfortunately saw a negative effect on wall-clock time, if more open nodes are used than available cores (except when hyper-threading is available). Thus the effectiveness of this approach w.r.t. speed-up is not really understood yet. On processors with four cores and no hyper-threading, TREENGELING with a maximum of four open nodes, is expected to perform slightly better than PLINGELING, e.g., substantially, but not dramatically better than plain LINGELING.

VI. ACKNOWLEDGEMENTS

This work heavily depends on research results obtained with my collaborators Marijn, Matti, Oliver, Siert, and Peter, and of course the whole SAT community. A full list of proper references can be found in the following papers.

REFERENCES

[1] Armin Biere. Lingeling and friends at the SAT Competition 2011. FMV Report Series Technical Report 11/1, Johannes Kepler University, Linz, Austria, 2011.

[2] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In *SAT 2004 Selected Papers*, volume 3542 of *LNCS*, pages 345–359. Springer, 2005.

[3] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guiding CDCL SAT solvers by lookaheads. In *Proc. HVC 2011*, 2012. To appear.

[4] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In *Proc. IJCAR'12*. To appear.

[5] Peter van der Tak, Marijn Heule, and Armin Biere. Concurrent Cube-and-Conquer. Submitted.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

march_nh

Marijn J. H. Heule
Department of Software Technology,
Delft University of Technology,
The Netherlands
Email: marijn@heule.nl

I. INTRODUCTION

The march_nh SAT solver is the latest version of the successful march lookahead solver, which won several awards at the SAT 2004, 2005, 2007, 2009 and 2011 competitions. For the latest detailed description of the various techniques used in march, we refer to [1], [2], [3], [4]. Like its predecessors, march_nh 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 enhancements (apart from bug fixes) in march_nh are:

- support for the cube-and-conquer technique [5]. This technique splits the input formula in multi subformulas (each subformula expressed as a cube) which is written to file (iCNF format[1]) to be used by in incremental conflict-driven clause learning solver.
- hyper binary resolvents [6] are added much more aggressively compared to earlier versions, especially in the nodes near the root of the search tree.

II. PRE-PROCESSING

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

III. PARTIAL LOOKAHEAD

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

[1]http://users.ics.tkk.fi/swiering/icnf/

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
```

IV. 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.
- Binary equivalences: If both $x_i \rightarrow x_j$ and $\neg x_i \rightarrow \neg x_j$ are detected during the look-ahead on x_i and $\neg x_i$, binary equivalence $x_i \leftrightarrow x_j$ is propagated in the CoE data-structure to reduce the length of some equivalence clauses.
- 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 PARTIAL-LOOKAHEAD, all satisfied ternary clauses of the prior node are removed from the active data-structure to speed-up the lookahead.

REFERENCES

[1] M. J. H. Heule, J. E. van Zwieten, M. Dufour, and H. van Maaren, "March_eq: Implementing additional reasoning into an efficient lookahead SAT solver," in *SAT 2004*, ser. Lecture Notes in Computer Science, H. H. Hoos and D. G. Mitchell, Eds., vol. 3542. Springer, 2005, pp. 345–359.

[2] M. J. H. Heule and H. van Maaren, "March_dl: Adding adaptive heuristics and a new branching strategy," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 47–59, mar 2006.

[3] S. Mijnders, B. de Wilde, and M. J. H. Heule, "Symbiosis of search and heuristics for random 3-sat," in *Proceedings of the Third International Workshop on Logic and Search (LaSh 2010)*, D. Mitchell and E. Ternovska, Eds., 2010.

[4] M. J. H. Heule and H. van Maaren, "Whose side are you on? finding solutions in a biased search-tree," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 117–148, 2008.

[5] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding cdcl sat solvers by lookaheads," in *Best paper award at HVC 2011, to appear*, 2012. [Online]. Available: http://www.st.ewi.tudelft.nl/ marijn/publications/cube.pdf

[6] 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.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Minifork

Yuko Akashi
Kyushu University, Japan
2ie11001y@s.kyushu-u.ac.jp

I. Introduction

Minifork is a parallel SAT solver based on a CDCL solver MiniSat2.2.0[1]. It parallelization is performed by using a UNIX system call 'fork'. The fork is invoked just after selecting a variable to assign. After the fork, the parent process deals with a case where the variable is assigned '1' while the child process deals with another case where the variable is assigned '0'. Thus, both the parent and child processes search distinct search spaces and can run independently. Their synchronization occurs only when they finish their jobs or fork again. The parent process waits until its all child processes terminate in order to prevent them from becoming zombie processes. Before a process forks, it checks whether the number of active SAT processes less than the number of the cores in order to prevent the system from being overload.

No parameter of MiniSat is changed for any SAT instances. Minifork is written in C++ and its compiling method is the same as MiniSat.

II. Timing of Fork

To start with we invoke a SAT process. After its ten restarts, it executes fork. Just before the fork, we select a variable using pickBranchLit(). After the fork, the parent process solves the case where the variable is assigned '1' while the child process solves another case where the variable is assigned '0'. The child process has a copy of learned clauses which may reduce search space afterward.

After the first fork, every process executes fork at every restart if the number of the current SAT processes is less than the number of the cores. Thus, we can keep the number of SAT processes being less than or equal the number of the cores. Because several SAT processes may execute fork simultaneously, these processes need to synchronize. The synchronization is realized by using a semaphore. The number of the current SAT processes is memorized in a semaphore. The number is increased by 1 when a fork succeeds while it is decreased by 1 when a SAT process is finished.

A semaphore is useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment. The semaphore in Miniforks is implemented by including $< sys/sem.h >$ of the C language. The number of the current SAT processes is memorized in a semaphore which is shared by all SAT processes. The share is implemented by including $< sys/shm.h >$ of the C language. The critical section is protected with the semaphore.

III. Termination

If a SAT process find a model, the other processes do not need to continue their jobs any more. In order to tell the processes that the model is found, we also use the same semaphore introduced in the previous section. When a process find a model, it writes a big number, which indicates SATISFIABLE, to the semaphore instead of the decrement. The other processes notice the fact at their next restarts. Thus, the processes do not terminate immediately.

IV. Conclusion

Minifork implementation is simple in the sense that there is no complicated synchronization mechanism, and light in the sense that the SAT processes do not share any learned clauses after forks. We plan to introduce a light mechanism for sharing useful learned clauses, and solve open problems with Minifork.

Acknowledgment

My deepest appreciation goes to Prof. Hasegawa whose enormous support and insightful comments were invaluable during the course of my study. I am also indebt to Associate Prof.Fujita and Assistant Prof.Koshimura whose comments made enormous contribution to my work. This work was supported by JSPS KAKENHI(20240003, 21300054).

References

[1] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In SAT 2003, 2003.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Parallel CIR MiniSAT

Tomohiro Sonobe
Graduate School of Information Science and Technology, University of Tokyo
Email: tominlab@gmail.com

***Abstract*—We introduce a new parallel solver based on counter implication restart (CIR) MiniSAT which we submitted to SAT Competition 2011. In this description, we explain the details of CIR and its role in parallel context.**

I. Introduction

In SAT Competition 2011, we submitted a sequential SAT solver that employs Counter Implication Restart (CIR). CIR is a novel restart policy for escaping from wrong branches vigorously. It consists of the standard restart and bumping VSIDS scores in order to change decision orders after the restart by analyzing the implication graph. We found that CIR is effective for time-consuming instances.

We implement a new parallel solver, Parallel CIR MiniSAT (ParaCIRMiniSAT), by modifying MiniSAT 2.2 [4] and using OpenMP. ParaCIRMiniSAT is based on portfolio approach. In portfolio approach, diversification and intensification [2] of the searching are important. We believe that CIR is useful for the diversification.

II. Counter Implication Restart

One of the important objectives of restart is to move different search space by changing the shallow level decisions which may be kept during bactracks. In recent year, frequent restart policies such as nested restart [1] and Luby restart [3] were turned out to be effective for many instances. However for some instances, we think they are insufficient to escape from large wrong branches where neither a solution nor useful learnt clauses exist. It can often occur because the search after the restart can be affected by the search before the restart, in other word, VSIDS scores are taken over to the next search.

For this issue, we proposed CIR that consists of standard restart and changing VSIDS scores of variables. In order to change VSIDS scores reasonably, CIR analyzes the implication graph constructed right before the restart. When CIR is invoked, CIR traverses whole implication graph and increases each VSIDS score of each variable, proportional to its indegree. A variable with large indegree is a unit literal in a large clause, and we believe that this variable has an important role. Finally, CIR conduct the standard restart.

The pseudo code of the function of CIR is shown below. This function is called just before the restart routine.

```
1. int run_count = 0;
2. CounterImplicationRestart() {
3.   if (run_count++ % INTERVAL == 0) {
4.     int indegree[nVar] = {0};
5.     int max_indegree = 0;
6.     [calculate indegree for each variable
        and max_indegree]
7.     for each variable var
8.       bumpScore(var, BUMP_RATIO *
              indegree[var] / max_indegree);
9.   }
10.  restart();
11. }
```

The variable "run_count" on the first line stands for the number of calls of this function. The constant number "INTERVAL" on third line limits the number of the executions of this function. In other words, the main part of this function is executed for every "INTERVAL" restart. On sixth line, the indegree is calculated by traversing the implication graph. Then, on seventh and eighth line, the VSIDS score for each variable is bumped according to the number of indegree. For this process, "BUMP_RATIO" stands for a relatively large constant number. Thus, the scores are bumped drastically. We found that the value of "INTERVAL" affects the overall performance.

Since CIR can change the search after the restart strongly, we are sure that CIR is useful for the diversification of the search in parallel context.

III. ParaCIRMiniSAT

The base solver of ParaCIRMiniSAT is MiniSAT 2.2. We use OpenMP for parallelization. In ParaCIRMiniSAT, one thread run the original MiniSAT 2.2 and the others run the MiniSAT 2.2 with CIR changing some parameters such as "INTERVAL", base number of Luby restart, and so on. Each thread shares learnt clauses whose size doesn't exceed 8.

References

[1] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, Vol. 4, No. 2-4, pp. 75–97, 2008.

[2] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and intensification in parallel sat solving. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pp. 252–265, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, Vol. 47, No. 4, pp. 173–180, 1993.

[4] Niklas Sorensson. Minisat 2.2 and minisat++ 1.1. *A short description in SAT Race 2010*, 2010.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Parallel Semi-Static Satisfiability Solver Selector (p3S-semistat)

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Brown University, Dept. of Computer Science, Providence, RI 02912, USA
Email: ynm@cs.brown.edu
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

The Parallel Semi-Static Satisfiability Solver Selector (p3S-semistat) is a parallel portfolio solver that dynamically selects and schedules various solvers across multiple cores, depending on the input instance. p3S-semistat version 2.1 participated in the Parallel Solvers track of SAT Challenge 2012.

I. Solving techniques

The Parallel Satisfiability Solver Selector (p3S) is a generalization of its sequential version 3S [2], [3]. p3S schedules both sequential and parallel "pure" solvers across multiple compute cores available on one machine. These solvers are determined by solving a corresponding optimization problem modeled as an Integer Program (IP).

The semi-static version of p3S, named p3S-semistat, schedules a pre-determined set of solvers on some of the available cores and a dynamically computed schedule on the remaining cores. p3S-semistat is built upon a variety of conflict directed clause learning SAT solvers (both sequential and parallel solvers), lookahead based solvers, and local search solver.

Similar to 3S, p3S works in two phases, an offline learning phase, and an online execution phase. We refer the reader to the solver description of 3S [3] for details, including a list of the 38 sequential baseline solvers used.

II. Implementation details

Since p3S builds upon the implementation of 3S, many of the implementation specific details remain the same as in 3S [3], including the benchmark set and timeout used in the offline training phase of p3S. The launching of (single-core and multi-core) solvers in parallel is managed by extending the Python 2.6 based sequential launcher of 3S, using the Subprocess and Signal packages. In essence, the launcher for p3S creates one schedule for each core and launches one independent copy of 3S to execute that schedule.

In addition to the 38 sequential solvers used by 3S (used both with and without preprocessing, resulting in 76 sequential solver choices), p3S also uses multi-core "pure" solvers Plingeling [1] and CryptoMiniSat [4]. Both of these solvers were run using 1, 2, 3, and 4 cores. Each of these variants, taken with and without preprocessing, counts towards the set of baseline solver for p3S, resulting in a total of 92 baseline solvers.

III. SAT Challenge 2012 specific details

The command line used to launch p3S-semistat in SAT Challenge 2012 was:

python p3Ssemistat-2.1.py –scale 1.3
–tmpdir TEMPDIR INSTANCE

Please refer to the solver description of 3S [3] for an explanation of the scaling parameter.

A pre-determined schedule was used for 7 cores and a dynamic schedule computed for the 8th core. The specifics of the preschedule used for the first 7 cores may be found in the file named preschedules/sch-p3Ssemistat.txt of the submitted solver. One invariant in this preschedule was that if, say, a 3-core solver is launched by core number k at some point of time T, then it was made sure that at least two other cores were free and available at time T to help execute the 3-core solver launched by core k. In the 8th core, along with a dynamic selection of solvers, the technique of clause forwarding was also employed, whereby clauses of size up to 10 learnt by certain solvers were "forwarded" to the next solver in the schedule.

Note that the offline training of p3S, similar to 3S, was done using a 2,000 timeout on the training machines and the 8-core schedule eventually used is optimized for the competition timeout of 900 seconds on the competition machines. Execution with different timeouts or on other machines will likely result in reduced performance.

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 "pure" 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] A. Biere. PLingeling, *SAT Race 2010 solver descriptions*, 2010.
[2] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.
[3] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Satisfiability Solver Selector (3S). *SAT Challenge 2012 solver descriptions*, 2012.
[4] M. Soos. CryptoMiniSat. *SAT Race solver descriptions*, 2010.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Parallel Static Satisfiability Solver Selector (p3S-stat)

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Brown University, Dept. of Computer Science, Providence, RI 02912, USA
Email: ynm@cs.brown.edu
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

The Parallel Static Satisfiability Solver Selector (p3S-stat) is a parallel portfolio solver that statically selects and schedules various solvers across multiple cores. p3S-stat version 2.1 participated in the Parallel Solvers track of SAT Challenge 2012.

I. Solving techniques

The Parallel Satisfiability Solver Selector (p3S) is a generalization of its sequential version 3S [2], [3]. p3S schedules both sequential and parallel "pure" solvers across multiple compute cores available on one machine. These solvers are determined by solving a corresponding optimization problem modeled as an Integer Program (IP).

The static version of p3S, named p3S-stat, schedules a predetermined set of solvers on all of the available cores. p3S-stat is built upon a variety of conflict directed clause learning SAT solvers (both sequential and parallel solvers), lookahead based solvers, and local search solver.

Similar to 3S, p3S works in two phases, an offline learning phase, and an online execution phase. We refer the reader to the solver description of 3S [3] for details, including a list of the 38 sequential baseline solvers used.

II. Implementation details

Since p3S builds upon the implementation of 3S, many of the implementation specific details remain the same as in 3S [3], including the benchmark set and timeout used in the offline training phase of p3S. The launching of (single-core and multi-core) solvers in parallel is managed by extending the Python 2.6 based sequential launcher of 3S, using the Subprocess and Signal packages. In essence, the launcher for p3S creates one schedule for each core and launches one independent copy of 3S to execute that schedule.

In addition to the 38 sequential solvers used by 3S (used both with and without preprocessing, resulting in 76 sequential solver choices), p3S also uses multi-core "pure" solvers Plingeling [1] and CryptoMiniSat [4]. Both of these solvers were run using 1, 2, 3, and 4 cores. Each of these variants, taken with and without preprocessing, counts towards the set of baseline solver for p3S, resulting in a total of 92 baseline solvers.

III. SAT Challenge 2012 specific details

The command line used to launch p3S-stat in SAT Challenge 2012 was:

python p3Sstat-2.1.py –scale 1.3
–tmpdir TEMPDIR INSTANCE

Please refer to the solver description of 3S [3] for an explanation of the scaling parameter.

A pre-determined schedule, generated by solving an IP optimization problem during the offline training phase, was used for all 8 cores. The specifics of the preschedule may be found in the file named preschedules/sch-p3Sstat.txt of the submitted solver. One invariant in this preschedule was that if, say, a 3-core solver is launched by core number k at some point of time T, then it was made sure that at least two other cores were free and available at time T to help execute the 3-core solver launched by core k.

Note that the offline training of p3S, similar to 3S, was done using a 2,000 timeout on the training machines and the 8-core schedule eventually used is optimized for the competition timeout of 900 seconds on the competition machines. Execution with different timeouts or on other machines will likely result in reduced performance.

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 "pure" 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] A. Biere. PLingeling, *SAT Race 2010 solver descriptions*, 2010.
[2] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.

[3] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Satisfiability Solver Selector (3S). *SAT Challenge 2012 solver descriptions*, 2012.
[4] M. Soos. CryptoMiniSat. *SAT Race solver descriptions*, 2010.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

PeneLoPe, a parallel clause-freezer solver

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 new 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 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 [3].

In addition, PeneLoPe includes a recent technique for its learnt clause database management. Roughly, this new approach follows this schema: each learnt clause c is periodically evaluated with a so-called *psm* measure [1], 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 [1] for more details).

Besides the *psm* technique, PeneLoPe also makes use of the *lbd* value defined in [2]. *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 the 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 [5].
- *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* [2]: 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 [2].

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 [4]. 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.

Acknowledgment

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

References

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

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

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

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

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


```
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

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

pfolioUZK: Solver Description

Andreas Wotzlaw*, Alexander van der Grinten*, 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

pfolioUZK is a portfolio SAT solver based on the portfolio SAT solver ppfolio developed by Olivier Roussel [1]. It can be used either as a parallel portfolio SAT solver on multicore systems, or as a sequential portfolio SAT solver. Here, the number of cores that may be used by pfolioUZK can be selected on the command line with the parameter `-c <number of cores>`. Like ppfolio, it is a simple computer program that starts SAT solvers from the available portfolio in parallel, among others an instance of our new complete SAT solver satUZK [2].

Currently, to the portfolio belong the following SAT solvers:

- satUZKs: a version of the complete SAT solver satUZK developed by Alexander van der Grinten and Andreas Wotzlaw, see [2] for a detailed description,
- glucose 2.0: a complete SAT solver by Gilles Audemard and Laurant Simon [3],
- lingeling 587 and plingeling 587: two complete SAT solvers by Armin Biere [4],
- contrasat: a complete SAT solver by Allen van Gelder,
- march_hi 2009: a complete SAT solver by Marijn Heule and Hans Van Maaren,
- TNM 2009: an incomplete SAT solver by Wanxia Wei and Chu Min Li [5],
- MPhaseSAT_M: a complete SAT solver by Jingchao Chen [6], and
- sparrow2011: an incomplete SAT solver developed by Dave Tompkins using the sparrow algorithm of Adrian Balint and Andreas Fröhlich [7].

The solvers have been chosen on the basis of their performance on the SAT Competition 2011. The type and the number of solvers that are started depend on the number of allocated cores and on the uniformity of the input instance. A CNF formula is *uniform* if all its clauses have exactly the same length. In case the input instance is uniform we start parallel only march_hi 2009, TNM 2009, MPhaseSAT_M, and sparrow2011, when possible each on a separate core. For all other instances, we use the following predefined configurations:

- 1 core or `-c 1`: satUZK, lingeling 587, TNM 2009, and MPhaseSAT_M are started on the same core (this configuration constitutes a sequential version of pfolioUZK),
- 2 cores or `-c 2`: satUZK and TNM 2009 on the first core, and glucose 2.0 and MPhaseSAT_M on the second core,
- 4 cores or `-c 4`: satUZK, glucose 2.0, contrasat, and lingeling 587, all on separate cores,
- 8 cores or `-c 8`: satUZK, glucose 2.0, contrasat, and four instances of plingeling 587 are started for CNF formulas with up to 12 millions clauses, all on their own cores. For larger formulas, satUZK is not used due to memory limitations.

For the SAT Challenge 2012 in tracks "Parallel Solvers - Application SAT+UNSAT" and "Sequential Portfolio Solvers" we have submitted both precompiled (with gcc 4.4.3 and -O3) and statically linked binaries (32- and 64-bit) as well as all sources (C/C++ programs and shell scripts). We consider to make the source code available online.

REFERENCES

[1] O. Roussel, "Description of ppfolio," SAT Competition 2011, 2011.
[2] A. van der Grinten, A. Wotzlaw, E. Speckenmeyer, and S. Porschen, "satUZK: Solver description," SAT Challenge 2012, 2012.
[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solver," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, 2009, pp. 399–404.
[4] A. Biere, "Lingeling, plingeling, picoSAT and precoSAT at SAT Race 2010," Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, FMV Reports Series 10/1, August 2010.
[5] W. Wei and C. M. Li, "Switching between two adaptive noise mechanisms in local search for SAT," SAT Competition 2009, 2009.
[6] J. Chen, "Phase selection heuristics for satisfiability solvers," *CoRR*, 2011. [Online]. Available: http://arxiv.org/abs/1106.1372
[7] 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 (SAT'10)*, ser. Lecture Notes in Computer Science, vol. 6175, 2010, pp. 10–15.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Description of ppfolio 2012

Olivier Roussel
CRIL - CNRS UMR 8188
olivier.roussel@cril.univ-artois.fr

ppfolio (Pico Portfolio or also Parallel Portfolio) is a naive parallel portfolio. It is meant to identify a lower bound of what can be achieved either with portfolios, or with parallel solvers.

ppfolio by itself is just a program that starts SAT solvers in parallel. It only involves system programmation and knows nothing about the SAT problem. The number of cores that may be used can be selected on the command line.

ppfolio does not try to be clever in any way. Its role is just to run solvers in parallel. Several obvious improvements are possible (detecting the type of SAT instance and choosing the appropriate solver, improving the scheduling of the solvers, sharing of the formula,...) but were not considered because the goal of this solver is uniquely to identify a lower bound of the performances that can be achieved using a few lines of plain system programmation. It is of course expected that access to main memory will be a bottleneck that will significantly impact each individual solver performances.

The submitted version uses the following solvers

- glueminisat *2.2.5* (Hidetomo NABESHIMA)
- MPhaseSAT_M *2011-02-16* (Jingchao chen)
- sparrow2011 *sparrow2011_ubcsat1.2_2011-03-02* (Adrian Balint, Andreas Froehlich, Dave Tompkins, Holger Hoos)
- CryptoMiniSat *Strange-Night-2-mt* (Mate Soos)
- Plingeling *587f* (Armin Biere)

These solvers have been chosen on the basis of their results on the 2011 competition benchmark, taking into account the reduced timeout of the SAT Challenge. For the sequential portfolio track, one solver per category of benchmarks was selected (the non portfolio solver giving the largest number of answers within 300 seconds). For the parallel track, two different configurations were considered: one where solvers could use up to 2 cores (hence a 900 second WC limit and a 1800 s CPU limit) and another one where solvers could use up to 4 cores (hence a 900 second WC limit and a 3600 s CPU limit). In each configuration, the solver answering on the largest number of instances was initially added to the selection. Whenever another solver was able to solve several instances that the solvers in the current selection didn't solve, it was added to the selection. In the end, 4 solvers were selected in the first configuration, and only 2 in the second one. As the number of instances solved in the two configuration was almost equal, the configuration with only 2 solvers running on 4 cores was chosen.

The solvers that are started in this version of ppfolio only depend on the number of allocated cores :

- *1 core*: glueminisat, MPhaseSAT_M and sparrow2011 are started, but since there is only one core, the solver is essentially sequential.
- *>1 core*: CryptoMiniSat and Plingeling are started, each on one half of the cores

More information on ppfolio can be found on http://www.cril.univ-artois.fr/∼roussel/ppfolio.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Relback: Relevant Backtracking 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

I. Major solving techniques

The following description concerns the two submitted solvers: *relback* and *relback_m*. These two solvers are based on existing implementations of a CDLC like solver. Indeed, *relback* is implemented under the *glucose* solver (without SatElite formula simplification [1]) and *relback_m* is based on *minisat 2.2* .

In *glucose* [2] and *minisat 2.2* [3], when a conflict is reached, during the propagation phase of the enqueued literals, the First Implication Point [4] is used in order to learn a clause and define a backjumping level.

The main purpose of our solvers 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. Parameter description

We give here the use of the new backtracking scheme in the two submitted solvers:

1) *relback*: backtracking according the to nearest decision variable is applied twice at each luby restart achieved by the solver. Moreover, each time the weight of the variables is initialized, the solver authorizes a new (twice) application of our backtracking scheme.
2) *relback_m*: the application of our backtracking scheme is more restrictive. Indeed, it is applied only once at the first conflict reached during each luby restart.

Moreover, we apply the progress saving for polarity variable selection.

III. Special Algorithms, Data Structures and Features

In both solvers, there is no preprocessing step. The data structures are strictly similar to the existing ones in *minisat* with additional ones to deal with our backtracking scheme.

IV. Implementation detail

1) The programming language used is C++
2) The solvers are based on *glucose 2* and *minisat 2.2* with the additional features explained above.

V. SAT Challenge 2012 Specifics

1) The two solvers are submitted in "Solver Testing Track" including : Random SAT, Hard Combinatorial SAT+UNSAT and Application SAT+UNSAT.
2) The used compiler is *g++*
3) The optimization flag used is "-O3". The compilation options are the same as the used existing solvers.

VI. Availability

Our solvers are not yet publicly available.

Acknowledgment

We would like to thank the authors of *minisat 2.2*[1] and *glucose*[2] for making available the source code of their solvers.

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] N. E. en and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003, pp. 502–518.
[4] 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://minisat.se/
[2] Available on http://www.lri.fr/∼simon/

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Solver Description of RISS 2.0 and PRISS 2.0

Norbert Manthey
Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

***Abstract*—The SAT solver RISS 2.0 and its concurrent parallelization PRISS 2.0 are described in the configuration they have been submitted to the SAT Challenge 2012.**

I. The sequential SAT solver RISS 2.0

Based on the CDCL procedure, this solver has been implemented as a module based system. The routines for the decision procedure, the learned clause management, unit propagation, preprocessor and the event heuristics for restart and removal can be exchanged easily. This style of implementation comes to a cost, namely the communication overhead among the components. Whereas plain SAT solver implementations can alter for example the watched list of the unit propagation immediately when a clause should be removed, RISS 2.0 has to store this data first, pass it to the unit propagation module and afterwards this module can execute the wanted operation. Based on this overhead, the implementation is a trade-off between providing as many features as possible and having a good performance on application instances. To still achieve a high performance, RISS 2.0 is equipped with a strong preprocessor COPROCESSOR 2.1 [13], which is also be used during search to simplify the formula and the set of learned clauses.[1]

A. Features of RISS 2.0

The main goal if RISS 2.0 is to solve formulas in CNF. Furthermore, the solver is used as research platform and thus provides many parameters to enable further techniques. These techniques are not present in general SAT solvers:

- Enumeration of all solutions of the input formula
- Loading and storing learned clauses of a run
- Searching for a solution with a set of assumed literals
- Passing an initial model to the solver that should be tested first

Additionally to the named features, RISS 2.0 implements many deduction techniques on top of CDCL that can be enabled. Among them there are *On-the-fly Self-Subsumption* (OTFSS) [7], *Lazy Hyper-Binary-Resolution*(LHBR) [3] and *Dominator Analysis* [6]. To speed up search, most of the techniques that are available in COPROCESSOR 2.1 can be used for simplifying the formula during search. The implementation and handling of data structures and memory accesses is based on the insights that have been published in [10]. The solver furthermore uses *Blocking Literals* introduced in [16] and *Implicit Binary Clauses*(e.g. [15]) to speed up unit propagation and conflict analysis.

The submitted configuration uses the Luby series with a factor 32 as a restart strategy and a geometric series starting with 3000 and an increment factor of 1.1 as removal schedule. The removal is mainly based on the LBD measure [2], but also short clauses are kept. Both OTFSS and LHBR are enabled.

We started to implemented RISS from scratch in 2009 as a teaching system in C++. The binary of the tool we provided for the SAT Challenge has been compiled with the GNU compiler and the optimization -O3. Although plenty of parameters are implemented in both RISS 2.0 and its preprocessor automated parameter setting has not been done yet. This is considered the next step, because parameter setup is not considered to be trivial but has high potential to improve the solvers performance.

B. Features of Coprocessor

The internal preprocessor of RISS 2.0 implements many simplification techniques, that are executed in the specified order. Whenever a technique can reduce the formula, the process is started from the top.

1) Unit propagation
2) Pure literal detection
3) Self-subsuming resolution
4) Equivalence elimination [5]
5) Unhiding [9]
6) Hidden tautology elimination [8]
7) Blocked clause elimination [11]
8) Variable elimination [4]
9) An algorithm based on extended resolution
10) Failed literal probing [12]
11) Clause vivification [14]

Equivalent literal detection is done based on binary clauses and on output literals of gates in the formula. The algorithm based on extended resolution to simplify the formula is unpublished, but submitted for publication. Each technique can be limited so that the consumed run time remains reasonable. After preprocessing, COPROCESSOR allows to shrink the formula so that all assigned or eliminated variables are removed and the resulting formula contains consecutive variables again.

II. The parallel SAT solver PRISS 2.0

The SAT solver PRISS 2.0 is a portfolio SAT solver based on RISS 2.0 and supports up to 64 parallel solver incarnations. After using COPROCESSOR on the input formula, n incarnations

[1]Both the solver and its preprocessor as well as descriptions are available at tools.computational-logic.org.

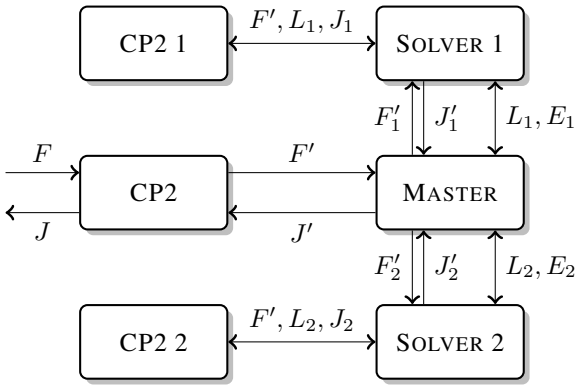


Fig. 1. Components in the PRISS 2.0 framework

of RISS are started concurrently, where each of the incarnations uses its own preprocessor to simplify the formula during search. Learned clauses are shared among the incarnations. The exchange is filtered both on the sender and the receivers side. Submitting clauses is based on the length of the clause and its activity. Whenever the length of a candidate clause is shorter than the average length since the last restart, the clause is a candidate to be submitted to the shared storage. Another criterion is the activity based on the LBD. The reception of clauses from the storage is based on the same criteria again. Furthermore, the PSM [1] is used to reject not useful clauses. In addition to clauses, the RISS incarnations share informations about equivalent literals, which are found during search by the simplification methods. Since the simplification might also add or remove variables from the formula of a certain thread, only information about common variables is shared – a clause that contains an eliminated variable will be rejected by the receiving thread. Based on the current portfolio implementation, this problem cannot be fixed easily. For the future it is wanted to integrate the common preprocessor also as common simplifier, so that all clauses can be shared again.

Figure 1 shows the a pictogram of the components and their communication. After the input formula F is processed by the preprocessor, each solver incarnations is started in a thread with a physical copy of the formula (F'_1 and F'_2). For inprocessing each solver has its private preprocessor. Learned clauses and equivalent literals are shared with the master (e.g. L_1 and E_1). When a solver finds a solution, its preprocessor reconstructs eliminated variables, equivalent variables and literals from blocked clauses. The processed model is passed back to the master, which stops all other solver incarnation and also reconstructs the final assignment.

The submitted configuration of the solver uses only 5 cores out of the 8 available cores. Each incarnation has a slightly different configuration. The first incarnation uses the default configuration. The next solver uses *permuted trail* restarts [17]. The third incarnation keeps 50 % of its learned clause data based instead of 25 %. The fourth solver uses the PSM value for removing clauses and bumps variables twice, if they are used during conflict analysis, are assigned at the conflict level and if the activity of their reason clause is comparably high. Finally, the fifth configuration exchanges the VSIDS heuristic by the VMTF heuristic for variable activities. If more cores should be used, the next configuration alters the implementation of the unit propagation by preferring satisfied literals in clauses to be watched. All further configurations are similar to the default configuration except the fact that one percent of their decisions is done randomly to not result in the same search.

REFERENCES

[1] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 188–200, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[3] Armin Biere. Lazy hyper binary resolution. In *Algorithms and Applications for Next Generation SAT Solvers*, number 09461, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

[4] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT'05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[5] Allen Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Ann. Math. Artif. Intell.*, 43(1):239–253, 2005.

[6] HyoJung Han, HoonSang Jin, and Fabio Somenzi. Clause simplification through dominator analysis. In *DATE*, pages 143–148. IEEE, 2011.

[7] Hyojung Han and Fabio Somenzi. On-the-fly clause improvement. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 209–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause Elimination Procedures for CNF Formulas. In Christian Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *LNCS*, pages 357–371. Springer, 2010.

[9] Marijn Heule, Matti Järvisalo, and Armin Biere. Efficient CNF Simplification based on Binary Implication Graphs. In K.A. Sakallah and L. Simon, editors, *SAT 2011*, volume 6695 of *LNCS*, page 201–215. Springer, 2011.

[10] Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware sat solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 519–534. Springer, 2010.

[11] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked Clause Elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 129–144. Springer, 2010.

[12] Inês Lynce and João Marques-Silva. Probing-Based Preprocessing Techniques for Propositional Satisfiability. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '03, pages 105–. IEEE Computer Society, 2003.

[13] Norbert Manthey. Coprocessor 2.0 – A flexible CNF Simplifier (Tool Presentation), 2012. Submitted to SAT 2012.

[14] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs. Vivifying propositional clausal formulae. In *18th European Conference on Artificial Intelligence(ECAI'08)*, pages 525–529, Patras (Greece), jul 2008.

[15] Mate Soos. Cryptominisat 2.5.0. In *SAT Race competitive event booklet*, July 2010.

[16] Niklas Sörensson and Niklas Eén. MiniSat 2.1 and MiniSat++ 1.0 — SAT Race 2008 Editions. Technical report, 2008.

[17] Peter van der Tak, Antonio Ramos, and Marijn J.H. Heule. Reusing the assignment trail in cdcl solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:133–138, 2011. system description.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Satisfiability Solver Selector (3S)

Yuri Malitsky*, Ashish Sabharwal†, Horst Samulowitz†, and Meinolf Sellmann†
*Brown University, Dept. of Computer Science, Providence, RI 02912, USA
Email: ynm@cs.brown.edu
†IBM Watson Research Center, Yorktown Heights, NY 10598, USA
Email: {ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

The Satisfiability Solver Selector (3S) is a portfolio solver that dynamically selects and schedules various baseline solvers depending on the input instance. 3S version 2.1 participated in the Sequential Portfolio Solvers track of SAT Challenge 2012, and is built upon a variety of conflict directed clause learning SAT solvers, lookahead based solvers, and local search solvers.

I. Solving techniques

3S works in two phases, an offline learning phase, and an online execution phase.

At Runtime: In the execution phase, as all dynamic solver portfolios, 3S first computes features of the given problem instance. In particular, 3S uses a subset of the 48 base features introduced by Xu et al. [4]. Then, 3S selects $k \in \mathbb{N}$ instances that are most "similar" to the given instance in a training set of SAT instances for which 3S knows all runtimes of all its constituent solvers. Similarity in 3S is determined by the Euclidean distance of the (normalized) feature vectors of the given instance and the training instances. 3S selects the solver that can solve most of these k instances within the given time limit (ties are broken by shorter runtime). Finally, 3S 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. [2] for details).

Offline: In the learning phase, which takes place during the development of the portfolio solver, 3S considers three tasks:

1) Computation of features and simulation of solvers on all instances to determine their runtime on all training instances.
2) Computation of a desirable size k of the local neighborhood of a given instance. To this end, 3S employs a cross validation by random subsampling. That is, 3S repeatedly splits the training set into a base and a validation set and determines which size of k results in the best average validation set performance when using only the base set training instances to determine the long running solver.
3) Lastly, 3S computes the fixed schedule of solvers that are run for roughly 10% of the competition runtime. The objective when producing this schedule is to maximize the number of instances that can be solved within this reduced time limit. Among schedules that can solve the same maximum number of instances, 3S selects one that minimizes the runtime of the schedule and then scales this shorter schedule back to the 10% time limit by increasing the runtime of each solver in the schedule proportionally

For more detailed information on the internals of 3S, please refer to [2].

II. Implementation details

The main launcher script of 3S is written in Python 2.6. This script orchestrates launching of solvers and preprocessors, forwarding of clauses (if turned on), conversion of solutions of the simplified formulae back to the solutions of the original formulas, etc. The solver selector/scheduler program, called 'coach', is written in C++ and compiled with GNU g++ 4.4.5 with options "-O3 -fexpensive-optimizations -static" for the x86-64 architecture. The preprocessor SatELite [1] 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 3S were themselves written mainly in C/C++, and are listed below.

Baseline Solvers: The portfolio 3S is composed of the following 38 baseline solvers (proper references omitted due to lack of space):

1) adaptg2wsat2009
2) adaptg2wsat2009++
3) clasp (ver: 1.3.6-x86-linux)
4) CryptoMinisat (ver: 2.9.0)
5) EagleUP (ver: 1.565.350)
6) ebminisat_static (ver: SAT Comp 2011)
7) Glucose_static (ver: 2.0)
8) gnovelty+2 (ver: 2.0)
9) gnovelty+2-H (ver: 2.0)
10) hybridGM3
11) kcnfs04SAT07
12) lingeling (ver: 587f)
13) LySATc
14) LySATi (ver: 0.1)
15) march_dl2004
16) march_hi
17) march_nn
18) minisat (ver: 2.2.0)
19) minisat20SAT07 (ver: 2.0 from SAT Comp 2007)
20) mxc-sat09 (ver: SAT Comp 2009)
21) mxc-sr08 (ver: SAT Race 2010)
22) picomus

23) picosat (ver: 846)
24) picosat (ver: 936)
25) precosat (ver: 570)
26) SatELite (ver: 2005, modified to not map variables)
27) TNM
28) tts-4-0
29) vallst (ver: 0.9.258)
30) zchaff_rand (ver: SAT 2005)
31) zchaff07
32) saps (ver: 1.2.0b)
33) SATensteinAshiFact (ver: 1.1.0tt2)
34) SATensteinAshiCbmc (ver: 1.1.0tt2)
35) SATensteinAshiR3fix (ver: 1.1.0tt2)
36) SATensteinAshiHgen (ver: 1.1.0tt2)
37) SATensteinAshiSwgcp (ver: 1.1.0tt2)
38) SATensteinAshiQcp (ver: 1.1.0tt2)

All training instances are preprocessed with SatELite, version 1.0, with default option "+pre" [1], which results in a total of 76 solvers in our portfolio.

Training Instances: We selected $6,667$ instances from all SAT Competitions and Races during 2002 and 2010 [3], whereby we discarded all instances that cannot be solved by any of the aforementioned solvers within a time limit of $2,000$ seconds (on the hardware used for training).

III. SAT Challenge 2012 specific details

The command line used to launch 3S in SAT Challenge 2012 was:

python 3S-2.1.py –scale 1.3 –tmpdir TEMPDIR INSTANCE

The scaling parameter was used to adjust for the difference in the speed of the Challenge machines and the machines on which 3S was developed. When used on other machines, this scaling factor would need to be adjusted.

Fixed Preschedule: As mentioned earlier, 3S runs a fixed preschedule of solvers for roughly 10% of the total runtime. The specific preschedule used in SAT Challenge 2012 was: EagleUP (14 sec), march_hi (22 sec), picosat936 (2 sec), precosat570 (31 sec), TNM (3 sec), tts-4-0 (2 sec), AshiFact (2 sec), AshiR3fix (2 sec), AshiHgen (2 sec), AshiQcp (2 sec), and gnovelty+2-H (2 sec).

Note that this pre-schedule is optimized for the competition timeout of 900 seconds on the competition machines. Execution with different timeouts or on other machines will likely result in reduced performance.

After these solvers have run, the 'coach' selector schedules a single long-running solver (in rare cases two such solvers) for the remainder of the runtime based on the dynamically computed features of the given instance.

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 "pure" 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] N. Een, A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. *SAT*, pp. 61-75, 2005.
[2] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann. Algorithm Selection and Scheduling. *CP*, pp. 454-469, 2011.
[3] SAT Competition 2011. *http://www.satcomptition.org*.
[4] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Sat4j 2.3.2-SNAPSHOT SAT solver

Daniel Le Berre
Université Lille - Nord de France, CRIL-CNRS UMR 8188
Université d'Artois, Lens,France
Email: leberre@cril.fr

I. About the Sat4j library

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. The Sat4j library started in 2004 as an implementation in Java of the Minisat specification[1]. It has been developed since then with the spirit to allow testing various combinations of features developed in new SAT solvers while keeping the technology easily accessible to a newcomer. For instance, it allows the Java programmer to express constraints on objects and hides all the mapping to the various research community input formats from the user. 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. 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.

II. About the solver submitted to the challenge

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. Here are the settings used in the solver submitted to the SAT challenge 2012.

The *rapid restarts* strategy used is the in/out one proposed by Armin Biere in Picosat[3]. We have been using that default setting in Sat4j since 2007 (Sat4j 1.7).

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]. We have been using that default setting in Sat4j since 2007 (Sat4j 1.7) too. However, a bug has been introduced in Sat4j 2.0 while refactoring the implementation of that feature, that ended up forcing the solver to always branch first on negative literals. We discovered that bug only a few months ago, because it does not impact the correctness of the solver. Sat4j 2.3.2 will be the first release of Sat4j 2.x that fixes such bug.

Finally, the solver keeps derived clause with literals from few different decision levels as proposed in 2009 award winner Glucose [6] using the settings "start cleanup at 5000 conflicts and increase that bound by 1000 conflicts when reached" provided by Armin Biere.

Note that unlike most other SAT solvers, Sat4j does not use any preprocessor, because none is currently available in the library.

III. Conclusion

In 2011, a fully equivalent Java implementation of Minisat 2.2 by Carsten Sinz allowed us to compute that a SAT solver in Java is roughly 3.25 times slower than its counterpart in C/C++ on the SAT 2009 competition application benchmarks. Furthermore, Sat4j works on generic constraints, not only clauses: for that reason, the library makes an heavy use of polymorphism (late binding). As such, we do not expect Sat4j to compete with the best solvers. We submitted Sat4j to the SAT challenge both to allow Sat4j users to have an idea of the efficiency of Sat4j compared to the state-of-the-art and to allow us to study the behavior of our solver on new hardware and benchmarks.

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 *Proc. of SAT'03*, 2003, pp. 502–518. [Online]. Available: http://www.math.chalmers.se/%7Eeen/Satzoo/An_Extensible_SAT-solver.ps.gz
[2] D. Le Berre and A. Parrain, "The sat4j library, release 2.2 system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010.
[3] A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.
[4] N. Sörensson and A. Biere, "Minimizing learned clauses," in *Proc. of SAT'09*, 2009, pp. 237–243.
[5] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Proc. of SAT'07*, 2007, pp. 294–299.
[6] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solver," in *Proc. of IJCAI'09*, jul 2009, pp. 399–404.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Description of Sattime2012

Chu Min LI & Yu LI
MIS, Université de Picardie Jules Verne, France, {chu-min.li, yu.li}@u-picardie.fr

***Abstract*—This document describes the SAT solver "Sattime2012", 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. Major solving techniques in Sattime2012

Sattime2012 is a SLS solver using a preprocessing to simplify the input formula by propagating all unit clauses and detecting all failed literals in the input formula.

II. Special Algorithms, Data Structures and Features

Sattime2012 is the new version of Sattime [1]. It is based on Novelty+ [2], while Sattime is based on Novelty++ [3]. In addition, we have made some code optimization in Sattime2012. Given a SAT instance ϕ to solve, it 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) Otherwise, randomly pick an unsatisfied clause c;
3) 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.

The score of a variable is the decrease of the number of unsatisfied clauses in ϕ if the variable is flipped. A satisfying variable of a clause is the variable whose flipping made the clause from unsatisfied to satisfied. The promising decreasing variable was defined in [3]. Probability p is adapted according to the improvement in the number of unsatisfied clauses during search according to [4], and wp=p/10.

In Novelty [5], Novelty+ and Novelty++, when the best variable is not the most recently flipped in c, it is flipped. Otherwise the second best variable in c is flipped with probability p and the best variable is flipped with probability 1-p. Since c is unsatisfied, the most recently flipped variable in c is necessarily the last falsifying variable whose flipping made c from satisfied to unsatisfied. Sattime2012 is different from Novelty, Novelty+ and Novelty++ in that it considers the last satisfying variable instead of the last falsifying one in c. Note that the last satisfying variable in c is not necessarily the most recently flipped in c. The intuition of Sattime2012 is to avoid repeatedly satisfying c using the same variable.

III. Parameter description

There are two parameters used in Hoo's adaptive noise mechanism in Sattime2012, Φ and Θ, used to specify the noise variation in each noise adapting and the maximum length of a period in which the noise is not adapted. In Sattime2012 submitted for the challenge, Φ=10 and Θ=5. The two parameters do not depend on instance properties. The performance of Sattime2012 is not very sensitive to the variation in the value of these parameters.

Other parameters include: -cutoff a, -tries b, -seed c, allowing to run b times Sattime2012 for at most a steps each time, the random seed of the first run being c.

IV. Implementation detail

Sattime2012 is implemented in C and is based on g2wsat [3].

V. SAT Challenge 2012 Specifics

Sattime2012 is submitted to three tracks in the challenge: satisfiable random SAT, SAT+UNSAT hard combinatorial and SAT+UNSAT industrial. Because of the preprocessing, Sattime2012 may prove the unsatisfiability of an instance.

Two binaries of Sattime2012 were submitted to the challenge.

`sattime2012` is a 64 bit binary obtained using

```
gcc -O3 -DNDEBUG -fno-strict-aliasing -m64
sattime2012.c -o sattime2012
```

`sattime2012b32` is a 32 bit binary obtained using

```
gcc -O3 -DNDEBUG -fno-strict-aliasing -m32
sattime2012.c -o sattime2012b32
```

Sattime2012 should be called in the challenge using

sattime2012 input-instance -seed RANDOMSEED

In the challenge, Sattime2012 will run until a contradiction is found (by the preprocessing), or a solution is found, or the cutoff time is reached. The code source of Sattime2012 is not yet available, but it will be.

References

[1] C. M. LI and Y. LI, "Satisfying versus falsifying in local search for satisfiability," in *Proceedings of SAT-2012, to appear*. Springer, 2012.
[2] H. Hoos, "On the run-time behavior of stochastic local search algorithms for sat," in *Proceedings of AAAI-99*, 1999, pp. 661–666.
[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] H. Hoos, "An adaptive noise mechanism for walksat," in *Proceedings of AAAI-02*. AAAI Press / The MIT Press, 2002, pp. 655–660.
[5] D. McAllester, B. Selman, and H. Kautz, "Evidence for invariant in local search," in *Proceedings of AAAI-97*, 1997, pp. 321–326.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

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

I. 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.

A. 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. 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. The available time must be specified to the solver with a command-line parameter `-budget <time in sec>`.

By default, the preprocessing phase is disabled and can be activated with a command-line parameter `-preproc-adaptive`.

B. 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.

For learned clause deletion the solver can use a MiniSAT-like learned clause deletion strategy or a more aggressive literal blocks distance based deletion strategy [7]. The first strategy is the default one, whereas the latter one can be enabled with command-line parameters `-clause-red-agile -restart-glucose -learn-minimize-glucose -learn-bump-glue-twice`.

Both Luby restarts and glucose-like dynamic restarts are implemented [7].

C. 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.

II. SAT Challenge 2012 Specifics

For the challenge in tracks "Hard Combinatorial SAT+UNSAT" and "Application SAT+UNSAT" we have submitted three parametrized versions of our solver, started with the following commands:

- satUZK: `satUZK -budget 900 -preproc-adaptive -show-model <cnf instance>`
- satUZKg: `satUZK -budget 900 -show-model -preproc-adaptive -clause-red-agile -restart-glucose -learn-minimize-glucose -learn-bump-glue-twice <cnf instance>`
- satUZKs: `satUZK_wrapper satUZK -budget 900 -preproc-adaptive <cnf instance>`

The last solver uses first SatELite [2] for preprocessing of the input instance before the satUZK solver is called.

All three solvers have been submitted both as precompiled (with gcc 4.4.3 and -O3) and statically linked 64-bit binaries as well as sources written in C++.

References

[1] 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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Parallel SAT Solver SatX10-EbMiMiGlCiCo 1.0

Bard Bloom, David Grove, Benjamin Herta, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat
IBM Watson Research Center, NY, USA
Email: {bardb,groved,bherta,ashish.sabharwal,samulowitz,vsaraswa}@us.ibm.com

`SatX10-EbMiMiGlCiCo` is the first instantiation of a parallel SAT solver built using the `SatX10` framework [2]. The `SatX10` framework is based on the `X10` programming language designed specifically for programming on multi-core and clustered systems easily [8]. The framework provides various facilities to conveniently run algorithms (here SAT solvers) in parallel, along with a communication infrastructure.

Version 1.0 of this solver participated in the Parallel Solvers track of SAT Challenge 2012.

I. Solving techniques

`SatX10-EbMiMiGlCiCo`, as its long name suggests, is composed of 6 distinct `MiniSat`-based conflict directed clause learning SAT solvers:

1) `EB MiniSat` [5]
2) `Circuit MiniSat` [3]
3) `Contra MiniSat` [4]
4) `Glucose 2.0` [1]
5) `Minisat 2.0` [7]
6) `Minisat 2.2.0` [6]

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`.

As a large number of solvers running in parallel can sometimes take up prohibitive amounts of memory, the number of solvers launched is decided following a simple static rule based on the number of clauses of the input formula and can vary anywhere from 1 to 8.

II. Implementation details

The solver `SatX10-EbMiMiGlCiCo` is built using the `SatX10` framework [2], heavily utilizing the mechanisms it provides for incorporating new solvers and sharing information amongst solvers. The version of the `X10` language used was 2.2.2 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++ 4.4.5 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 Challenge 2012 specific details

The parallel solver is launched in SAT Challenge 2012 using a Bash wrapper script called runSatX10.sh. The script decides whether to launch 8, 7, 6, 5, 4, 3, 2, or 1 solver(s) based on whether the CNF header in the formula reports at most 8M, 12M, 16M, 20M, 24M, 29M, 33M, or more clauses, resp. All solvers are launched with the parameter `-verb=0`. Solvers `Glucose 2.0` and `EB MiniSat` are also deployed with a second configuration for each, namely, `-phase-saving=1 -no-luby` and `-rcheck`, resp. Clauses of maximum length 10 are shared with other solvers.

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] 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.
[3] Circuit Minisat Solver Description. SAT Competition 2011.
[4] Contrasat Solver Description. SAT Competition 2011.
[5] EBMinisat Solver Description. SAT Competition 2011.
[6] N. Een and N. Sorensson An Extensible SAT-solver [ver 2.0] *http://www.minisat.se*.
[7] N. Een and N. Sorensson An Extensible SAT-solver [ver 2.2.0] *http://www.minisat.se*.
[8] 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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

SATzilla2012: Improved Algorithm Selection Based on Cost-sensitive Classification Models

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

1 Introduction

Empirical studies often observe that the performance of different algorithms across problem instance distributions can be quite uncorrelated. When this occurs, there is an incentive to investigate the use of portfolio-based approaches that draw on the strengths of multiple algorithms. `SATzilla` is such a portfolio-based approach for SAT; it was first deployed in the 2003 and 2004 SAT competitions [5], and later versions won a number of prizes in the 2007 and 2009 SAT competitions [10, 8, 11], including gold medals in the random, crafted and application categories in 2009.

Different from previous versions of `SATzilla`, which utilized *empirical hardness models* [4, 6] for estimating each candidate algorithm's performance on a given SAT instance, `SATzilla2012` is based on cost-sensitive classification models [7]. We also introduced a new procedure that generates a stand-alone `SATzilla` executable based on models learned within Matlab. Finally, we used new component algorithms and training instances.

Overall, `SATzilla2012` makes use of the same methodology as described in [9].

Offline, as part of algorithm development:

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

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

1. Predict whether the feature computation time is above 90 CPU seconds. If the feature computation is too costly, run the backup solver identified in Step 6 above; otherwise continue with the following steps.
2. Run the presolvers in the predetermined order for up to their predetermined fixed cutoff times.
3. Compute feature values. If feature computation cannot be completed due to an error, select the backup solver identified in Step 6 above; otherwise continue with the following steps.
4. For every pair of solvers, predict which solver performs better using the DF trained in Step 8 above, and cast a vote for it.
5. Run the solver that received the highest number of votes. If a solver fails to complete its run (e.g., it crashes), run the solver with the second-highest number of votes. If that solver also fails, run the backup solver.

2 SATzilla2012 vs SATzilla2009

SATzilla2012 implements a number of improvements over SATzilla2009.

New algorithm selector. Our new selection procedure uses an explicit cost-sensitive loss function—punishing misclassifications in direct proportion to their impact on portfolio performance—without predicting runtime. We introduced this approach in [12, 9]. To the best of our knowledge, this is the first time this approach is applied to algorithm selection: all other existing classification approaches use a simple 0–1 loss function that penalizes all misclassifications equally, whereas previous versions of SATzilla used regression-based runtime predictions. We construct cost-sensitive DFs as collections of 99 cost-sensitive decision trees [7], following standard random forest methodology [2].

New `SATzilla` executable. Our SATzilla version used in [9] was based on classification models built in Matlab, and its execution required the installation of the free Matlab runtime environment (MRE). In order to avoid the need for installing MRE, we now converted our Matlab-built models to Java and provide Java code to make predictions using them. Thus, running SATzilla2012 now only requires the scripting language Ruby (which is used for running the SATzilla pipeline).

New component algorithms and training instances. We updated the component solvers used in SATzilla2009 with the 31 newest publicly-available SAT solvers. These include 28 solvers from [9], the two versions of Spear optimized for software and hardware verification in [3], and MXC 0.99 [1] (the list of solvers can also be found in the execution script of SATzilla2012).

Our training set is based on a collection of SAT instances that includes all instances from all three SAT competitions and three SAT Races since 2006: 1362 instances for Random SAT, 767 instances for Crafted SAT+UNSAT, and 1167 instances for Application SAT+UNSAT. We droppped instances that could not be solved by any of our 31 solvers within 900 CPU seconds. For training a general version of SATzilla2012 that works well across categories, we used 1614 instances: 538 randomly sampled instances from each of Crafted, Application, and Random (SAT+UNSAT).

3 Running SATzilla2012

We submit a package containing one main executable for SATzilla2012 that can be customized for each of the four categories in the 2012 SAT challenge by an input parameter. The callstring for SATzilla2012 is: `ruby SATzilla12.rb <type> <cnf file>`, where `<type>` should be chosen as INDU for target category Application SAT+UNSAT, HAND for Hard Combinatorial SAT+UNSAT, RAND for Random SAT, and ALL for the Special Track for Sequential Portfolio Solvers. The source code of SATzilla2012 is available online at http://www.cs.ubc.ca/labs/beta/Projects/SATzilla. In order to run properly, subdirectory bin should contain all binaries for SATzilla's component solvers and its feature computation; subdirectory models should contain all models for algorithm selection and predicting the cost of feature computation. We note that SATzilla2012 has an optional 3rd input parameter `<seed>` that will be forwarded to any randomized component solver it runs; by default, that seed is set to 1234.

References

[1] D. R. Bregman. The SAT solver MXC, version 0.99. Solver description, SAT competition 2009, 2009.

[2] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting Verification by Automatic Tuning of Decision Procedures. In *Proc. of FMCAD'07*, pages 27–34, 2007.

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

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

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

[7] K. M. Ting. An instance-weighting method to induce cost-sensitive trees. *IEEE Trans. Knowl. Data Eng.*, 14(3):659–665, 2002.

[8] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla2009: an Automatic Algorithm Portfolio for SAT. Solver description, SAT competition 2009, 2009.

[9] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Evaluating component solver contributions to algorithm selectors. In *Proc. of SAT 2012*, 2012. Under review.

[10] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla2007: a new & improved algorithm portfolio for SAT. Solver description, SAT competition 2007, 2004.

[11] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, June 2008.

[12] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

SimpSat 1.0 for SAT Challenge 2012

Cheng-Shen Han and Jie-Hong R. Jiang
Department of Electrical Engineering / Graduate Institute of Electronics Engineering
National Taiwan University, Taipei 10617, Taiwan
{hankf4@gmail.com, jhjiang@cc.ee.ntu.edu.tw}

I. Introduction

Recent research on Boolean satisfiability (SAT) reveals modern solvers' inability to handle formulae in the abundance of parity (XOR) constraints. To overcome this limitation, SimpSat [1] integrates SAT solving tightly with Gaussian elimination in the style of Dnatzing's simplex method. SimpSat aims to achieve fast and complete detection of XOR-inferred implications and conflicts. It was implemented in the C++ language based on CryptoMiniSat 2.9.2 [2], [3], which is a successful CDCL SAT solver equipped with state-of-the-art solving techniques. A C++ compiler and GNU make are applied to build and compile the codes for 64-bit machines.

II. Satisfiability Solving under XOR Constraints

Similar to other modern SAT solvers, SimpSat adopts the conflict-driven clause learning (CDCL) mechanism. Figure 1 sketches the pseudo code, where lines 2 and 13-16 are inserted for special XOR-handling. SimpSat extracts XOR-clauses from a set of regular clauses, similar to CryptoMiniSat. A set of m XOR-clauses over n variables $\vec{x} = \{x_1, \ldots, x_n\}$ can be considered as a system of m linear equations over n unknowns. Hence the XOR-constraints can be represented in a matrix form as $A\vec{x} = \vec{b}$, where A is an $m \times n$ matrix and $\vec{b}$ is an $m \times 1$ constant vector of values in $\{0, 1\}$.

In line 2, XOR-clauses are extracted from the input formula ϕ. Let $A\vec{x} = \vec{b}$ be a system of linearly independent equations derived from these XOR-clauses. Then $M = [A|\vec{b}]$. If M is empty, lines 13-16 take no effect and the pseudo code works same as the standard CDCL procedure. On the other hand, when M contains a non-empty set of linear equations, the procedure *Xorplex* in line 13 deduces implications or conflicts whenever exist from M with respect to a given variable assignment α. In the process, matrix M may be changed along the computation. When implication (or propagation) happens, α is expanded to include newly implied variables. If any implication or conflict results from *Xorplex*, in line 15 essential information is added to ϕ in the form of learnt clauses, which not only reduces search space but also facilitates future conflict analysis.

III. XOR Reasoning

The efficacy of XOR-handling in the pseudo code of Figure 1 is mainly determined by the procedure *Xorplex*. In essence, two factors, deductive power and computational efficiency, need to be considered in realizing *Xorplex*.

```
SimpSat
  input: Boolean formula φ
  output: SAT or UNSAT
  begin
01  α := ∅;
02  M := ObtainXorMatrix(φ);
03  repeat
04    (status, α) := PropagateUnitImplication(φ, α);
05    if status = conflict
06      if conflict at top decision level
07        return UNSAT;
08      φ := AnalyzeConflict&AddLearntClause(φ, α);
09      α := Backtrack(φ, α);
10    else
11      if all variables assigned
12        return SAT;
13      (status, α) := Xorplex(M, α);
14      if status = propagation or conflict
15        φ := AddXorImplicationConflictClause(φ, M, α);
16        continue;
17      α := Decide(φ, α);
  end
```

Fig. 1. Algorithm: SimpSat

The following example motivates SimpSat's adoption of Gauss-Jordan elimination, rather than Gaussian elimination as used in CryptoMiniSat.

Example 1: Consider the following matrix triangularized by Gaussian elimination.

$$[A|\vec{b}] = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

No implication can be deduced from it. With Gauss-Jordan elimination, however, it is reduced to the following diagonal matrix.

$$[A'|\vec{b}'] = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

The values of the first three variables can be determined from the four equations. Therefore Gaussian elimination is strictly weaker than Gauss-Jordan elimination in detecting implications and conflicts.

We extend the two-literal watching scheme in unit propagation to incremental Gauss-Jordan elimination in a way similar

to the simplex method to support lazy update. Consequently, *Xorplex* can be implemented efficiently and has complete power deducing implications and conflicts whenever exist.

In the simplex method, the variables of the linear equations $A\vec{x} = \vec{b}$ are partitioned into m *basic variables* and $(n-m)$ *nonbasic variables* assuming that the $m \times (n+1)$ matrix $[A|\vec{b}]$ is of full rank and $m < n$. Matrix $[A|\vec{b}]$ is diagonalized to $[I|A'|\vec{b'}]$, where I is an $m \times m$ identity matrix and A' is an $m \times (n-m)$ matrix, by Gauss-Jordan elimination such that the m basic and $(n-m)$ nonbasic variables correspond to the columns of I and A', respectively. Note that diagonalizing $[A|\vec{b}]$ to $[I|A'|\vec{b'}]$ may incur column permutation, which is purely for the ease of visualization to make the columns indexed by the basic variables adjacent to form the identity matrix. In practice, such permutation is unnecessary and not performed. By the simplex method, a basic variable and a nonbasic variable may be interchanged in the process of searching for a feasible solution optimal with respect to some linear objective function. The basic variable to become nonbasic is called the *leaving variable*, and the nonbasic variable to become basic is called the *entering variable*. Although the simplex method was proposed for linear optimization over the reals, the matrix operation mechanism works for our considered XOR-constraints, i.e., linear equations over $\mathrm{GF}(2)$.

To equip complete power in deducing implications and conflicts, procedure *Xorplex* of Figure 1 maintains $M|_\alpha$, the induced matrix M of linear equations subject to some truth assignment α on variables, in a reduced row echelon form. Since *Xorplex* is repeatedly applied under various assignments α during SAT solving, Gauss-Jordan elimination needs to be made fast. A two-literal watching scheme is proposed to make incremental update on M in a lazy fashion, thus avoiding wasteful computation. Essentially, the following invariant is maintained for M at all time.

Invariant:

> For each row r of $M = [A|\vec{b}]$, two non-assigned variables are watched. Particularly, the first watched variable (denoted $w_1(r)$) must be a basic variable and the second watched variable (denoted $w_2(r)$) must be a nonbasic variable.

When the two watched variables of some row in M are non-assigned, no action needs to be taken on this row for Gauss-Jordan elimination. On the other hand, actions need to be taken to maintain the invariant for the following two cases. Firstly, when variable $w_2(r)$ is assigned, another non-assigned nonbasic variable in row r is selected as the new second watched variable. No other rows are affected by this action. Secondly, when $w_1(r)$ is assigned and thus becomes the leaving variable, a non-assigned nonbasic variable in row r needs to be selected as the entering variable. The column c of the entering variable then undergoes the *pivot operation*, which performs row operations (additions) forcing all entries of c to be 0 except for the only 1-entry appearing at row r. Note that the pivot operation may possibly cause the vanishing of variable $w_2(r')$ from another row r'. In this circumstance a new non-assigned nonbasic variable needs to be selected for the second watched variable in row r', similar to the required action when $w_2(r')$ is assigned.

When the invariant can no longer be maintained on some row r of M under α, either of the following two cases happens. Firstly, all variables of r are assigned. In this case the linear equation of r is either satisfied or unsatisfied. For the former, no further action needs to be applied on r; for the latter, *Xorplex* returns the detected conflict. Secondly, only variable $w_1(r)$ (respectively variable $w_2(r)$) is non-assigned. In this case, the value of $w_1(r)$ (respectively $w_2(r)$) is implied. Accordingly, α is expanded with $w_1(r)$ (respectively $w_2(r)$) assigned to its implied value.

Upon termination, procedure *Xorplex* leads to one of the four results: 1) propagation, 2) conflict, 3) satisfaction, and 4) indetermination. Only the first two cases yield useful information for CDCL SAT solving. The information is provided by procedure *AddXorImplicationConflictClause* in line 15 of the pseudo code in Figure 1. In the propagation case, the corresponding rows in M that implications occur are converted to learnt clauses. In the conflict case, the conflicting row in M is converted to a learnt clause.

IV. Conclusions

Boolean satisfiability solving integrated with Gauss-Jordan elimination can be powerful in solving hard real-world instances involving XOR-constraints. With two-variable watching and simplex-style matrix update, Gauss-Jordan elimination can be made fast for complete detection of XOR-inferred implications and conflicts.

References

[1] C.-S. Han and J.-H. R. Jiang. When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way. To appear in *Proc. Int'l Conf. on Computer Aided Verification* (CAV), 2012.

[2] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing* (SAT), 2009.

[3] M. Soos. Enhanced Gaussian elimination in DPLL-based SAT solvers. In *Proc. Pragmatics of SAT*, 2010.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

SINN

Takeru Yasumoto
Kyushu University, Japan
yasumoto.kyushu@gmail.com

I. Introduction

SINN is based on MiniSat2.2[1]. The SINN 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 Restart and freqRST 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. SINN has two phases called Luby Phase and freqRST Phase. Luby Phase uses Luby restart as its restart strategy and RHPolicy for determining the number of learnt clauses that will be deleted. freqRST Phase uses a frequent restart strategy and RQPolicy as a method to delete learnt clauses.

A. Luby Phase

1) Luby Restart: Luby Restart in SINN is the same as that in MiniSat2.2.

2) RHPolicy: The solver deletes the first half of learnt clauses at deletion time. This policy is based on MiniSat2.2 but SINN will not delete more than half of learnt clauses like MiniSat.

B. freqRST Phase

1) freqRST Restart: freqRST Restart is a static restart strategy. The solver restarts every 50 conflicts.

2) RQPolicy: The solver deletes 3 quarters of learnt clauses at deletion time. This policy is based on GlueMiniSat2.2.5[3].

III. TLBD

True LBD, TLBD for short, is a kind of LBD[2]. TLBD is different from LBD in the manner of updating its value. TLBD ignores literals assigned 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.

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 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. And I wish to thank Mr. Okugawa for his assistant.

References

[1] N. Eén and N. Sörensson. An extensible SAT-solver. In proceedings of SAT, pages 502-518, 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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Splitter – a Scalable Parallel SAT Solver Based on Iterative Partitioning

Antti E. J. Hyvärinen
Aalto University
Department of Information and Computer Science
PO Box 15400, FI-00076 AALTO, Finland
antti.hyvarinen@aalto.fi

Norbert Manthey
Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

***Abstract*—This document briefly describes the SAT solver SPLITTER in the configuration it has been submitted to the SAT Challenge 2012. SPLITTER is a search space splitting solver based on iterative partitioning.**

I. The Basic Design of Splitter

The implementation of SPLITTER [3] is based on MINISAT 2.2 [6], [5]. MINISAT 2.2 has been extended with a component that takes care of handling multiple incarnations of the solver in the multi-core environment. This component, called *master*, furthermore maintains a tree with the initial formula as root. Each node of this tree represents a sub formula that has been created by splitting the parent node. To create child nodes, a thread is created to split the formula into sub formulas. Depending on the available nodes that need to be solved, the master assigns available resources either for splitting another node or to solve a sub formula. For both of these tasks time limits are set, so that a working thread might be aborted if its current formula seems to be too hard to be solved. The splitting timeout is multiplied with the numbers of sub formulas that should be created during splitting a node. Thus, the time to create a single node is the same for each node. When a working thread has finished its task, learned unit clauses are attached to the currently handled node. These unit clauses can be used by threads that solve nodes in the sub tree of that node to prune their search space further.

II. Splitting a Formula

A formula is split into sub formula by iterative partitioning [2]. The *Iterative Partitioning Approach* is based on solving a hierarchical *partition tree* in a breadth-first order. Given a formula ϕ, iteratively constructed derived formulas can be presented by a *partition tree* T_ϕ. Each node ν_i is labeled with a set of clauses $Co(\nu_i)$ so that the root ν_0 is labeled with $Co(\nu_0) = \phi$, and given a node ν_k and a rooted path $\nu_0, \ldots, \nu_{k-1}$ to its parent, the label of ν_k is $Co(\nu_k) = \kappa_i$, where κ_i is one of the constraints given by $P(\bigwedge_{j=0}^{k-1} Co(\nu_j), n)$. Each node ν_k with a rooted path $\nu_0, \ldots, \nu_k$ represents the formula $\phi_{\nu_k} = \bigwedge_{i=0}^{k} Co(\nu_i)$. For creating new nodes we use the scattering approach as in [2]. Solving is attempted for each ϕ_{ν_k} in the tree in a breadth-first order. The approach terminates if a satisfying assignment is found, or all rooted paths to the leaves contain a node ν_j such that ϕ_{ν_j} is shown unsatisfiable.

III. The Solver Configuration

Since the implementation of MINISAT 2.2 is very close to GLUCOSE 2 [1], but GLUCOSE 2 performed better during recent competitions, we decided to adapt the modifications of GLUCOSE 2 into our solver. We set the time limit for solving is set to 15 seconds, and the time limit for creating a single sub formula is set to two seconds. These values are quite small, but we want to ensure that a large part of the search tree is visited within the 900 seconds timeout. The number of used threads is set to 8.

Before the parallel solver is started, the formula is given to the preprocessor COPROCESSOR 2.1 [4] with its default configuration.

The solver is implemented in C++and compiled with the gnu compiler as 64 bit binary by using the -O3 flag[1].

References

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[2] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Proc. LPAR-17*, volume 6397, pages 372–386, 2010.

[3] Antti E. J. Hyvärinen and Norbert Manthey. Designing scalable parallel sat solvers, 2012. Submitted to SAT 2012.

[4] Norbert Manthey. Coprocessor 2.0 – A flexible CNF Simplifier (Tool Presentation), 2012. Submitted to SAT 2012.

[5] Niklas Sörensson. Minisat 2.2 and minisat++ 1.1. http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf, 2010.

[6] Niklas Sörensson and Niklas Eén. MiniSAT 2.1 and MiniSAT++ 1.0 - SAT race 2008 editions. SAT 2009 Competitive Event Booklet, http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf, 2009.

[1]The solver is available at http://tools.computational-logic.org.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

The Stochastic Local Search Solver: SSA

Robert Stelzmann
Knowledge Representation and Reasoning Group
Technische Universitt Dresden, 01062 Dresden, Germany

***Abstract*—This document briefly describes the SLS solver SSA in the configuration it has been submitted to the SAT Challenge 2012.**

I. Major solving techniques

The solver SSA is an implementation of the SPARROW [1] algorithm with some minor enhancements.

II. Parameter description

We basically took the whole parameter setup from the reference SPARROW implementation [2]. Further we slightly adapted the parameter c_3 on 3-SAT instances to 200000, since it seems that the original setting of 100000 is not optimal on large 3-SAT instances.

III. Special Algorithms, Data Structures and Features

We replaced the list data structure used by UBCSAT SPARROW [3] to maintain the promising variables by a more efficient heap. Consequently we got a performance boost on large 3-SAT instances regarding to the number of flips per second.

IV. Implementation detail

The complete solver is completely written from scratch. The primary goal was the development of an easy applicable and robust framework to investigate new procedures and explore different configurations and parameter setups in the future. The solver is highly modularized and allows an easy exchange and extension of single components like the variable selection heuristic or the clause weighting strategy. Still, solving performance is a design criterion - the solver should be competitive with state-of-the-art SLS solver. As an implementation of SPARROW , the solver is capable of the follow state-of-the-art SLS techniques:

- G2WSAT -like [4] greedy search
- PAWS -like [5] clause weighting
- SPARROW -like [1] selection heuristic

V. SAT Challenge 2012 Specifics

Our solver is written in C++11 and was built as a static -O3 64-bit binary by gcc 4.6.2 on a linux machine, running kernel 2.6.32. We submitted our solver to the random track.

VI. Availability

The SSA framework together with some documentation can be found at: http://tools.computational-logic.org.

Acknowledgment

The authors would like to thank Norbert Manthey for his advice.

References

[1] A. Balint and A. Fröhlich, "Improving Stochastic Local Search for SAT with a New Probability Distribution," in *Theory and Applications of Satisfiability Testing - SAT 2010*, ser. Lecture Notes in Computer Science, O. Strichman and S. Szeider, Eds. Springer Berlin / Heidelberg, 2010, vol. 6175, pp. 10–15, 10.1007/978-3-642-14186-7_3. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14186-7_3

[2] A. Balint, A. Fröhlich, D. A. D. Tompkins, and H. H. Hoos, "Sparrow2011," Solver Description, SAT 2011 Competition Booklet, 2011.

[3] D. A. D. Tompkins and H. H. Hoos, "UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT," in *Revised Selected Papers from the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, ser. Lecture Notes in Computer Science, H. Hoos and D. Mitchell, Eds., vol. 3542. Springer Berlin / Heidelberg, 2005, pp. 306–320.

[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. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 158–172. [Online]. Available: http://dx.doi.org/10.1007/11499107_12

[5] J. Thornton, D. N. Pham, S. Bain, and V. Ferreira, "Additive versus multiplicative clause weighting for SAT," in *Proceedings of the 19th national conference on Artifical intelligence*, ser. AAAI'04. AAAI Press, 2004, pp. 191–196. [Online]. Available: http://dl.acm.org/citation.cfm?id=1597148.1597181

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

TENN

Takeru Yasumoto
Kyushu University, Japan
yasumoto.kyushu@gmail.com

I. Introduction

TENN is based on MiniSat2.2[1]. The TENN system employs a double BTLV MA Restart strategy, a strategy for aggressively keeping learnt clauses, FIRM LBD and evaluation values of learnt clause, NTLBD, LTLBD, HTLBD and LBDSUM.

II. TLBD

True LBD, TLBD for short, is a kind of LBD[2]. TLBD is different from LBD in the manner of updating its value. TLBD ignores literals assigned at level 0. And TENN utilizes three kinds of TLBD and use TLBDSUM which is the sum of them.

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, takes the worst NTLBD of a learnt clause so far.

III. FIRM LBD

FIRM LBD is a strategy for keeping good clauses. If the TLBDs for a learnt clause satisfy one of the following conditions, it will not be deleted.

1) NTLBD is lower than 3.
2) LTLBD is lower than 3 and HTLBD is lower than 11.
3) TLBDSUM is lower than 11.

IV. double BTLV MA Restart

double BTLV MA Restart is a dynamic restart strategy. If one of the following conditions is satisfied, then a restart is forced.

1) an average of backtrack levels in the last 50 conflicts is greater than an average of backtrack levels over the last 1000 conflicts.
2) an average of backtrack levels in the last 50 conflicts is greater than the global average $\times$ 1.05.

Acknowledgment

I wish to express my gratitude to Mr. Hasegawa, Mr. Fujita, Mr. Koshimura for valuable advices and comments. And I wish to thank Mr. Okugawa for his assistant.

References

[1] N. Eén and N. Sörensson. An extensible SAT-solver. In proceedings of SAT, pages 502-518, 2003.
[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In proceedings of IJCAI, 2009.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

ZENN

Takeru Yasumoto
Kyushu University, Japan
yasumoto.kyushu@gmail.com

I. Introduction

ZENN is based on MiniSat2.2[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 SINN will not delete more than half of learnt clauses like MiniSat.

B. LBD+CDLV Phase

1) LBD+CDLV Restart: LBD+CDLV restart is a dynamic restart strategy used by GlueMiniSat2.2.5[3]: 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 learnt clauses at deletion time. This policy is based on GlueMiniSat2.2.5.

III. TLBD

True LBD, TLBD for short, is a kind of LBD[2]. TLBD is different from LBD in the manner of updating its value. TLBD ignores literals assigned 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.

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 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. And I wish to thank Mr. Okugawa for his assistant.

References

[1] N. Eén and N. Sörensson. An extensible SAT-solver. In proceedings of SAT, pages 502-518, 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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

ZENNfork

Yuko Akashi
Kyushu University, Japan
2ie11001y@s.kyushu-u.ac.jp

Takeru Yasumoto
Kyushu University, Japan
yasumoto.kyushu@gmail.com

I. Introduction

ZENNfork is a parallel SAT Solver based on ZENN (This is based on MiniSat2.2.0[1].). Parallelization is performed by ‘ fork ’ which is a UNIX system call. After fork, the parent process deals with a case where a variable is assigned ‘ 1 ’ while the child process deals with a case where the variable is assigned ‘0’. The parent process and the child process can run independently. No synchronization occurs until the processes finish their jobs in principles. The parent process waits until the all the child processes terminate in order to prevent them from becoming zombie processes.

II. Timing of Fork

Initially a SAT process is invoked. After ten restarts, we execute fork. Just before fork, we choose a desicion variable by pickBranchLit(). Before fork the parent process solves the case where the variable is assigned ‘1’ while the child process solves the case where the variable is assigned ‘ 0 ’ . The child process has a copy of learned clauses which may reduce search space afterward.

After the first fork, at every restart we execute fork if the number of current SAT process is less than the number of the cores. Therefore the number of SAT processes is always less than or equal the number of the cores. This needs synchronization of SAT processes. The synchronization is realized by using semaphore. The number of the current processes is memorized in memory. The number is increased when fork succeeds while it is decreased when a SAT process is finished.

III. Termination

If a sat process find a model, other processes do not need to continue. In order to let the processes know that the model is found, we use semaphore. When a process find a model, the process write a big number, which indicates SATISFIABLE, to the memory instead of decrement. Other processes notice the fact at the next restart. Thus every process does not terminate immediately.

IV. Features of ZENN

A. 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 amount of learnt clauses deletion. LBD+CDLV Phase uses LBD+CDLV restart for restart strategy and RQPolicy as deletion learnt clause management.

B. Luy SE Phase

1) Luby SE Phase: Luby SE Restart is a restart strategy based on Luby Restart. Luby Restart use a sequence that has cycles. Luby SE Restart shorten length of every cycle and let solver search more deeply.

2) RHPolicy: The solver deletes half learnt clauses at deletion time. This policy is based on MiniSat2.2 but SINN will not delete more than half learnt clauses like MiniSat.

C. LBD+CDLV Phase

1) LBD+CDLV Restart: LBD+CDLV restart is a dynamic restart strategy used by GlueMiniSat[3]: if one of the following conditions is satisfied, then a restart is forced.

(a) an average of decision levels over the last 50 conflicts is greater than the global average.

(b) an average of LBDs over the last 50 conflicts is greater than the global average $\times$ 0.8.

2) RQPolicy: The solver deletes 3 quarters of learnt clauses at deletion time. This policy is based on glueminisat2.2.

D. TLBD

TLBD is a kind of LBD[2]. TLBD is different from LBD in the manner of updating its value. TLBD ignores literals assigned at level 0.

1) NTLBD: NTLBD is a kind of TLBD. NTLBD takes the latest TLBD of a learnt clause.

2) LTLBD: LTLBD is also a kind of TLBD. LTLBD takes the best NTLBD of a learnt clause so far.

E. SAFE LBD

Safe LBD is a criterion for freezing learnt clause. When learnt clauses are about to being deleted, if its LTLBD is lower than SAFE LBD, it will not be deleted but be detached and kept for possible activation in the future.

Acknowledgment

Our deepest appreciation goes to Prof.Hasegawa whose enormous support and insightful comments were invaluable during the course of my study. We are also indebt to Associate Prof.Fujita and Assistant Prof.Koshimura whose comments made enormous contribution to our work.

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.

BENCHMARK DESCRIPTIONS

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Application and Hard Combinatorial Benchmarks in SAT Challenge 2012

Adrian Balint
Ulm University
Germany

Anton Belov
University College Dublin
Ireland

Matti Järvisalo
HIIT & Dept. Comp. Sci.
University of Helsinki
Finland

Carsten Sinz
Karlsruhe Institute of Technology
Germany

***Abstract*—We outline the selection process of application and hard combinatorial benchmarks for SAT Challenge 2012, and present a description of the benchmark sets. The 600 selected benchmarks in each category were used in Application SAT+UNSAT, Hard Combinatorial SAT+UNSAT, Parallel Application SAT+UNSAT, and Sequential Portfolio tracks of the event.**

I. Selection Process Outline

The two sets of benchmarks were selected using the process described in this section. Specific details for the individual sets are provided in the subsequent sections.

The selection process was driven by the following (sometimes conflicting) requirements:

(i) 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—including the weakest—submitted solvers. This requirement is due to the fact the SAT solvers in SAT Challenge 2012 are ranked using the the solution count ranking.

(ii) The selected set should contain as many benchmarks as possible that were not used in the previous SAT competitions — we refer to these benchmarks as *unused* from now on.

(iii) The selected set should not contain a dominating number of benchmarks from the same source (domain, benchmark submitter).

The benchmarks were drawn from a pool containing benchmarks that either (i) were used in the past five competitive SAT events (SAT Competitions 2007, 2009, 2011 and SAT Races 2008, 2010); (ii) were submitted to these 5 events but not used (*unused* benchmarks); (iii) new benchmarks submitted to SAT Challenge 2012 (for which separate individual descriptions are provided in these proceedings).

The empirical hardness of the benchmarks (the benchmark *rating*) was evaluated using a selection of well-performing SAT solvers from SAT Competition 2011. Our first attempt to select the *state-of-the-art (SOTA) contributors* [1] from the second phase of the competition failed due to the fact that *all* solvers from the second phase turned out to be SOTA contributors (using the definition in [1]). Driven by the restrictions on computational resources, we ultimately selected five SAT solvers among the best performing solvers from both the application and crafted tracks of the 2011 SAT competition. Preference was given to solvers that solved one or more benchmarks uniquely. The selected solvers for each track (application based on the 2011 application track solvers, hard combinatorial based on the 2011 crafted track solvers) are listed in the subsequent sections.

The benchmarks were rated using the same execution environment as the one used for SAT Challenge 2012. The rating was done according to difficulty as follows:

`easy` — benchmarks that were solved by all 5 solvers in under 90 seconds. These benchmarks are extremely unlikely to contribute to the (solution-count) ranking of SAT solvers in the Challenge, as all reasonably efficient solvers are expected to solve these instances within the 900 seconds timeout enforced in the Challenge.

`medium` — benchmarks that were solved by all 5 solvers in under 900 seconds. Though these benchmarks are expected to be solved by the top-performers in the Challenge, they can help to rank the weaker solvers.

`too-hard` — benchmarks that were not solved by any solver within 2700 seconds (3 times the timeout used in the Challenge). These benchmarks are expected to be unsolved by all solvers in the Challenge, 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 2700 seconds, and were not solved by at least one solver within 900 seconds. These benchmarks are expected to be the most useful for ranking the top-performing solvers submitted to the Challenge.

This rating of the benchmarks is similar to the used in SAT Competition 2009 and 2011 [1], except that by singling out and disregarding the benchmarks that would almost certainly not be solved by any submitted solver (these are the `too-hard` benchmarks), we intended to increase the effectiveness of the selected sets for the ranking the solvers.

Once the hardness of the benchmarks in the pool was established, 600 benchmarks were selected from the pool. During the selection we attempted to keep the 50-50 ratio

[1] http://www.satcompetition.org/2009/BenchmarksSelection.html

TABLE I
PROPERTIES OF THE SELECTED 600 "APPLICATION" BENCHMARKS

	Rating				Satisfiability			Status	
	easy	medium	hard	too-hard	SAT	UNSAT	UNKNOWN	used	unused
Num. instances	57	246	291	6	264	333	3	289	311

TABLE II
PROPERTIES OF THE SELECTED 600 "HARD COMBINATORIAL" BENCHMARKS

	Rating				Satisfiability			Status	
	easy	medium	hard	too-hard	SAT	UNSAT	UNKNOWN	used	unused
Num. instances	52	39	503	6	368	226	6	284	316

between the medium and hard benchmarks, and, at the same time, to make sure that no benchmarks from the same source are over-represented ($> 10\%$ of the selected set). This latter requirement has forced us to select about 10% of easy and a number of too-hard benchmarks as well. The details for each selected set differ, and are provided in the following.

II. APPLICATION INSTANCES

The five SAT solvers used to evaluate the hardness of the application instances are: CryptoMiniSat (ver. Strange-Night2-st), Lingeling (ver. 587f), glucose (ver. 2), QuteRSat (ver. 2011-05-12), RestartSAT (ver. B95). All solvers were obtained from SAT Competition 2011 website [2]. The set of application benchmarks was drawn from the pool of 5472 instances. Some of the statistics on the set of 600 selected instances are presented in Table I. The distribution of the selected benchmarks among the various sources is presented in the table below.

Source	Count
2D strip packing	10
Bioinformatics	28
Diagnosis	59
FPGA routing	2
Hardware verification: BMC	11
Hardware verification: BMC, IBM benchmarks	60
Hardware verification: CEC	20
Hardware verification: pipelined machines (P. Manolios)	60
Hardware verification: pipelined machines (M. Velev)	54
Planning	46
Scheduling[3]	9
Software verification: bit verification	60
Software verification: BMC	14
Termination	33
Crypto: AES[3]	11
Crypto: DES	10
Crypto: MD5	14
Crypto: SHA	10
Crypto: VMPC	13
Miscellaneous/unknown	76

Overall, we achieved fairly balanced mix between medium and hard benchmarks, SAT and UNSAT benchmarks, and among the various sources. The proportion of previously used benchmarks is, despite our best efforts, too high — we take this opportunity to encourage the community to contribute new application benchmarks.

[2]http://www.satcompetition.org/2011

[3]Includes new benchmarks submitted to SAT Challenge 2012. Separate descriptions of the benchmarks are provided in these proceedings.

III. HARD COMBINATORIAL INSTANCES

The five SAT solvers used to evaluate the hardness of the application instances are: clasp_2.0 (ver. R4092-crafted), SArTagnan (ver. 2011-05-15), MPhaseSAT (ver. 2011-02-15), sattime (ver. 2011-03-02), Sparrow UBC (ver. SATComp11). Note that we added the SLS-based solver Sparrow UBC to the set — this is due to the fact that some of the benchmarks in the hard combinatorial category are random-like. All solvers were obtained from SAT Competition 2011 website[2]. The set of hard combinatorial benchmarks was drawn from the pool of 1743 instances. Table II presents some of the statistics on the set of 600 selected instances. The distribution of the selected benchmarks among the various sources is presented in the following table.

Source	Count
Automata synchronization	8
Edge matching	32
Ensemble computation[3]	12
Factoring	43
Fixed-shape forced satisfiable[3]	29
Games: Battleship	28
Games: Hidoku[3]	3
Parity games	26
Pebbling games	13
Horn backdoor detection via vertex cover[3]	59
MOD circuits	35
Parity (MDP)	7
Quasigroup	40
Ramsey cube	8
rbsat	53
sgen[3]	47
Social golfer problem	2
Sub-graph isomorphism	46
Van der Waerden numbers	41
XOR chains	2
Miscellaneous	66

Note that while the selected benchmarks are balanced well among various sources, the proportion of hard benchmarks is very high. This is due to the fact, among the 1743 benchmarks in the pool, there are only 39 instances of medium difficulty. Approximately 1/3 of the pool consists of easy instances, 1/3 of hard, and 1/3 of too-hard. Thus, we expect the selected set to be more difficult for the solvers in the Challenge than the set of application instances. The dis-balance between SAT

and UNSAT instances is explained by the fact that a large proportion of the `hard` instances are satisfiable.

ACKNOWLEDGMENTS

The authors would like to thank the bwGRID [3] for providing the computational resources to evaluate the instances. The first author acknowledges funding from the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1, the second author from Science Foundation of Ireland (grant BEACON 09/IN.1/I2618), and the third author from Academy of Finland (grants 132812 and 251170).

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] 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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

SAT Challenge 2012 Random SAT Track: Description of Benchmark Generation

Adrian Balint
Ulm University
Germany

Anton Belov
University College Dublin
Ireland

Matti Järvisalo
HIIT & Dept. Comp. Sci.
University of Helsinki
Finland

Carsten Sinz
Karlsruhe Institute of Technology
Germany

***Abstract*—The SAT Challenge 2012 random SAT track benchmark set contains 600 instances, generated according to the uniform random generation model. The instances were divided in five major classes: k-SAT for $k = 4, 5, 6, 7$. Each class contains ten subclasses with varying clauses-to-variables ratios and numbers of variables. Each subclass contains 12 instances. Within this description we provide insights about the generation algorithm, the model according to which the size and properties of instances were chosen, and also about the filtering process.**

I. Uniform random k-SAT

The (uniform) random k-SAT problem is arguably the most studied class of random SAT instances. A random k-SAT generator takes the following inputs.

1) n, the number of variables
2) m, the number of clauses, or the ratio $\alpha = \frac{m}{n}$
3) k, the number of literals in each clause

A generation procedure for uniform random k-SAT instances is described as Algorithm 1.

Algorithm 1: Uniform k-SAT generator

Input : n, m, k
Output: uniform random k-SAT instance

```
F = ∅ ;
i = 0;
while i < m do
    C_i = ∅;
    j = 0 ;
    while j < k do
        v = variable index chosen uniformly at random
        from {1 ... n};
        l = literal chosen uniformula at random from
        {v, v̄};
        if (l, (l̄ ∉ C_i)) then
            C_i = C_i ∨ l ;
            j++;
    if (C_i ∉ F) then
        F = F ∧ C_i;
        i++;
return F;
```

A random k-SAT instance can be characterized by its size n and by the clauses-to-variables ratio α. The satisfiability status of such an instance is not know a priori, although for each k there exists a threshold value α_t for the clauses-to-variables ratio such that all instances generated with an $\alpha < \alpha_t$ are with high probability satisfiable, and for all instances generated with an $\alpha > \alpha_t$ are with high probability unsatisfiable.

II. The Random Category in the SAT Competitions 2007-2011

The random instances generated and used in the last three SAT Competitions [1] mainly follow the uniform random generation model described before (with a small exception: the $2 + p$ instances used in 2007). The range of number of variables and clauses-to-variable ratios of the instances used in the competitions have been as follows.

1) 3-SAT
   a) $\alpha = 4.26$ on threshold and $200 < n < 800$
   b) $\alpha = 4.2$ near threshold and $2000 < n < 50000$
2) 5-SAT
   a) $\alpha = 21.3$ on threshold and $90 < n < 175$
   b) $\alpha = 20$ near threshold and $600 < n < 2000$
3) 7-SAT
   a) $\alpha = 89$ on threshold and $50 < n < 90$
   b) $\alpha = 85$ near threshold and $140 < n < 400$

Note that only k-SAT instances for $k = 3, 5, 7$ were used in these competitions, and for each class only two different ratios were considered (one also containing unsatisfiable instances). For further background, we refer to [2] for details on the random instances used in the 2005 SAT Competition.

III. Size and Ratios of the SC2012 Instances

For SAT Challenge 2012, we generated k-SAT instances for $k = 3, 4, 5, 6, 7$.

Instances generated at the threshold ratios or near them are the most challenging instances for complete and local search methods. For large n, the best approximations of the threshold ratios are given in [3] and listed in Table I.

TABLE I: Threshold values α_k for different k

k	3	4	5	6	7
α_k	4.267	9.931	21.117	43.37	87.79

Starting from these values, we applied the following generation model: For each k, two extreme points (α_k, n_k) and (α_{nt}, n_{nt}) were defined:

- n_k is the largest number of variables a formula generated at the threshold α_k is allowed to have (estimated based on e.g. the performance of the best solvers for the random category in the latest SAT Competition).
- α_{nt} is the largest clauses-to-variables ratio for the number of variables n_{nt} (again based on our estimate of the behaviour of best known solvers).

We ended up using the following values for the different k's:

k	α_k	n_k	α_{nt}	n_{nt}
3	4.267	2000	4.2	40000
4	9.931	800	9.0	10000
5	21.117	300	20	1600
6	43.37	200	40	400
7	87.79	100	85	200

For each k, the following 10 (α, n) combinations were chosen from on the line between (α_k, n_k) and (α_{nt}, n_{nt}), totalling at 50 combinations.

Set	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
1	4.2	9	20	40	85
	40000	10000	1600	400	200
2	4.208	9.121	20.155	40.674	85.558
	35600	8800	1420	360	180
3	4.215	9.223	20.275	41.011	85.837
	31400	7800	1280	340	170
4	4.223	9.324	20.395	41.348	86.116
	27200	6800	1140	320	160
5	4.23	9.425	20.516	41.685	86.395
	23000	5800	1000	300	150
6	4.237	9.526	20.636	42.022	86.674
	18800	4800	860	280	140
7	4.245	9.627	20.756	42.359	86.953
	14600	3800	720	260	130
8	4.252	9.729	20.876	42.696	87.232
	10400	2800	580	240	120
9	4.26	9.83	20.997	43.033	87.511
	6200	1800	440	220	110
10	4.267	9.931	21.117	43.37	87.79
	2000	800	300	200	100

For each (α, n) combination, we generated 100 instances, resulting in a total of 1000 instances per k value, and thus a total of 5000 instances. For the generation process we used the pseudo random number generator SHA1PRNG part of Sun Java implementation.

IV. The Filtering Procedure

For filtering out unsatisfiable instances within the generated 5000 instances just described, we used the best performing solvers from the SAT Competition 2011[1] random track: Sparrow2011, sattime2011, EagleUP and adaptG2WSAT2011. Additionally, we used survey propagation [4], adaptive WalkSAT [5], adaptive probSAT [5] and adaptnovelty+ from UBCSAT [6]. Each solver was run only once on each instance using a cutoff of 2700 seconds (3 times more than the SAT Challenge timeout). If a instance was solved by at least one solver, it is considered satisfiable, and otherwise the satisfiability status of the instance is marked as unknown.

[1]http://www.satcompetition.org

V. The Final Benchmark Set

From each of the 50 sets of instances generated for each (α, n) combination, we randomly chose 12 instances that were determined as satisfiable in the filtering phase. Notice that this random selection was done independently of the running times of the solvers used in filtering on the instances. *The resulting set of a total of 600 instances constitutes the benchmark set used in the Random SAT Track of SAT Challenge 2012*.

The complete set of instances before and after filtering are available at http://baldur.iti.kit.edu/SAT-Challenge-2012/.

Acknowledgments

The authors would like to thank the bwGRID [7] for providing the computational resources to filter the instances. The first author acknowledges funding from the Deutsche Forschungsgemeinschaft (DFG) (grant SCHO 302/9-1), the second author from Science Foundation of Ireland (grant BEACON 09/IN.1/I2618), and the third author from Academy of Finland (grants 132812 and 251170).

References

[1] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon, "The international SAT solver competitions," *AI Magazine*, vol. 33, no. 1, pp. 89–92, 2012.

[2] O. Kullmann, "The sat 2005 solver competition on random instances," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 1-4, pp. 61–102, 2006.

[3] 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, 2006.

[4] A. Braunstein, M. Mézard, and R. Zecchina, "Survey propagation: An algorithm for satisfiability," *Random Struct. Algorithms*, vol. 27, no. 2, pp. 201–226, 2005.

[5] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *Proc. SAT*, ser. LNCS. Springer, 2012, to appear.

[6] D. A. Tompkins and H. H. Hoos, "UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT," in *Proc. SAT*, 2004.

[7] 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, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Advanced Encryption Standard II benchmarks

Matthew Gwynne
Computer Science Department
College of Science, Swansea University
Swansea, SA2 8PP, UK
email: csmg@swansea.ac.uk

Oliver Kullmann
Computer Science Department
College of Science, Swansea University
Swansea, SA2 8PP, UK
http://cs.swan.ac.uk/~csoliver

I. Background

The Advanced Encryption Standard (AES), described in [1], is a popular encryption cipher used in a variety of areas including wireless, disk, and network encryption. AES was designed to replace the Data Encryption Standard (DES), which is now vulnerable to brute force attacks, and to provide an alternative to the computationally expensive Triple-DES. Small-scale variants of the AES were introduced in [2]. Such small-scale variations allow the scaling of the AES block and key sizes while maintaining the cryptographic properties of the cipher. They were designed to offer instances which were more easily analysed, due to their smaller size, but which shared the same fundamental structure and features as the AES.

We present benchmarks which are translations of small-scale AES into SAT. These benchmarks are generated via a general translation framework within the `OKlibrary`[1]. Small-scale variants are used, instead of the standard AES, to provide instances which are likely to be solvable by SAT solvers within the time-frame of the SAT Challenge 2012.

All work presented is available within the `OKlibrary` (see [3]), a research platform for hard problems; preliminary results were presented in [4], and a full exploration will be available in [5] (forthcoming).

II. Small-scale AES

A small-scale AES cipher $\mathrm{aes}(m, r, c, e)$ for $m, r, c, e \in \mathbb{N}$, encrypts, by a series of m rounds. a plaintext P to ciphertext C using key K, where P, K and C are b-bit inputs and $b := r \cdot c \cdot e$. Standard 128-bit AES corresponds to $\mathrm{aes}(10, 4, 4, 8)$ (with a small irrelevant change; see below). The b-bit inputs (P, K and C) constitute a matrix with r rows and c columns, and with elements in the *byte field*, a finite field of order 2^e with base-set $\{0,1\}^e$ (so "bytes" have length e). These matrices are formed in column-major order, that is, column-by-column, top-to-bottom, and left-to-right. For example, the binary string `0000 0001 0010 1111` for the $\mathrm{aes}(m, 2, 2, 4)$ ciphers corresponds to the matrix

$$\begin{pmatrix} \mathtt{0} & \mathtt{2} \\ \mathtt{1} & \mathtt{F} \end{pmatrix}$$

where the elements are given in hexadecimal. The cipher $\mathrm{aes}(m, r, c, e)$ has the following structure:

1) The inputs are the b-bit plaintext P and key K.
2) The *key schedule* generates $m + 1$ b-bit round-keys $K_0, \ldots, K_m$ from K, where $K_0 := K$. K_{i+1} is generated via applications of the *S-box* and XOR from K_i.
3) The following *round function* is iterated m times. The i-th round, $i = 1, \ldots, m$, consists of:
   a) Addition (XOR) of K_{i-1} to the input.
   b) Applying a permutation over $\{0,1\}^e$ (the *S-box*) to each byte in the matrix(-representation).
   c) Permutation of bytes in the rows of the matrix.
   d) Multiplication of each column by a fixed (invertible) $r \times r$-matrix of bytes.

   The input to the first round is P, and then each round output becomes the input of the next round.
4) The output of the last round is additionally XORed with K_m to generate the ciphertext C.

The third step of a round, the row-permutation (ShiftRows), is suppressed in our system, as it is directly applied to the variable-indices.

As mentioned, there is a small deviation for standard AES: The final round of the standard AES omits the multiplication by the fixed matrix (MixColumns), however for simplicity this is *not* the case in the small-scale AES. Within the `OKlibrary` we say the 10-round standard AES has $9+\frac{2}{3}+\frac{1}{3}$ rounds, denoting that it has 9 full rounds, then the addition and S-box application (two thirds of a round), and a final addition (one third of a round). As we only consider the small-scale AES here, we just speak of "m rounds" in this document, meaning actually $m + \frac{1}{3}$ rounds (as specified above). We remark that the "$+\frac{1}{3}$" step, the final addition of the last round key, has considerable effect on the hardness, while sometimes "m rounds" in the literature *exclude* this final addition.

In the following, we will treat the S-box and finite field operations as boolean functions to be translated directly to

[1] http://ok-sat-library.org/

SAT, not further broken down. The specification for each of these operations can be found in [2].

III. CNF TRANSLATION

The instances are generated by decomposing the small-scale AES ciphers based on the standard definition from [2] into small constituent boolean functions: S-boxes, XORs and field multiplications. These small boolean functions are then translated to CNFs (without new variables) with the minimum number of clauses. This differs from the approaches taken in [2], [6], which apply global algebraic methods and completely "rewrite" the AES system.

Our translation produces a template-CNF for which the first $3 \cdot b$ variables correspond to the plaintext, key and ciphertext bits respectively (in this order). That CNF is satisfied by a total assignment iff that assignment corresponds to plaintext, key and ciphertext binary-strings P, K and C such that $\mathrm{aes}(m, r, c, e)(P, K) = C$, while the auxiliary variables for the encryption steps are set according to their definitions.

IV. BENCHMARKS

We provide *key-discovery instances* for a subset of the small-scale AES parameter space, yielding benchmarks with a range of difficulties. A key-discovery instance is obtained from the template-CNF by applying the partial assignment given by the plaintext and ciphertext. The SAT solver has then to compute the (or a) key. We provide benchmarks (see Fig 1) based on 5 randomly generated plaintext-key pairs for each of the following small-scale ciphers:

1) 16-bit: $\mathrm{aes}(m, 2, 1, 8)$ for $m \in \{3, 4, 5, 6, 10\}$;
2) 20-bit: $\mathrm{aes}(m, 1, 5, 4)$ for $m = 20$;
3) 24-bit: $\mathrm{aes}(m, 1, 3, 8)$ for $m \in \{3, 4\}$;
4) 32-bit: $\mathrm{aes}(m, 2, 2, 8)$ for $m \in \{1, 2\}$.

Calculating the number of variables in each instance:

1) $3 \cdot b$ input variables (P, K, C);
2) $m \cdot b$ round-key variables (b per round);
3) $m \cdot b$ round output variables (b per round);
4) $m \cdot b$ variables for the S-box outputs (b per round);
5) $m \cdot b$ variables for MixColumns output (b per round);
6) $e \cdot m$ variables for round constants in the key-schedule (e per round);
7) S-box output variables in key-schedule:
   a) if $c > 1$ then $r \cdot e \cdot m$ ($r \cdot e$ per round);
   b) otherwise $e \cdot m$ (e per round);
8) if $r \geq 2$ then $r \cdot r \cdot e \cdot c \cdot m$ variables for the output of multiplication boxes in MixColumns ($r \cdot r \cdot e \cdot c$ per round).[2)]

[2)]Note that in general, if $r = 4$, the number of variables due to the MixColumns is more complicated, but such examples do not occur here and so we avoid the issue

Benchmark	n	c	Difficulty
aes_16_3_keyfind_*.cnf	384	4,464	easy
aes_16_4_keyfind_*.cnf	496	5,920	medium-easy
aes_16_5_keyfind_*.cnf	608	7,376	medium
aes_16_6_keyfind_*.cnf	720	8,832	medium-hard
aes_16_10_keyfind_*.cnf	1,168	14,656	hard
aes_20_20_keyfind_*.cnf	1,820	7,160	medium-easy
aes_24_3_keyfind_*.cnf	408	4,512	medium
aes_24_4_keyfind_*.cnf	520	5,968	medium-hard
aes_32_1_keyfind_*.cnf	312	2,604	medium
aes_32_2_keyfind_*.cnf	528	5,016	medium-hard

Fig. 1. Small-scale AES benchmarks, statistics for benchmarks, and estimate of difficulty (time) to solve; all are satisfiable. "n" and "c" denote the number of variables and number of clauses respectively for each instance. Difficulty: easy = <1 minute, medium-easy = >5 but <15 minutes, medium = >15 but <60 minutes, medium-hard = between 1 and 3 hours, hard = >3 hours. To determine the difficulty, all instances were solved using `minisat-2.2.0` on a machine with a 3.0Ghz Core2Duo and 4GB of RAM.

Calculating the number of clauses in each instance:

1) $(r \cdot c \cdot m + r \cdot m) \cdot X$ clauses from the S-boxes[3)].
2) If $r = 2$ then
   a) $r \cdot c \cdot m \cdot M_2$ clauses from multiplications by `02`.
   b) $r \cdot c \cdot m \cdot M_3$ clauses from multiplications by `03`.
3) $r \cdot c \cdot e \cdot m \cdot 2^r$ clauses from matrix additions.
4) $(m + 1) \cdot r \cdot c \cdot e \cdot 4$ clauses from key additions.
5) Clauses from additions in the key-schedule:
   a) if $c = 1$ then $e \cdot m \cdot 4$,
   b) otherwise $e{\cdot}m{\cdot}8{+}e{\cdot}(r{-}1){\cdot}c{\cdot}m{\cdot}4{+}e{\cdot}(c{-}1){\cdot}m{\cdot}4$,
6) $e \cdot m$ clauses encoding the round constants.
7) $2 \cdot r \cdot c \cdot e$ clauses for the plaintext-ciphertext pair.

where X, M_2 and M_3 are the sizes of the e-bit S-box, multiplication by `02` and `03` respectively.

So, for example, $\mathrm{aes}(2, 2, 2, 8)$ has:

1) $3 \cdot 32 + 4 \cdot 2 \cdot 32 + 8 \cdot 2 + 2 \cdot 8 \cdot 2 + 2 \cdot 2 \cdot 8 \cdot 2 \cdot 2 = 528$ variables;
2) $(2{\cdot}2{\cdot}2{+}2{\cdot}2){\cdot}294{+}2{\cdot}2{\cdot}2{\cdot}20{+}2{\cdot}2{\cdot}2{\cdot}36{+}2{\cdot}2{\cdot}8{\cdot}2{\cdot}2^2{+}3{\cdot}2{\cdot}2{\cdot}8{\cdot}4{+}8{\cdot}2{\cdot}8{+}8{\cdot}1{\cdot}2{\cdot}2{\cdot}4{+}8{\cdot}1{\cdot}2{\cdot}4{+}8{\cdot}2{+}2{\cdot}2{\cdot}2{\cdot}8 = 5016$ clauses.

These benchmark instances, called `aes_b_m_keyfind_s.cnf` ("s" for "seed"), are found in the `Benchmarks/b/` directory.

V. GENERATING BENCHMARKS

For each benchmark

`Benchmarks/b/aes_b_m_keyfind_*.cnf`,

[3)]from round + key schedule

Boolean function	n	|DNF|	$\|\mathrm{prc}_0(f)\|$	|CNF| (min)
4-bit S-box	8	16	147	22
4-bit $\times$02	8	16	14	9
4-bit $\times$03	8	16	120	16
8-bit S-box	16	256	136,253	≤ 294
8-bit $\times$02	16	256	58	20
8-bit $\times$03	16	256	5,048	≤ 36

Fig. 2. Statistics for small-scale AES boxes. n is the number of variables; other columns list the number of clauses in the (unique) DNF, set of prime implicates ($\mathrm{prc}_0(f)$) and minimum CNF (used in the translation) for each box. In the case that a CNF being minimum is only conjectured we denote this by $\leq$.

the corresponding uninstantiated template-CNF for m rounds is provided in `Formulas/b/aes_b_m.cnf`. The associated plaintext-ciphertext assignment, given as clause-sets with $2 \cdot b$ unit clauses, is provided in

`Assignments/b/aes_ass_b_m_keyfind_s.cnf`

The provided `AppendDimacs` tool can be used to combine the instance file and the assignment to generate the benchmark. The hexadecimal representations of the plaintext, key and ciphertext are given in the first comment line of the assignment files.

More detailed information on the generation of all instances can be found in the documentation of the `OKlibrary`[4]. The Git-ID of the state of the `OKlibrary` is e3896a3b8e00cf8c46739fded03a6835117c01cb (15.4.2012).

REFERENCES

[1] J. Daemen and V. Rijmen, *The Design of Rijndael*. Berlin: Springer, 2001, iSBN 3-540-42580-2; QA76.9.A25 D32 2001.

[2] C. Cid, S. Murphy, and M. Robshaw, *Algebraic Aspects of the Advanced Encryption Standard*. Springer, 2006, iSBN-10 0-387-24363-1.

[3] O. Kullmann, "The `OKlibrary`: Introducing a "holistic" research platform for (generalised) SAT solving," *Studies in Logic*, vol. 2, no. 1, pp. 20–53, 2009.

[4] M. Gwynne and O. Kullmann, "Towards a better understanding of SAT translations," in *Logic and Computational Complexity (LCC'11), as part of LICS 2011*, U. Berger and D. Therien, Eds., June 2011, 10 pages, available at http://www.cs.swansea.ac.uk/lcc2011/.

[5] ——, "Attacking DES + AES via SAT: Applying better box representations," arXiv, Tech. Rep. arXiv:??? [cs.DM], 2012, in preparation.

[6] P. Jovanovic and M. Kreuzer, "Algebraic attacks using SAT-solvers," *Groups-Complexity-Cryptology*, vol. 2, no. 2, pp. 247–259, December 2010.

[4] Available at http://cs.swan.ac.uk/~csoliver/ok-sat-library/internet_html/doc/doxygen_html/dc/d1e/OKlib_2Experimentation_2Benchmarks_2docus_2general_8hpp.html

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Horn backdoor detection via Vertex Cover: Benchmark Description

Marco Gario
Fondazione Bruno Kessler, Trento, Italy

***Abstract*—In this document we describe the benchmark submitted to the SAT'12 competition. The benchmark is constituted by hard instances of vertex cover, that originate from the problem of Horn backdoor detection.**

INTRODUCTION

The notion of a *backdoor* was first introduced in [10] and refers to a set of variables that, once it is assigned a value, leaves the rest of the problem in some polynomial-time solvable class. Among the possible classes for which we can study backdoors, our interest in Horn backdoors is mainly given by the simplicity of the decision procedure for Horn formulas and the existence of an elegant reduction of the detection problem to the well-known vertex cover problem [7].

The benchmark presented here originates from experiments done on Horn backdoor detection by means of the reduction to Vertex Cover. In particular, we encode the problem of Horn backdoor detection into a vertex cover problem, and afterwards we encode this problem into SAT.

We provide a short theoretical background concerning backdoors and the reduction in Section I. In Section II we provide more detailed informations on the benchmark, including experimental results.

I. BACKGROUND

In this section we present some background knowledge necessary to understand the benchmark domain. This includes a short introduction on backdoors[1], the reduction to vertex cover, and the reduction from vertex cover to SAT.

A. Definitions

Let F be a propositional formula in conjunctive normal form and J a (partial or total) interpretation from the set of variables occurring in F to the set $\{\top, \bot\}$ of truth values. We use $F|_J$ to denote the *reduct* of F w.r.t. J.

We can group formulas together into a *class* depending on some characteristic they posses. For example, we say that a formula is *Horn* iff each of its clauses is Horn; in turn, we say that a clause is Horn iff it has at most one positive literal.

Backdoors are defined with respect to a class of problems for which there exists a *subsolver* that, roughly, is an algorithm that can decide any instance of the class in polynomial time or reject it as not being part of the class. Here we consider Horn backdoors. For Horn formulas, suitable subsolvers exist (see e.g., [6]). In the sequel, $Horn$ shall denote a subsolver for Horn formulas.

A non-empty subset B of the variables occurring in F is a *strong $Horn$-backdoor*[2] for F iff for all total interpretations J from B to $\{\top, \bot\}$ the subsolver $Horn$ returns a satisfying assignment or concludes unsatisfiability of $F|_J$; in other words, $F|_J$ is a Horn formula. For the same instance there might be several $Horn$-backdoors, possibly of different size. We call a $Horn$-backdoor *minimal* iff no proper subset is a $Horn$-backdoor; we call it *smallest* iff it is minimal and there is no other $Horn$-backdoor of smaller cardinality.

Given F and a set of variables V, we denote with $F-V$ the formula obtained from F by replacing each occurrence of the literals x and $\neg x$ in F by $\bot$ for all $x \in V$, and simplifying the formula. A non-empty subset B of the variables occurring in F is a *deletion $Horn$-backdoor* for F iff $F - B \in Horn$. For Horn formulas the concepts of deletion and strong backdoors are equivalent [5]. This result is important because it is easier to verify whether a backdoor B is a deletion backdoor rather than a strong backdoor: we just need to verify the membership in $Horn$ of $F-B$, and not of all the possible $2^{|B|}$ assignments of B.

We are particularly interested in the *(strong/deletion) $Horn$-backdoor detection problem*. It consists of a SAT instance F and an integer $k \geq 0$, and is the question whether there exists a (strong/deletion) $Horn$-backdoor for F of size at most k. The backdoor detection problem is NP-hard. This follows immediately from the fact that, once a backdoor is known, the problem can be solved in polynomial time.

Let $G = (V, E)$ be a graph and $R \subseteq V$. R is a *vertex cover* of G iff for all $e \in E$ there exists a $v \in R$ such that $v \in e$. Let $|R|$ be the *size* of R. The *vertex cover problem* consists of a graph G and an integer k, and is the question whether there exits a vertex cover of G of size at most k. The *minimum vertex cover* is the optimization version of the problem, in which the goal is to minimize the value of k.

B. From Horn backdoor detection to Vertex Cover

In [7] the Horn backdoor detection problem is reduced to the vertex cover problem as follows: Let F be a SAT instance and $G_F = (V, E)$, where V is the set of variables occurring in F and $E = \{\{u, v\} \mid$ there exists a clause $C \in F$ such that $u, v \in C\}$. As stated in [7], B is a strong Horn backdoor for F iff B is a vertex cover of G_F.

[1] A more detailed introduction to the topic is available in Chapter 13 of [2]

[2] We provide definitions w.r.t. Horn formulas, but this definitions extend to any other class for which a subsolver exists.

As an example consider $F = (x \vee y \vee z) \wedge (\overline{x} \vee z \vee w) \wedge (y \vee w)$. The corresponding graph G_F and a vertex cover for G_F is shown in Fig. 1. The set $\{y, z\}$ is a vertex cover and we obtain $F - \{y, z\} = x \wedge (\overline{x} \vee w) \wedge w$. Thus, $\{y, z\}$ is a deletion Horn backdoor and, hence, a strong Horn backdoor for F. Moreover, solving the minimum vertex cover problem for G_F leads to a smallest strong Horn backdoor.

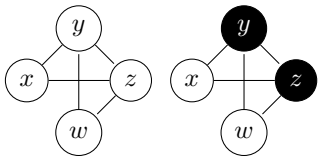


Fig. 1. The graph G_F corresponding to F (left) and a vertex cover for G_F consisting of the filled vertices (right).

The existence of this reduction allows existing theories and algorithms from the vertex cover problem to be applied for solving the strong Horn backdoor detection problem.

In particular, in this work we consider the reduction of the vertex cover problem to SAT. The encoding is quite straightforward: we build binary clauses representing the existence of an edge among two vertices and a cardinality constraint, saying that we can select at most k vertices. In order to encode the *at most* constraint we use the encoding suggested in [8] using $O(kn)$ clauses and $O(kn)$ new variables, where n is the total number of vertices.[3]

II. Details of the Benchmark

The SAT benchmark presented here originates from the Horn backdoor detection problem of some instances from SATLib, SATCompetition and Car Configuration [3], [1], [4]. In particular, we computed a Horn backdoor of these instances by other means (i.e., local search) and we were interested to show minimality of the result. Minimality of the backdoor is equivalent to minimality of the vertex cover. In particular, if the smallest solution for an instance has size k, then we need to test whether there is any solution of size $k - 1$. If this is not the case, then the solution of size k is minimal. The instances presented here are a subset of the ones we used in our experiments; these instances seem to be hard for other types of algorithms for vertex cover[4]. By construction we expect the instances to be unsatisfiable, however we were not able to confirm this. We tried to solve the instances with Cryptominisat 2.9.2 [9], with a time-out of 2 hours on a Intel i7 2.93 GHz with 8GB Ram. The complete benchmark is composed of 61 instances, only 9 were solved (as unsatisfiable) while the majority timed-out. Information on the runtime of the solved instances is presented in Fig. 2. The name of the instances in the benchmark includes the original name (e.g., 3blocks.cnf) and the target cover size (mis-137 for $k = 137$).

[3]Robert Stelzmann kindly provided us with an implementation of this algorithm.

[4]In particular, fixed-parameter tractable algorithms

Name	Time (seconds)
3blocks.cnf.mis-137.cnf	33.71
ais8.cnf.mis-97.cnf	20.56
C171_FR.cnf.mis-128.cnf	75.23
C250_FV.cnf.mis-138.cnf	61.46
C250_FW.cnf.mis-138.cnf	55.38
hole10.cnf.mis-98.cnf	7.77
sgen1-unsat-103-100.cnf.mis-78.cnf	4259.75
sgen1-unsat-85-100.cnf.mis-63.cnf	695.25
sgen1-unsat-97-100.cnf.mis-72.cnf	30.50

Fig. 2. Runtime information of the solved instances

References

[1] *SAT'11 Competition*. http://www.satcompetition.org/, 2011.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[3] Holger Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In I.P Gent, H.v. Maaren, and T. Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000.

[4] W. Küchlin and Carsten Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1):145–163, 2000.

[5] Naomi Nishimura and Prabhakar Ragde. Solving #SAT using vertex covers. *Acta Informatica*, 44(7):509–523, 2007.

[6] Naomi Nishimura, Prabhakar Ragde, and Stefan Szeider. Detecting backdoor sets with respect to Horn and binary clauses. *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[7] Marko Samer and Stefan Szeider. Backdoor trees. *Proceedings of the 23rd Conference on Artificial*, pages 363–368, 2008.

[8] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. *Theory and Practice of Constraint Programming-CP 2005*, pages 1–5, 2005.

[9] M. Soos. CryptoMiniSat 2.5. 0. *SAT Race competitive event booklet*, 2010.

[10] Ryan Williams, C.P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *Proceeding of IJCAI-03*, volume 18, pages 1173–1178, 2003.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Finding Circuits for Ensemble Computation via Boolean Satisfiability

Matti Järvisalo*, Petteri Kaski†, Mikko Koivisto*, and Janne H. Korhonen*
*HIIT & Department of Computer Science, University of Helsinki, Finland
†HIIT & Department of Information and Computer Science, Aalto University, Finland

***Abstract*—This note describes a family of moderately difficult Boolean satisfiability instances based on the NP-complete Ensemble Computation problem.**

I. Introduction

In the study of exact exponential algorithms, one often encounters situations where one could obtain an improved algorithm for a problem if certain sums can be computed with sufficiently few addition operations. This kind of summation problems can be, for fixed parameters, encoded as Boolean satisfiability, resulting in large structured SAT instances. In this note, we describe one such construction giving SAT instances that can serve as benchmarks for SAT solvers. For further background, details, and results, please refer to Järvisalo et al. [1].

Our starting point is the NP-complete *Ensemble Computation* problem [2]:

> *(SUM-)Ensemble Computation.* Given as input a collection $\mathcal{Q}$ of nonempty subsets of a finite set P and a nonnegative integer b, decide (yes/no) whether there is a sequence
>
> $$Z_1 \leftarrow L_1 \cup R_1,\ Z_2 \leftarrow L_2 \cup R_2,\ \ldots\ ,\ Z_b \leftarrow L_b \cup R_b$$
>
> of union operations, where
>
> (a) for all $1 \leq j \leq b$ the sets L_j and R_j belong to $\{\{x\} : x \in P\} \cup \{Z_1, Z_2, \ldots, Z_{j-1}\}$,
> (b) for all $1 \leq j \leq b$ the sets L_j and R_j are disjoint, and
> (c) the collection $\{Z_1, Z_2, \ldots, Z_b\}$ contains $\mathcal{Q}$.

It is also known that *SUM-Ensemble Computation* remains NP-complete even if the requirement (b) is removed, that is, the unions need not be disjoint [2]; we call this variant *OR-Ensemble Computation*. Stated in different but equivalent terms, each set A in $\mathcal{Q}$ in an instance of *SUM-Ensemble Computation* specifies a subset of the variables in P whose sum must be computed. The question is to decide whether b arithmetic gates suffice to evaluate all the sums in the ensemble. An instance of *OR-Ensemble Computation* asks the same question but with sums replaced by ORs of Boolean variables, and with SUM-gates replaced by OR-gates.

In the rest of this note, we describe how Ensemble Computation instances can be encoded as Boolean satisfiability instances, and present certain interesting hand-picked instances that can be used as satisfiability solver benchmarks. In Section II, we give necessary basic definitions, and in Section III we present a SAT encoding for the Ensemble Computation problem. Finally in Section IV, we describe the benchmark instances.

II. Definitions

A *circuit* is a directed acyclic graph C whose every node has in-degree either 0 or 2. Each node of C is a *gate*. The gates of C are partitioned into two sets: each gate with in-degree 0 is an *input gate*, and each gate with in-degree 2 is an *arithmetic gate*. The *size* of C is the number $g = g(C)$ of gates in C. We write $p = p(C)$ for the number of input gates in C.

The *support* of a gate z in C is the set of all input gates x such that there is a directed path in C from x to z. The *weight* of a gate z is the size of its support. All gates have weight at least one, with equality if and only if a gate is an input gate.

In what follows we study two classes of circuits, where the second class is properly contained within the first class. First, every circuit is an *OR-circuit*. Second, a circuit C is a *SUM-circuit* if for every gate z and for every input gate x it holds that there is at most one directed path in C from x to z.

Let $(P, \mathcal{Q})$ be an instance of ensemble computation, that is, let P be a finite set and let $\mathcal{Q}$ be a set of nonempty subsets of P. We adopt the convention that for a SUM-ensemble all circuits considered are SUM-circuits, and for an OR-ensemble all circuits considered are OR-circuits. We say that a circuit C *solves* the instance $(P, \mathcal{Q})$ if (a) the set of input gates of C is P; and (b) for each $A \in \mathcal{Q}$, there exists a gate in C whose support is A. The *size* of the solution is the size of C. A solution to $(P, \mathcal{Q})$ is *optimal* if it has the minimum size over all possible solutions.

III. SAT Encoding

We next develop a SAT encoding for deciding whether a given ensemble has a circuit of a given size.

A. Basic Encoding

We start by giving a representation of an OR- or SUM-circuit as a binary matrix. This representation then gives us a straightforward way to encode the circuit existence problem as a propositional formula.

Let $(P, \mathcal{Q})$ be an OR- or SUM-ensemble and let C be a circuit of size g that solves $(P, \mathcal{Q})$. For convenience, let us assume that $|P| = p$, $|\mathcal{Q}| = q$ and $P = \{1, 2, \ldots, p\}$. Furthermore, we note that outputs corresponding to sets of size 1 are directly provided by the input gates, and we may thus assume that $\mathcal{Q}$ does not contain sets of size 1. The circuit

C can be represented as a $g \times p$ binary matrix M as follows. Fix a topological ordering $z_1, z_2, \ldots, z_g$ of the gates of C such that $z_i = i$ for all i with $1 \leq i \leq p$ (we identify the input gates with elements of P). Each row i of the matrix M now corresponds to the support of the gate z_i so that for all $1 \leq j \leq p$ we have $M_{i,j} = 1$ if j is in the support of z_i and $M_{i,j} = 0$ otherwise. In particular, for all $1 \leq i \leq p$ we have $M_{i,i} = 1$ and $M_{i,j} = 0$ for all $j \neq i$.

We now have that C (viewed as an OR-circuit) solves $(P, \mathcal{Q})$ if and only if the matrix M satisfies

(a) for all i with $1 \leq i \leq p$ it holds that $M_{i,i} = 1$ and $M_{i,j} = 0$ for all $j \neq i$,
(b) for all i with $p+1 \leq i \leq g$ there exist k and ℓ such that $1 \leq k < \ell < i$ and for all j with $1 \leq j \leq p$ it holds that $M_{i,j} = 1$ if and only if $M_{k,j} = 1$ or $M_{\ell,j} = 1$, and
(c) for every set A in $\mathcal{Q}$ there exists an i with $1 \leq i \leq g$ such that for all j with $1 \leq j \leq p$ it holds that $M_{i,j} = 1$ if $j \in A$ and $M_{i,j} = 0$ otherwise.

Similarly, we have that C (viewed as a SUM-circuit) solves $(P, \mathcal{Q})$ if and only if the matrix M satisfies conditions (a), (c), and

(b') for all i with $p+1 \leq i \leq g$ there exist k and ℓ such that $1 \leq k < \ell < i$ and for all j with $1 \leq j \leq p$ it holds that $M_{i,j} = 1$ if and only if $M_{k,j} = 1$ or $M_{\ell,j} = 1$ and that $M_{k,j} = 0$ or $M_{\ell,j} = 0$.

Based on the above observations, we encode an ensemble computation instance as SAT instance as follows. Given an OR-ensemble $(P, \mathcal{Q})$ and integer g as input, we construct a propositional logic formula φ over variables $M_{i,j}$, where $1 \leq i \leq g$ and $1 \leq j \leq p$, so that any assignment into variables $M_{i,j}$ satisfying φ gives us a matrix that satisfies conditions (a)–(c). We encode condition (a) as

$$\alpha = \bigwedge_{i=1}^{p} \Big(M_{i,i} \wedge \bigwedge_{j \neq i} \neg M_{i,j} \Big) .$$

Similarly, we encode the condition (b) as

$$\beta = \bigwedge_{i=p+1}^{g} \bigvee_{k=1}^{i-2} \bigvee_{\ell=k+1}^{i-1} \bigwedge_{j=1}^{p} \Big((M_{k,j} \vee M_{\ell,j}) \leftrightarrow M_{i,j} \Big) ,$$

and condition (c) as

$$\gamma = \bigwedge_{A \in \mathcal{Q}} \bigvee_{i=p+1}^{g} \Big[\Big(\bigwedge_{j \in A} M_{i,j} \Big) \wedge \Big(\bigwedge_{j \notin A} \neg M_{i,j} \Big) \Big] .$$

The desired formula φ is then $\varphi = \alpha \wedge \beta \wedge \gamma$. For a SUM-ensemble, we replace β with

$$\beta' = \bigwedge_{i=p+1}^{g} \bigvee_{k=1}^{i-2} \bigvee_{\ell=k+1}^{i-1} \bigwedge_{j=1}^{p} \Big(((M_{k,j} \vee M_{\ell,j}) \leftrightarrow M_{i,j}) \wedge (\neg M_{k,j} \vee \neg M_{\ell,j}) \Big) .$$

B. Practical Considerations

There are several optimisations that can be used to tune this encoding to speed up SAT solving. The resulting SAT instances have a high number of symmetries, as any circuit can be represented as a matrix using any topological ordering of the gates. This makes especially the unsatisfiable instances difficult to tackle with SAT solver. To alleviate this problem, we constrain the rows i for $p+1 \leq i \leq g$ appear in lexicographic order, so that any circuit that solves $(P, \mathcal{Q})$ has a unique valid matrix representation. Indeed, we note that the lexicographic ordering of the gate supports (viewed as binary strings) is a topological ordering. We insert this constraint to the SAT encoding as the formula

$$\bigwedge_{i=p+2}^{g} \bigwedge_{k=p+1}^{i-1} \Big[(M_{i,1} \vee \neg M_{k,1}) \wedge \bigwedge_{j_1=2}^{p} \Big(\Big(\bigwedge_{j_2=1}^{j_1-1} (M_{i,j_2} \leftrightarrow M_{k,j_2}) \Big) \rightarrow (M_{i,j_1} \vee \neg M_{k,j_1}) \Big) \Big] .$$

We obtain further speedup by constraining the first t arithmetic gates to have small supports. Indeed, starting from input gates, the ith arithmetic gate in any topological order has weight at most $i+1$. Towards this end, we fix $t = 6$ in the experiments and insert the formula

$$\bigwedge_{i=1}^{t} \bigwedge_{\substack{S \subseteq P \\ |S| = i+2}} \neg \Big(\bigwedge_{j \in S} M_{p+i,j} \Big) .$$

Further tuning is possible if $\mathcal{Q}$ is an *antichain*, that is, if there are no distinct $A, B \in \mathcal{Q}$ with $A \subseteq B$. In this case an optimal circuit C has the property that every gate whose support is in $\mathcal{Q}$ has out-degree 0. Thus, provided that we do not use the lexicographical ordering of gates as above, we may assume that the gates corresponding to sets in $\mathcal{Q}$ are the last gates in the circuit, and moreover, their respective order is any fixed order. This means that if we know that $\mathcal{Q} = \{A_1, A_2, \ldots, A_q\}$ is an antichain, we can replace γ with

$$\bigwedge_{i=1}^{q} \Big[\Big(\bigwedge_{j \in A_i} M_{g-q+i,j} \Big) \wedge \Big(\bigwedge_{j \notin A_i} \neg M_{g-q+i,j} \Big) \Big]$$

to obtain a smaller formula. Finally, we note that we can be combine this with the lexicographic ordering by requiring that only rows i for $p+1 \leq i \leq g-q$ are in lexicographic order.

IV. Instances

In this section, we describe our benchmark instances. These instances were selected so that most of them should be solvable with modern SAT solvers in 500–2000 seconds, though some are more challenging.

Our instances were obtained from a given ensemble $(P, \mathcal{Q})$ and a target number of gates g by first generating a Boolean circuit based on the encoding of Section III. This circuit was then transformed into CNF using the `bc2cnf` encoder (http:

//users.ics.tkk.fi/tjunttil/circuits/), which implements the standard Tseitin encoding [3].

The benchmark instance families themselves are described in detail below.

1) `jkkk-random`: This family was obtained by generating random ensembles $(P, \mathcal{Q})$ by setting $P = \{1, 2, \ldots, p\}$ and drawing uniformly at random a $\mathcal{Q}$ consisting of q subsets of P of size at least 2. We generated 1,000 ensembles for $p = q = 10$ and generated SAT instances for both OR- and SUM-circuits and varying values of g. Finally, we hand-picked instances that matched our criteria.
2) `jkkk-one-one`: This family was generated from an ensemble family parameterised by the number of inputs p, with $P = \{1, 2, \ldots, p\}$ and $\mathcal{Q} = \{P \setminus \{i\} \colon i \in P\}$. The CNFs are generated using the SUM-encoding and the antichain optimisation. The selected instances have $p = 10$ or $p = 11$, and g was again chosen so that the instances were solvable in about 500–2000 seconds.
3) `jkkk-challenge`: These very large instances are based on a generalisation of the construction used in the `jkkk-one-one` instances; we omit the details. We note that we were not able to solve them in our own experiments.

Acknowledgments

Work supported in part by Academy of Finland (grants 132812 and 251170 (MJ), 252083 and 256287 (PK), and 125637 (MK)), and by Helsinki Doctoral Programme in Computer Science - Advanced Computing and Intelligent Systems (JK).

References

[1] M. Järvisalo, P. Kaski, M. Koivisto, and J. H. Korhonen, "Finding efficient circuits for ensemble computation," in *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 369–382.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[3] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*. Springer, 1983, pp. 466–483.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Fixed-shape Forced Satisfiable CNF Benchmarks

Anton Belov
Complex and Adaptive Systems Laboratory
University College Dublin, Ireland

***Abstract*—This note describes the `fixed-shape-forced` benchmark class submitted to the Hard Combinatorial track of SAT Challenge 2012.**

I. Introduction

The class of *fixed-shape random formulas* was introduced in [1] as a generalization of uniform random CNF formulas to the domain of non-clausal propositional formulas in the Negation Normal Form (NNF). The structure of the formulas is determined by the following parameters: the number of variables n, the number of *hyper-clauses* m, and the *shape* of hyper-clauses given by a tuple of positive integers $\langle s_1, \ldots, s_k \rangle$. Each hyper-clause of shape $\langle s_1, \ldots, s_k \rangle$ can be viewed as a k-level balanced tree with alternating levels of disjunctions and conjunctions, where the i-th level is a s_i-child disjunction if i is odd ($i = 1$ is the root of the tree), and s_i-child conjunction if i is even. The leafs of the tree a drawn uniformly from the set of $2n$ literals on the n variables, avoiding the complimentary literals that share the same parent. The final formula is constructed by taking m such hyper-clauses.

In [1], authors performed a detailed empirical study of fixed-shape random formulas of certain shapes, and conjectured phase transitions at particular ratios of hyper-clauses to variables, similar to that of uniform random CNF formulas. The formulas with $n = 200$, $m = 560$, and shape $\langle 3, 3, 2 \rangle$, as well as $n = 300$, $m = 354$, and shape $\langle 2, 2, 3, 2 \rangle$ were used in [2] to evaluate the performance of non-clausal SLS-based solver `polSAT`.

II. Submitted Instances

The set of 110 `fixed-shape-forced` CNF benchmarks submitted to SAT Challenge 2012 was generated by applying Plaisted-Greenbaum CNF transformation [3] to a set of fixed-shape non-clausal formulas with $n = 300$, $m = 354$, and shape $\langle 2, 2, 3, 2 \rangle$. The CNF transformation was optimized to avoid the introduction of the unnecessary variables, as described in [1]. In addition, the generated formulas were *forced* to be satisfiable by seeding them with two complimentary assignments — this technique was suggested in [4] to produce forced satisfiable formulas that are difficult for local-search methods. Thus, while the generated CNF formulas do exhibit a lot of "randomness", at the same time they possess certain circuit-like structure. The tool used for generation of formulas — `forgen` — is available for download at http://anton.belov-mcdowell.com. The tool can also be used to generate other classes of non-clausal, and the corresponding CNF, formulas.

References

[1] J. Navarro and A. Voronkov, "Generation of hard non-clausal random satisfiability problems," in *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005)*, 2005, pp. 436–442.

[2] Z. Stachniak and A. Belov, "Speeding-up non-clausal local search for propositional satisfiability with clause learning," in *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT 2008)*, 2008, pp. 257–270.

[3] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, 1986.

[4] D. Achlioptas, H. Jia, and C. Moore, "Hiding satisfying assignments: Two are better than one," *Artificial Intelligence Research*, vol. 24, no. 1, pp. 623–639, 2005.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

Solving Logic Puzzles with SAT

Norbert Manthey and Van Hau Nguyen
Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

***Abstract*—This description explains how the submitted SUDOKU and HIDOKU puzzles have been generated.**

I. Motivation

Brain jogging has received attention recently by the popular puzzle SUDOKU(see Figure 1). The puzzles are based on very simple constraints. A SUDOKU of size n is a square with n times n fields. In each field there needs to be a number between 1 and n, where for each column, row and block no number appears twice. With a few simple deduction rules those puzzles can be solved. Most often, there exists only a single solution, so that there even is no need to guess the value of certain fields.

A very familiar looking puzzles are HIDOKUs (see Figure III). Again, there is a square of n time n fields. The numbers from 1 to n^2 have to be assigned to the fields such that each number appears exactly once. Furthermore, if a field contains number i, then one of its eight neighboring fields needs to contain the number $i + 1$. Differently to SUDOKUs, HIDOKUs are directly related to the Hamilton path, which also requires to visit each node in a graph exactly once. The neighborhood relation is defined by the edges in the graph. Thus, HIDOKUs can be seen as a special case of the Hamilton path problem.

Encoding HIDOKU to SAT has been done for generating solvable puzzles. A formula describing an empty HIDOKU is generated and afterwards solved by a SAT solver. The solution is then shrinked so that a partially puzzle remains. During that task it has been noticed that by increasing the size of the puzzle, the solving time does not increases linear.

II. Encoding Sudokus

Solving Sudokus using SAT solvers is a well known technique [1]. The direct encoding was chosen to encode the SUDOKUs to SAT [4]. To encode all rules of the puzzle, for each cell the *at-least-one* constraint and the *at-most-one* constraint have been encoded. For each column, row and block only the *at-most-one* constraint has been encoded.

Based on this encoding, the generation of the SUDOKUs has been done according to the following steps: (1) generating a random Sudoku of size 9, (ii) remove values from fields randomly and (iii) measure the time to solve the partially filled SUDOKU. If this solving time is below a certain *threshold*, more cells are erased, otherwise the SUDOKU is kept.

The given instances encodes a Sudoku, where the size of a small square is 9×9. The overall Sudoku has a size 81×81. Surprisingly, the given Sudoku is very hard to solve for CDCL solvers, although similar puzzles with a similar size can be solved quite easily. The submitted instance has been preprocessed with COPROCESSOR 2 by using only equivalence preserving techniques [2].

				2				
1								3
			1					4
						1		
		1			2			
				1				7
					1	9		8
						5		
							6	

Fig. 1. SUDOKU with size 9 times 9

III. Encoding Hidokus

Based on the hamilton path encoding from [3], a formula F can be generated that represents an empty HIDOKUof size n. The encoding uses additional variables that indicate possible successors and predecessors in the Hamiltonian cycle. The following Boolean variables are used: the variable $index(x, y, z) = 1$ is assigned *true* whenever the cell (x, y) is assigned a value v. If the cell (x_2, y_2) is the successor of cell (x_1, y_1), then the variable $next(x_1, y_1, x_2, y_2) = 1$. Therefore, we specify the set of cell neighbors M. For each cell (x, y), the set $M_{(x,y)}$ contains all the neighboring cells in the grid.

Similar as for SUDOKU, each cell can contain only a single value (between 1 and n^2), because it can have only a single position in the path. This fact is encoded by using the *at-most-k* constraint. Usually, as also used in [3], for m elements the naive encoding uses a quadratic amount of clauses ($\frac{m(m-1)}{2}$). We replaced this encoding by a nested encoding of the *at-most-k* constraint by introducing new variables. By doing so, the number of clauses is reduced to $3m$.

To encode the neighborhood relationship of cells, the following implication is generated for each neighboring cell pair and value:

$$index(x_1, y_1, z) \wedge next(x_1, y_1, x_2, y_2) \Longrightarrow index(x_2, y_2, z + 1),$$

which is equivalent to

$$\bigwedge_{\substack{x_1=1,\\ y_1=1}}^{n} \bigwedge_{\substack{(x_2,y_2)\in\\ M_{(x_1,y_1)}}} \bigwedge_{z=1}^{n^2-1} (\neg index(x_1,y_1,z) \vee \neg next(x_1,y_1,x_2,y_2) \vee index(x_2,y_2,z+1)).$$

Likewise, the backward adjacency constraints are encoded:

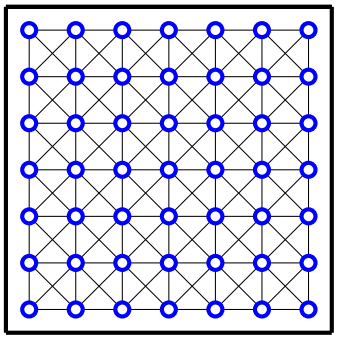
$$\bigwedge_{\substack{x_1=1,\\ y_1=1}}^{n} \bigwedge_{\substack{(x_2,y_2)\in\\ M_{(x_1,y_1)}}} \bigwedge_{z=2}^{n^2} (\neg index(x_1,y_1,z) \vee \neg next(x_1,y_1,x_2,y_2) \vee index(x_2,y_2,z-1)).$$

Encoding an HIDOKU of size n with the usual *at-most-k* encoding requires $n^4 + 8n^2$ variables and $\frac{1}{2}n^6 + 8n^4$ clauses. A model for this formula F contains the position of each cell in the HIDOKU, which can be extracted by the variables $index(x, y, z)$. Thus, encoding a partially filled HIDOKU can be done by adding these values as unit clauses to the formula as well.

The submitted HIDOKU-instances encode empty HIDOKUs of various sizes. From the authors experience, the solving time for these puzzles (or Hamilton path problems) does not scale polynomial with their size, although there exist many solutions per instance and from humans there are obvious ways how to solve these empty puzzles.

47				**34**		**1**
48	**40**			**43**		**32**
	49		**44**			
				28		**30**
10					**29**	
		17	**20**	**15**	**22**	**25**
						24

(a) HIDOKU



(b) Hamilton Path Problem

Fig. 2. The human solvable HIDOKU (a) and its representation as Hamilton path (b)

REFERENCES

[1] Inês Lynce and Joël Ouaknine. Sudoku as a sat problem. In *ISAIM*, 2006.

[2] Norbert Manthey. Coprocessor 2.0 – A flexible CNF Simplifier (Tool Presentation), 2012. Submitted to SAT 2012.

[3] Miroslav N. Velev and Ping Gao. Efficient sat techniques for absolute encoding of permutation problems: Application to hamiltonian cycles. In *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009*. AAAI, 2009.

[4] Toby Walsh. Sat v csp. In *in Proc. CP-2000*, pages 441–456. Springer-Verlag, 2000.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

sgen4: A generator for small but difficult satisfiability instances

Ivor Spence
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
i.spence@qub.ac.uk

I. Introduction

This is a development of the previous generators sgen1 [1] and sgen2 [2] and generates both satisfiable and unsatisfiable instances. The instances are not derived solving any particular problem but are crafted to be as difficult as possible to solve. In all cases the variables in the generated instance are partitioned into small groups and for each group there are clauses to define relationships among the group. This is repeated for two or more partitions.

II. Unsatisfiable instances

Generating small yet difficult unsatisfiable instances requires a balance between the following constraints:

- To keep the instance short, each clause should eliminate a large number of possibilities.
- To make the instance hard to solve, the variables in each clause should not be "related", that is occur together in other clauses.

Unfortunately, these constraints conflict. Having unrelated variables tends to preclude a clause eliminating a large number of possible assignments. In the spirit of Hirsch's hgen8 program the compromise used here is to partition the variables into groups of size four and five (in two different partitions) and have multiple clauses re-use the variables in each group.

The basic technique here is identical to previous versions of the generator. In each partition one group contains five variables and the remaining groups each contain four. Thus the number of variables must be of the form $4g + 1$ where $g \in \mathbb{N}$. If the number of variables requested is not of this form the generator will use the next larger possibility - thus the generated instance will contain at least the requested number of variables, but may contain up to three more.

Within each group of four variables the idea is to permit at most two variables to be `false` For each group of size four, generate all possible 3-clauses of positive literals, i.e.

$$(a \vee b \vee c) \wedge (a \vee b \vee d) \wedge (a \vee c \vee d) \wedge (b \vee c \vee d)$$

This permits at most two variables from the set $\{a, b, c, d\}$ to be `false`. For the group of size five again generate all possible 3-clauses of positive literals (10 clauses), meaning that again only two variable from this group can be `false`. In total therefore, only $2(g-1)+2 = 2g = (n-1)/2$ variables can be `false`.

Now partition the variables into a different collection of $(g-1)$ groups of size four and one of size five and again for each group of variables generate all possible 3-clauses except this time use all negative literals. Thus now only $(n-1)/2$ variables can be `true`. Taking these two sets of clauses together it can be seen that it is not possible to assign a value to every variable since at most $(n-1)/2$ can be `true` and also at most $(n-1)/2$ can be `false`. Thus the generated instance is unsatisfiable.

If the number of variables is $4g + 1$ then the number of groups is g and the total number of clauses is $8g + 12$. Each clause has three literals, so if the number of variables is n, then as n increases the number of clauses is approximately $2n$ and the number of literals is approximately $6n$.

III. Satisfiable instances

To generate difficult satisfiable instances we use three partitions of the variables, all in groups of size five. Clauses generated from the first partition permit at most one variable from each group to be positive and clauses from the second and third partitions both require at least one variable per group to be positive. In principle therefore the instance may be satisfiable.

If the number of groups is g (so that $n = 5g$) then the instance will contain $10g$ binary clauses (from the first partition) and $2g$ 5-clauses (g from each of the second and third paritions), giving a total of $12g = 12n/5$ clauses and $6n$ literals (see Table I).

For each group in the first partition, we generate all possible binary clauses of negative literals (10 clauses for each group). This permits at most one `true` variable per group, that is a maximum of $g = n/5$ `true` variables overall.

For each group in the second and third partitions, we generate one 5-clause of all the positive literals. The $n/5$ `true` variable permitted by the first set of clauses might be enough to satisfy these subsequent clauses if they can be allocated as one per group.

If more and more collections of 5-clauses of positive literals are added it is less and less likely that the formula will remain satisfiable. Empirical results indicate that two collections give the most difficult instances for their size.

IV. Partitioning

For both satisfiable and unsatisfiable cases, to create difficult instances we need to ensure that there is as little connection as

possible between the different partitions. For the first partition natural ordering is used, e.g. $\{1,2,3,4\},\{5,6,7,8\},\ldots$. Second and subsequent partitions are obtained by using simulated annealing, with a weight function which tries to minimise the extent to which the original partition is reflected in the second one. The difference between sgen4 and earlier versions lies in this function, which now tries to minimise the correlation between the second and third partitions as well as between first and second and between first and third. It is anticipated that this may make instances even harder to solve, but there is not yet sufficient empirical evidence to confirm this.

To ensure that satisfiable instances are created when requested, i.e. that it is possible to choose the g positive variables so that there is one per group in each of the three partitions, the technique used is to make an initial choice of the $n/5$ `true` variables and restrict the partitioning process to keep these variables in different groups. If the option -m model-file is chosen when generating a satisfiable instance, a satisfying model will be written to model-file.

Different partitions can be forced by using -s to specify the seed for the random number generation. If -s is omitted a value of 1 is used.

V. Parameters

The possible parameters are:

- sat — Requests that a satisfiable instance be generated.
- unsat — Requests that an unsatisfiable instance must be generated. Exactly one of `sat` and `unsat` must be specified.
- reorder — Having generated the clauses as described up, a random permutation of variables and clauses is applied. This option is enabled by default.
- n — Specify the minimum number of variables to be generated. Mandatory.
- s — Specify a seed for random number generation. Defaults to 1.
- m — Specify a filename for a satisfying model to be written to. Requires `sat` to be specified.

For example, typical invocations might be:

```
sgen4 -unsat -n 49 >u49.cnf
sgen4 -sat -n 120 -m s120.cnf >s120.cnf
```

VI. Results

Table I gives an example of a 9-variable unsatisfiable formula and a 10-variable satisfiable one.

Unsatisfiable (9 variables)	Satisfiable (10 variables)
p cnf 9 28	p cnf 10 24
-2 -3 -4 0	-1 -2 0
-1 -3 -4 0	-1 -3 0
-1 -2 -4 0	-1 -4 0
-1 -2 -3 0	-1 -5 0
-5 -6 -7 0	-2 -3 0
-5 -6 -8 0	-2 -4 0
-5 -6 -9 0	-2 -5 0
-5 -7 -8 0	-3 -4 0
-5 -7 -9 0	-3 -5 0
-5 -8 -9 0	-4 -5 0
-6 -7 -8 0	-6 -7 0
-6 -7 -9 0	-6 -8 0
-6 -8 -9 0	-6 -9 0
-7 -8 -9 0	-6 -10 0
4 2 6 0	-7 -8 0
7 2 6 0	-7 -9 0
7 4 6 0	-7 -10 0
7 4 2 0	-8 -9 0
3 8 5 0	-8 -10 0
3 8 9 0	-9 -10 0
3 8 1 0	9 6 3 5 1 0
3 5 9 0	4 2 8 10 7 0
3 5 1 0	7 5 3 9 2 0
3 9 1 0	10 4 8 1 6 0
8 5 9 0	
8 5 1 0	
8 9 1 0	
5 9 1 0	

TABLE I
Example instances

References

[1] I. Spence, "sgen1: A generator of small but difficult satisfiability benchmarks," *J. Exp. Algorithmics*, vol. 15, pp. 1.2:1.1–1.2:1.15, mar 2010. [Online]. Available: http://doi.acm.org/10.1145/1671970.1671972

[2] A. V. Gelder and I. Spence, "Zero-one designs produce small hard sat instances," in *SAT*, ser. Lecture Notes in Computer Science, O. Strichman and S. Szeider, Eds., vol. 6175. Springer, 2010, pp. 388–397.

Appears in A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz (eds.), *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of Department of Computer Science Series of Publications B, University of Helsinki 2012. ISBN 978-952-10-8106-4

SAT Instances for Traffic Network Scheduling Problems

Peter Großmann
Faculty of Transport and Traffic Science
Technische Universität Dresden, 01062 Dresden, Germany
pg@janeway.inf.tu-dresden.de

Norbert Manthey
Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

I. Motivation

Many real-world applications like computing a time table for a given railway network or setting up a traffic light systems are based on periodic events and constraints imposed on these events. Events and their constraints can be modeled by so-called periodic event networks. The periodic event scheduling problem (PESP) consists of such a network and is the question, whether all the events can be scheduled such that a set of constraints – specified by the network – is satisfied. The problem is $\mathcal{NP}$-complete [5] and the currently best solutions are obtained by constraint-based solvers notably PESPSOLVE [4], [6] or by LP solvers, which solve linearized PESP instances by introducing modulo parameters [3]. However, these solvers are still quite limited in the size of the problem which they can tackle. For example, PESPSOLVE is able to schedule the inter city express trains, but cannot schedule all passenger trains in Germany.

II. Encoding PESP

Encoding a PESP instance into SAT requires to encode all its constraints, namely the departure of a vehicle from a certain node in the network. Furthermore, time consuming constraint have to be considered. These constraints state the time a vehicle needs to travel between two nodes, which has to be traveled conflict free and with a correct route. Thus, a constraints includes for each node the departure times for all vehicles, which have to be found, such that all the conditions concerning the network are met. The encoding is described in more details in [2], [1].

An encoded SAT instance is satisfiable if a valid periodic departure time schedule for all the nodes in the network exist. In that case an instance can have multiple solutions. If no conflict free departure time schedule exists, the SAT instance is unsatisfiable and vice versa.

III. Solving PESP

To the best knowledge of the authors, PESP is typically solved with classical search strategies, e.g. CSP and decision trees or mixed integer problems. Empirical studies showed that the SAT encoded problem can be solved orders of magnitudes faster. Still, there exist traffic networks that are very challenging for SAT solvers.

The submitted instances encode parts of the railway network of Germany, following [2]. The suffixes of the instances can be divided in the following categories:

- Small hard subnetworks (SHN)
- Large complex networks (LCN).

For SHN, the number of nodes ranges from 23 to 76 and the number of arcs is about 13 times larger. The LCN instances have many more nodes, ranging from 500 to 7000 with a number of arcs between 7000 and 50000. Out of the available scheduling problems we selected a subset that is harder to solve than the average of the instances.

References

[1] Peter Großmann. Polynomial Reduction from PESP to SAT. Technical Report 4, Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, October 2011.

[2] Peter Großmann, Steffen Hölldobler, Norbert Manthey, Karl Nachtigall, Jens Opitz, and Peter Steinke. Solving Periodic Event Scheduling Problems with SAT. In *IEA AIE, accepted*, 2012.

[3] 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. Springer-Verlag, 2007.

[4] Karl Nachtigall. *Periodic Network Optimization and Fixed Interval Timetable*. Habilitation thesis, University Hildesheim, 1998.

[5] Michiel A. Odijk. *Construction of periodic timetables, Part 1: A cutting plane algorithm*. 1994.

[6] Jens Opitz. *Automatische Erzeugung und Optimierung von Taktfahrplänen in Schienenverkehrsnetzen*. Reihe: Logistik, Mobilität und Verkehr. Gabler Verlag — GWV Fachverlage GmbH, 2009.

Solver Index

Benchmark Index

Author Index