Proceedings of

SAT COMPETITION 2018
Solver and Benchmark Descriptions

Marijn J. H. Heule, Matti Järvisalo, and Martin Suda *(editors)*

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2018-1

Helsinki 2018

PREFACE

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

SAT Competition 2018 (SC 2018; `http://sat2018.forsyte.tuwien.ac.at`), a competitive event for SAT solvers, was organized as a satellite event of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT 2018), Oxford, UK, as part of the Federated Logic Conference (FLoC 2018). SC 2018 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held 2002-2005, biannually during 2007-2013, and 2014, 2016-2017; the SAT-Races held in 2006, 2008, 2010, and 2015; and SAT Challenge 2012.

SC 2018 consisted of four tracks: In addition to the Main Track for sequential SAT solvers, a Random SAT track (for solvers focusing on efficiently solving satisfiable randomly generated SAT instances), Parallel track (for parallel SAT solvers designed for computers with multiple CPUs or CPU cores), and a special No-Limits track where solver source code and solution certificates are not required and portfolios of any nature are allowed) were organized. Additionally, "Glucose hacks", i.e., solvers based on small modifications to the Glucose 3.0 CDCL SAT solver, were encouraged to be submitted to the main track with the intention of separately awarding the best Glucose hack.

There were two ways of contributing to SC 2018: by submitting one or more solvers for competing in one or more of the competition tracks, and by submitting interesting benchmark instances on which the submitted solvers could be evaluated on in the competition. Following the tradition put forth by SAT Challenge 2012, the rules of SC 2018 invited all contributors to submit a short, 1-2 page long description as part of their contribution. This book contains these non-peer-reviewed descriptions in a single volume, providing a way of consistently citing the individual descriptions.

Participants of the main track of the SAT Competitions are required to submit at least ten interesting benchmarks since 2017. This policy resulted in a significant increase in the number of submitted formulas. The organizers planned to modify all known benchmarks in the 2018 suite in a satisfiability-preserving way to avoid overfitting. However, the number of submitted benchmarks allowed selecting a benchmark suite of 400 formulas that consists only of unknown (submitted in 2018 or earlier, but never used) instances. The hardness of all submitted instances was determined by average the solving time of a mix of SAT Competition 2017 solvers. The selected benchmark suite is a balanced mixed between medium, hard, and challenging (too hard) instances as well as an intended 50%-50% mix of SAT and UNSAT instances. The exact balance cannot be determined as several formulas are too hard for existing solvers.

Successfully running SC 2018 would not have been possible without active support from the community at large. We would like to thank the StarExec initiative (`http://www.starexec.org`) for the computing resources needed to run the Main, No-Limits, and Random tracks. Many thanks go to Aaron Stump and Patrick J. Hawks for their invaluable help in setting up StarExec to accommodate for the competition's needs. We also acknowledge the Texas Advanced Computing Center (TACC, `http://www.tacc.utexas.edu`) at The University of Texas at Austin for providing grid resources for running the Parallel Track. Finally, we would like to emphasize that a competition does not exist without participants: we thank all those who contributed to SC 2018 by submitting either solvers or benchmarks and the related description.

Marijn J. H. Heule, Matti Järvisalo, & Martin Suda
SAT Competition 2018 Organizers

Contents

SOLVER DESCRIPTIONS

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

AbcdSAT and Glucose hack: Various Simplifications and Optimizations for CDCL SAT solvers

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

Abstract—**This article focuses on the decision variable branching heuristic and learnt clause maintenance for CDCL (conflict-driven clause-learning) SAT solvers. We improve the existing variable branching heuristics and learnt clause reduction via various simplification tricks. On the basis of that, we develop multiple improved versions of abcdSAT and a hack version of Glucose, which are submitted to main, no-limit, Glucose hack and parallel track at the SAT Competition 2018.**

I. Introduction

In this article, we focus on how to simplify the existing techniques such as variable branching heuristic and learnt clause reduction etc. VSIDS (Variable State Independent Decay Sum) is a prevalent decision variable branching policy. The learning rate based branching heuristic (LRB) [9] proposed in the recent year can outperform in some cases. So we decide to simplify the LRB heuristic. The existing three-tiered learnt clause management scheme is sophisticated and difficult to use. Here we simplify it into a one-tiered learnt clause management scheme so that a general CDCL SAT solver can use it also. In addition, we introduce a new re-learning technique.

II. Simplifying Learning Rate Based Branching Heuristic

The learning rate based branching heuristic (LRB) [9] is a variant of the conflict history-based branching heuristic (CHB) [8]. The score A_v of each variable v is computed using the following reinforcement learning formula.

$$A_v = (1-\alpha)A'_v + \alpha r_v$$

where A'_v is the old score of variable v. The difference between LRB and CHB is the computation of r_v. In CHB, r_v is a constant that is either 1 or 0.9. However, in LRB, r_v is defined as

$$r_v = \frac{C_v + P_v}{T}$$

where C_v and P_v is the number of conflict clauses and reason clauses v participated in since v is assigned, and T is the interval time that is defined as $T = conflictCounter - assigedTime[v]$. Our SAT solver modifies r_v as

$$r_v = \frac{C'_v + S_v + P'_v}{T'}$$

where C'_v, S_v and P'_v is the number of conflict clauses, seen clauses and reason clauses v participated in since v is picked, $T' = conflictCounter - pickedTime[v]$. Notice, the time v is assigned is not necessarily equal to the time it is picked. In general, S_v and C'_v overlap and interweave. Computing $C'_v + S_v + P'_v$ is easier than computing $C_v + P_v$. In analyzing a conflict clause, we compute C'_v. In collecting reason clauses, we compute $S_v + P'_v$. In the detailed implementation, we use one counter to store the value of $C'_v + S_v + P'_v$.

III. Simplifying hybrid Branching Heuristic

Many CDCL SAT solvers use two branching heuristics LRB and VSIDS (Variable State Independent Decay Sum) to pick a branching variable. To maintain the priority of variables, they construct two order heap data structures, which are called $order\_heap\_VSIDS$ and $order\_heap\_LRB$, respectively. Our SAT solver uses also mixed heuristics LRB and VSIDS. However, we merge two order heaps into one order heap called $order\_heap$. VSIDS mode and LRB mode build $order\_heap$ via VSIDS scoring scheme and LRB scoring scheme, respectively. such merging has no impact on the solving performance.

IV. Simplifying LBD based three-tiered learnt clause management scheme

LBD (literal block distance) is defined as the number of decision variables in a clause. According to LBD values, the CoMiniSatPS [6] solver classifies learnt clauses into three categories *low*-LBD, *mid*-LBD and *high*-LBD. The *low*-LBD clause is also called *core* clause, whose LBD value is less than 4. The *high*-LBD clause is also called *local* clause, whose LBD value is greater than 6. CoMiniSatPS uses three lists *learnts_core*, *learnts_tier2* and *learnts_local* to store separately learnt clauses. Our SAT solver uses also the three-tiered learnt clause management scheme [7]. However, we merge three lists into one list calles *learnts*, and set a mark for each learnt clause to distinguish which category a learnt clause belongs. In the other words, although we use the same learnt clause management scheme as CoMiniSatPS, our data structure is simpler than that of CoMiniSatPS. Except that each clause has an additional mark, our data structure storing learnt clauses is the same as that of Glucose.

CoMiniSatPS has two database maintenance subroutines. One tries to halve the number of *local*-tier learnt clauses at every 15,000 conflicts. The other checks *mid*-tier clauses for reduction at every 10,000 conflicts. The *mid*-tier clauses not used in the past 30,000 conflicts are moved to *local* -tier.

We merge such two subroutines into one. In details, at every 15,000 conflicts, we halve the number of *local* -tier learnt clauses, and move simultaneously *mid*-tier clauses not used in the past 26,000 conflicts to *local* -tier.

V. RE-LEARNING

Our re-learning notion is similar to learnt clause minimization (LCM) given in [10]. Let $\mathcal{F}$ be a current formula, $C = x_1 \vee x_2 \vee \cdots \vee x_n$ be a learnt clause. The basic idea of LCM is that if $(\mathcal{F} - C) \cup \{\neg x_1, \neg x_2, \ldots, \neg x_n\}$ derives an empty clause, we can obtain a new learnt clause L by analyzing this conflict. If the LBD value of L is smaller than that of C, we replace C with L. In [10], this idea is applied to only the learnt clauses with small LBD. In our re-learning policy, we apply this idea to all the learnt clauses. The other difference between LCM in [10] and our re-learning policy is that we can remove some redundant learnt clauses by a subsumption operation on-the-fly, while LCM in [10] cannot. As a inprocessing, our re-learning policy cannot be applied to the whole solving procedure. We apply the re-learning policy only when the number of conflicts is less than 5×10^5.

VI. ABCDSAT *r18*

This is submitted to the main track. Compared to abcdSAT *r17* [5], it adds a simplified three-tiered learnt clause management scheme, a hybrid branching heuristic and a re-learning strategy given above. The solver runs in the Minisat-VSIDS [3] scoring scheme for the first 5×10^4 conflicts. Afterwards the scoring scheme is switched to the LRB scoring scheme. When the number of conflicts reaches 5×10^6 for a large formula or 1.5×10^7 for a small formula, the scoring scheme is switched to the Glucose-VSIDS [4] scoring scheme. Like the *r17* version, in the tree-based search, the solver produces also DRAT proofs. However, its tree-based search branching is different from that of the *r17* version. This version uses two scoring policies ACE (Approximation of the Combined lookahead Evaluation) [2] and LRB. In details, it selects a tree node variable using ACE scores when the average LBD is greater than 11, and LRB scores otherwise. In general, the solver does not adopt a bit-encoding phase selection strategy [1] except that it runs in the Glucose-VSIDS scoring scheme.

VII. SMALLSAT

Smallsat is a simplified version of abcdSAT *r18*. It is also submitted to the main track. Compared to abcdSAT *r18*, it removes inprocessing techniques such as lifting, probing, distillation, elimination, complex hyper binary resolution, equivalent literal search, unhiding redundancy etc. The variable branching heuristic based on blocked clause decomposition is given up also. This solver has no Glucose-VSIDS scoring scheme. It runs in the Minisat-VSIDS scoring scheme when the number of conflicts is less than 5×10^4 or greater than 1.8×10^7, and in in the LRB scoring scheme otherwise. It contains the tree-based search, but uses only the LRB scoring policy, excluding the ACE scoring policy.

VIII. ABCDSAT *n18*

This is similar to abcdSAT *r18*, but does not output a DRAT proof. So this solver is submitted to the no-limit track. Except for the symmetry breaking preprocessing and XOR Gaussian elimination etc, the simplification technique used in this solver is basically the same as one used in abcdSAT *r18*. Like the previous no-limit version [5], it divides the whole solving process into three phases. In the first phase, it uses a Minisat-VSIDS scoring scheme and a LRB scoring scheme to search a solution. The second phase simplifies the formula generated in the first phase, using various simplification technique including XOR and cardinality constraint simplification. The second phase uses a Glucose-VSIDS scoring scheme and a LRB scoring scheme to solve the simplified problem. The tree-based search is almost same as that in abcdSAT *r18*.

IX. ABCDSAT *p18*

This is a parallel version of abcdSAT *n18*. Compared with the last year's the parallel version, this year's version does not use the master-thread to solve the original problem. This solver uses at most 25 threads. Let the i-th pivot be p_i, and input formula F. 20 out of 25 threads solve the subproblem $F \wedge p_i$. The other 4 threads solve either the original problem or the simplified problem. Once the thread of a subproblem ended, we re-use it to solve the simplified problem with learnt clauses generated so far. We use C++ *pthread_cond_timedwait* to detect which thread has terminated already. However, it failed sometimes to detect. This may be a C++ bug. For this reason, we use also the solving status of each thread to detect whether a thread has ended already. Only one thread applies bit-encoding phase selection strategies [1], The symmetry breaking preprocessor is also applied to only another thread. Except the two policies, abcdSAT *p18* uses almost the same inprocessing techniques as version *p17* [5].

X. *glu_mix*

glu_mix is a hack version of Glucose. It made modifications on learnt clause management and branching heuristic. *glu_mix* uses a simplified three-tiered learnt clause management scheme and a hybrid branching heuristic given above. It uses a VSIDS scoring scheme when the number of conflicts is less than 2×10^6 or greater than 2×10^7, and a LRB scoring scheme otherwise. To be able to implement simultaneously such two policies in edit distance 1000 benefits from our simplification techniques here.

REFERENCES

[1] J.C. Chen:A bit-encoding phase selection strategy for satisfiability solvers,in *Proceedings of Theory and Applications of Models of Computation (TAMC'14)*, ser. LNCS 8402, 2014, pp. 158–167.

[2] Chen, J.C.: Building a Hybrid SAT Solver via Conflict-driven, Lookahead and XOR Reasoning Techniques, SAT 2009.

[3] N. Eén, N. Sörensson: An extensible SAT-solver, SAT 2003. LNCS, vol. 2919, 2004, pp. 502–518.

[4] G. Audemard and L. Simon: predicting learnt clauses quality in modern SAT solvers, IJCAI 2009.

[5] J.C. Chen: Improving abcdSAT by Weighted VSIDS Scoring Schemes and Various Simplifications, Proceedings of the SAT Competition 2017.

[6] C. Oh: Between SAT and UNSAT: The fundamental difference in CDCL SAT, SAT 2015.
[7] C. Oh: Patching MiniSat to Deliver Performance of Modern SAT Solvers, Proceedings of the SAT Competition 2015.
[8] J.H. Liang, V. Ganesh, P. Poupart, K. Czarnecki: Exponential Recency Weighted Average Branching Heuristic for SAT Solvers, AAAI 2016.
[9] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki: Learning Rate Based Branching Heuristic for SAT Solvers, SAT 2016.
[10] F. Xiao, M. Luo, C.M. Li, F. Manyà, Z.P. Lü: MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width in SAT Competition 2017, Proceedings of the SAT Competition 2017.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018

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

***Abstract*—This note documents the versions of our SAT solvers submitted to the SAT Competition 2018, which are CaDiCaL, Lingeling, its two parallel variants Treengeling and Plingeling, and our local search solver YalSAT.**

Lingeling,Plingeling, Treengeling,YalSAT

Compared to the version of Lingeling submitted last year [1] we added *Satisfication Driven Clause Learning* (SDCL) [2], which however due to its experimental nature is disabled (option "`--prune=0`"). We further disabled blocked clause removal (option "`--block=0`") [3], binary blocked clause addition (option "`--bca=0`") [4], as well as on-the-fly subsumption (option "`--otfs=0`") [5], since all of them can not be combined with SDCL style pruning.

As in our new version of CaDiCaL we also experimented with bumping reason side literals too, as suggested in [6]. See below for the motivation to include this feature. There is also a slight change in the order how literals are bumped: previously they were bumped in trail order and are now bumped in variable score order.

Since already last year's version of Lingeling [1] was almost identical to that from the SAT 2016 Competition [7], it is fair to say that Lingeling and also its parallel extensions Plingeling and Treengeling essentially did not change since 2016. This applies even more to the submitted version of YalSAT, which surprisingly won the random track in 2017, even though it did not change since 2016.

CaDiCaL

As explained in our last year's solver description [1] the goal of developing CaDiCaL was to produce a radically simplified CDCL solver, which is easy to understand and change. This was only partially achieved, at least compared to Lingeling. On the other hand the solver became competitive with other state-of-the-art solvers, actually surpassing Lingeling in performance in the last competition, while being more modular, as well as easier to understand and change.

We also gained various important new insights starting to develop a SAT solver (again) from scratch [1], particularly how inprocessing attempts for variable elimination and subsumption should be scheduled, and how subsumption algorithms can be improved (see again [1] for more details).

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

On the feature side not much changed, since CaDiCaL still does not have a complete incremental API (assumptions are missing). However, the non-incremental version was used as back-end of Boolector in the SMT 2017 Competition [8] in the quantifier-free bit-vector track (QF_BV), where it contributed to the top performance of Boolector (particularly compared to the version with Lingeling as back-end).

Our analysis of the SAT 2017 Competition [9] results revealed that the technique of *bumping reason side literals* [6] of MapleSAT [6] and successors [10], [11] has an extremely positive effect on the selected benchmarks. It consists of going over the literals in learned (minimized 1st UIP) clauses and "bumping" [12] all other literals in their reason clauses too. Even though MapleSAT actually only uses this technique with the new variable scoring scheme proposed in [6], it is already effective in combination with the VMTF scheme [12] used in CaDiCaL (and probably for VSIDS too).

Last year's success of the MapleLCM solver [11], which is an extension of MapleSAT by a different set of authors, also showed that vivification [13] of *learned* clauses as described by the authors of MapleLCM in their IJCAI paper [14] can be quite useful. In last year's version of CaDiCaL we already had a fast implementation of vivification [1], but only applied it to *irredundant* clauses. During inprocessing [15] our new version of CaDiCaL has two vivification phases, the first phase working on all including *redundant* clauses and the second phase works as before only on irredundant clauses.

Furthermore, all the top performing configurations of MapleSAT and MapleLCM made use of the observation of Chanseok Oh [16], that a CDCL solver should alternate between "quiet" no-restart phases and the usual fast restart scheduling [17]. This also turns out to be quite beneficial for last year's selection of benchmarks and we added such "stabilizing" phases scheduled in geometrically increasing conflict intervals.

Then, we experimented with "rephasing", which in arithmetically increasing conflict intervals overwrites all saved phases [18] and either (i) restores the initial phase (default *true* in CaDiCaL), (ii) flips the current saved phase, (iii) switches to the inverted initial phase (thus *false*), or (iv) picks a completely random phase. This technique gives another (but smaller) boost to the performance of CaDiCaL on last year's benchmarks compared to the other new techniques above.

Finally, we observed, that for very long running instances (taking much longer than the 5000 seconds time limit used in the competition), the standard arithmetic increase [19] of the limit on kept learned clauses increases memory usage over time substantially and slows down propagation. Therefore we flush all redundant clauses (including low glucose level clauses) in geometrically increasing conflict intervals too. This should happen less than a dozen of times during each competition run though.

LICENSE

The default license of YALSAT, LINGELING, PLINGELING and TREENGELING did not change in the last three years. It allows the use of these solvers for research and evaluation but not in a commercial setting nor as part of a competition submission without explicit permission by the copyright holder. However, as part of our new open source release of BOOLECTOR 3.0 [20] we also plan to release LINGELING under an open source MIT style license, which for CADICAL continues to be the case.

REFERENCES

[1] A. Biere, "Deep Bound Hardware Model Checking Instances, Quadratic Propagation Benchmarks and Reencoded Factorization Problems Submitted to the SAT Competition 2017," in *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2017-1. University of Helsinki, 2017, pp. 40–41.

[2] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere, "Pruning through satisfaction," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 10629. Springer, 2017, pp. 179–194.

[3] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 129–144.

[4] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370.

[5] H. Han and F. Somenzi, "On-the-fly clause improvement," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 209–222.

[6] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140.

[7] A. Biere, "Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016," in *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2016-1. University of Helsinki, 2016, pp. 44–45.

[8] "Boolector at the SMT competition 2017," FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep., 2017.

[9] T. Balyo, M. Heule, and M. Järvisalo, Eds., *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, vol. B-2017-1. University of Helsinki, 2017.

[10] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS," in *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2017-1. University of Helsinki, 2017, pp. 20–21.

[11] F. Xiao, M. Luo, C.-M. Li, F. Manyà, and Z. Lü, "MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width in SAT Competition 2017," in *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2017-1. University of Helsinki, 2017, pp. 22–23.

[12] A. Biere and A. Fröhlich, "Evaluating CDCL variable scoring schemes," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9340. Springer, 2015, pp. 405–422.

[13] C. Piette, Y. Hamadi, and L. Sais, "Vivifying propositional clausal formulae," in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, vol. 178. IOS Press, 2008, pp. 525–529.

[14] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 703–711.

[15] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *IJCAR*, ser. Lecture Notes in Computer Science, vol. 7364. Springer, 2012, pp. 355–370.

[16] C. Oh, "Between SAT and UNSAT: the fundamental difference in CDCL SAT," in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Heule and S. Weaver, Eds., vol. 9340. Springer, 2015, pp. 307–323.

[17] A. Biere and A. Fröhlich, "Evaluating CDCL restart schemes," in *Proceedings POS-15. Sixth Pragmatics of SAT workshop*, 2015, to be published.

[18] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, ser. Lecture Notes in Computer Science, J. Marques-Silva and K. A. Sakallah, Eds., vol. 4501. Springer, 2007, pp. 294–299.

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

[20] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2, BtorMC and Boolector 3.0," in *Computer Aided Verification - 30th International Conference, CAV 2018*, ser. Lecture Notes in Computer Science. Springer, 2018, to appear.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

System Description of Candy Kingdom – A Sweet Family of SAT Solvers

Markus Iser* and Felix Kutzner†
Institute for Theoretical Computer Science, Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: *markus.iser@kit.edu, †felix.kutzner@qpr-technologies.de

***Abstract*—Candy is a branch of the Glucose 3 SAT solver and started as a refactoring effort towards modern C++. We replaced most of its custom lowest-level data structures and algorithms by their C++ standard library equivalents and improved or reimplemented several of its components. New functionality in Candy is based on gate structure analysis and random simulation.**

I. Introduction

The development of our open-source SAT solver **Candy**[1] started as a branch of the well-known **Glucose** [1], [2] CDCL SAT solver (version 3.0). With Candy, we aim to facilitate the solver's development by refactoring the Glucose source code towards modern C++ and by reducing dependencies within the source code. This involved replacing most custom lowest-level data structures and algorithms by their C++ standard library equivalents. The refactoring effort enabled high-level optimizations of the solver such as inprocessing and cache-efficient clause memory management. We also increased the extensibility of Candy via static polymorphism, e.g. allowing the solver's decision heuristic to be customized without incurring the overhead of dynamic polymorphism. This enabled us to efficiently implement variants of the Candy solver. Furthermore, we modularized the source code of Candy to make its subsystems reusable. The quality of Candy is assured by automated testing, with the functionality of Candy tested on different compilers (Clang, GCC, Microsoft C/C++) and operating systems (Linux, Apple macOS, Microsoft Windows) using continuous integration systems.

II. Clause Memory Management

Unlike Glucose, we use regular pointers to reference clauses in Candy. To reduce the memory access overhead, we introduced a dedicated cache-optimized clause storage system. To this end, we reduced the memory footprint of clauses by shrinking the clause header, in which only the clause's size, activity and LBD values as well as a minimal amount of flags are stored. For clauses containing 500 literals or less, our new clause allocator preallocates clauses in buckets of same-sized clauses. Clauses larger than 500 literals are individually allocated on the heap. Buckets containing small clauses are regularly sorted by their activity in descending order to group frequently-accessed clauses, thereby concentrating memory accesses to smaller memory regions. Moreover, the watchers are regularly sorted by clause size and activity.

[1]https://github.com/udopia/candy-kingdom

III. Improved Incremental Mode

We enabled several clause simplifications in Candy's incremental mode that had been deactivated in Glucose's incremental mode. Also, certificates for unsatisfiability can be generated in incremental mode for sub-formulas not containing assumption literals. This is achieved by suppressing the emission of learnt clauses containing assumption literals as well as the output of the empty clause until no assumptions are used in the resolution steps by which unsatisfiability is deduced.

IV. Inprocessing

We improved the architecture of clause simplification such that Candy can now perform simplification based on clause subsumption and self-subsuming resolution during search. The original problem's clauses are included as well as learnt clauses that are persistent in the learnt clause database, i.e. clauses of size 2 and clauses with an LBD value no larger than 2.

V. Improvments Candy 2018

The approach to modularize the Glucose codebase was further pursued and realized. We improved a lot in separation of concerns and building of separate components for heuristics and algorithmic methods.

Candy now initializes variable ordering based on problem structure by default. This should make its default heuristic configuration more stable against problem scrambling.

References

[1] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence. pp. 399–404. IJCAI'09, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009)

[2] Eén, N., Sörensson, N.: An extensible sat-solver. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. pp. 502–518 (2003)

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

CBPeneLoPe2018 and CCSPeneLoPe2018 at the SAT Competition 2018

Tomohiro Sonobe
National Institute of Informatics, Japan
JST, ERATO, Kawarabayashi Large Graph Project, Japan
Email: tominlab@gmail.com

***Abstract*—In this description, we provide a brief introduction of our solvers: CBPeneLoPe2018 and CCSPeneLoPe2018, in the SAT Competition 2018. CBPeneLoPe2018 and CCSPeneLoPe2018 are based on the parallel SAT solver PeneLoPe. Both solvers use SatELite [4] as a preprocessor.**

I. CBPeneLoPe2018

CBPeneLoPe2018 is a parallel portfolio SAT solver based on PeneLoPe [2] and a new version of ones submitted in the SAT Competition 2014, SAT Race 2015, SAT Competition 2016, and SAT Competition 2017. CBPeneLoPe2018 implements *community branching* [7], a diversification [5] technique using community structure of SAT instances [1]. Community branching assigns a different set of variables (community) to each worker and forces them to select these variables as decision variables in early decision levels, aiming to avoid overlaps of search spaces between the workers more vigorously than the existing diversification methods.

In order to create communities, we construct a graph where a vertex corresponds to a variable and an edge corresponds to a relation between two variables in the same clause, proposed as Variable Incidence Graph (VIG) in [1]. After that, we apply Louvain method [3], one of the modularity-based community detection algorithms, to identify communities of a VIG. Variables in a community have strong relationships, and a distributed search for different communities can benefit the whole search.

In addition, CBPeneLoPe2018 uses *community-based learnt clause sharing (CLCS)* for learnt clause sharing between workers. CLCS restricts the sharing of each learnt clause to workers that conducts the search for the variables related with communities in the target learnt clause. By combining community branching, CLCS distributes the target clauses to the workers with related communities. For example, if a learnt clause $(a \vee b \vee c)$ is to be shared among the workers, and the variable a and b belong to a community C_1 and the variable c belongs to a community C_2, this clause is distributed only to the workers that are assigned the community C_1 or C_2 by community branching.

II. CCSPeneLoPe2018

CCSPeneLoPe2018 is a parallel portfolio solver based on PeneLoPe. The features of CCSPeneLoPe2018 are as follows.

- Conflict history-based branching heuristic (CHB) [6] for some workers
- CLCS prioritizing high VSIDS or CHB scores

CHB is good at cryptographic instances in [6]. In CCSPeneLoPe2018, some workers use this heuristic with different sets of its parameters. For CLCS, each worker calculates an average activity score (VSIDS or CHB) of variables for each community and chooses the highest scored community as a "desired community". CLCS distributes the target clause to the workers that desire to share that clause (i.e., including the variables that belong to the desired community).

III. Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 17K12742.

References

[1] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of SAT formulas. In *Theory and Applications of Satisfiability Testing*, SAT'12, pages 410–423, 2012.

[2] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In *Theory and Applications of Satisfiability Testing*, SAT'12, pages 200–213, 2012.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):10008, 2008.

[4] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Theory and Applications of Satisfiability Testing*, SAT'05, pages 61–75, 2005.

[5] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Saïs. Diversification and intensification in parallel SAT solving. In *Principles and practice of constraint programming*, CP'10, pages 252–265, 2010.

[6] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *AAAI Conference on Artificial Intelligence*, AAAI'16, 2016.

[7] Tomohiro Sonobe, Shuya Kondoh, and Mary Inaba. Community branching for parallel portfolio SAT solvers. In *Theory and Applications of Satisfiability Testing*, SAT'14, pages 188–196, 2014.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

The CryptoMiniSat 5.5 set of solvers at the SAT Competition 2018

Mate Soos, National University of Singapore

I. Introduction

This paper presents the conflict-driven clause-learning SAT solver CryptoMiniSat v5.5 (*CMS*) as submitted to SAT Competition 2018. CMS aims to be a modern, open-source SAT solver that allows for multi-threaded in-processing techniques while still retaining a strong CDCL component. In general, CMS is a inprocessing SAT solver that uses optimised data structures and finely-tuned timeouts to have good control over both memory and time usage of simplification steps. Below are the changes to CMS compared to the SAT Competition 2016 version.

II. Major Improvements

A. *Careful code review*

Over the years, much cruft has accumulated in CryptoMiniSat. This has left serious bugs in the implementation in important parts of the solver such as clause cleaning and restarting. This has lead to low performance. A code review of the most important parts of the solver such as bounded variable elimination, restarting, clause cleaning and variable activities has been conducted.

B. *Integration of ideas from Maple_LCM_Dist*

Some of the ideas from Maple_LCM_Dist[2][4] have been included into CMS. In particular, the clause cleaning system, the radical in-process distillation and the Maple-based variable activities are all used.

C. *Cluster Tuning*

The author has been generously given time on the ASPIRE-1 cluster of the National Supercomputing Centre Singapore[1]. This allowed experimentation and tuning that would have been impossible otherwise. CMS has not been tuned on a cluster for over 6 years and the difference shows. A slightly interesting side-effect is that the parameters suggested by the cluster are non-intuitive, such as not simplifying the CNF straight away, but rather CDCL-solving it first. Another interesting effect is that intree probing[3] seems to be very important.

D. *Parallel Solving*

As in previous competitions, CMS only shares unit and binary clauses, and shares them very rarely. The different threads, however, are run with very different, hoping to be orthogonal, parameters varying everything from clause cleaning strategies to default polarities.

E. *Automatic Tuning*

The "autotune" version of the solver measures internal solving parameters and re-configures itself after a preset number of conflicts to a configuration that has been suggested by the parameters and the machine learning algorithm C4.5[5].

III. General Notes

A. *On-the-fly Gaussian Elimination*

On-the-fly Gaussian elimination is again part of CryptoMiniSat. This is explicitly disabled for the competition, but the code is available and well-tested. This allows for special uses of the solver that other solvers, without on-the-fly Gaussian elimination, are not capable of.

B. *Robustness*

CMS aims to be usable in both industry and academia. CMS has over 150 test cases and over 2000 lines of Python just for fuzzing orchestration, and runs without fault under both the ASAN and UBSAN sanitisers of clang. It also compiles and runs under Windows, Linux and MacOS X. This is in contrast many academic winning SAT solvers that produce results that are non-reproducible, cannot be compiled on anything but a few select systems, and/or produce segmentation faults if used as a library. CryptoMiniSat has extensive fuzzing setup for library usage and is very robust under strange/unexpected use cases.

IV. Thanks

This work was supported in part by NUS ODPRT Grant, R-252-000-685-133. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore[1]. The author would also like to thank all the users of CryptoMiniSat who have submitted over 400 issues and many pull requests to the GitHub CMS repository[6].

References

[1] ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), https://www.nscc.sg/about-nscc/overview/

[2] Balyo, T., Heule, M.J.H., Jarvisalo, M.: MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width in SAT Competition 2017. In: Proceedings of SAT Competition 2017 (2018)

[3] Heule, M., Järvisalo, M., Biere, A.: Revisiting Hyper-Binary Resolution. In: Gomes, C.P., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7874, pp. 77–93. Springer (2013), https://doi.org/10.1007/978-3-642-38171-3_6

[4] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning Rate Based Branching Heuristic for SAT Solvers. In: Creignou, N., Le Berre, D. (eds.) Theory and Applications of Satisfiability Testing – SAT 2016. pp. 123–140. Springer International Publishing, Cham (2016)

[5] Salzberg, S.L.: C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993. Machine Learning 16(3), 235–240 (Sep 1994), https://doi.org/10.1007/BF00993309

[6] Soos, M.: CryptoMiniSat SAT solver GitHub page (2018), https://github.com/msoos/cryptominisat

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

COMiniSatPS Pulsar and GHackCOMSPS in the 2018 Competition

Chanseok Oh
Google
New York, NY, USA

***Abstract*—COMiniSatPS is a patched MiniSat generated by applying a series of small diff patches to the last available version (2.2.0) of MiniSat that was released several years ago. The essence of the patches is to include only minimal changes necessary to make MiniSat sufficiently competitive with modern SAT solvers. One important goal of COMiniSatPS is to provide these changes in a highly accessible and digestible form so that the necessary changes can be understood easily to benefit wide audiences, particularly starters and non-experts in practical SAT. As such, the changes are provided as a series of incrementally applicable diff patches, each of which implements one feature at a time. COMiniSatPS has many variations. The variations are official successors to an early prototype code-named SWDiA5BY that saw great successes in the past SAT-related competitive events.**

I. Introduction

It has been shown in many of the past SAT-related competitive events that very simple solvers with tiny but critical changes (e.g, MiniSat [1] hack solvers) can be impressively competitive or even outperform complex state-of-the-art solvers [2]. However, the original MiniSat itself is vastly inferior to modern SAT solvers in terms of actual performance. This is no wonder, as it has been many years since the last 2.2.0 release of MiniSat. To match the performance of modern solvers, MiniSat needs to be modified to add some of highly effective techniques of recent days. Fortunately, small modifications are enough to bring up the performance of any simple solver to the performance level of modern solvers. COMiniSatPS [3]. adopts only simple but truly effective ideas that can make MiniSat sufficiently competitive with recent state-of-the-art solvers. In the same minimalistic spirit of MiniSat, COMiniSatPS prefers simplicity over complexity to reach out to wide audiences. As such, the solver is provided as a series of incremental patches to the original MiniSat. Each small patch adds or enhances one feature at a time and produces a fully functional solver. Each patch often changes solver characteristics fundamentally. This form of source distribution by patches would benefit a wide range of communities, as it is easy to isolate, study, implement, and adopt the ideas behind each incremental change. The goal of COMiniSatPS is to lower the entering bar so that anyone interested can implement and test their new ideas easily on a simple solver guaranteed with exceptional performance.

The patches first transform MiniSat into Glucose [4] and then into SWDiA5BY. Subsequently, the patches implement new techniques described in [5], [2], and [6] to generate the current form of COMiniSatPS.

COMiniSatPS is a base solver of the MapleCOMSPS solver series [7], [8], [9] that participated in SAT Competition 2016, 2017, and 2018.

II. COMiniSatPS Pulsar

This year's solver is basically identical to the last year's solver, fixing only two minor bugs:

- Correctly reports UNSAT when a problem is determined to be UNSAT during CNF parsing if Gaussian elimination is enabled. (The Gaussian elimination is enabled when not generating UNSAT proof.)
- Correctly generates UNSAT proof when a problem is solved by pre-processing alone.

III. GHackCOMSPS

This year's solver is identical to the last year's solver [10] (which is in turn identical to the 2016 version). GHackCOMSPS qualifies as a Glucose hack.

IV. Availability and License

Source is available for download for all the versions described in this paper. Note that the license of the M4RI library (used to implement the Gaussian elimination) is GPLv2+.

Acknowledgment

We thank specifically the authors of Glucose, GlueMiniSat, Lingeling, CryptoMiniSat, and MiniSat.

References

[1] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003.
[2] C. Oh, "Improving SAT solvers by exploiting empirical characteristics of CDCL," Ph.D. dissertation, New York University, 2016.
[3] ——, "Patching MiniSat to deliver performance of modern SAT solvers," in *SAT-RACE*, 2015.
[4] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI*, 2009.
[5] C. Oh, "Between SAT and UNSAT: The fundamental difference in CDCL SAT," in *SAT*, 2015.
[6] ——, "COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS," in *SAT Competition*, 2016.
[7] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *SAT Competition*, 2016.
[8] ——, "MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS," in *SAT Competition*, 2017.
[9] J. H. Liang, C. Oh, K. Czarnecki, P. Poupart, and V. Ganesh, "MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS in the 2018 Competition," in *SAT Competition*, 2018.
[10] C. Oh, "COMiniSatPS Pulsar and GHackCOMSPS," in *SAT Competition*, 2017.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

dimetheus

Oliver Gableske
oliver@gableske.net

***Abstract*—This document describes the `dimetheus` SAT solver as submitted to the random SAT track of the SAT Competition 2018.**

I. Introduction

Please note that this article must be understood as a rather brief overview of the `dimetheus` SAT solver. Additional information regarding its functioning, a comprehensive quick-start guide, as well as the source-code of the latest version of the solver can be found on the authors website.[1] Additionally, the author elaborates on the theoretical background of the solver in his Ph.D. thesis [1] which can be found online.[2] A preliminary overview of the applied techniques can be found in [2], [3].

This article will first cover the main techniques that the solver applies in Section II. Afterwards, a brief overview of the parameter settings are discussed in Section III. This is followed by a brief explanation of the programming language and the compiler relevant parameters in Section IV. Additionally, several SAT Competition relevant details are discussed in Section V. The article is concluded by a few remarks on the availability and the license of the solver in Section VI.

II. Main Techniques

The `dimetheus` solver runs in various phases as depicted in Figure 1.

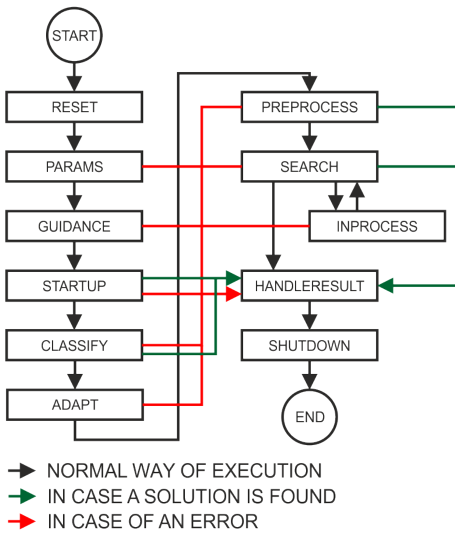


Fig. 1. A flow chart that visualizes the execution of `dimetheus`.

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

In each phase, the solver must fulfill a pre-defined task. The first four of theses phases (reset, params, guidance, startup) are not discussed here in detail. At the end of the startup phase the solver has loaded the formula and is able to work with it.

The solver will then execute the classification phase in order to determine determine what type of CNF formula was provided. Since the solver is submitted to the random SAT track of the SAT Competition it will determine what type of random formula it has to solve (e.g., it will determine the size of the formula, the clause lengths, the clauses-to-variables ratio). The result of the classification phase is then forwarded to the adaptation phase. In this phase, the solver adapts a wide variety of internal parameters in order to initialize its internal heuristics and search algorithm.

Afterwards, preprocessing is performed. The preprocessing is kept simple and includes pure literal elimination and the removal of duplicate clauses.

Preprocessing is then followed by the search phase in which the solver tries to find a satisfying assignment for the formula (inprocessing is turned off when the solver solves random formulas). The approach that the solver applies is best understood as bias-based decimation followed by stochastic local search. The bias-based decimation applies a Message Passing algorithm to calculate biases for individual variables. These biases indicate how likely it is to observe a variable assigned to one or zero when taking into account the models of the formula. For more information see [1]. Afterwards, a fraction of the variables with the largest bias are assigned and unit propagation (UP) is performed which then leads to a simplified remaining formula. The bias calculation and the UP-based assignment of variables with the largest bias is repeated until one of two cases occurs. First, a model is found. In this case the solver merely outputs the model and terminates. Second, UP runs into a conflict. In this case the solver will undo all assignments and initializes an SLS solver. The starting assignment for the SLS is comprised of all the assignments made until the confilct arose as well as random assignments to the remaining variables. From this point onwards the SLS takes place until either a time-out is hit or a model is found. The `dimetheus` solver, as it runs in the SAT Competition 2018, is therefore an incomplete solver that cannot detect unsatisfiability.

III. Main Parameters

The solver is started with the two following parameters.

> -formula STRING: The STRING points to the file that contains the formula in DIMACS CNF input format.

-classifyInputDomain 10: This tells the classifier that it can assume the formula to be a random formula when determining what specific type of formula it is.

As mentioned in the previous section, the solver will use the information that was gathered in the classification phase in order to enforce an optimal parameter setting to a variety of internal parameters [1]. Unfortunately, it is not possible to correctly explain the abundance of parameters here which is why the reader is referred to the given reference for details.

IV. Implementation Details

The `dimetheus` solver is implemented in C. The Message Passing algorithm that is applied to calculate the biases is an interpolation of Belief Propagation and Survey Propagation [1], [4]. The SLS serach follows the `probSAT` approach [5].

V. SAT Competition 2018 Specifics

The `dimetheus` solver was submitted to the random SAT track. It was compiled on the StarExec Cluster using `gcc` with the compile flags `-std=c99 -O3 -static -fexpensive-optimizations -flto -fwhole-program -march=native -Wall -pedantic`. The result is a 64-bit binary.

VI. Availability and License Information

The `dimetheus` solver is publicly available and can be downloaded from https://www.gableske.net/dimetheus. The solver is provided under the Creative-Commons Non-Commercial Share-Alike license version 3.0 (CCBYNCSA3).

Acknowledgments

The author hereby expresses his gratitude towards the organizers of the SAT Competition and the SAT Conference. Your hard work is much appreciated.

References

[1] O. Gableske, “Sat solving with message passing,” Ph.D. dissertation, Ulm University, Germany, May 2016.

[2] ——, “An ising model inspired extension of the product-based mp framework for sat,” *Theory and Application of Satisfiability Testing*, vol. LNCS 8561, pp. 367–383, 2014.

[3] ——, “On the interpolation of product-based message passing heuristics for sat,” *Theory and Application of Satisfiability Testing*, vol. LNCS 7962, pp. 293–308, 2013.

[4] A. Braunstein, M. Mézard, and R. Zecchina, “Survey propagation: an algorithm for satisfiability,” *Journal of Random Structures and Algorithms*, vol. 27, pp. 201–226, 2005.

[5] A. Balint and U. Schöning, “Choosing probability distributions for stochastic local search and the role of make versus break,” *Theory and Application of Satisfiability Testing*, vol. LNCS 7137, pp. 16–29, 2012.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Description of *expSAT* Solvers

Md Solimul Chowdhury
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mdsolimu@ualberta.ca

Martin Müller
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mmueller@ualberta.ca

Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
jyou@ualberta.ca

***Abstract*—*expSAT* is a novel CDCL SAT solving method, which performs random-walk based explorations of the search space w.r.t the current search state to guide the search. It uses a new branching heuristics, called `expVSIDS`, which combines the standard variable selection heuristic VSIDS, which is based on search performance so far, with heuristic scores derived from random samples of possible future search states. This document describes the *expSAT* approach and four CDCL SAT solvers based on this approach, which we have submitted for the SAT competition-2018.**

I. The *expSAT* Approach

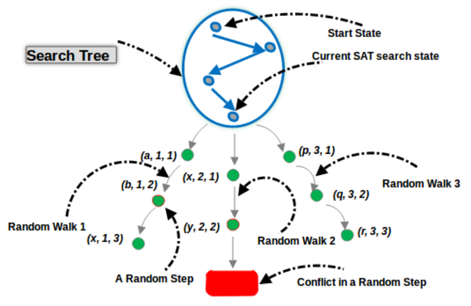
This section presents the *expSAT* approach, part of which is to appear in [1].

A. *expSAT algorithm*

Given a CNF SAT formula $\mathcal{F}$, let $vars(\mathcal{F})$, $uVars(\mathcal{F})$ and $assign(\mathcal{F})$ denote the set of variables in $\mathcal{F}$, the set of currently unassigned variables in $\mathcal{F}$ and the current partial assignment, respectively. In addition to $\mathcal{F}$, *expSAT* also accepts five *exploration parameters* $nW, lW, \theta_{stop}, p_{exp}$ and ω, where $1 \leq nW, lW \leq uVars(\mathcal{F})$, $0 < \theta_{stop}, p_{exp}, \omega \leq 1$. These parameters control the exploration aspects of *expSAT*. The details of these parameters are given below.

Given a CDCL SAT solver, *expSAT* modifies it as follows: (I) Before each branching decision, if the search-height, $\frac{|assign(\mathcal{F})|}{|vars(\mathcal{F})|} \leq \theta_{stop}$, with probability p_{exp}, *expSAT* performs an *exploration episode*, consisting of a fixed number *nW* of random walks. Each walk consists of a limited number of *random steps*. Each such step consists of (a) the uniform random selection of a currently unassigned *step variable* and assigning a boolean value to it using a standard CDCL *polarity* heuristic, and (b) a followed by Unit Propagation (UP). A walk terminates either when a conflict occurs during UP, or after a fixed number *lW* of random steps have been taken. Figure 1 illustrates an exploration episode. (II) In an exploration episode of nW walks of maximum length lW, the *exploration score expScore* of a decision variable v is the average of the *walk scores ws*(v) of all those random walks within the same episode in which v was one of the randomly chosen decision variables. *ws*(v) is computed as follows: (a) $ws(v) = 0$ if the walk ended without a conflict. (b) Otherwise, $ws(v) = \frac{\omega^d}{lbd(c)}$, with decay factor $0 < \omega \leq 1$, $lbd(c)$ the LBD score of the clause c learned for the current conflict, and $d \geq 0$ the *decision distance* between variable v and the conflict which ended the current walk: If v was assigned at some step j during the current walk, and the conflict occurred after step $j' \geq j$, then $d = j' - j$. We assign credit to all the step variables in a walk that ends with a conflict and give higher credit to variables closer to the conflict. (III) The novel branching heuristic `expVSIDS` adds VSIDS score and *expScore* of the unassigned variables. At the current state of the search, the variable bumping factor of VSIDS is g^z, where $g > 1$ and $z \geq 1$ is the count of conflicts in the search so far. To achieve a comparable scale for $expScore$ and VSIDS score, we scale up the *expScore* by g^z before adding these scores. A variable v^* with maximum combined score is selected for branching. (IV) All other components remain the same as in the underlying CDCL SAT solver.

Fig. 1: An exploration episode with $nW = 3$ walks and a maximum of $lW = 3$ random steps per walk. (v, i, j) represents that the variable v is randomly decided at the j^{th} step of i^{th} walk.



B. *Exploration Parameter Adaptation*

In *expSAT*, $(nW, lW, \theta_{stop}, p_{exp}, \omega)$, the set of exploration parameters, governs the exploration. The first two parameters dictate *how* much exploration to perform per episode, the third and fourth parameter dictate *when* to trigger an exploration episode. ω controls how exploration scores are computed.

How to adapt these parameters during the SAT search is an interesting question, which is not addressed in [1]. The *expSAT* based solvers submitted for this competition uses a simple local search algorithm to adapt the first four exploration parameters $P = (nW, lW, \theta_{stop}, p_{exp})$ to dynamically control when to trigger exploration episodes and how much exploration to perform in an exploration episode. This local search algorithm executes in parallel to the SAT search in *expSAT*.

a) The Adaptation Algorithm: The idea of this algorithm is to start with an initial value $val(P, 1) = (nW^1, lW^1, \theta^1_{stop}, p^1_{exp})$ of the exploration parameters and iteratively update the values between two consecutive restarts, based on the performance of exploration in the previous restarts.

Assume *expSAT* has just performed the j^{th} ($j \geq 2$) restart. Let σ_j the performance of exploration between $(j-1)^{th}$ and j^{th} restarts. We define σ^j as follows:

$$\sigma^j = w_1 * \frac{c^j}{rSteps^j} + w_2 * \frac{gc^j}{rSteps^j} + w_3 * \frac{1}{\overline{rsLBD^j}}$$

Here, $rSteps^j$ is the number of random steps taken during the exploration episodes, which has occurred between $(j-1)^{th}$ and j^{th} restarts, c^j, gc^j, $\overline{rsLBD^j}$ are the number of conflicts, number of glue-clauses and the mean LBD value of the (learned) clauses identified in these $rSteps^j$ steps, respectively. w_1, w_2 and w_3 are three fixed weights.

After restart j, just before starting SAT search, the algorithm updates the exploration parameter values by comparing σ_j and σ_{j-1}. Let $val(P, j) = (nW^j, lW^j, \theta^j_{stop}, p^j_{exp})$ is the updated value of exploration parameters before the search begins, just after the $(j-1)^{th}$ restart.

- If $\sigma^j < \sigma^{j-1}$, then the performance of exploration deteriorates after the $(j-1)^{th}$ restart. In this case, we perform two operations on the exploration parameters after j^{th} restart:
  - **Decrement**: Let $dp \in P$ the parameter whose value was increased from x to x' after the $(j-1)^{th}$ restart. We attribute this deterioration of performance to this update. We revert the value of dp back to x from x'.
  - **Increment**: Randomly select a parameter $rp \in P$ and increase its value to y' from y, where $rp \neq dp$.
- If $\sigma^j - \sigma^{j+1} = 0$, then we only perform the **Increment** operation, as we do not know whom to blame for the stall.
  The updated value of these parameters remain effective until the $(j+1)^{th}$ restart.
- If $\sigma^j > \sigma^{j-1}$, then the performance of exploration is increasing after the $(j-1)^{th}$ restart and we do not change any parameter value as the current value of the exploration parameters leads to better performance.

For changing the value of a parameter $x \in P$, we associate a step size s_x with x. Also, in order to prevent the unbounded growth/shrink of the parameters we associate a lower and upper bound with each of the parameters. That is, for $x \in P$, we have a $[l_x, u_x]$. Whenever the value of x exceeds u_x OR the value of x is less than l_x, then the value of x is reset to its initial value x^1.

II. *expSAT* Solvers

We have submitted four CDCL SAT solvers based on the expSAT approach, which are implemented on top of Glucose, MapleCOMPSPS_LRB and MapleCOMPSPS. In the following, we describe our solvers:

a) ***expGlucose:*** expGlucose is an extension of Glucose, where we replace VSIDS by *expVSIDS* and and have kept everything else the same as in Glucose.

b) ***expMC_LRB_VSIDS_Switch:*** The corresponding baseline system MapleCOMPSPS_LRB switches between branching heuristics LRB and VSIDS in between restarts. In expMC_LRB_VSIDS_Switch, we replace VSIDS with *expVSIDS* and have kept everything else the same as in MapleCOMPSPS_LRB.

c) ***expMC_LRB_VSIDS_Switch_2500:*** The corresponding baseline system MapleCOMPSPS has three switches between VSIDS and LRB (i) VSIDS for initialization (first 50,000 conflicts), (ii) then run LRB for 2,500 seconds, and (iii) then switches to VSIDS for rest of the execution of the solver. In expMC_LRB_VSIDS_Switch_2500, we replace VSIDS with *expVSIDS* for (iii) and have kept everything else the same as in MapleCOMPSPS.

d) ***expMC_VSIDS_LRB_Switch_2500:*** This system is a variant of expMC_LRB_VSIDS_Switch_2500. It uses *expVSIDS* for the first 2,500 seconds and then switches to LRB for the rest of its execution.

References

[1] MS Chowdhury and M. Müller and J. You, "Preliminary results on exploration driven satisfiability solving (Student Abstract).", AAAI-2018.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Glucose and Syrup: Nine years in the SAT competitions

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

Laurent Simon
Bordeaux-INP / Univ. Bordeaux
LaBRI/CNRS UMR 5800
lsimon@labri.fr

***Abstract*—Glucose is a CDCL solver developped on top of Minisat nine years ago, with a special focus on removing useless clauses as soon as possible, and an original restart scheme based on the quality of recent learnt clauses. Syrup is the parallel version of Glucose, with an original lazy clauses exchanges policy, thanks to a one-watched scheme. We describe in this short solver description the small novelties introduced this year for the SAT 2018 competition.**

I. Introduction

Glucose is a CDCL (Conflict Driven Clause Learning) solver introduced in 2009 that tries to center all the components of the SAT solver around a measure of learnt clause quality, called LBD, for Literal Block Distance. This measure allows to delete a lot of learnt clauses from the beginning of the computation. From a practical point of view, it seems that this feature allows Glucose to produce more shorter proofs, which probably explains why Glucose and Syrup won a number of competitions in the last 9 years. A recent survey paper summarizes most of the improvements we added to the original Glucose [1]. Of course, the current short description does not mean to be exhaustive and the interested reader should refer to the previous paper.

In a few words, however, Glucose enters SAT competitions/races [2], [3] every years since its creation. Glucose is based on the internal architecture of minisat [4] (especially for the VSIDS implementation, the 2-Watched scheme and the memory management of clauses (garbage collection, ...)). It is based on the notion of Literal Block Distance, as aforementioned, a measure that is able to estimate the quality of learnt clauses [5]. This measure simply counts the number of distinct decision levels of literals occurring in learnt clauses, at the time of their creation. Thanks to that, new strategies for deleting clauses were proposed. Moreover, the solver constantly watch the quality of the last learnt clauses and triggers a restart when the quality is worst than the average. Recent developments includes a way of postponing restarts when the number of assigned literals suddenly increases without conflicts (a SAT solution may be then expected). In the last version of Glucose, a special strategy allowed the solver to decide which strategy to use with respect to a set of identified extreme case [6].

Indeed, learnt clauses removal, restarts, small modifications of the VSIDS heuristic are based on the concept of LBD. The core engine of Glucose (and Syrup) is 8 years old. Syrup is a major improvement of Glucose on which we focused most of our efforts in the last years.

II. New components

The 2018 version of Glucose and Syrup are very similar to the 2016 ones, with two improvements. The main modifications are based on the extension of the recent LCM strategies proposed last year [7] (which ”revived” the vivification technique [8]). We observed that the LCM strategy was not always performed on clauses of small LBD only, because LCM was not triggered right after clause database reduction, and thus the order of clauses traversed by the LCM was not based on a sorted order of learnt clauses. However, we observed that LCM was more efficient when not always run on good clauses only (LCM can replace clauses, and thus may delete a good clause). We observed that LCM was more efficient when active clauses were kept, in addition to clauses of small LBD. Glucose is now keeping 10% of the most active clauses in addition to the usual LBD based ranking. In addition, we integrated the LCM technique into Syrup (good clauses found during the LCM reduction were not shared initially). The integration into Syrup was not obvious because LCM seemed to be much more efficient on sequential solvers only. LCM was thus not activated on all the cores in the parallel version.

III. Parallel version of Glucose

We used a version with 24 and 48 cores this year for the parallel versions of Glucose (called Glucose-Syrup).

IV. Algorithm and Implementation details

Glucose uses a special data structure for binary clauses, and a very limited self-subsumption reduction with binary clauses, when the learnt clause is of interesting LBD. The parallel version uses a special data structure for sharing clauses that may prevent some clause to be shared when it is full.

V. Acknowledgments

Recent developments of Glucose-Syrup is supported by the French Project SATAS ANR-15-CE40-0017. We would also like to thanks the organizers of the competition for their efforts.

References

[1] G. Audemard and L. Simon, “On the glucose sat solver,” vol. 27, no. 01.

[2] ——, “Glucose: a solver that predicts learnt clauses quality,” *SAT Competition*, pp. 7–8, 2009.

[3] ——, “Glucose 2.3 in the sat 2013 competition,” *Proceedings of SAT Competition*, pp. 42–43, 2013.

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

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

[6] ——, “Extreme Cases in SAT,” in *19th International Conference on Theory and Applications of Satisfiability Testing(SAT'13)*, 2013, p. To appear.

[7] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, “An effective learnt clause minimization approach for CDCL SAT solvers,” in *Proceedings of IJCAI 2017*, 2017, pp. 703–711.

[8] C. Piette, Y. Hamadi, and L. Sais, “Vivifying propositional clausal formulae,” in *ECAI 2008 - 18th European Conference on Artificial Intelligence*, 2008, pp. 525–529.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

GluHack

Aolong Zha
Kyushu University, Japan
aolong.zha@inf.kyushu-u.ac.jp

Abstract—**GluHack is a SAT solver submitted to the Glucose Hack Track of the SAT Competition 2018. It updates Glucose 3.0 in the following aspects: searching watch list, conflict learning and learnt clause database reduction.**

I. Introduction

Glucose [1] is an open-source CDCL-based SAT solver [2] that has achieved numerous excellent performance in past SAT Competition. In *Unit Propagation* (UP), the head of trail queue of assigned literal will be specified as a watched literal for searching its corresponding watch list, which is in order to propagate the next unit unassigned literal in a clause. This procedure will not stop until all variables are assigned, unless a conflict occurs. Let's consider the situation which is to decide literal a is assigned to true in UP. Then by searching watch list we know that all literals in clause C_1 have been assigned to false including literal $\neg a$. Obviously, a conflict occurs. Glucose stops searching watch list, returns the reason of this conflict, the clause C_1, and goes to the learning phase. Under current assignment gluHack keeps searching watch list until all conflicts are detected, then stores all corresponding reasons $C_1, C_2, \ldots, C_n$ into a vector of clauses and returns it.

II. Implementation

GluHack stops unit propagating (assigning), but keeps searching watch list when first conflict occurs. All detected conflicts are stored into a vector of clauses.

Modification 1 CRef Solver::$propagate()$

```
Initialize: bool firstConflOccur ← true;
            vec ⟨CRef⟩ conflCRefList;
 1: ···
 2: // Did not find watch – clause is unit under assignment:
 3: *j++ ← w;
 4: if $value$(first) = l_False then
 5:     if firstConflOccur = true then
 6:         firstConflOccur ← false;
 7:         confl ← cr;
 8:     end if
 9:     conflCRefList.$push$(cr);
10:     qhead ← trail.$size$();
11: else if firstConflOccur = true then
12:     $uncheckedEnqueue$(first, cr);
13: end if
14: NextClause:;
15: ···
16: return confl;
```

The $analyze()$ function will generate corresponding learnt clauses for each conflict in this vector. Finally, we evaluate and filter some effective learnt clauses, whose the number of literal contained ≤ 2, and add them into the learnt clause database.

Modification 2 lbool Solver::$search$(int)

```
Initialize: vec ⟨Lit⟩* learnt_clauseList ←
            new vec ⟨Lit⟩[conflCRefList.$size()$];
 1: ···
 2: for $i = 1$ to conflCRefList.$size()$ do
 3:     $analyze$(conflCRefList[$i$], learnt_clause, selectors,
        backtrack_level, nblevels, szWoutSelectors); ···
 4:     for $j = 0$ to learnt_clause.$size()$ do
 5:         learnt_clauseList[$i$].$push$(learnt_clause[$j$]);
 6:     end for
 7: end for
 8: $analyze$(confl, learnt_clause, selectors, backtrack_level,
    nblevels, szWoutSelectors);
 9: ···
10: if learnt_clause.$size() = 1$ then
11:     $uncheckedEnqueue$(learnt_clause[0]); nbUn++;
12: else
13:     for $i = 1$ to conflCRefList.$size()$ do
14:         if learnt_clauseList[$i$].$size() \leq 2$ then
15:             CRef cr = ca.$alloc$(learnt_clauseList[$i$], true); ···
16:         end if
17:     end for
18:     CRef cr = ca.$alloc$(learnt_clause, true); ···
19: end if
20: ···
```

In order to keep high quality of learnt clauses, we change the rate of learnt clause database reduction from 50% to 70%.

Modification 3 void Solver::$reduceDB()$

```
1: ···
2: int limit = learnts.$size()/10 * 7$; ···
```

The different between this hack version and original sources of Glucose 3.0 is total 779 non-space characters.

References

[1] G. Audemard and L. Simon, "Predicting Learnt Clauses Quality in Modern SAT Solvers," in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 2009, pp. 399–404.
[2] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-Driven Clause Learning SAT Solvers," in *Handbook of Satisfiability*, 2009, pp. 131–153.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

ManyGlucose 4.1-2

Yuuya Gotou
University of Yamanashi, JAPAN

Hidetomo Nabeshima
University of Yamanashi, JAPAN

***Abstract*—ManyGlucose 4.1-2 is a deterministic parallel SAT solver based on Glucose syrup 4.1 with an efficient clause exchange mechanism that is a refinement of ManySAT algorithm.**

I. Introduction

ManyGlucose 4.1-2 is a deterministic parallel SAT solver. Given an instance, a deterministic solver has reproducible results in terms of solution (satisfying assignment or proof of unsatisfiability) and running time. ManyGlucose supports such reproducible behavior. ManySAT 2.0 [2] is a representative deterministic parallel solver which is built on MiniSat 2.2 [3] with a deterministic clause exchange algorithm [1]. ManyGlucose 4.1-2 has a refined algorithm for clause exchange and is built on Glucose syrup 4.1 [4].

II. Main Techniques

ManySAT 2.0 has reproducible behavior by synchronizing among threads and exchanging learning clauses among them after synchronization. Each thread synchronizes every execution interval called *period*, which is defined as the number of conflicts. Since the generation speed of conflicts depends on the search space of each thread, the execution time of a period is different in each thread. Thus, each thread often has an idle time for synchronization.

In order to reduce the idle time, we introduce two improvements to ManySAT algorithm.

1) Refinement of period: a period is defined as the number of scanned literals in unit propagations instead of the number of conflicts.
2) Lazy clause exchange: each thread receives learned clauses obtained in m periods ago of the other threads. This eliminates the need to wait if the gap of the period of each thread is less than or equal to m.

III. Main Parameters

We define a period as 2 million scanned literals and use 20 as the margin for lazy clause exchange. Each thread uses different random seeds to hold the diversity of solvers. We submit ManyGlucose 4.1-2 with 24 threads and with 48 threads to Parallel track.

Acknowledgment

This research is supported in part by Grant-in-Aid for Scientific Research (No. 17K00300 and 16H02803) from Japan Society for the Promotion of Science.

References

[1] Y. Hamadi, S. Jabbour, C. Piette, and L. Sais, "Deterministic parallel DPLL," *Journal on Satisfiability*, vol. 7, no. 4, pp. 127–132, 2011.
[2] ——, "ManySAT 2.0," http://www.cril.univ-artois.fr/~jabbour/manysat.htm, 2011.
[3] N. Sörensson and N. Eén, "MiniSat2.1 and MiniSat++1.0 — SAT Race 2008 Editions," http://baldur.iti.uka.de/sat-race-2008/descriptions/solver_25.pdf, 2008, SAT Race 2008 Solver Description.
[4] G. Audemard and L. Simon, "Glucose and syrup in the SAT'17," in *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*. University of Helsinki, 2017, pp. 16–17. [Online]. Available: http://hdl.handle.net/10138/224324

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Maple_LCM_M1 and Maple_LCM+BCrestart

Zhen Li
School of Computer Science & Technology
Huazhong University of Science & Technology
Wuhan, China
violetcrestfall@hotmail.com

Kun He
School of Computer Science & Technology
Huazhong University of Science & Technology
Wuhan, China
brooklet60@hust.edu.cn

***Abstract*—Maple_LCM_M1 is a patched Maple_LCM solver by slightly modifying the conflict analyzing function. Maple_LCM+BCrestart adds a new restarting strategy to the Maple_LCM solver. Maple_LCM+BCrestart_M1 combines both patches in one solver.**

I. INTRODUCTION

The Maple_LCM solver submitted to the SAT Competition 2017, developed by Chu-Min Li's team, won the first prize of the 2017 competition. Our team is trying to improve its performance by applying small patches to the original solver. In Maple_LCM_M1, one of the new versions, we adjusted the coefficients of activity bumping process for each variable involved in conflicts. The farther the clause containing the variable from the conflict clause, the less activity it will gain. In Maple_LCM_BCrestart, we present a new restarting strategy that the solver will restart once the average conflicts per decision level reach threshold. The third new solver Maple_LCM+BCrestart_M1 combines the modifications in the two new solvers.

II. MAPLE_LCM_M1

When a conflict is reached, the original behavior of Maple_LCM is to bump the activity of all variables involved in the conflict 0.5 times var_inc. In the Maple_LCM_M1 version, the multiplier 0.5 varies from clause to clause. The initial value of the multiplier is 0.5 when processing variables of the initial conflict clause. It will be assigned to 0.9 times itself at each round of iteration during the learnt clause generation process.

We think the modification makes sense, as the farther an assignment of a variable in the conflict, the less impact it has to contribute the conflict. And, the multiplier decaying rate might be adjusted for a better performance.

III. MAPLE_LCM+BCRESTART

Maple_LCM+BCrestart is also based on Maple_LCM. It calculates the total conflicts reached after the last restart and checks the average conflicts per decision level. If the average conflicts reaches some threshold, we will run a restart. Maple_LCM+BCrestart divide the search into three phases. The first phase let the search to learn from the conflicts, then in the second phase, Maple_LCM+BCrestart will restart when the average of conflicts per decision level reaches a suit threshold. At last, when the search comes to the leaves of search tree, reduce the restarts. The value of average conflicts illustrates the impact of a decision variable on constructing new conflicts. The search tree is better when it is shorter and more balanced. Given the limit amount of learnt information, this number can be used to determine whether the current search space is worth for keeping the current searching process.

IV. MAPLE_LCM+BCRESTART_M1

This solver just combines the modifications in Maple_LCM_M1 and Maple_LCM+BCrestart.

REFERENCES

[1] Mao Luo, Chu-Min Li, Fan Xiao, Felip Many, Zhipeng Lu. An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17), 2017, Melbourne, Pages 703-711.

[2] JIA H L,VIJAY G,PASCAL P et al. Exponential recency weighted average branching heuristic for SAT solvers. [C]// Proceedings of 30th AAAI Conference on Artificial Intelligent. New York: AAAI Press, 2016:3434-3440.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

`Maple_LCM_Dist_ChronoBT`: Featuring Chronological Backtracking

Vadim Ryvchin and Alexander Nadel
Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
Email: {vadim.ryvchin,alexander.nadel}@intel.com

***Abstract*—This is the system description of the solver `Maple_LCM_Dist_ChronoBT`, submitted to the SAT Competition 2018. We have integrated chronological backtracking [3] into the SAT Competition 2017 [1] winner, `Maple_LCM_Dist` [2].**

I. Description

The goal of our submission is to test the performance of Chronological Backtracking (CB) [3] in the settings of the latest SAT competition. In our solver–`Maple_LCM_Dist_ChronoBT`, we updated `Maple_LCM_Dist` [2] with CB (configuration $\{\texttt{T} = 100, \texttt{C} = 4000\}$) based on the results in [3].

References

[1] M. Heule, M. Järvisalo, and T. Balyo. Sat competition 2017. https://baldur.iti.kit.edu/sat-competition-2017/.
[2] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 703–711. ijcai.org, 2017.
[3] A. Nadel and V. Ryvchin. Chronological backtracking. In *Theory and Applications of Satisfiability Testing - SAT 2018. Proceedings*, 2018. Accepted to publication.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Maple_LCM_Scavel and Maple_LCM_Scavel_200

1st Yang Xu, 4th Shuwei Chen
School of Mathematics
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
xuyang@home.swjtu.edu.cn, swchen@home.swjtu.edu.cn

2nd Guanfeng Wu, 3rd Qingshan Chen
School of Information Science and Technology
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
wl520gx@gmail.com, qschen@home.swjtu.edu.cn

***Abstract*— This document describes Maple_LCM_Scavel SAT solver.**

I. Introduction

Maple_LCM_Scavel is an improved SAT solver based on Maple_LCM_Dist. The improvements are made mainly from the following two aspects. First, we changed the evaluation method of VSIDS, so that the better branch decision variable can be selected. The second is that we improved the evaluation method of learning clauses so that better clauses can be preserved.

II. Two Improvement

A. Dynamic comprehensive variable activity evaluation

LIANG et al. [1] analyzed the role of the number of conflicts in the global conflict, where the variable was used in the latest conflict analysis, and proposed the Conflict History-based Branching strategy (CHB). The increment of variable activity is calculated based on the formula (1).

$$s' = (1-\alpha)\cdot s + \alpha \cdot r \qquad (1)$$

Where α is the interval of increment whose initial value is 0.4 and the attenuation rate is 10-6 after each conflict, r is the reward value. The variable is used for conflict analysis and derivation of learning clauses, which should be endowed dynamic activity. Therefore, we put forward a heuristic branch decision algorithm that provides a comprehensive evaluation of the decision level of the effective decision variable and its role in the conflict, called as Dynamic Activity algorithm (DA)[2]. Its framework is similar to that of EVSIDS, and is expressed as follows.

DA set an activity counter ‘s’ for every variable, the increment of ‘s’ can be calculated as (3).

$$s' = s + W(v)\cdot f \quad (0 < f < 1) \qquad (2)$$

Where f is the increment factor , W is a function of decision level and conflict level, and is defined as (4).

$$W(v) = \alpha \cdot \frac{vLevel}{nLevels} + (1-\alpha)\cdot \frac{vConflict}{nConflicts} \quad (0 \le \alpha \le 1) \qquad (3)$$

Where $vLevel$ is the latest decision level when the variable is treated as decision variable. $nLevels$ is the current decision level. $vConflict$ is the total number of conflicts when taking this decision variable. $nConflicts$ is the global number of conflicts. α is the regulation factor, and normally takes the value of 0.7. The $vLevel$ of a variable is set to 0 when its value is unassigned. If a variable is in the learn clauses or is used in the conflict analysis, its activity will be updated automatically.

B. Improved learning clause management strategy

Both the activity evaluation and LBD evaluation adopted a more aggressive deletion strategy. According to the experiments, LBD[3] or Activity[4] based solvers continually produce a large number of learning clauses, at the same time delete clauses in frequently. The clauses which have high priority to be deleted have less opportunity to be used in conflict analysis. In other words, the probability of the good clauses to be preserved will usually become low. It can be inferred that the probability of a clause being deleted is closely related to the number of times it is used.

Maple_LCM_Scavel sets a threshold for the number of the learning clauses being used in conflict analysis, *NBused*, named “*NB_threshold_value*”. The initial value of *NBused* of each learned clauses is 0. If the learned clause is used in conflict analysis, increase the value of *NBused* by 1. In the compression procedure (in “reducedDB” method), when the clause’s *NBused* value reachs the specified threshold *NB_threshold_value*, it will be deleted. The experimental results show that, the threshold value being 150 or 200 will get best results.

References

[1] LIANG J H, GANESH V, POUPART P, et al. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers[C]//AAAI. 2016: 3434-3440.

[2] Qingshan Chen, Yang Xu, Guanfeng Wu, Xingxing He. Conflicting Rate Based Branching Heuristic for CDCL SAT Solvers. In Proceedings of 12th International Conference on Intelligent Systems and Knowledge Engineering, 2017, 361-365.

[3] SIMON L, AUDEMARD G. Predicting learnt clauses quality in modern sat solver [C]. In Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09). Menlo Park: AAAI Press, 2009: 399-404.

[4] Minisat homepage [EB/OL]. http://minisat.se/MiniSat.html.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS in the 2018 Competition

Jia Hui Liang*‡, Chanseok Oh†‡, Krzysztof Czarnecki*, Pascal Poupart*, Vijay Ganesh*
* University of Waterloo, Waterloo, Canada
† Google, New York, United States
‡ Joint first authors

***Abstract*—This document describes the SAT solvers MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS that implement our machine learning branching heuristics called the *learning rate branching heuristic* (LRB) and the *conflict history-based branching heuristic* (CHB).**

I. Introduction

A good branching heuristic is vital to the performance of a SAT solver. Glancing at the results of the previous competitions, it is clear that the VSIDS branching heuristic is the de facto branching heuristic among the top performing solvers. We are submitting two unique solvers with a new branching heuristic called the *learning rate branching heuristic* (LRB) [1] and another solver with the *conflict history-based branching heuristic* (CHB) [2].

Our intuition is that SAT solvers need to prune the search space as quickly as possible, or more specifically, learn a high quantity of high quality learnt clauses. In this perspective, branching heuristics can be viewed as a bi-objective problem to select the branching variables that will simultaneously maximize both the quantity and quality of the learnt clauses generated. To simplify the optimization, we assumed that the first-UIP clause learning scheme will generate good quality learnt clauses. Thus we reduced the two objectives down to just one, that is, we attempt to maximize the quantity of learnt clauses.

II. Learning Rate Branching

We define a concept called *learning rate* to measure the quantity of learnt clauses generated by each variable. The learning rate is defined as the following conditional probability, see our SAT 2016 paper for a detailed description [1].

$$\begin{aligned}learningRate(x) = \mathbb{P}(&Participates(x) \mid \\ &Assigned(x) \wedge SolverInConflict)\end{aligned}$$

If the learning rate of every variable was known, then the branching heuristic should branch on the variable with the highest learning rate. The learning rate is too difficult and too expensive to compute at each branching, so we cheaply estimate the learning rate using multi-armed bandits, a special class of reinforcement learning. Essentially, we observe the number of learnt clauses each variable participates in generating, under the condition that the variable is assigned and the solver is in conflict. These observations are averaged using an exponential moving average to estimate the current learning rate of each variable. This is implemented using the well-known *exponential recency weighted average algorithm* for multi-armed bandits [3] with learning rate as the reward.

Lastly, we extended the algorithm with two new ideas. The first extension is to encourage branching on variables that occur frequently on the reason side of the conflict analysis and adjacent to the learnt clause during conflict analysis. The second extension is to encourage locality of the branching heuristic [4] by decaying unplayed arms, similar to the decay reinforcement model [5], [6]. We call the final branching heuristic with these two extensions the *learning rate branching heuristic*.

III. Conflict History-Based Branching

The *conflict history-based branching heuristic* (CHB) precedes our LRB work. CHB also applies the exponential recency weighted average algorithm where the reward is the reciprocal of the number of conflicts since the assigned variable last participated in generating a learnt clause. See our paper for more details [2].

IV. Solvers

All the solvers are modifications of COMiniSatPS [7]. We used the same COMiniSatPS version that also participates in the competition [8]. This year's solvers are basically identical to the last year's solvers, fixing only a few minor bugs:

- Fixed the bug in the last year's MapleCOMSPS_LRB_VSIDS [9] that crippled the on-the-fly probing techniques.
- Fixes two minor bugs in the base solver COMiniSatPS [8].

V. Availability and License

Source is available for download for all the versions described in this paper. All the solvers use the same license as COMiniSatPS. Note that the license of the M4RI library (which COMiniSatPS uses to implement Gaussian elimination) is GPLv2+.

Acknowledgment

We thank specifically the authors of Glucose, GlueMiniSat, Lingeling, CryptoMiniSat, and MiniSat.

References

[1] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning Rate Based Branching Heuristic for SAT Solvers," in *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'16, 2016.

[2] ——, "Exponential Recency Weighted Average Branching Heuristic for SAT Solvers," in *Proceedings of AAAI-16*, 2016.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[4] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, "Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers," in *Hardware and Software: Verification and Testing*. Springer, 2015, pp. 225–241.

[5] I. Erev and A. E. Roth, "Predicting How People Play Games: Reinforcement Learning in Experimental Games with Unique, Mixed Strategy Equilibria," *American Economic Review*, vol. 88, no. 4, pp. 848–881, 1998.

[6] E. Yechiam and J. R. Busemeyer, "Comparison of basic assumptions embedded in learning models for experience-based decision making," *Psychonomic Bulletin & Review*, vol. 12, no. 3, pp. 387–402.

[7] C. Oh, "Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL," Ph.D. dissertation, New York University, 2016.

[8] ——, "COMiniSatPS Pulsar and GHackCOMSPS in the 2018 Competition," in *SAT Competition*, 2018.

[9] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Pascal, "MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS," in *SAT Competition*, 2017.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

painless-mcomsps and painless-mcomsps-sym

Ludovic Le Frioux*†, Hakan Metin†, Souheib Baarir*† ‡, Maximilien Colange*, Julien Sopena†§, Fabrice Kordon†
*LRDE, EPITA, Kremlin-Bicêtre, France
†Sorbonne Université, LIP6, CNRS, UMR 7606, Paris, France
‡Université Paris Nanterre, France
§INRIA, Delys Team, Paris, France

***Abstract*—This paper describes the solvers painless-mcomsps, and painless-mcomsps-sym submitted to the parallel track of the SAT Competition in 2018. They are parallel solvers instantiated with PArallel INstantiabLE Sat Solver (PaInleSS) framework and using MapleCOMSPS as core sequential solver.**

I. Introduction

painless-mcomsps and painless-mcomsps-sym are parallel SAT solvers built by instantiating components of the PaInleSS parallel framework [1]. They are Portfolio based solvers implementing a diversification strategy, fine control of learnt clause exchanges, and using MapleCOMSPS [2] as a core sequential solver. Moreover, painless-mcomsps-sym included dynamic symmetry breaking [3] by using the Cosy library.

Section II gives an overview on PaInleSS framework. Section III details the implementation of painless-mcomsps using PaInleSS and MapleCOMSPS. Section IV explains how dynamic symmetry breaking has been incorporated in painless-mcomsps to give the solver painless-mcomsps-sym.

II. Description of PaInleSS

PaInleSS is a framework that aims at simplifying the implementation and evaluation of parallel SAT solvers for many-core environments. Thanks to its genericity and modularity, the components of PaInleSS can be instantiated independently to produce new complete solvers.

The main idea of the framework is to separate the technical components (e.g., those dedicated to the management of concurrent programming aspects) from those implementing heuristics and optimizations embedded in a parallel SAT solver. Hence, the developer of a (new) parallel solver concentrates his efforts on the functional aspects, namely parallelization and sharing strategies, thus delegating implementation issues (e.g., data concurrent access protection mechanisms) to the framework.

Three main components arise when treating parallel SAT solvers: *sequential engines*, *parallelization*, and *sharing*. These form the global architecture of PaInleSS.

A. Sequential Engines

The core element that we consider in our framework is a sequential SAT solver. This can be any CDCL state-of-the art solver. Technically, these engines are operated through a generic interface providing basics of sequential solvers: *solve, interrupt, add clauses*, etc.

Thus, to instantiate PaInleSS with a particular solver, one needs to implement the interface according this engine.

B. Parallelization

To built a parallel solver using the aforementioned engines, one needs to define and implement a parallelization strategy. Portfolio and Divide-and-Conquer are the basic known ones. Also, they can be arbitrary composed to form new strategies.

In PaInleSS, a strategy is represented by a tree-structure of arbitrary depth. The internal nodes of the tree represent parallelization strategies, and leaves are core engines. Technically, the internal nodes are implemented using WorkingStrategy component and the leaves are instances of SequentialWorker component.

Hence, to develop its own parallelization strategy, the user should create one or more strategies, and build the required tree-structure.

C. Sharing

In parallel SAT solving, the exchange of learnt clauses warrants a particular focus. Indeed, beside the theoretical aspects, a bad implementation of a good sharing strategy may dramatically impact the solver's efficiency.

In PaInleSS, solvers can export (import) clauses to (from) the others during the resolution process. Technically, this is done by using lockfree queues [4]. The sharing of these learnt clauses is dedicated to particular components called Sharers. Each Sharer in charge of sets of producers and consumers and its behaviour reduces to a loop of sleeping and exchange phases.

Hence, the only part requiring a particular implementation is the exchange phase, that is user defined.

III. painless-mcomsps

This section describes the overall behaviour of our competing instantiation named painless-mcomsps. Its architecture is highlighted in Fig. 1.

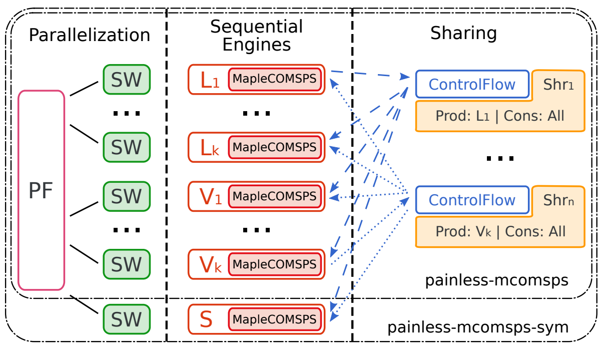


Fig. 1. Architecture of `painless-mcomsps`.

A. *Sequential Engines: `MapleCOMSPS`*

`MapleCOMSPS` is a sequential solver that finished second of the main track of the SAT Competition 2017. It is based on `MiniSat` [5], and uses as decision heuristics the classical Variable State Independent Decaying Sum (VSIDS) [6], and newly defined Learning Rate Branching (LRB) [7]. These heuristics are used in one-shot phases: first LRB, then VSIDS. Moreover, it uses Gaussian Elimination (GE) at preprocessing time.

We adapt this solver for the parallel context as follows: (1) we parametrized the solver to select either LRB, or VSIDS for all solving process (noted respectively, `L` and `V`); (2) we added callbacks to export and import clauses; (3) we added an option to use or not the GE preprocessing.

B. *Parallelization: Portfolio and Diversification*

`painless-mcomsps` is a solver implementing a basic Portfolio strategy (`PF`), where the underlying core engines are either `L` or `V` instances.

For each type of instances, we apply a sparse random diversification similar to the one introduced in [8]. That is for each group of k solvers, the initial phase of a solver is randomly set according the following settings: every variable gets a probability $1/2k$ to be set to false, $1/2k$ to true, and $1 - 1/k$ not to be set.

Moreover, only one of the solvers performs the GE preprocessing.

C. *Sharing: Controlling the Flow of Shared Clauses*

In `painless-mcomsps`, the sharing strategy `ControlFlow` is inspired from the one used by [8]. We instantiate a `Sharer` per solver (the producer). It gets clauses from this producer and exports some of them to all others (the consumers).

The exchange strategy is defined as follows: each solver exports clauses having a LBD value under a given threshold (2 at the beginning). Every 0.5 seconds, 1500 literals (the sum of the size of the shared clauses) are selected by the `Sharer` and dispatched to consumers. The LBD threshold of the concerned solver is increased if an insufficient number of literals (less than 1200) are dispatched.

IV. `PAINLESS-MCOMSPS-SYM`

This section describes the overall behaviour of our competing instantiation named `painless-mcomsps-sym`. Its architecture is highlighted in Fig. 1.

A. *Dynamic Symmetry Breaking*

The idea we bring is to break symmetries *on the fly*: when the current partial assignment can not be a prefix of a *lex-leader* (of an orbit), a constraint called *esbp* is generated. This constraint prunes this forbidden assignment and all its extensions.

B. *Integration to `painless-mcomsps`*

`Cosy`, a C++ library, provides dynamic symmetry breaking primitives. We integrated the library into `MapleCOMSPS`, and we added a parameter to activate or not dynamic symmetry breaking mode.

In `painless-mcomsps-sym`, there is only one solver that used dynamic symmetry breaking, we call it `S` in the Fig. 1. This solver uses the VSIDS heuristics.

The solver `S`, receives clauses from all the others, but it does not export clauses.

Acknowledgment

We would like to thank Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart, the authors of `MapleCOMSPS`.

References

[1] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, “Painless: A framework for parallel sat solving,” in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 233–250, Springer, 2017.

[2] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, “Maplecomsps lrb vsids, and maplecomsps chb vsids,” *SAT COMPETITION 2017*, pp. 20–21.

[3] M. Hakan, B. Souheib, C. Maximilien, and K. Fabrice, “Cdclsym: Introducing effective symmetry breaking in sat solving,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2018. To appear.

[4] M. M. Michael and M. L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 267–275, ACM, 1996.

[5] N. Eén and N. Sörensson, “An extensible sat-solver,” in *Theory and applications of satisfiability testing*, pp. 502–518, Springer, 2003.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *38th annual Design Automation Conference*, pp. 530–535, ACM, 2001.

[7] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Learning rate based branching heuristic for sat solvers,” in *Theory and Applications of Satisfiability Testing*, pp. 123–140, Springer, 2016.

[8] T. Balyo, P. Sanders, and C. Sinz, “Hordesat: A massively parallel portfolio sat solver,” in *int. conf. on Theory and Applications of Satisfiability Testing*, pp. 156–172, Springer, 2015.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

probSAT

Adrian Balint
Ulm University
Ulm,Germany

Uwe Schöning
Ulm University
Ulm,Germany

***Abstract*—We describe some details about the SLS solver probSAT, a simple and elegant SLS solver based on probability distributions, a heuristic first presented in the SLS solver Sparrow [3].**

I. Introduction

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

II. Main Techniques

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

Algorithm 1: ProbSAT

```
Input  : Formula $F$ , $maxTries$, $maxFlips$
Output: satisfying assignment a or UNKNOWN
1  for $i = 1$ to $maxTries$ do
2      a ← randomly generated assignment
3      for $j = 1$ to $maxFlips$ do
4          if (a is model for $F$) then
5              return a
6          $C_u$ ← randomly selected unsat clause
7          for $x$ in $C_u$ do
8              compute $f(x, \mathbf{a})$
9          $var$ ← random variable $x$ according to
           probability $\frac{f(x,\mathbf{a})}{\sum_{z \in C_u} f(z,\mathbf{a})}$
10         flip($var$)
11 return UNKNOWN;
```

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

$$f(x, \mathbf{a}) = (c_b)^{-break(x,\mathbf{a})}$$

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

III. Parameter settings

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

k	fct	cb	ϵ
3	0	2.06	0.9
4	1	3	-
5	1	3.88	-
6	1	4.6	-
≥ 7	1	4.6	-

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

IV. Further Details

ProbSAT is implemented in C and uses a new XOR implementation scheme for the flip procedure described in detail in [4].

Acknowledgment

This project was funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1.

References

[1] Balint, A. et al: EDACC - An advanced Platform for the Experiment Design, Administration and Analysis of Empirical Algorithms In: *Proceedings of LION5*, pages 586–599.
[2] Adrian Balint, Uwe Schöning: Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break Lecture Notes in Computer Science, 2012, Volume 7317, Theory and Applications of Satisfiability Testing - SAT 2012, pages 16-29
[3] Balint, A., Fröhlich, A.: Improving stochastic local search for SAT with a new probability distribution. *Proceedings of SAT 2010*, pages 10–15, 2010.
[4] Balint, A., Biere, A., Fröhlich, A., Schöning, U.: Efficient implementation of SLS solvers and new heuristics for k-SAT with long clauses. *Proceedings of SAT 2014*

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Riss 7.1

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—The sequential SAT solver RISS combines a heavily modified Minisat-style solving engine of GLUCOSE 2.2 with a state-of-the-art preprocessor COPROCESSOR and adds many modifications to the search process. RISS allows to use inprocessing based on COPROCESSOR. Based on this RISS, we create a parallel portfolio solver PRISS, which allows clause sharing among the incarnations, as well as sharing information about equivalent literals.**

I. Introduction

The CDCL solver RISS is a highly configurable SAT solver based on MINISAT [1] and GLUCOSE 2.2 [2], [3], implemented in C++. Many search algorithm extensions have been added, and RISS is equipped with the preprocessor COPROCESSOR [4]. Furthermore, RISS supports automated configuration selection based on CNF formulas features, emitting DRAT proofs for many techniques and comments why proof extensions are made, and incremental solving. The solver is continuously tested for being able to build, correctly solve CNFs with several configurations, and compile against the IPASIR interface. For automated configuration, RISS is also able to emit its parameter specification on a detail level specified by the user. The repository of the solver provides a basic tutorial on how it can be used, and the solver provides parameters that allow to emit detailed information about the executed algorithm in case it is compiled in debug mode (look for "debug in the help output). While RISS also implements model enumeration, parallel solving, and parallel model enumeration, this document focusses only on the differences to RISS 7, which has been submitted to SAT Competition 2017.

II. SAT Competition Specifics

While last years submissions did not make full use of the implemented formula simplification techniques, the configuration submitted to the NoLimit track of the competition now uses XOR reasoning [5] and cardinality reasoning [6] again. These techniques have been disabled last year, as they can not print DRAT proofs efficiently.

III. Modifications of the Search - LCM

Last years winning solver family was the "Maple_LCM" solvers, which are based on learned clause minimization (LCM) [7]. The implementation of LCM in RISS has a few modifications to those in the original publication, namely:

1) apply LCM after every second reduction
2) when simplifying a clause, try to simplify it in reverse order as well
3) when a clause could be reduced, use a resolution based simplification to reduce the size further

The first modification helps to reduce the overhead LCM might introduce on clauses that would be removed in the next clause removal phase. The second modifications uses a Bloom filter like effect following the assumption: if a clause can be reduced by performing vivification in one particular order, then using the reverse order might allow to drop even more literals. On the other hand, for clauses that cannot be reduced, no additional cost is introduced. Hence, the second modification focusses on clauses that can be reduced. Finally, the last modification makes the reduction more effective: while vivification stops when a conflict is found and proceeds with the current set of literals, applying conflict analysis with resolution allows to remove further redundant literals. Cycles for reduction are only spend on clauses that could be reduced in the first place.

IV. Modifications of the Simplifier

To be able to emit DRAT proofs for the main track, many simplification techniques of Coprocessor had to be disabled, among them reasoning with XORs and cardinality constraints [6]. In the NoLimit track, these techniques are enabled again.

V. Availability

The source of the solver is publicly available under the LGPL v2 license at https://github.com/conp-solutions/riss. The version with the git tag "v7.1.0" is used for the submission. The submitted starexec package can be reproduced by running "./scripts/make-starexec.sh" on this commit.

Acknowledgment

The author would like to thank the developers of GLUCOSE 2.2 and MINISAT 2.2. The development of this project was supported by the DFG grant HO 1294/11-1.

References

[1] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Heidelberg: Springer, 2004, pp. 502–518.
[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI 2009*, C. Boutilier, Ed. Pasadena: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.
[3] ——, "Refining restarts strategies for sat and unsat," in *CP'12*, 2012, pp. 118–126.
[4] N. Manthey, "Coprocessor 2.0 – a flexible CNF simplifier," in *SAT 2012*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Heidelberg: Springer, 2012, pp. 436–441.
[5] M. Soos, K. Nohl, and C. Castelluccia, "Extending sat solvers to cryptographic problems," in *SAT 2009*, O. Kullmann, Ed. Heidelberg: Springer, 2009, pp. 244–257.

[6] A. Biere, D. Le Berre, E. Lonca, and N. Manthey, “Detecting cardinality constraints in CNF,” in *Theory and Applications of Satisfiability Testing – SAT 2014*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer International Publishing, 2014, pp. 285–301. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09284-3_22

[7] F. X. F. M. Z. L. Mao Luo, Chu-Min Li, “An effective learnt clause minimization approach for cdcl sat solvers,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 703–711. [Online]. Available: https://doi.org/10.24963/ijcai.2017/98

SparrowToRiss 2018

Adrian Balint
adrian.c.balint@gmail.com

Norbert Manthey
nmanthey@conp-solutions.com

***Abstract*—SparrowToRiss is a combination of the solver Sparrow and Riss. SparrowToRiss is first trying to solve the problem with Sparrow, limiting its execution to $5 \cdot 10^8$ flips and then passes the assignment found to the CDCL solver Riss, which uses this information for initialization and then tries to solve the problem.**

The SLS solver Sparrow is the same version as used in 2014. The solver Riss is used in version 7.1, which is also submitted to this competition.

I. Introduction

While in 2014, where this solver combination was submitted for the first time, the benchmark was split in industrial and combinatorial families, recent competitions did not insist on this split any more. As we do not know the origin of submitted and used benchmarks for this years competition, and did not see solver from previous competitions that performed well on crafted formulas, we submit SparrowToRiss again, to validate whether the combination of a SLS and a CDCL solver is still a reasonable solving approach. The submission is furthermore supported be the results found in [1], where formula simplification is used to boost the efficiency of SLS solvers on crafted families. The best found technique together with Sparrow represents the basis of our solver Sparrow+CP3. As SLS solvers cannot show unsatisfiability, we run a CDCL solver after a fixed amount of $5 \cdot 10^8$ flips, so that the overall solver behavior stays deterministic.

II. Main Techniques

Sparrow is a clause weighting SLS solvers that uses promising variables and probability distribution based selection heuristics. It is described in detail in [2]. Compared to the original version, the one submitted here is updating weights of unsatisfied clauses in every step where no promising variable can be found.

The built-in preprocessor CP3 is an extension of Coprocessor 2 [20], and received updates. Compared to the submitted version of Riss to the SAT competition 2017, no new techniques have been added.

The CDCL solver Riss uses the Minisat search engine [17], more specifically the extensions added in Glucose 2.2 [18], [19]. Furthermore, Riss is equipped with the preprocessor Coprocessor.

The combination of the Sparrow and Riss, called SparrowToRiss, does not simply execute the two solvers after each other, but also forwards information from the SLS solver to the CDCL solver: when Sparrow terminates, it outputs its last full assignment in chronological order (i.e. the oldest variable first), which is used to initialize the phase saving of Riss, such that the first decisions of Riss follow this assignment. In a brief empirical evaluation this communication turned out to be useful. The solvers are also able to forward the information about the age of the variables in the SLS search. This data could be used to initialize the activities of the variables inside Riss. However, this feature is not enabled in the used configuration.

III. Main Parameters

Sparrow is using the same parameters as Sparrow 2011.

The configuration of CP3 has been tuned for Sparrow in [1] on the SAT Challenge 2012 satisfiable hard combinatorial benchmarks. The configuration used in 2018 is the same configuration used in the version of 2014.

The main parameters of Riss control how the formula simplification of CP3 is executed. A major modification of Riss is the addition learned clause minimization [**?**], which has been slightly modified. The configuration of CP3 has been tuned for Glucose 2.2 in [1] on the SAT Challenge 2012 application benchmark. The final setup of the preprocessor inside Riss uses the following techniques: UP, SUB+STR (producing all resolvents for ternary clauses), Unhide without *hidden literal elimination* [10] and 5 iterations, BVE without on the fly BCE. Furthermore, if no proof should be emitted, Gaussian Elimination and Cardinality Constraint [21] reasoning is applied, as well as Covered Literal Elimination [22].

For SparrowToRiss it can be chosen whether to forward the last assignment, or the activity information.

IV. Implementation details

Sparrow is implemented in C. The solver Riss is build on top of Minisat 2.2 and Glucose 2.2, and is implemented in C++.

V. Availability

The source code of Riss (including CP3) is available at https://github.com/conp-solutions/riss under LGPL v2.1. Sparrow is available at https://github.com/adrianopolus/Sparrow.

References

[1] A. Balint and N. Manthey, "Boosting the Performance of SLS and CDCL Solvers by Preprocessor Tuning," in *Pragmatics of SAT*, 2013.

[2] A. Balint and A. Fröhlich, "Improving stochastic local search for sat with a new probability distribution," in *Proceedings of the 13th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 10–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14186-7_3

[3] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 61–75. [Online]. Available: http://dx.doi.org/10.1007/11499107_5

[4] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 129–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_10

[5] N. Manthey, M. J. H. Heule, and A. Biere, "Automated reencoding of boolean formulas," in *Proceedings of Haifa Verification Conference 2012*, 2012.

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

[7] M. Heule, M. Järvisalo, and A. Biere, "Covered clause elimination," *CoRR*, vol. abs/1011.5202, 2010.

[8] M. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for cnf formulas," in *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 357–371. [Online]. Available: http://dl.acm.org/citation.cfm?id=1928380.1928406

[9] A. V. Gelder, "Toward leaner binary-clause reasoning in a satisfiability solver," *Ann. Math. Artif. Intell.*, vol. 43, no. 1, pp. 239–253, 2005.

[10] M. J. H. Heule, M. Järvisalo, and A. Biere, "Efficient cnf simplification based on binary implication graphs," in *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, ser. SAT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 201–215. [Online]. Available: http://dl.acm.org/citation.cfm?id=2023474.2023497

[11] W. Wei and B. Selman, "Accelerating random walks," in *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, ser. CP '02. London, UK, UK: Springer-Verlag, 2002, pp. 216–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=647489.727142

[12] N. Manthey and P. Steinke, "Quadratic Direct Encoding vs. Linear Order Encoding," in *First International Workshop on the Cross-Fertilization Between CSP and SAT(CSPSAT'11)*, 2011.

[13] M. N. V. Van Hau Nguyen and S. Hölldobler, "Application of hierarchical hybrid encoding to efficient translation of a csp to sat," Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, Tech. Rep., 2013.

[14] A. del Val, "On 2-sat and renamable horn," in *AAAI/IAAI*, H. A. Kautz and B. W. Porter, Eds. AAAI Press / The MIT Press, 2000, pp. 279–284.

[15] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *AAAI*, B. Hayes-Roth and R. E. Korf, Eds. AAAI Press / The MIT Press, 1994, pp. 337–343.

[16] K. Gebhardt and N. Manthey, "Parallel Variable Elimination on CNF Formulas," in *Pragmatics of SAT*, 2013.

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

[18] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. 21st Int. Joint Conf. on Artifical Intelligence (IJCAI '09)*. Morgan Kaufmann, 2009, pp. 399–404.

[19] ——, "Refining restarts strategies for sat and unsat," in *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming*, ser. CP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 118–126. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33558-7_11

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

[21] A. Biere, D. Le Berre, E. Lonca, and N. Manthey, "Detecting cardinality constraints in CNF," in *Theory and Applications of Satisfiability Testing – SAT 2014*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer International Publishing, 2014, pp. 285–301. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09284-3_22

[22] N. Manthey and T. Philipp, "Formula simplifications as DRAT derivations," in *KI 2014: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, C. Lutz and M. Tielscher, Eds., vol. 8736. Springer Berlin Heidelberg, 2014, pp. 111–122.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

BreakIDGlucose and BreakIDGlucoseSEL

Jo Devriendt
University of Leuven
Leuven, Belgium

Bart Bogaerts
University of Leuven
Leuven, Belgium

Abstract—**BreakIDGlucose and BreakIDGlucoseSEL combine the Glucose SAT solver with the symmetry detection tool BreakID. The former breaks symmetry statically by employing classic symmetry breaking formulas, while the latter handles it dynamically by employing symmetric explanation learning.**

I. Introduction

Many real-world problems exhibit symmetry, but the SAT competition and SAT race seldomly feature solvers who are able to exploit symmetry properties. Similarly to 2013 and 2016, we submit a static symmetry breaking approach BreakIDGlucose. This year, we also submit the dynamic symmetry handling approach BreakIDGlucoseSEL.

II. Main Techniques

As symmetry breaking preprocessor we use BreakID 2.3 [1]. Compared to BreakID 2.2 used in 2016, version 2.3 fixes a small bug in the automorphism graph construction routine. As SAT-solver we employ Glucose 4.0 [2] which is modified to support *symmetric explanation learning* (SEL) [3]. [1] The difference between the approaches is that BreakIDGlucose employs BreakID's symmetry breaking formulas and disables SEL, while BreakIDGlucoseSEL has SEL activated and forgoes BreakID's symmetry breaking formulas.

The reason we use the modified version of Glucose as backend for BreakIDGlucose is that we want to compare both static and dynamic versions of symmetry handling as equally as possible, which requires exactly the same backend solver.

III. Main Parameters

The main user-provided parameters control:

- How much time is allocated to symmetry detection. The builtin graph automorphism tool Saucy [4] gets 100 seconds to detect symmetry generators.
- How large the symmetry breaking formulas are allowed to grow, measured in the number of auxiliary variables introduced by a symmetry breaking formula. We limit this to 50 auxiliary variables.
- How many generators BreakIDGlucoseSEL uses to handle row interchangeability symmetry groups. We employ a quadratic number of row-swaps (e.g., swapping every two pigeons of a pigeonhole problem). The alternative would have been a linear amount of swaps (e.g., swapping every two *consecutive* pigeons of a pigeonhole problem).

[1]We also implemented a small Glucose hack called inIDGlucose, which is presented in a corresponding system description.

IV. Special Algorithms, Data Structures, and Other Features

BreakIDGlucoseSEL employs a second *symmetrical clause store* for clauses symmetrical to the ones that are asserting in the current search state. These symmetrical clauses $\sigma(c)$ are added to the main learned clause store only when they become unit or conflicting, and otherwise are quickly forgotten after a backjump causes the original clause c to revert to non-unit status. As usual, a two-watched literal scheme keeps track of the truth value of any clause in the symmetrical clause store.

V. SAT Competition 2018 Specifics

BreakIDGlucose and BreakIDGlucoseSEL participate in the No-Limit track since BreakIDGlucoseSEL constructs proofs using a symmetry rule not present in the DRAT format.

VI. Availability

Source code and documentation for BreakID is available under a non-commercial license [5]. Source code and documentation for the extension of Glucose with symmetric explanation learning is freely available [6].

Acknowledgment

We would like to thank the authors of Saucy, MiniSat [7] and Glucose.

References

[1] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker, "Improved static symmetry breaking for SAT," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 104–122. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-40970-2

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

[3] J. Devriendt, B. Bogaerts, and M. Bruynooghe, "Symmetric explanation learning: Effective dynamic symmetry handling for SAT," in *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, ser. Lecture Notes in Computer Science, S. Gaspers and T. Walsh, Eds., vol. 10491. Springer, 2017, pp. 83–100. [Online]. Available: https://doi.org/10.1007/978-3-319-66263-3

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

[5] J. Devriendt and B. Bogaerts, "BreakID, a symmetry breaking preprocessor for SAT solvers," bitbucket.org/krr/breakid, 2015.

[6] J. Devriendt, "Glucose-SEL, an implementation of symmetric explanation learning in glucose 4.0," bitbucket.org/krr/glucose-sel, 2016.

[7] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

inIDGlucose

Jo Devriendt
University of Leuven
Leuven, Belgium

***Abstract*—inIDGlucose (pronounced “init-glucose”) is a submission to the Glucose Hack Track that initializes variable activity and phase based on a weighted literal occurrence count on the original CNF.**

I. Main Technique

The *activity* of a variable is the priority given to the variable when the solver selects a variable to decide. The *phase* of a variable is the value assigned to it when the solver decides the variable.

MiniSat-based solvers have an initial phase of false and an initial activity of 0 for all variables. As this does not take any information from the instance at hand into account, we hope to improve upon this by counting the occurrences of each literal in a clause the CNF, taking into account clause length as well.

The general idea behind our activity initialization is that variables whose literals occur both positively and negatively in short clauses are probably hard to decide a good value for, and might be part of a lot of failing search branches. As activity tracks the amount of conflicts a variable is part of, such variables seem good candidates to start out with a high activity.

The general idea behind our phase initialization is that we want to maximize the number of satisfied clauses by assigning true or false to a variable, since this might speed up the derivation of satisfying assignments. Variables that occur mostly positively (resp. negatively) then should be assigned true (resp. false). Generally speaking, long clauses are easier to satisfy than short ones, so again the occurrence count of literals in short clauses should be more important than in long clauses.

II. Special Algorithms, Data Structures, and Other Features

We weigh the occurrence of a literal l by the inverse square of the length of the clause c, as this strongly reduces the importance of occurrences in long clauses. The *total weighted occurrence* (two) for a literal then is:

$$two(l) = \sum_{c \text{ s.t. } l \in c} \frac{1}{length(c)^2}$$

The initial activity for a variable v simply is the product

$$two(v) * two(\bar{v})$$

which is a measure for both the total weighted occurrence of a variable as well as the difference between the positive and negative occurrences of the variable. E.g., a variable v occurring as a pure literal will have an initial activity of 0, as either $two(v)$ or $two(\bar{v})$ will be 0.

The initial phase for a variable v is the truth value of

$$two(v) > two(\bar{v})$$

which measures whether the variable occurs mostly positively or mostly negatively. Note that in case of a tie, the inital phase is negative.

Note that the activity and phase are only *initialized* with the above values. During search, Glucose runs its customary phase caching and activity updating schemes.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

SCALOPE , PENELOPE_MDLC and GLUCOSE-3.0_PADC in SC18

Rodrigue Konan Tchinda and Clémentin Tayou Djamegni
University of Dschang
{rodriguekonanktr, dtayou}@gmail.com

***Abstract*—We provide in this paper a short description of our solvers SCALOPE , PENELOPE_MDLC and GLUCOSE-3.0_PADC submitted to the SC18. The first solver SCALOPE is a simplified implementation of the one we submitted to the SC17 [1]. The second solver PENELOPE_MDLC is a new one also based on PENELOPE 2014 [2] which aims to minimize clause duplication by using two learned clause databases: one for the clauses derived from conflicts analysis and the other for the imported clauses. This latter is subject to a special cleaning strategy since it can contains duplicate or already subsumed clauses. The third solver GLUCOSE-3.0_PADC is a GLUCOSE3.0 hack with periodical steps of deep database cleaning.**

I. INTRODUCTION

SAT solvers have become very efficient today. This efficiency is the result of a subtle combination of several feature such as unit propagation through watched literals, restarts strategies, dynamic branching and polarity heuristics, conflict clause analysis and clause learning. This latter is one of the most important feature of modern SAT solvers. In fact after each found conflict, a procedure is invoked to analyze it in order to produce an asserting clause. This asserting clause is then added to a learned clause database (i.e. this clause is learned) and is used to redirect the back-jumping level as well as to prevent the same conflict in the future as long as this clause is kept in the learned clause database. However during the search there can be a huge amount of learned and keeping all these clauses will have a negative impact on the performances of the solver by slowing down unit propagations. In parallel solvers, this issue is even more important since beside learned clause there are also imported clauses coming from other threads of the portfolio that are also added to the learned clause database. To cope with this issue, modern SAT solvers use learned clause database cleaning strategies where the common clause quality measures are the size or the LBD of the clause. Our solvers SCALOPE , PENELOPE_MDLC and GLUCOSE-3.0_PADC presented in this paper mainly use several forms of learned clause database management to achieve different objectives. These Solvers are described in details in the subsequent sections.

II. SCALOPE

The solver SCALOPE [1] that we submitted to SC17 was designed to improve the scalability of the portfolio by organizing threads in teams. It allowed an intensive communication between threads of the same team and limited communication between threads of different teams. This version used an explicit division of thread in team by providing to each team a different cooperation object. Hence there were in each team a particular thread that was in charge of exporting and importing information from and to other teams. Once the information were imported by this thread, the others could simply retrieve them via classical communications within the team. This year, we changed this implementation. Instead, we use one Cooperation object and dedicated channels between each pair of threads in order to share learned clauses as in PENELOPE. However the amount of clauses shared via a channel depends on the proximity of the threads i.e. whether they are in the same team or not. This simplification really reduce the complexity of the code. Furthermore inter-team communication is no longer restricted to unit clauses sharing as it was in the previous version. We now use the LBD [3] as a discriminant when sharing clauses. Hence only clauses that have good LBD scores are exported outside the team. In the current version, every learned clause with the LBD score lower than or equal to 2 is exported outside the teams.

III. PENELOPE_MDLC

PENELOPE_MDLC is a built on top of PENELOPE [2]. Unlike classical SAT solvers, PENELOPE_MDLC manages two separate learned clause databases per thread: one for the clauses that are derived from conflicts by the thread itself and the other for the clauses that were imported from other threads. PENELOPE_MDLC allows the move of a clause from the imported clause database to the learned clause database where the lifetime is greater for good quality clauses. Let Δ be the learned clause database and Γ the imported clause database. A clause c is moved from Γ to Δ whenever it propagates or is conflicting while none clause in $F \cup \Delta$ could do so. Hence no clause in $F \cup \Delta$ will be duplicated nor subsumed by another one. We could then apply different cleaning strategy on Δ and Γ. As such, it might be possible to have a more aggressive strategy in Γ since it can possibly contains duplicated or subsumed clauses. In the current version of PENELOPE_MDLC , Γ is implemented as a Queue which means that the FIFO strategy is used to manage the imported clause database. At each clause database cleaning step, a fraction (which we called $reduceFactor$ and which is such that $0 \leq reduceFactor \leq 1$) of the total number of clauses to delete is removed from Δ on the less useful clauses and the rest is removed from Γ on the oldest clauses. The rationale is that if a clause has not propagated any literal since its entrance in the queue, there can be more chances that the literals it could propagate are already propagated by other clauses present in $F \cup \Delta$. This can be the case when the clause is duplicated or is subsumed by another one. An Exception is however made on clauses with LBD ≤ 2. These latter are given a longer lifetime in Γ. Concretely, just before insertion in Γ during imports, each clause with $LBD \leq 2$ is given a maximum number of deletion attempts — which we call its lifetime — before definitely delete it. After each unsuccessful deletion attempt the clause is reinserted in the queue while decreasing its lifetime. This procedure is sketched in algorithm 1. Algorithm 2 as far as it is concerned, describes the propagation phase of each solver in PENELOPE_MDLC . PENELOPE_MDLC also includes some optimization techniques such as learned clause minimization with binary clause resolution. It also uses a special data structure to handle binary clauses — as in GLUCOSE — in order to check them first during unit propagations.

Algorithm 1: Reduce DB

```
Input: The Learned clause database $\Delta$ and the imported
       clause database $\Gamma$
1  begin
2    $totalClauseToDelete := (|\Delta| + |\Gamma|)/2$;
3    $nLt := totalClauseToDelete * reduceFactor$;
4    $nImp := totalClauseToDelete - nLt$;
5    if $|\Gamma| < nImp$ then
6      $nImp := |\Gamma|$;
7      $nLt := totalClauseToDelete - nImp$;
8    for ($i := 0$ ; $i < nImp$ ; $i$++) do
9      $c := \Gamma.pop()$;
10     if $lbd(c) > 2$ or $c.lifetime \leq 0$ then
11       delete(c);
12     else
13       $\Gamma.insert(c)$;
14       $c.lifetime$ --;
15   sort $\Delta$ according to clauses' LBD;
16   remove from $\Delta$ the $nLt$ clauses with bad LBD;
17 end
```

IV. GLUCOSE-3.0_PADC

GLUCOSE-3.0_PADC is a GLUCOSE3.0 hack. It simply allow the solver to periodically run a deep cleaning of the learned clause database. Concretely after each K execution of the cleaning procedure, it deletes all the clauses in the database except those which are of very high quality — such as clauses with $LBD \leq 2$ — and those that actually participate to the construction of the implication graph.

V. SAT COMPETITION 2018 SPECIFICS

We submitted two versions of GLUCOSE-3.0_PADC : GLUCOSE-3.0_PADC_3 and GLUCOSE-3.0_PADC_10 with respectively the parameter $K = 3$ and $K = 10$. SCALOPE was tuned to use 24 core and PSM [4]. As far as

Algorithm 2: Propagation phase

```
Input: A formula $F$, a learned clause database $\Delta$ and
       the imported clause database $\Gamma$
1  begin
2    repeat
3      while $\exists$ a unit or a falsified clause $c$ in $F \cup \Delta$ do
4        if $c$ is unit then
5          add the unassigned literal of $c$ to the
           current interpretation;
6        else
7          return c;
8      if $\exists$ a unit or conflicting clause $c \in \Gamma$ then
9        move $c$ to $\Delta$;
10   until no new clause has been moved to $\Delta$ ;
11   return noConflictClauseFound;
12 end
```

PENELOPE_MDLC is concerned, we tune it to use the $reduceFactor$ of 0.6 and each imported clause with $LBD \leq 2$ was given a lifetime of 1.

VI. ACKNOWLEDGEMENT

We would like to thanks authors of MINISAT[5] , GLUCOSE[3], MANYSAT[6] and PENELOPE[2] for the great work they have done.

REFERENCES

[1] K. T. Rodrigue and T. D. Clémentin, "Scalope: A scalable parallel sat solver based on penelope," *SAT COMPETITION 2017*, p. 32.

[2] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette, "Penelope in sat competition 2014," *SAT COMPETITION 2014*, p. 58.

[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers." in *IJCAI*, vol. 9, 2009, pp. 399–404.

[4] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs, "On freezing and reactivating learnt clauses," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2011, pp. 188–200.

[5] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

[6] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2008.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Maple_CM, Maple_CM_Dist, Maple_CM_ordUIP and Maple_CM_ordUIP+ in the SAT Competition 2018

Mao Luo[1], Fan Xiao[1], Chu-Min Li[2], Felip Manyà[3], Zhipeng Lü[1], Yu Li[2]
[1]School of Computer Science, Huazhong University of Science and Technology, Wuhan, China
{maoluo,fanxiao,zhipeng.lv}@hust.edu.cn
[2]MIS, University of Picardie Jules Verne, Amiens, France
{chu-min.li, yu.li}@u-picardie.fr, **corresponding authors**
[3]Artificial Intelligence Research Institute (IIIA-CSIC), Barcelona, Spain. felip@iiia.csic.es

1. Introduction

The CDCL SAT solver Maple_LCM won the gold medal of the main track of the SAT Competition 2017. It was implemented on top of the solver MapleCOMSPS_DRUP [1], [2] by integrating the effective learnt clause minimization approach described in [3]. In the current competition, we propose Maple_CM, which is a new solver on top of Maple_LCM that extends clause minimization to original clauses. Moreover, we propose three variants of Maple_CM: Maple_CM_Dist: is Maple_CM but uses a special branching heuristic called Distance for the first 50,000 conflicts; Maple_CM_ordUIP: is Maple_CM but reorders the first UIPs implying some conflicts selected under specific conditions; and Maple_CM_ordUIP+: is Maple_CM_ordUIP but selects the conflicts to re-order the first UIPs conflict in function of the branching heuristics VSIDS and LRB. All these solvers are described in the remaining sections.

2. Clause Minimization in Maple_CM

Clause minimization based on unit propagation (UP) can be described as follows: Given a clause $C = l_1 \vee l_2 \vee \cdots \vee l_k$, if UP$(F \cup \{\neg l_1, \neg l_2, \dots, \neg l_i\})$ $(i \le r)$ derives an empty clause and $\{\neg l'_1, \neg l'_2, \dots, \neg l'_{i'}\}$ is the subset of literals in $\{\neg l_1, \neg l_2, \dots, \neg l_i\}$ that are responsible of the conflict, we replace C by $\{l'_1 \vee l'_2 \vee \ldots \vee l'_{i'}\}$. This clause minimization is not applied to every clause at every restart because it is costly. It works as follows in Maple_CM:

- During preprocessing, each original clause is minimized. The minimization process stops when the total number of unit propagations is greater than 10^8.
- During the search, Maple_CM organizes the learnt clauses in three sets as MapleCOMSPS_DRUP: CORE, TIER2 and LOCAL. The sets CORE and TIER2 roughly store the learnt clauses with LBD≤ 6, where LBD refers to the number of decision levels in a clause [4]. It also identifies a subset of original clauses called *useful clauses* that are used to derive at least one learnt clause of LBD≤ 20 since the last clause minimization. Then, before a restart, Maple_CM minimizes each clauses C such that function liveClause(C) returns true, provided that the number of clauses learnt since the last clause minimization is greater than or equal to $\alpha + 2 \times \beta \times \sigma$, where $\alpha = \beta = 1000$ and σ is the number of minimizations executed so far. Algorithm 1 defines liveClause(C).

Algorithm 1: liveClause(C)

```
   Input: A clause $C$
   Output: $true$ or $false$
1  begin
2      if $C$ is a learnt clause then
3          if $C$ is in CORE or TIER2 then
4              if $C$ was never minimized, or the LBD of $C$ is decreased 2 times since its last minimization, or the LBD of $C$ is decreased to 1 since its last minimization then
5                  Return $true$;
6          Return $false$;
7      else
8          if $C$ was used to derive at least one learnt clause of LBD$\leq 20$ since the last clause minimization then
9              if $C$ was never minimized during search, or the LBD of $C$ is decreased 3 times since its last minimization, or the LBD of $C$ is decreased to 1 since its last minimization then
10                 Return $true$;
11         Return $false$;
12 end
```

The condition to select a restart for triggering a clause minimization process in Maple_CM is the same as in

Maple_LCM. However, Maple_CM selects the clauses to be minimized differently from Maple_LCM. First, Maple_LCM only minimizes the learnt clauses in CORE and TIER2, whereas Maple_CM also minimizes the useful original clauses, because original clauses can also contain redundant literals. Second, a learnt clause is minimized at most once in Maple_LCM, whereas a clause, either learnt or original, can be minimized more than once in Maple_CM under some conditions specified in terms of the decrease of its LBD.

The rationale behind the re-minimization of a clause is that further redundant literals can be detected, using unit propagation, after adding additional learnt clauses since its last minimization. Maple_CM re-minimizes a learnt (original) clause if its LBD was decreased two (three) times since its last minimization, because UP probably becomes more powerful in this case. The condition to re-minimize an original clause is stronger because an original clause presumably contains fewer redundant literals.

A particular case is a clause with LBD 1. This clause is probably very powerful in unit propagation and the LBD value cannot be decreased anymore. So, a clause will be re-minimized if its LBD becomes 1 since its last minimization, no matter how many times the LBD value was decreased.

3. *Distance*: A New Branching Heuristic Based on Implication Graphs in Maple_CM_Dist

Standard branching heuristics in CDCL solvers, such as VSIDS [5] and LRB [1], select a variable based on the behaviour of this variable in the past. However, at the beginning of search, very few things have happened. So, Maple_CM_Dist uses a new heuristic, different from VSIDS and LRB, at the beginning of the search.

Maple_CM_Dist constructs a complete implication graph for each conflict of the first 50,000 conflicts detected during the search. Each vertex in the implication graph corresponds to a variable and also to a clause. For each variable in the graph, we can collect the set D of clauses in all paths from the variable to the conflict. We then say that the variable depends on the clauses in D to contribute to the conflict. A variable depending on fewer clauses to contribute to a conflict is probably more powerful to contribute to a future conflict. Thus, the cardinality $|D|$ can be viewed as a measure of the strength with which a variable contributes to a conflict. Unfortunately, we are not aware of any linear-time algorithm that computes $|D|$ for all the variables.

Therefore, Maple_CM_Dist constructs the complete implication graph and computes the number of vertices in the longest path, denoted by longDist$[x]$, from each variable x to the conflict, based on the observation that a variable depending on many clauses probably needs a longer path to reach the conflict. Then, it computes the *Distance* score for each variable x, denoted by distAct$[x]$ and initialized to 0, as follows: When x contributes to a conflict, Maple_CM_Dist calls Algorithm 2 to increment $distAct(x)$ by $inc * 1/longDist[x]$, where inc is a global variable initialized to 1, and $dist\_Decay$ is intended to give more importance to recent conflicts, similarly to the var_decay parameter in VSIDS. The default value of $dist\_Decay$ is 0.95 as the default value of var_decay in MiniSAT.

Algorithm 2: updateDistanceScore(C)

```
  Input: $C$, a clause in which all literals are falsified
         by the current partial assignment
1 begin
2     Construct the complete implication graph $G$ that
      falsifies $C$ and compute longDist$[x]$ for each
      variable $x$ occurring in $G$;
3     for each variable $x$ occurring in $G$ do
4         $distAct[x] \leftarrow$
          $distAct[x] + inc * 1/longDist[x]$;
5     $inc \leftarrow inc/dist\_Decay$;
6 end
```

Maple_CM_Dist computes $distAct[x]$ and branches on the variable x with maximum $distAct[x]$ only for the first 50,000 conflicts. We limit this heuristic to the first 50,000 conflicts because constructing the complete implication graph is time-consuming, and VSIDS and LRB perform well after 50,000 conflicts. Maple_CM_Dist behaves like Maple_CM after the first 50,000 conflicts.

4. Blocking Restarts and Re-ordering UIPs in Maple_CM_ordUIP

The block-restart mechanism was introduced in Glucose 2.1 [6] for postponing a restart, when the solver is estimated to be approaching a global solution using a heuristic, in order to find quickly the global solution.

In Maple_CM_ordUIP, we push further the block-restart mechanism of Glucose. When the solver derives a conflict but is estimated to be approaching a global solution, it is forbidden to restart the search at least for the next 50 conflicts. In addition, it constructs a complete implication graph of the conflict to collect the first UIP (Unique Implication Point [7]) of every decision level involved in the conflict, in order to re-order these first UIPs and re-produce the conflict as described below.

Recall that a UIP in a decision level is a literal l such that every path from the decision literal of the level to the conflict goes through l. The first UIP (denoted by 1UIP) in the level is the closest UIP to the conflict in the level. Literal l_1 is said to imply literal l_2 if l_2 occurs in a path from l_1 to the conflict. When the solver identifies the 1UIP in a level, it estimates also the total number of vertices it implies.

Then, the solver backtracks so that all decisions in the complete implication graph are canceled, and continues the search from there by picking the decision literals from the collected 1UIPs in decreasing order of the total number of literals they imply, breaking ties in favor of the 1UIP with the smallest decision level in the implication graph.

Concretely, each time Maple_CM_ordUIP derives a conflict after the first 10^5 conflicts, it calls Algorithm 3 before analyzing the conflict. In the algorithm, nbC_For_restart and nbC_For_reOrder are two global variables that are incremented by 1 upon each conflict. Restart is possible only if nbC_For_Restart is greater than 50. In other words, when the 1UIPs are re-ordered, the next restart or the next 1UIP re-ordering is possible only after the next 50 conflicts. If unit propagation derives an empty clause C', function unitPropagate(l) returns C'; otherwise, it returns "no conflict".

```
Algorithm 3: reOrderUIPs(C)
   Input: C, a clause in which all literals are falsified
          by the current partial assignment
   Output: C', a clause in which all literals are falsified
           by the current partial assignment or the
           value "no conflict"
1  begin
2      if nbC_For_ReOrder> 50 and the number of
       assigned variables is two times greater than the
       average number of assigned variables in the
       previous 5000 conflicts then
3          nbC_For_Restart ← 0;
4          nbC_For_ReOrder ← 0;
5          Empty the vector UIPs;
6          Construct the complete implication graph G
           from C and push the 1UIP of each decision
           level of G into vertor UIPs;
7          Sort vector UIPs in decreasing order of the
           number of literals the 1UIPs imply, breaking
           ties in favor of the 1UIP with the smallest
           decision level;
8          Backtrack to cancel all decisions in G;
9          for i ← 0 to UIPs.size() do
10             if UIPs[i] is free then
11                 C' ← unitPropagate(UIPs[i]);
12                 if C' is a falsified clause then
13                     Return C';
14     Return "no conflict";
15 end
```

If Algorithm 3 returns a falsified clause C', Maple_CM_ordUIP analyzes C' instead of C. The conflict analysis stops at the last picked 1UIP, which implies fewer literals among the picked literals and may result in a shorter learnt clause. Otherwise, it continues the search by picking the decision literals among the free literals as Maple_CM.

The intuition behind re-ordering the 1UIPs can be described as follows: When a solver is approaching a global solution but derives a conflict, the conflict may simply be due to the ordering of the 1UIPs in the complete implication graph. In fact, the 1UIPs constitute the direct reason of the conflict. The 1UIP implying the greatest number of literals is probably the most constrained and should be satisfied first, leading more easily to the global solution.

5. Blocking Restarts and Re-ordering UIPs in Maple_CM_ordUIP+

Maple_CM, like MapleCOMSPS_DRUP and Maple_LCM, first uses the LRB heuristic for 2500 seconds and switches to the VSIDS heuristic for the remaining run time. It appears that the LRB heuristic is more powerful to solve satisfiable instances, whereas the VSIDS heuristic is more powerful to solve unsatisfiable instances. So, we implement Maple_CM_ordUIP+, that is Maple_CM_ordUIP but selects the conflict to re-order the 1UIPs differently in function of the branching heuristic used to derive the conflict.

- In a restart using VSIDS, Maple_CM_ordUIP+ re-orders the 1UIPs implying the first conflict as in Algorithm 3.
- In a restart using LRB, Maple_CM_ordUIP+ behaves as Maple_CM_ordUIP.

Intuitively, the first conflict is the most important in a restart, because it determines the search direction in the restart. Maple_CM_ordUIP+ aims at reinforcing the capability of the VSIDS heuristic to find a global solution of the instance, by branching first on the 1UIPs implying the greatest number of literals in a restart.

Acknowledgments

Research supported by the National Natural Science Foundation of China (Grants no. 61370183, 61370184, 61472147), MINECO-FEDER project RASO TIN2015-71799-C2-1-P, EU Horizon 2020 project LOGISTAR and the Matrics platform of University of Picardie Jules Verne.

References

[1] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Proceedings of SAT*, 2016, pp. 123–140.

[2] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maple-COMSPS, MapleCOMSPS LRB, MapleCOMSPS CHB," in *SAT Competition*, 2016, p. 52.

[3] M. Luo, C.-M. Li, F. Xiao, F. Manya, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *Proceedings of IJCAI*, 2017, pp. 703–711.

[4] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of IJCAI-2009*, 2009, pp. 399–404.

[5] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of DAC-2001*, 2001.

[6] G. Audemard and L. Simon, "Refining restarts strategies for SAT and UNSAT," in *Proceedings of CP*, 2012, pp. 118–126.

[7] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *Proceedings of ICCAD-2001*, 2001, pp. 279–285.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Glucose Hacks and TopoSAT2 Description

Thorsten Ehlers
Kiel University
Kiel, Germany

Dirk Nowotka
Kiel University
Kiel, Germany

***Abstract*—This document describes the parallel SAT solver TopoSAT2 and the Glucose Hack which we submit to the SAT Competition 2018.**

I. Introduction

This short paper describes the Glucose hack and the version of our parallel solver TopoSAT2 submitted to the SAT Competition 2018.

II. Glucose Hack

Our Glucose Hack is considered with the preprocessor, which mainly is code from MiniSAT [1] and SATElite [2]. This code still contains old comments like

```
// FIX: this is too inefficient but would be nice
to have (properly implemented)
// if (!find(subsumption_queue, &c))
```

We suggest to improve the preprocessor code at three places.

1) We add a boolean flag to the clause header which prevents clauses from being put on the subsumption queue several times. Though not a big issue on many formulas, this actually is a big problem on some formulas which may cause the preprocessor to be stuck for hours.
2) The subsumption check for clauses a, b has a worst-case running time of $|a| \cdot |b|$. Using a simple lookup-array, this can be reduced to $|a| + |b|$.
3) The same applies for the merge-procedure which computes the resolvent of two clauses.

We think that this is particularly interesting as many current solvers are built on top of MiniSAT and Glucose. The required changes are rather small (the edit distance is 892), and can easily be applied to every solver which is built upon MiniSAT and Glucose.

III. TopoSAT2

TopoSAT2 was mainly designed to run in a massively-parallel environment (> 1000 solver threads). Thus, we are curious to see how it performs on a shared-memory system. It is built on top of Glucose 3.0, but uses a bug-fixed version of the lockless clause sharing mechanism from ManySAT [3] for communication on one compute node rather than the lock-based implementation from Glucose Syrup. The communication between nodes uses MPI, but this is not used for the competition.

It comes with two features which we hope will be especially useful in the competition.

A. Diversification

The first portfolio solvers used different sequential solvers, or different settings of one sequential solver. We somewhat go back to the roots and diversify the search of the solver threads by the following parameters.

- Branching: Some solver threads use VSIDS, whereas other use LRB [4], as this branching heuristic was quite successful in the past SAT competitions. As VSIDS still works better on a significant amount of benchmarks, we use both.
- Restarts: We use the inner/outer restart scheme [5], Luby restarts, and the adaptive restart strategy from Glucose [6].
- Learnt Clause DB management: Some solver threads use a scheme similar to the one suggested in [7]: Clauses with very low LBD (≤ 3) are stored permanently. Clauses of intermediate LBD are stored at least for some time, and there is a small activity-based clause storage. The LBD of clauses imported from other solver threads are initialised with the size of the clause. Thus, the clause must be used in order to update its LBD, and allowing it to be stored for a longer time. Some other solver threads use the default clause management strategy from Glucose [8].

B. Lifting exported clauses

Wieringa et. al suggested to use some threads of a parallel SAT solver to strengthen learnt clauses [9]. Similarly, in [10] some of the learnt clauses are strengthened during search. We use this technique when exporting clauses. Whenever one solver threads learns a clause of sufficiently low LBD, it is stored in an extra buffer. After the next restart, the clauses from this buffer are strengthened, and the results are exported to the other solver threads.

C. Submitted versions

We submit one version of TopoSAT2 to the SAT Competition. However, we submit two different scripts to start it with different parameters. These parameters are concerned with the way in which clauses are import by the solver. The first version uses a variation of the clause import strategy of ManySAT. During search, the trail size is monitored. Clauses are imported when some time has passed and the solver is somewhat close to the root of the search tree. In this way, we try to prevent the solver from backtracking too often when imported clauses are unit under the current assignment.

The second version imports more often. However, clauses which are unit under the current assignment and would thus require a backtrack are imported as one-watched clauses as in Glucose Syrup [11].

Acknowledgment

This work is funded by the German Federal Ministry of Education and Research, combined project 01IH15006A.

References

[1] N. Eén and N. Sörensson, “An extensible sat-solver,” in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518. [Online]. Available: https://doi.org/10.1007/b95238

[2] N. Eén and A. Biere, “Effective preprocessing in SAT through variable and clause elimination,” in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75. [Online]. Available: https://doi.org/10.1007/b137280

[3] Y. Hamadi, S. Jabbour, and L. Sais, “Manysat: a parallel SAT solver,” *JSAT*, vol. 6, no. 4, pp. 245–262, 2009.

[4] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Learning rate based branching heuristic for SAT solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140.

[5] A. Biere, “Picosat essentials,” *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[6] G. Audemard and L. Simon, “Refining restarts strategies for SAT and UNSAT,” in *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Milano, Ed., vol. 7514. Springer, 2012, pp. 118–126.

[7] C. Oh, “Between SAT and UNSAT: the fundamental difference in CDCL SAT,” in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Heule and S. Weaver, Eds., vol. 9340. Springer, 2015, pp. 307–323.

[8] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, C. Boutilier, Ed., 2009, pp. 399–404.

[9] S. Wieringa and K. Heljanko, “Concurrent clause strengthening,” in *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. Järvisalo and A. V. Gelder, Eds., vol. 7962. Springer, 2013, pp. 116–132.

[10] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, “An effective learnt clause minimization approach for CDCL SAT solvers,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 703–711. [Online]. Available: http://www.ijcai.org/Proceedings/2017/

[11] G. Audemard and L. Simon, “Lazy clause exchange policy for parallel SAT solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer, 2014, pp. 197–205.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Varisat, a SAT Solver Written in Rust

Jannis Harder
Email: me@jix.one

***Abstract*—Varisat is a CDCL-based SAT solver written from scratch in the programming language Rust. It is still in an early stage of development, offering little beyond the essentials. Nevertheless, it hopefully shows that Rust is a suitable programming language for implementing a SAT solver.**

I. Introduction

Most state-of-the-art SAT solvers are implemented in C or C++. This is not surprising, as optimizing data layout, memory access patterns and other low-level optimizations are important for the performance of SAT solvers.

Recently the programming language Rust [1] has seen increased use in areas where before mostly C and C++ were popular. Rust is a systems programming language with a focus on memory safety and performance, which is sponsored by Mozilla Research.

Varisat is my attempt to implement a SAT solver in Rust, which I began on one hand to become familiar with the details of implementing a modern SAT solver and on the other hand to show that Rust is a suitable language for implementing a SAT solver.

II. Implemented Techniques

Varisat is a CDCL-based SAT solver [2] written from scratch. It it is still in an early stage of development and implements only basic techniques. Among the already implemented techniques are: VSIDS branching heuristic, Luby series restarts, minimization of learned clauses [3], LBDs [4] and a 3-Tiered learned clause database [5].

No pre- or inprocessing techniques are implemented yet. Also Varisat has seen very little benchmarking and thus very little parameter tuning.

III. Source Code

The source code is licensed under the MIT license and available at https://jix.one/sc18/varisat. It uses some Rust features that are not yet part of the stable release. The submitted version was developed using the nightly release 2018-02-08.

References

[1] Rust project developers. (n.d.). The Rust Programming Language, [Online]. Available: https://www.rust-lang.org/.

[2] J. Marques-Silva, I. Lynce, and S. Malik, “CDCL Solvers,” in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., IOS Press, 2009, pp. 131–150.

[3] N. Sörensson and A. Biere, “Minimizing Learned Clauses,” in *Theory and Applications of Satisfiability Testing - SAT 2009*, ser. LNCS, Springer, 2009, pp. 237–243.

[4] L. Simon and G. Audemard, “Predicting Learnt Clauses Quality in Modern SAT Solvers.,” in *Twenty-First International Joint Conference on Artificial Intelligence (IJCAI’09)*, 2009.

[5] C. Oh, “Between SAT and UNSAT: The Fundamental Difference in CDCL SAT,” in *Theory and Applications of Satisfiability Testing – SAT 2015*, ser. LNCS, Springer, 2015, pp. 307–323.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

satUZK-ddc at SAT Competition 2018

Alexander van der Grinten
University of Cologne

***Abstract*—We describe the current version of our distributed satUZK-ddc solver. Since SAT Competition 2017, we have augmented the solver with a parallel preprocessing technique that we call parallel distillation. This algorithm performs the same CNF simplifications as the conventional CNF distillation algorithm, namely removing or shortening clauses of the input formula according to certain redundancy properties. While sequential CNF distillation is known to be expensive, due to parallelism, it becomes feasible to preprocess even large real-world CNF formulas with our algorithm.**

I. Overview

We submitted two solver configuration to the SAT Competition 2018. Both configurations are based on the *satUZK-ddc* engine that was also submitted to last year's SAT Competition [1]. The first configuration is the current default configuration of satUZK-ddc. This configuration corresponds to the satUZK-ddc configuration from last year, with minor adjustments and bug fixes. The second configuration (satUZK-ddc `--ddist`) is augmented with our *parallel distillation* preprocessing technique. In contrast to last year, we did not submit a sequential version as our sequential algorithm is mostly unchanged from the SAT Competition 2017 version.

II. DDC algorithm

The *distributed divide and conquer* (DDC) algorithm was shortly described in [1] and is described in more detail in [2]. We summarize the main ideas of the algorithm here.

The DDC algorithm uses a parallel lookahead technique to partition the search space until there are at least as many subproblems as processors. Those subproblems are solved by CDCL. In contract to Cube and Conquer [3], the DDC algorithm does not rely on work stealing for load balancing; instead, the algorithm maintains a distributed divide and conquer tree and routes individual processors through this tree. As the routing procedure uses only local information and does not require extensive synchronization, this approach is able to fully utilize all processors at every step of the algorithm.

The basic algorithm is extended with preprocessing, inprocessing, an LBD [4] score that takes the incremental nature of the algorithm into account, clause sharing and diversification. Again, for details on those extensions, we refer to [2]. satUZK-ddc is an MPI-based implementation of the DDC algorithm that builds on our earlier CDCL implementation in satUZK.

III. Parallel distillation

Our most significant contribution to the SAT Competition 2018 is the parallel distillation algorithm described below.

CNF distillation: In the following, we shortly review the traditional *distillation* [5] procedure[1] and state it in the language of [7], [8]. Let ϕ be the input formula (in CNF) and let $C \in \phi$ be a clause. If $C' \subsetneq C$ is a subset of C and unit propagation on $\phi \setminus \{C\} \cup \{\{\bar{\ell}\} : \ell \in C'\}$ assigns all literals of C to false, then C can be shortened to C' while preserving equivalence. This technique is called *asymmetric literal elimination* (ALE). Note that C' is not necessarily unique for each $C \in \phi$ (not even if we require C' to be minimal). Thus, implementations of ALE usually pick an arbitrary suitable C'.

On the other hand, if unit propagation on $\phi \setminus \{C\} \cup \{\{\bar{\ell}\} : \ell \in C\}$ leads to a conflict, then C can be removed from ϕ while preserving equivalence. This process is called *asymmetric tautology elimination* (ATE).

Distillation is a systematic procedure to perform ALE and ATE. To accomplish this, unit propagation is used to detect opportunities for ALE and ATE. In particular, unit propagation is applied at least once for each clause in ϕ. In the sequential case, this is the main bottleneck of the algorithm; therefore, our parallel version tries to parallelize the detection of ALE and ATE opportunities.

Commutation properties: In order to parallelize the distillation algorithm, we observe the following commutation properties (see [2] for details):

ALE vs. ALE If ALE transforms $\phi \cup \{C_1\}$ to $\phi \cup \{C_2'\}$ and also $\phi \cup \{C_2\}$ to $\phi \cup \{C_2'\}$ (i.e. ALE shortens the clauses C_1 to C_1' and C_2 to C_2', respectively), then the two formulas $\phi \cup \{C_1, C_2\}$ and $\phi \cup \{C_1', C_2'\}$ are equivalent[2].

ALE vs. ATE If ALE transforms $\phi \cup \{C\}$ to $\phi \cup \{C'\}$ and ATE transforms $\phi \cup \{D\}$ to ϕ (i.e. ATE removes the clause D), then the two formulas $\phi \cup \{C, D\}$ and $\phi \cup \{C'\}$ are equivalent. Specifically, ALE can only expand the set of literals that are fixed by unit propagation on $\phi \cup \{\{\bar{\ell}\} : \ell \in D\}$.

ATE vs. ATE On the other hand, if ATE transforms $\phi \cup \{D_1\}$ to ϕ and also $\phi \cup \{D_2\}$ to ϕ, the two formulas $\phi \cup \{D_1, D_2\}$ and ϕ are not necessarily equivalent. However, they are not equivalent only if D_1 participates in the conflict[3] which is derived by applying unit propagation to $\phi \cup \{\{\bar{\ell}\} : \ell \in D_2\}$, or vice versa. Thus, this situation must be detected to determine if replacing $\phi \cup \{D_1, D_2\}$ by ϕ is sound.

Parallel algorithm: Let p denote the number of processors that are available to the solver. Initially, each processor has its own copy of the input formula ϕ. The parallel distillation algorithm now works as follows: The set of clauses of ϕ is

[1]We use the term *distillation* synonymously with *vivification* [6].

[2]This is not particularly surprising: In fact, this property holds true for every equivalence-preserving simplification technique that only shortens clauses.

[3]We say that a clause participates in a conflict if it is part of the conflict graph [9].

partitioned into p subsets $\phi_1, \ldots, \phi_p$. ϕ_i will be the set of clauses that are processed by processor i. For each clause $C \in \phi_i$, processor i tries to shorten C to a subset $S(C) \subsetneq C$ by applying ALE to ϕ. If C cannot be shortened by ALE, we set $S(C) = C$. Furthermore, processor i determines a flag $r(C)$ that is true if and only if C can be removed from ϕ using ATE. If that is the case, a set $\mathcal{D}(C)$ is determined. $\mathcal{D}(C)$ consists of exactly those clauses that participate in the conflict which is derived by unit propagation on $\phi \cup \{\{\bar{\ell}\} : \ell \in C\}$. We call $\mathcal{D}(C)$ the set of *dependencies* of C. Indeed, because of the last commutation property, the removal of C depends on the clauses from $\mathcal{D}(C)$ in the following sense: C can be safely removed from ϕ if all clauses from $\mathcal{D}(C)$ are retained in ϕ. We remark that our algorithm parallelizes at the clause level; like in the case of traditional distillation, ALE and ATE checks are performed by standard unit propagation.

After the computation is finished on all processors, the information $S(C)$, $r(C)$ and $\mathcal{D}(C)$ are gathered on all processors, for every clause $C \in \phi$. The processors agree on an order of those clauses and all processors inspect the clauses in the same order. Now, for each clause $C \in \phi$, the flag $r(C)$ is checked to determine if C is a candidate for removal. If $r(C)$ is true, all processors remove C if and only if (i) no dependency in $\mathcal{D}(C)$ has already been removed and (ii) C was not a dependency of any clause that has already been removed. If C is not removed, we replace C by $S(C)$. The commutation properties discussed above ensure that this algorithm results in a formula that is equivalent to the input formula ϕ. As all processors apply modifications in the same order, the resulting formulas are identical. Note that the effectiveness of the algorithm potentially depends on the order in which clauses are inspected; however, optimizing this order did not turn out to be necessary in practice.

Optimizations: We discuss some modifications of the parallel distillation algorithm that aim to improve its empirical performance. First, we modify the algorithm to not remove binary clauses using ATE. This allows us to omit binary clauses from the $\mathcal{D}(C)$ sets. In the case of real-world CNF instances, this often yields a significant size reduction, as more than half of the clauses are binary in many of those instances. We note that this optimization only marginally decreases the strength of distillation when it is combined with the (considerably faster) *unhiding* [10] algorithm: In fact, we can expect that many binary clauses that are removed by ATE can also be removed by the weaker *hidden tautology elimination* (HTE) technique. The established *failed literal elimination* (FLE) and *hyper-binary-resolution* (HBR) [11] techniques allow to turn additional ATEs into HTEs.

Secondly, we want to avoid removing *useful* clauses using ATE. For example, note that ATE would be strong enough to remove all learned clauses from a formula. Certainly, removing all learned clauses impairs the performance of CDCL solvers. Thus, instead of removing clauses after ATE, we only mark them as non-redundant and depend on the usual clause database reduction heuristics to remove those clauses eventually.

Furthermore, to increase the opportunities for ALE and ATE, we apply a parallel FLE algorithm before running parallel distillation. This FLE algorithm is a probing-based algorithm that relies on the tree-based lookahead approach [11] to check, for each literal ℓ, if ℓ is a failed literal. If that is indeed the case, the clause $\{\ell\}$ is added to the input formula ϕ. As adding the clauses to ϕ can only enlarge the set of failed literals, it is possible to traverse multiple lookahead trees in parallel while preserving correctness.

Finally, instead of partitioning ϕ into p fixed subsets, we employ a work-stealing load balancer to assign clauses to processors. As we expect the running time of unit propagations to differ substantially depending on affected variables and clauses, this improves processor utilization compared to static partitioning.

References

[1] A. van der Grinten, “satUZK-seq and satUZK-ddc: Solver description,” in *Proceedings of SAT Competition 2017*, T. Balyo, M. J. H. Heule, and M. Järvisalo, Eds., 2017.

[2] ——, “Design, implementation and evaluation of a distributed CDCL framework,” 2017. [Online]. Available: https://kups.ub.uni-koeln.de/8281/

[3] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, “Cube and Conquer: Guiding CDCL SAT solvers by lookaheads,” in *Proceedings of the 7th international Haifa Verification conference on Hardware and Software: Verification and Testing*, ser. HVC ’12. Springer, 2012, pp. 50–65.

[4] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, ser. IJCAI’09. Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[5] H. Han and F. Somenzi, “Alembic: An efficient algorithm for CNF preprocessing,” in *44th ACM/IEEE Design Automation Conference*. ACM, 2007, pp. 582–587.

[6] C. Piette, Y. Hamadi, and L. Saïs, “Vivifying propositional clausal formulae,” in *Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence*. Amsterdam, The Netherlands: IOS Press, 2008, pp. 525–529.

[7] M. Heule, M. Järvisalo, and A. Biere, “Clause elimination procedures for CNF formulas,” in *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2010, pp. 357–371.

[8] M. Järvisalo, M. J. H. Heule, and A. Biere, “Inprocessing rules,” in *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. Springer Berlin Heidelberg, 2012, pp. 355–370.

[9] J. a. P. M. Silva and K. A. Sakallah, “Grasp - a new search algorithm for satisfiability,” in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided Design*, ser. ICCAD ’96. IEEE, 1996, pp. 220–227.

[10] M. J. H. Heule, M. Järvisalo, and A. Biere, “Efficient CNF simplification based on binary implication graphs,” in *Theory and Applications of Satisfiability Testing - SAT 2011*. Springer, 2011, pp. 201–215.

[11] M. Heule, M. Järvisalo, and A. Biere, “Revisiting hyper binary resolution,” in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, C. Gomes and M. Sellmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 77–93.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

ReasonLS

Shaowei Cai and Xindi Zhang
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
School of Computer and Control Engineering, University of Chinese Academy of Sciences

***Abstract*—This note describes the SAT solver "ReasonLS", which is a hybrid solver that combines a semi-exact CDCL based solver and a local search solver.**

I. Introduction

An interesting direction for solving SAT is to combine the CDCL method and the local search method, as these two methods have different advantages. Previous works in this direction usually use a CDCL solver as a black box. On the other hand, our recent works show that initial assignments generated by unit propagation based procedures can significantly improve the performance of local search on solving industrial MaxSAT instances, even leading the algorithm to find the optimal solution [1].

ReasonLS is a hybrid solver that combines a CDCL based semi-exact solver and a local search solver. We develops a backtracking procedure by modifying a CDCL solver to find a complete assignment with high consistency, which is then handed to a local search solver trying to find a solution. ReasonLS works by interleaving between theses two solvers. In each round (except for the first one), some restarting strategies are used to restart the backtracking procedure. ReasonLS uses a parameter (a real number from 0 to 1) to control to what level of consistency that we require the backtracking style procedure to find, and once the procedure finds such a complete assignment, it returns the assignment. Then ReasonLS calls a local search to find a solution, using the assignment found by the backtracking procedure.

ReasonLS can also be seen as a generalization of both CDCL solvers and local search solvers. When the parameter is set to 0, ReasonLS becomes a local search solver; when the parameter is set to 1, it becomes a CDCL solver.

II. A Semi-exact procedure

We modify a CDCL solver to make it allow conflicts in some cases. We use a parameter p (a real number from 0 to 1), and if the solver finds a path that assigns at least a portion p of the variables without generating a conflict, then it will finish the path by assigning the remaining variables, even if it may lead to conflicts.

In ReasonLS, we develop our semi-exact solver by modifying the CDCL solver named Maple_LCM_Dist [2], which is a state of the art CDCL solver and won the main track of SAT Competition 2017.

III. The Local Search Algorithm

As for the local search solver used in ReasonLS, we develop a variant of CCAnr [3] by forbidding the aspiration mechanism. The algorithm is describes as follows.

Starting from an initial assignment, the algorithm flips a variable in each step. Firstly, if there are configuration changed variables, it picks a configuration changed variable with the greatest score, breaking ties by favoring the oldest variable. Otherwise, the clause weights are updated according to a Threshold-based Smoothed Weighting (TSW) scheme; then, it picks a random unsatisfied clause and selects a variable in the clause with the greatest score, breaking ties by favoring the oldest variable.

IV. Main Parameters

There is one parameter p for controlling the cooperation of the backtracking style procedure and the local search solver. There are three parameters in the local search solver : the average weight threshold parameter γ, and the two factor parameters ρ and q. All of the three parameters are for the TSW weighting scheme. The parameters are set as follows: $\gamma = 50$; $\rho = 0.3$; $q = 0.7$. Meanwhile, there are a button ASP to control whether turn on or turn off the process of aspiration of local search, and a parameter $VSIDS$ to decide when change search strategy of CDCL to VSIDS.

In our solver, we call the ReasonLS solver to solve an instance with different parameter settings as follow. Each thread runs the solver with one setting.

1) $p = 1$; $VSIDS = 2500s$ (ReasonLS becomes the CDCL solver named Maple_LCM_Dist, thus no need to set the parameters for local search.)
2) $p = 1$; $VSIDS = 0s$.
3) $p = 0$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.
4) $p = 0$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0$.
5) $p = 0$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0.7$.
6) $p = 0$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0$.
7) $p = 0$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.
8) $p = 0$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0$.
9) $p = 0$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0.7$.

10) $p = 0$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0$.
11) $p = 0.9$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.
12) $p = 0.9$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0$.
13) $p = 0.9$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0.7$.
14) $p = 0.9$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0$.
15) $p = 0.9$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.
16) $p = 0.9$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0$.
17) $p = 0.9$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0.7$.
18) $p = 0.9$; $\gamma = 300$; $VSIDS = 2500s$; $ASP = off$; $\rho = 0.3$; $q = 0$.
19) $p = 0.9$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$. (increase the number of flips of local search, and so dose the 20th.)
20) $p = 0.9$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$. (set more strict limit on the time of local search.)
21) $p = 0.7$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.
22) $p = 0.7$; $\gamma = 50$; $VSIDS = 0s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.
23) $p = 0.5$; $\gamma = 50$; $VSIDS = 2500s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.
24) $p = 0.5$; $\gamma = 50$; $VSIDS = 0s$; $ASP = on$; $\rho = 0.3$; $q = 0.7$.

V. Implementation Details

ReasonLS is implemented in C++. It is developed based on the codes of Maple_LCM_Dist [2] and CCAnr solver [3].

VI. SAT Competition 2018 Specifies

ReasonLS is submitted to “Parallel Track”. It is compiled by g++ with the ’O3’ optimization option.

Its running command is: ”./ReasonLS-run.sh $1”. $1 is the absolute path of input file. For a given input file ”∼/sc/a.cnf”, the call command is ”./ReasonLS-run.sh ∼/sc/a.cnf ”.

References

[1] S. Cai, C. Luo, and H. Zhang, “From decimation to local search and back: A new approach to MaxSAT,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, pp. 571–577.

[2] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, “An effective learnt clause minimization approach for CDCL SAT solvers,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, pp. 703–711.

[3] S. Cai, C. Luo, and K. Su, “CCAnr: A configuration checking based local search solver for non-random satisfiability,” in *Proceedings of18th International Conference on Theory and Applications of Satisfiability Testing, SAT 2015, Austin, TX, USA, September 24-27, 2015*, 2015, pp. 1–8.

BENCHMARK DESCRIPTIONS

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Generating the Uniform Random Benchmarks

Marijn J. H. Heule

Department of Computer Science,
The University of Texas at Austin, United States

***Abstract*—The benchmark suite of the Random Track of SAT Competition 2018 can be partitioned into three parts. The first part consists of uniform random k-SAT instances described below. The second part consists of benchmarks generated by a tool by Tomáš Balyo [1]. These benchmarks are similar as the ones used in the 2016 SAT Competition. The third part consists of random benchmarks contributed by Adrian Balint described on page 61 of these proceedings.**

Intro

This description explains how the benchmarks were created of the uniform random categories of the SAT Competition 2018. These categories consists of uniform random k-SAT instances with $k \in 3, 5, 7$ – Boolean formulas for which all clauses have length k. For each k the same number of benchmarks have been generated.

Generating the Satisfiable Benchmarks

The satisfiable uniform random k-SAT benchmarks are generated for two different sizes: medium and huge. The medium-sized benchmarks have a clause-to-variable ratio equal to the phase-transition ratio[1]. The number of variables differs for all the benchmarks. The huge random benchmarks have a few million clauses and are therefore as large as some of the application benchmarks. For the huge benchmarks, the ratio ranges from far from the phase-transition ratio to relatively close, while for each k the number of variables is the same. Table I shows the details.

No filtering was applied to construct the competition suite. As a consequence, a significant fraction (about 50%) of the medium-sized generated benchmarks is unsatisfiable.

TABLE I
Parameters of generating the satisfiable benchmarks

k	medium (40)	huge (20)
3	$r = 4.267$ $n \in \{5000, 5200, \dots, 12800\}$	$r \in \{3.86, 3.88, \dots, 4.24\}$ $n = 1,000,000$
5	$r = 21.117$ $n \in \{200, 210, \dots, 590\}$	$r \in \{16, 16.2, \dots, 19.8\}$ $n = 250,000$
7	$r = 87.79$ $n \in \{90, 92, \dots, 168\}$	$r \in \{55, 56, \dots, 74\}$ $n = 50,000$

[1]The observed clause-to-variable ratio for which 50% of the uniform random formulas are satisfiable. For most algorithms, formula generated closer to the phase-transition ratio are harder to solve.

References

[1] T. Balyo, "Using algorithm configuration tools to generate hard random satisfiable benchmarks," in *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, 2016, pp. 60–62.

Divider and Unique Inverse Benchmarks Submitted to the SAT Competition 2018

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

Our benchmark submission for the SAT 2018 Competition consist of two sets of word-level properties originally formulated as SMT problems in the quantifier-free theory of bit-vectors in BTOR [1] or SMTLIB [2] format. We then use our SMT solver Boolector [3] to synthesize AIGs [4], which in turn were translated to DIMACS format.

Division

The first set specifies word-level (modulo 2^n) division using multiplication for various bit-widths n in BTOR format [1].

We consider both *unsigned* and *signed* dividers. For unsigned division we check validity over unsigned n-bit bit-vectors ("$/_u$" denotes unsigned division):

$$y \neq 0 \quad \Rightarrow \quad (x - (x /_u y) \cdot y) <_u y$$

As common in bit-vector logics arithmetic operators take two n-bit bit-vectors as input and produce one n-bit bit-vector as output, with the effect, that there is no difference between signed and unsigned versions of multiplication nor subtraction.

For signed division it is more complicated and we have to take signs into account (now "$/_s$" denotes signed division):

$$y \neq 0 \quad \Rightarrow \quad |x - (x /_s y) \cdot y| <_u |y|$$

where "| |" is actually implemented with an if-then-else operator testing the argument to be smaller than zero (using signed "$<_s$" comparison) and if so negating it (two-complement). These signed benchmarks are as a consequence much harder.

Inversion

The second set of benchmarks checks that bit-vector multiplication modulo 2^n has unique inverses for odd numbers, which translates to the following SMT benchmark for $n = 32$ in SMTLIB format [2]:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(declare-fun z () (_ BitVec 32))
(assert (= (bvmul x y) (bvmul x z)))
(assert ((_ extract 0 0) x))
(assert (distinct y z))
(check-sat)
(exit)
```

References

[1] R. Brummayer, A. Biere, and F. Lonsing, "BTOR: Bit-precise modelling of word-level problems for model checking," in *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, ser. SMT '08/BPR '08. New York, NY, USA: ACM, 2008, pp. 33–38.

[2] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[3] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2, BtorMC and Boolector 3.0," in *Computer Aided Verification - 30th International Conference, CAV 2018*, ser. Lecture Notes in Computer Science. Springer, 2018, to appear.

[4] A. Biere, "The AIGER And-Inverter Graph (AIG) format version 20071012," FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep., 2007.

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

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Reversing Elementary Cellular Automata

Jannis Harder
Email: me@jix.one

I. Introduction

Elementary (i.e. one-dimensional two-state) cellular automata are systems with simple rules that can nevertheless show complex behavior. These benchmarks encode the problem of finding a sequence of predecessor states for a given target state and elementary automaton rule R. To allow for a finite encoding of states, we assume that all states, target and predecessors, have a fixed period n.

We first generate a pseudo-random state X from a seed s. Starting from X and advancing it f times, we generate a target state Y. We then ask whether there is an initial state I, that evolves to Y in exactly r steps. The sequence of states from I to Y does not have to contain the state X, but when $r \leq f$ such a sequence does exist.

The generated instance for the parameters (R, n, f, r, s) is named `ecarev-`R`-`n`-`f`-`r`-`s`.cnf`.

II. SAT-Encoding

We need to consider the sequence of states $S_1, \ldots, S_r$, where $S_1 = I$ and $S_r = Y$. Each state S_i is represented by the cell values for n consecutive cells $S_{i,1}, \ldots, S_{i,n}$. These cell values correspond to variables in the SAT-encoding. Since the states are periodic, $S_{i,j} = S_{i,j+n}$ holds and we will use both to represent the same variable in the encoding.

To encode the constraint that the states follow the automata rule, we need to encode the state transition $S_{i+1} = T_R(S_i)$. This relation can be decomposed into

$$\bigwedge_{1 \leq j \leq n} S_{i+1,j} = t_R(S_{i,j-1}, S_{i,j}, S_{i,j+1})$$

where t_R is specified by the automaton's rule table. It is encoded as a conjunction of all implied minimal clauses.

We then constrain the states to follow the automata's evolution by

$$\bigwedge_{1 \leq i < r} S_{i,j} = S_{i,j+n}.$$

Finally we add unit clauses to constrain $S_r = Y$.

The implementation of this encoding is written in Python 3 and available at https://jix.one/sc18/ecarev.py.

III. Selected Benchmarks

The submitted benchmarks are for the rule-110 automaton, which is capable of universal computation. The state period n was chosen large enough to make an exhaustive search infeasible. Apart from that the parameters were manually chosen to make the instances not too easy or too hard and to make it difficult to guess the outcome from the parameters alone.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Verifying Simple Floating-Point Programs

Joseph Scott, Jia Hui Liang, Vijay Ganesh
University of Waterloo, Waterloo, ON, Canada

***Abstract*—A brief description of the instances we submitted to the SAT Competition 2018 encoding bounded model checking of floating-point C programs.**

I. Background

Floating point data types use a finite number of bits to represent the set of real numbers within a fixed interval. Unlike its integer counterpart, not every real number within the interval has an exact representation in its floating point type. For example, the real number 0.1 does not have an exact representation in common binary floating point types.

Computer programs with floating point arithmetic frequently suffer from a (quiet) inexact exception where the floating point operation results in a real value that does not have an exact representation in its floating point type. A rounding decision must be made to set the result of the operation to a representable floating point value.

Floating-point operations and rounding behavior are dictated by the IEEE 754 standard and behave unintuitively under certain circumstances. Programmers who treat floating-point numbers as real numbers are surprised to learn that basic properties of real numbers such as associativity and distributivity are not respected in the floating-point space due to the required rounding. This often leads to hard-to-debug errors that expose themselves in floating-point programs under rare conditions. Additionally, floating-point adds special representations like not-a-number (NaN), positive and negative zero, and positive and negative infinity, which can cause several subtle errors when assuming similar behavior to familiar integer data types.

In program verification over integer data types, SMT solvers with rich theories can be used to guide a SAT solver. Due to the unintuitive nature of floating point and lack of established properties, solvers must use a more direct translation to SAT which makes the solvability of floating point instances desirable.

II. Benchmarks

We use CBMC (C Bounded Model Checker) by Kroening and Tautschnig [1] to translate 46 simple C-programs written by us involving floating-points into to SAT. These programs test for:

- Floating-point associative property
  - Exact associativity of $+$ and $\times$.
  - Associativity of $+$ and $\times$ with various acceptable relative error bounds.
- Floating-point commutative property
- Floating-point distributive property
  - Exact distributivity.
  - Distributivity with various acceptable relative error bounds.
- Finding floating-point roots of a quadratic polynomial
  - Testing that the roots are exact.
  - Testing that the roots when evaluated are within a threshold from zero.
- Inexactness of the square-root of floating-point numbers
- Triangle Inequality
  1) Exact triangle inequality.
  2) Triangle inequality with various added fixed constants for acceptable error.
- Perceptron classification with floating-points over a fixed data set.

The original C-programs are available here: https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/floatingpointsource.zip?attredirects=0&d=1. We used the command `cbmc source.c --dimacs` to generate the CNF files. The floating-point operations in the C programs are essentially bit-blasted down to propositional logic by CBMC. If the CNF is satisfiable, then the assertion in the C code is satisfiable at the point of execution. Otherwise the CNF is unsatisfiable.

References

[1] D. Kroening and M. Tautschnig, “CBMC – C Bounded Model Checker,” in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391.

[2] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, “Ieee standard for floating-point arithmetic,” *IEEE Std 754-2008*, pp. 1–70, 2008.

[3] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of floating-point arithmetic*. Springer Science & Business Media, 2009.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

$\mathbf{GrandTour^{obs}}$ Puzzle as a SAT Benchmark

Md Solimul Chowdhury
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mdsolimu@ualberta.ca

Martin Müller
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mmueller@ualberta.ca

Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
jyou@ualberta.ca

***Abstract*—GrandTour[1] is a puzzle game, which is usually played in a rectangular grid of points, where a player is posed with the challenge to find a closed loop that goes through each point exactly once. In GrandTour, the player is allowed to connect any pair of points to solve the game. We consider a slight variation of the GrandTour puzzle game, where a player is allowed to connect a restricted subset of pairs of points, as there are obstacles between some of the points. We call this puzzle GrandTour with obstacles or $\mathbf{GrandTour^{obs}}$. For SAT competition-2018, we provide 20 SAT instances of the $\mathbf{GrandTour^{obs}}$ puzzle.**

I. Encoding and Instance Generation

a) $\mathbf{GrandTour^{obs}}$ *as Hamiltonian Cycle Problem:* The problem of determining whether a Hamiltonian cycle exists in a given graph is a well-known NP-complete problem. This problem deals with the following question: is there a cycle in a given graph, in which each vertex is visited exactly once? While solving a **GrandTour** puzzle is equivalent to finding a Hamiltonian cycle in a complete graph[2] induced from its completely connected puzzle grid, solving $\mathbf{GrandTour^{obs}}$ problem is equivalent to finding a Hamiltonian cycle in an incomplete graph, which is induced from its incompletely connected puzzle grid. If a Hamiltonian cycle exists in the graph induced from the puzzle grid of a $\mathbf{GrandTour^{obs}}$ puzzle, then the puzzle is solvable, otherwise it is not.

b) Problem Encoding: A ground normal logic program P can be converted into a SAT formula S, from the Clark's completion C of P. For example, the Clark's completion of $P = \{a \leftarrow b., a \leftarrow c.\}$ is $C = \{a \leftarrow b \vee c.\}$ and C can be expressed into the SAT formula S in CNF form as: $(\neg b \wedge \neg c) \vee a \equiv (\neg b \vee a) \wedge (\neg c \vee a)$. This technique is utilized in ASSAT [1], a solver for answer set programs using a SAT solver as the underlying inference engine. Given a ground normal logic program P, ASSAT computes its stable models by using a SAT solver. As part of its solving process, ASSAT produces a SAT formula S from P. We exploit this feature of ASSAT to generate SAT instances of the $\mathbf{GrandTour^{obs}}$ problem from the normal logic program encoding of Hamiltonian cycle as proposed in [2].

c) Instance Generation: For the $\mathbf{GrandTour^{obs}}$, a puzzle grid of size $x \times y$ can be induced from a graph with n nodes, where n is an even number and $n = x * y$. The website for ASSAT[3] contains 32 normal logic program instances for the Hamiltonian cycle problem from two graphs with 60 and 50 vertexes, respectively. We treat these instances as the $\mathbf{GrandTour^{obs}}$ instances of grid size 10×6 and 10×5, respectively.

TABLE I: Details of the 20 SAT instances for the $\mathbf{GrandTour^{obs}}$ benchmark; An instance *gto_p***x***c***y** is based on a grid which has x points and y pairs of points are connected.

Problem/Grid Size	**Variables/Clauses**	**MiniSAT cpuTime (s)**	**MiniSAT Results**
gto_p60c229 / 10×6	1011/3473	101.95	UNSAT
gto_p60c231 / 10×6	1015/3493	351.36	UNSAT
gto_p60c241 / 10×6	1046/3673	481.81	UNSAT
gto_p60c239 / 10×6	1038/3627	717.90	UNSAT
gto_p60c231_1 / 10×6	1019/3517	833.89	UNSAT
gto_p60c243 / 10×6	1054/3721	881.04	UNSAT
gto_p60c233 / 10×6	1023/3533	1177.71	UNSAT
gto_p60c234 / 10×6	1027/3577	1897.77	UNSAT
gto_p60c235 / 10×6	1031/3583	3360.01	UNSAT
gto_p60c238 / 10×6	1043/3649	4232.67	UNSAT
gto_p60c295 / 10×6	635/3330	5000	UNKNOWN
gto_p60c343 / 10×6	729/4563	5000	UNKNOWN
gto_p50c291 / 10×5	607/3813	5000	UNKNOWN
gto_p50c345 / 10×5	703/5413	5000	UNKNOWN
gto_p50c311 / 10×5	623/4137	5000	UNKNOWN
gto_p50c312 / 10×5	643/4325	5000	UNKNOWN
gto_p50c314 / 10×5	635/4319	5000	UNKNOWN
gto_p50c314_1 / 10×5	639/4339	5000	UNKNOWN
gto_p50c345 / 10×5	605/3664	5000	UNKNOWN
gto_p50c307 / 10×5	613/4036	5000	UNKNOWN

These 32 instances are known to be hard for SAT solvers. In our experiment, MiniSat could not solve any of those within 5000 seconds. So, these instances are not *interesting* for the SAT competition-2018. As per the requirement of the benchmark submission, to generate 10 *interesting* SAT instances, we took a graph G, namely *hard.1.graph* in the ASSAT website, and repeated the following steps until we get 10 *interesting* instances: (i) randomly remove a few arcs to produce a new graph G', (ii) generate the SAT instance S from the normal logic program encoding of the Hamiltonian cycle program along with G' by using ASSAT, and (iii) test S with MiniSat to see if it is *interesting*. The first 10 rows in Table 1 show these *interesting* instances. The next 10 rows give the details of the 10 hard instances taken from the ASSAT website.

[1]http://curiouscheetah.com/Museum/Puzzle/Grandtour
[2]In a complete graph, each pair of vertexes are connected by an edge.
[3]http://assat.cs.ust.hk/hardsat.html

References

[1] F. Lin and Y. Zhao, “ASSAT: computing answer sets of a logic program by SAT solvers”, Artif. Intell. , vol. 157, pp. 115–137, 2004.

[2] I. Niemelä, “Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm”, Ann. Math. Artif. Intell., vol. 25, pp.241–273, 1999.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Polynomial Multiplication

Chu-Min Li[1], Fan Xiao[2], Mao Luo[2], Felip Manyà[3], Zhipeng Lü[2], Yu Li[1]
[1]MIS, University of Picardie Jules Verne, Amiens, France
chu-min.li@u-picardie.fr, corresponding author
[2]School of Computer Science, Huazhong University of Science and Technology, Wuhan, China
{maoluo,fanxiao,zhipeng.lv}@hust.edu.cn
[3]Artificial Intelligence Research Institute (IIIA-CSIC), Barcelona, Spain
felip@iiia.csic.es

***Abstract*—Multiplying two polynomials of degree $n-1$ can need n^2 coefficient products, because each polynomial of degree $n-1$ has n coefficients. If the coefficients are real numbers, the Fourier transformation allows to reduce the number of necessary coefficient products to $O(n*log(n))$. However, when the coefficients are not real numbers (e.g., the coefficients can be a matrix), the Fourier transformation cannot be used. In this case, reducing the number of necessary coefficient products can significantly speed up the multiplication of two polynomials. In this short paper, we reduce the problem of multiplying two polynomials of degree $n-1$ with t ($t \leq n^2$) coefficient products to SAT and provide 20 new crafted SAT instances.**

1. Introduction

A simple example of polynomial multiplication can be expressed using Equation 1:

$$(ax+b)(cx+d) = acx^2 + (ad+bc)x + bd \tag{1}$$

The trivial multiplication of the two polynomials of degree 1 needs 4 coefficient products: $\{ac, ad, bc, bd\}$. A smart multiplication of the two polynomials needs only 3 coefficient products $\{ac, (a+b)(c+d), bd\}$, as expressed in Equation 2:

$$(ax+b)(cx+d) = acx^2 + \Big((a+b)(c+d) - ac - bd\Big)x + bd \tag{2}$$

In Equation 2, we need more addition and subtraction operations than in Equation 1. However, multiplication is much more costly than addition and subtraction. So, we can multiply two polynomials of degree 1 more quickly using Equation 2 than using Equation 1.

In the general case, we want to multiply two polynomials of degree $n-1$ using fewer than n^2 coefficient products. If the coefficients are real numbers, the Fourier transformation allows to reduce the number of necessary coefficient products to $O(n*log(n))$. However, when the coefficients are not real numbers (e.g., the coefficients can be a matrix), the Fourier transformation cannot be used.

In the sequel, we describe how to reduce the problem of multiplying two polynomials of degree $n-1$ using t ($t \leq n^2$) coefficient products to SAT. When the obtained SAT instance is satisfiable, the SAT solution gives a way to multiply two polynomials of degree $n-1$ using t coefficient products. When the obtained SAT instance is unsatisfiable, we know that more than t coefficient products are needed. We refer to [1], [2] for other efficient algorithms for polynomials.

2. SAT Encoding of polynomial Multiplication Using t Products

Consider two polynomials of degree $n-1$:

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$$

$$B(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1x + b_0$$

Their product is

$$A(x) \times B(x) = c_{2n-2}x^{2n-2} + c_{2n-3}x^{2n-3} + \cdots + c_1x + c_0$$

We want to compute $A(x) \times B(x)$ using t ($t \leq n^2$) coefficient products: $P_1, P_2, \ldots, P_t$, where each P_l ($1 \leq l \leq t$) is of the form $(a'_1 + a'_2 + \cdots)(b'_1 + b'_2 + \cdots)$ with $a'_1, a'_2, \ldots \in \{a_{n-1}, a_{n-2}, \ldots, a_0\}$ and $b'_1, b'_2, \ldots \in \{b_{n-1}, b_{n-2}, \ldots, b_0\}$. Addition and subtraction of these products give the coefficients c_k ($0 \leq k \leq 2n-2$) of $A(x) \times B(x)$. The problem becomes to determinate a'_i and b'_j for each product. In order to solve the problem, we first define the following Boolean variables.

- $a_{il} = 1$ iff a_i is involved in product P_l;
- $b_{jl} = 1$ iff b_j is involved in product P_l;
- $c_{kl} = 1$ iff product P_l is used to compute c_k;
- $x_{ijkl} = 1$ iff a_i and b_j are involved in product P_l, and product P_l is used to compute c_k;

We then define the clauses of the CNF, which encode the following properties:

- $x_{ijkl} \equiv a_{il} \wedge b_{jl} \wedge c_{kl}$

- For each i and j $(0 \leq i, j \leq n-1)$ and for each k $(0 \leq k \leq 2n-2)$ such that $i+j \neq k$, if a_i and b_j are involved in product P_l (i.e., $a_{il} \wedge b_{jl}$ is implied) and P_l is used to produce c_k, then the product of a_i and b_j should be eliminated by subtraction using another product $P_{l'}$ involving a_i and b_j. If $i+j=k$, one product of a_i and b_j should remain in c_k. So,

$$\sum_{l=1}^{t} x_{ijkl} \bmod 2 = \begin{cases} 0 & \text{if } i+j=k \\ 1 & \text{otherwise} \end{cases}$$

3. Set of Submitted Instances

We generated 20 SAT instances, using the encoding of the previous section, by varying n and t as follows:

- $n = 8$, $t \in \{60, 61, 62, 63\}$
- $n = 11$, $t \in \{118\}$
- $n = 13$, $t \in \{165, 166\}$
- $n = 14$, $t \in \{194\}$
- $n \in \{23, 27, 29, 37, 39, 42, 44, 45, 49, 51, 52, 54\}$, $t = 6$

Each combination of n and t gives an instance pol$n$-t.

Table 1 shows, for each one of the 20 generated instances, its number of variables and clauses, the status of the formula (satisfiable, unsatisfiable or unknown), and the time needed by MiniSat [3] to solve the instance on a computer with Intel Westmere Xeon E5-2680 of 2.40GHz and 20GB of memory under Linux. The cutoff time is 3600 seconds.

TABLE 1. INFORMATION ABOUT THE GENERATED INSTANCES.

Instance	#Variables	#Clauses	Satisfiability	Time
Nb8T60	114180	453120	unkown	timeout
Nb8T61	116131	460800	unkown	timeout
Nb8T62	118082	468480	SAT	2737
Nb8T63	120033	476160	unkown	timeout
Nb11T118	597127	2378376	unkown	timeout
Nb13T165	1389990	5543200	unkown	timeout
Nb13T166	1398491	5577000	unkown	timeout
Nb14T194	2048090	8170848	unkown	timeout
Nb23T6	214791	952200	UNSAT	243.4
Nb27T6	348375	1545480	UNSAT	467.6
Nb29T6	432123	1917480	UNSAT	694.1
Nb37T6	900315	3997480	UNSAT	1441
Nb39T6	1054983	4684680	UNSAT	1343
Nb42T6	1318710	5856480	UNSAT	2354
Nb44T6	1516938	6737280	UNSAT	2295
Nb45T6	1623099	1623099	UNSAT	2184
Nb49T6	2097243	9315880	UNSAT	4301
Nb51T6	2365527	10508040	UNSAT	3185
Nb52T6	2507850	11140480	UNSAT	3310
Nb54T6	2809398	12480480	UNSAT	3249

Acknowledgments

Research partially supported by the National Natural Science Foundation of China (Grants no. 61370183, 61370184, 61472147), the MINECO-FEDER project RASO TIN2015-71799-C2-1-P, and the MeCS platform of university of Picardie Jules Verne.

References

[1] D. Bini and V. I. Pan, *Polynomial and Matrix Computations: Fundamental Algorithms*. Birkhauser Verlag Basel, Switzerland, 1994.

[2] R. Zippel, *Effective Polynomial Computation*. Springer Science & Business Media, 2012.

[3] N. Eén and N. Sörensson, "An extensible sat-solver," in *6th International Conference on Theory and Applications of Satisfiability Testing , SAT*, 2003, pp. 502–518.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Encodings of Relativized Pigeonhole Principle Formulas with Auxiliary Constraints

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

Abstract—**Although a relativized pigeonhole principle (RPHP) formula has been proven to require resolution proofs of size roughly n^k, if using its symmetry, a CDCL SAT solver solve easily it. To destroy its symmetry, here we add extra constraints so that the formula generated is a non-symmetry CNF.**

I. Introduction

The SAT-encoding of relativized pigeonhole principle (RPHP) formulas yields a symmetry CNF [1]. In theory, such a CNF requires resolution proofs of size roughly n^k [2]. However, if using a preprocessor such as BreakID [3] to detect symmetry and construct symmetry breaking formulas, it has polynomial-size bounded-depth proofs, and is solved easily by a general CDCL SAT solver [4]. In this documentation, we encode RPHP with additional constraints. The resulting CNF is non-symmetry.

II. SAT Encoding of RPHP with extra constraints

A relativized pigeonhole principle (RPHP) formula is defined as whether k pigeons can fly into $k-1$ holes via n "resting places". We denote such a claim by $RPHP^{k,n}_{k-1}$. Let $[n] = \{1, 2, \dots, n\}$. Let p and q be functions: $[k] \to [n]$ and $[n] \to [k]$, respectively. Furthermore, assume that p is one-to-one and defined on $[k]$, and q is one-to-one and defined on the range of p. We encode p , q and a superset of the range of p by Boolean variables $p_{u,v}$, $q_{v,w}$ and r_v. $RPHP^{k,n}_{k-1}$ is encoded as follows.

$$\begin{array}{ll}
p_{u,1} \vee p_{u,1} \vee \cdots \vee p_{u,n} & 1 \le u \le k \\
\overline{p}_{u,v} \vee \overline{p}_{u',v} & \text{for all } u \ne u', v \\
\overline{p}_{u,v} \vee r_v & \text{for all } u, v \\
\overline{r}_v \vee q_{v,1} \vee q_{v,1} \vee \cdots \vee q_{v,k-1} & 1 \le v \le n \\
\overline{r}_v \vee \overline{r}_{v'} \vee \overline{q}_{v,w} \vee \overline{q}_{v',w} & \text{for all } v \ne v', w
\end{array}$$

Let $m = n/2$. We add the following constraint clauses to $RPHP^{k,n}_{k-1}$.

$$\begin{array}{ll}
\overline{p}_{u,v} \vee x_v & 1 \le u \le k, 1 \le v \le m \\
\overline{x}_v \vee \overline{r}_{v'} \vee \overline{q}_{v,w} \vee \overline{q}_{v',w} & 1 \le v \le m < v' \le n, 1 \le w < k \\
\overline{x}_v \vee \overline{r}_{v'} \vee \overline{p}_{v,w} \vee \overline{p}_{v',w} & 1 \le v \le m < v' \le n, 1 \le w < k
\end{array}$$

where x_v is a additional variable different from $p_{u,v}$, $q_{v,w}$ and r_v. Clearly, $RPHP^{k,n}_{k-1}$ with the above constraint clauses is not symmetry.

$RPHP^{k,n}_{k-1}$ can be transformed into a 3-CNF by using extension variables to break up the long clauses. Clause $p_{u,1} \vee p_{u,1} \vee \cdots \vee p_{u,n}$ may be transformed into

$$\begin{array}{ll}
p_{u,1} \vee p_{u,1} \vee y_{u,2} & \\
\overline{y}_{u,v} \vee p_{u,v+1} \vee y_{u,v+1} & 2 \le v \le n-3 \\
\overline{y}_{u,n-2} \vee p_{u,n-1} \vee y_{u,n} &
\end{array}$$

Similarly, we can transform $\overline{r}_v \vee q_{v,1} \vee q_{v,1} \vee \cdots \vee q_{v,k-1}$ and $\overline{r}_v \vee \overline{r}_{v'} \vee \overline{q}_{v,w} \vee \overline{q}_{v',w}$ into a 3-CNF.

Although the 3-CNF version of $RPHP$ is also symmetry, not all symmetry relations were detected by BreakID [3]. So we submit also the 3-CNF version of $RPHP$ to the SAT competition 2018.

References

[1] A. Atserias, M. Lauria, J. Nordström: Narrow proofs may be maximally long, ACM Transactions on Computational Logic, vol. 17, pp. 19:1 – 19:30, May 2016, preliminary version in CCC' 14.
[2] J. Elffers, J. Nordström: Documentation of some combinatorial benchmarks, in *Proceedings of the SAT Competition 2016*, pp. 67–69.
[3] J. Devriendt and B. Bogaerts: BreakID, a symmetry breaking preprocessor for sat solvers, bitbucket.org/krr/breakid, 2015.
[4] J. Devriendt and B. Bogaerts: BreakIDCOMiniSatPS, in *Proceedings of the SAT Competition 2016*, pp. 31-32.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Random k-SAT q-planted solutions

Adrian Balint
independent researcher
Ulm, Germany

Hiding solutions in a randomly generated formula can be accomplished in different ways. An overview of the different possibilities is given in [1] Section 2.2.3.

One of the most promising approaches is the one proposed in [4]. The authors define a single parameter q. A randomly generated clause that has t satisfied literals in the planted solution is accepted with a probability proportional to q^t. The authors call this model the q-hidden model. To generate balanced instances (positive and negative literal probability should be the same) the authors show that q should be set to $q = (\sqrt{5} - 1)/2 = 0.618$, which is the golden ratio. To derive this value we need some equations. In case of 3-SAT we have seven possible clauses that have one, two or three satisfied literals in the planted solution. For each type of clause we assign the probabilities p_1 for 1-satisfied and accordingly p_2 and p_3. Due to probability normalization we have the equation:

$$3p_1 + 3p_2 + p_3 = 1 \tag{1}$$

For an arbitrary position in the clause we have to balance the probabilities of negative and positive occurrence of a variable. Setting the sum of these probabilities to be equal, we can add the additional equation:

$$p_1 + 2p_2 + p_3 = 2p_1 + p_2 \tag{2}$$

Adding to the Equations 1, 2 the additional equations:

$$p_2/p_1 = p_3/p_2 = q \tag{3}$$

and then solving this equation system, we will get the same value for q as mentioned above.

The value of q, which depends on k, can be computed as the positive root of the equation:

$$(1 - q)(1 + q)^{k-1} - 1 = 0 \tag{4}$$

To compute the values of p_t we can use the general form of the normalization Equation 1, which then results in:

$$p_t = \frac{q^t}{(1 + q)^k - 1} \tag{5}$$

k-SAT	3	4	5	6	7
q	0.6180	0.8392	0.9275	0.9659	0.9835

TABLE I
THE POSITIVE ROOTS OF EQUATION 5 FOR DIFFERENT VALUES OF k.

Table I shows the values of q for typical k-SAT. Smaller values for q will result in problems where local search algorithms are more likely to move away from the hidden solution then being attracted by it. This type of problems is also called *deceptive* formulas. A systematic construction of deceptive formulas that were shown to be very hard for local search solvers are also presented in [3].

A simple uniform k-SAT generator can be altered to generate planted solution instances according to the q-hidden model by adding an additional condition that checks if the clause is to be accepted or should be dropped depending on the planting model. After computing the number of true literals in the clause, the clause will be accepted according to an acceptance probability, which can be computed from the probability distribution $(p_1, \ldots, p_k)$ by normalizing the p-values with respect to p_1. This will result in an acceptance probability distribution of $(1, q, q^2, \ldots, q^{k-1})$ where the value of q can be found in Table I.

For the SAT Competition 2018 we submit ten 3-SAT instances (r=4.267, n= 7000), ten 5-SAT instances (r=21.117, n=250) and ten 7-SAT instances (r=87.79, n=120) generated according to the q-hidden model.

REFERENCES

[1] Adrian Balint: Engineering Stochastic Local Search for the Satisfiability Problem, PhD Thesis Ulm University 2013.
[2] Adrian Balint, Uwe Schöning: Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break Lecture Notes in Computer Science, 2012, Volume 7317, Theory and Applications of Satisfiability Testing - SAT 2012, pages 16-29
[3] Hirsch, E. A. SAT Local Search Algorithms: Worst-Case Study J. Autom. Reasoning, 2000, 24, 127-143
[4] Jia, H.; Moore, C. ; Strain, D. Generating hard satisfiable formulas by hiding solutions deceptively In AAAI, AAAI Press, 2005, 384-389

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

k-colorability benchmarks

Jo Devriendt
University of Leuven
Leuven, Belgium

***Abstract*—Deciding the k-colorability of a graph is a well-known NP-complete problem with symmetry properties.**

I. Description

The k-colorability problem consists of deciding whether there exists an assignment (a *coloring*) of a set of k colors to a graph's nodes, such that a different color is assigned to two neighboring nodes.

An instance of this problem is encoded to a CNF-formula by introducing Boolean variables x_{ij} that denote whether node i is assigned color j, by constructing for each node a clause that requires at least one color to be assigned to the node, and by introducing for each edge in the graph a binary clause excluding that its incident nodes are assigned the same color (for each color).

The k-colorability problem exhibits strong symmetry properties, as permuting the colors in a coloring has no influence on whether neighboring nodes are colored differently. In other words, the colors are *interchangeable*, and the larger the set of colors (the higher k), the more this symmetry property will hinder search.

Aside from symmetry induced by color interchangeability, certain node permutations might also induce symmetry, depending on the structure of the graph at hand.

To generate 20 k-coloring encodings, we took graphs from Michael Trick's Operations Research Page [1]. For each graph, we chose k such that the instance became unsatisfiable: too few colors are available for a valid coloring. In Table I, the specific k-values for each instance are presented.

Finally, we composed an instance set where Glucose 4.0 was able to solve 9 instances on an Intel(R) Xeon(R) E3-1225 cpu using 16 GB of memory and a timeout of 5000 seconds.

instance name	k-value	Glucose 4.0 solve time (s)
anna.col.11.cnf	10	101
david.col.11.cnf	10	103
huck.col.11.cnf	10	145
jean.col.10.cnf	9	8
le450_15a.col.15.cnf	14	5000+
le450_15b.col.15.cnf	14	5000+
le450_15c.col.15.cnf	14	5000+
le450_25a.col.25.cnf	24	5000+
le450_25b.col.25.cnf	24	5000+
le450_25d.col.25.cnf	24	5000+
myciel5.col.6.cnf	5	17
queen10_10.col.10.cnf	9	8
queen11_11.col.11.cnf	10	116
queen12_12.col.12.cnf	11	2429
queen13_13.col.13.cnf	12	5000+
queen14_14.col.14.cnf	13	5000+
queen15_15.col.15.cnf	14	5000+
queen8_12.col.12.cnf	11	2081
school1_nsh.col.14.cnf	13	5000+
school1.col.14.cnf	13	5000+

[1] mat.gsia.cmu.edu/COLOR/instances.html

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Searching for a Unit-Distance Graph with Chromatic Number 6

Marijn J. H. Heule

Department of Computer Science,
The University of Texas at Austin, United States

***Abstract*—This benchmark suite consists of determining whether some unit-distance graphs can be colored with 5 colors. Starting with a hard to color unit-distance graph, smaller graphs are produced by removing dozens of vertices at a time. All instances are satisfiable.**

Intro

The *chromatic number of the plane* (CNP), a problem first proposed by Edward Nelson in 1950 [1], asks how many colors are needed to color all points of the plane such that no two points at distance 1 from each other have the same color. Early results showed that at least four and at most seven colors are required. By the de Bruijn–Erdős theorem, the chromatic number of the plane is the largest possible chromatic number of a finite unit-distance graph [2]. The Moser Spindle, a unit-distance graph with 7 vertices and 11 edges, shows the lower bound [3], while the upper bound is due to a 7-coloring of the entire plane by John Isbell [1].

In a recent breakthrough for this problem, Aubrey de Grey improved the lower bound by providing a unit-distance graph with 1581 vertices with chromatic number 5 [4]. This graph was obtained by shrinking the initial graph with chromatic number 5 consisting of 20 425 vertices. The 1581-vertex graph is almost minimal: at most 4 vertices can be removed without introducing a 4-coloring of the remaining graph. The discovery by de Grey started a Polymath project to find smaller unit-distance graphs with chromatic number 5 and a graph with chromatic number 6 or 7.

Preliminaries

A graph for which all edges have the same length is called a *unit-distance graph*. A lower bound for CNP of k colors can be obtained by showing that a unit-distance graph has chromatic number k.

Given two sets of points A and B, the Minkowski sum of A and B, denoted by $A \oplus B$, equals $\{a + b \mid a \in A, b \in B\}$. Consider the sets of points $A = \{(0,0),(1,0)\}$ and $B = \{(0,0),(1/2,\sqrt{3}/2)\}$, then $A \oplus B = \{(0,0),(1,0),(1/2,\sqrt{3}/2),(3/2,\sqrt{3}/2)\}$.

Given a positive integer i, we denote by θ_i the rotation around point $(0,0)$ with angle $\arccos(\frac{2i-1}{2i})$ and by θ_i^k the application of θ_i k times. Let p be a point with distance $\sqrt{i}$ from $(0,0)$, then the points p and $\theta_i(p)$ are exactly distance 1 (unit distance) apart and thus would be connected with an edge in a unit-distance graph.

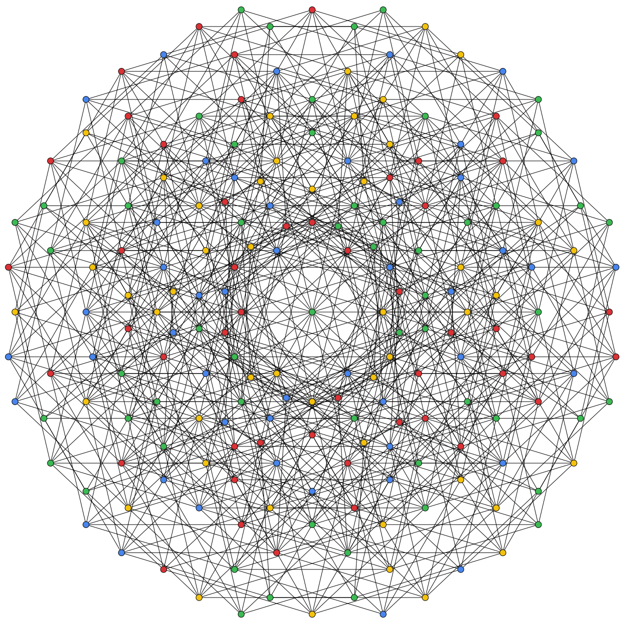


Fig. 1. A 4-coloring of the graph S_{199}.

Chromatic Number 6?

A recent article [5] produced an interesting symmetric graph, known as S_{199}. This graph is shown in Figure1. Finding a 5-coloring of graph $S_{199} \oplus \theta_4(S_{199})$ is hard. The benchmark suite contains 20 satisfiable formulas that encode whether subgraphs of this graph have a 5-coloring. Advances in solving these benchmarks may help finding a unit-distance graph with chromatic number 6.

References

[1] A. Soifer, *The Mathematical Coloring Book*, 2008, ISBN-13: 978-0387746401.

[2] N. G. de Bruijn and P. Erdős, "A colour problem for infinite graphs and a problem in the theory of relations," *Nederl. Akad. Wetensch. Proc. Ser. A*, vol. 54, pp. 371–373, 1951.

[3] L. Moser and W. Moser, "Solution to problem 10," *Can. Math. Bull.*, vol. 4, pp. 187–189, 1961.

[4] A. D. N. J. de Grey, "The chromatic number of the plane is at least 5," 2018, arXiv:1804.02385.

[5] M. J. H. Heule, "Computing small unit-distance graphs with chromatic number 5," 2018, arXiv:1805.12181.

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

SATcoin – Bitcoin mining via SAT

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

Jonathan Heusser
jonathan.heusser@gmail.com
London, England

***Abstract*—The main security properties of Bitcoin, censorship resistance and an immutable historical ledger of transactions is enforced by its mining algorithm. By design this "proof of work" requires a large amount of computational power to verify transactions. While current mining algorithms are based on brute force, this document briefly describes how the same problem can be solved via SAT. Most details of this document are taken from [1].**

I. Bitcoin Mining

A Bitcoin mining program essentially performs the following (in pseudo-code):

```
nonce = MIN
while(nonce < MAX):
  if sha(sha(block+nonce)) < target:
     return nonce
  nonce += 1
```

The task is to find a nonce which, as part of the bitcoin block header, hashes below a certain value.

This is a brute force approach to something like a preimage attack on SHA-256. The process of mining consists of finding an input to a cryptographic hash function which hashes below or equal to a fixed target value. At every iteration the content to be hashed is slightly changed to find a valid hash; there's no smart choice in the nonce. The choice is essentially random as this is the best one can do on such hash functions.

In [1], Heusser proposed an alternative mining algorithm which does not perform a brute force search. Instead, we utilize tools from the program verification domain to find bugs or prove properties of programs, see as example [2]. To find the correct nonce or prove the absence of a valid nonce, a model checker backed by a SAT solver is used on a C implementation of the hashing. In contrast to brute force, which actually executes and computes many hashes, the new approach is symbolically executing the hash function with added constraints that are inherent in the bitcoin mining process. The submitted benchmark is based on CNFs created by CBMC.

II. Bitcoin mining using SAT Solving and Model Checking

We take an existing C implementation of SHA-256 from a mining program and strip away everything but the actual hash function and the basic mining procedure of sha(sha(block)). This C file is the input to CBMC [3].

By adding the right assumptions and assertions to the implementation, i.e. MIN and MAX in the pseudo code above, we direct the SAT solver to find a nonce. Instead of a loop which executes the hash many times and a procedure which checks if we computed a correct hash, we add constraints that when satisfied implicitly have the correct nonce in its solution.

The assumptions and assertions can be broken down to the following ideas:

- The nonce is modelled as a non-deterministic value
- The known structure of a valid hash, i.e. leading zeros, is encoded as assumptions in the model checker
- An assertion is added stating that a valid nonce does not exist

More details about the translation, and a basic solver preformance comparison can be found in the full article [1].

III. SAT Competition Specifics

The used input file for CBMC implements the described analysis, and uses the so called *genesis block* of the bitcoin blockchain, the very first block of the chain. We provide a script that allows to produce a CNF based on the fact whether the formula should be satisfiable, or unsatisfiable. The formula is satisfiable, if the valid nonce is part of the given range. Hence, the range is constructed such that the valid nonce is right in the middle of the range to be analyzed. For unsatisfiable formulas, the range starts right after the valid nonce.

The code, as well as all scripts to create a benchmark are available at https://github.com/jheusser/satcoin.

References

[1] J. Heusser, "SAT solving - an alternative to brute force bitcoin mining," http://jheusser.github.io/2013/02/03/satcoin.html, 2013, accessed: 2018-04-15.

[2] J. Heusser and P. Malacaria, "Quantifying information leaks in software," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 261–269. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920300

[3] D. Kroening and M. Tautschnig, "CBMC - C bounded model checker - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 389–391. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_26

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Logical Cryptanalysis Benchmarks for Classical and Modern Hash Functions

Alexander Scheel, Iowa State University, alexander.m.scheel@gmail.com

I. Introduction

This collection of benchmarks focuses on new techniques in logical cryptanalysis for analyzing the structure of various hash functions. We offer as a benchmark a new technique analyzing the security of a hash function of the Merkle-Dåmgard construction which are broadly applicable to all such constructions and several benchmarks analyzing the various security properties of the Keccak hash function. While the latter techniques have not yeilded useful cryptanalysis results, they pose as a source of variable scalable benchmarks due to several choices of parameters (the bitwidth, w and the number of rounds).

II. Classical Hash Functions

The MD4 hash function is composed of 48 iterated rounds, each updating one of four 32-bit state variables [4]. While authors such as I. Mironov have applied SAT solvers to finding collisions given an existing differential path [3] and D. Jovanović has applied applied SAT solvers to brute force finding collisions and preimages in hash functions [2], these techniques either require pre-existing collisions or are too hard for SAT solvers on a large number of rounds.

One technique that is efficient for even a modest number of rounds (≤ 28) is the notion of a differential family search. In most cases, the differential path between two blocks b_1 and b_2 is the simple XOR difference between the intermediate rounds. For a given path p, there are often many such blocks which create a collision; I. Mironov showed that SAT solvers can find such blocks in MD4 and MD5 [3] relatively quickly. We can extend this concept to differential paths as well: given a differential path p, its family is the ordered tuple of indices of the rounds with a non-zero difference. For example, the family of the differential path introduced by M. Schlaffer (in [6]) is:

$$(1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 19, 20, 35, 36)$$

Thus for a given family, f, there are multiple possible differential paths which have a structure described by f.

By using this concept of a differential family, we can divide the search space into multiple different SAT problems: for a given number of rounds, r, there are 2^{r-4} possible differential families (versus 2^{32w-32} possible differential paths). For $r \leq 28$ it is possible to exhaustively search large portions of the family space; for $32 \leq r \leq 36$ it becomes possible to find in select cases, and for $r > 36$, it remains impossible in nearly all cases (due to the large time limit). Our benchmarks are a sample of possible differential families for $r = 24$, mixing both SAT and UNSAT results. These benchmarks can be found in the `families` folder [5].

III. Modern Hash Functions

The notion of a differential family is not as useful for Keccak due to the interaction between the sponge function and the internal permutation functions causing the collision space of families to be entirely different between successive rounds. From an algebraic perspective, and to study the security margin of the XOF consruct [1], it is important to show that the Keccak round functions are bijective and have high permutation order. Two of our benchmarks (`bijection` and `orders`) model these problems in SAT. In general, these instances are easy, but with a few notable outliers: for $w \geq 4$, proving the order of θ is $3w$ is difficult. Note that this should simplify to showing that $\forall x, x \neq x$ is UNSAT, and thus should be relatively trivial; however $w = 4$ produces runtimes in excess of one hour. Further, anything later involving θ (such as $\rho \circ \theta$, etc.) also becomes difficult for $w \geq 4$.

In `differences`, we study the differential properties of the permutation functions: for a given parameters $x, y \leq 25w$ and round function f, we seek to find witnesses a and b such that:

$$\#(1, a \oplus b) = x$$
$$\#(1, f(a) \oplus f(b)) = y$$

That is, find an input with difference x which produces an output with difference y. There are a few interesting outliers in this model: for $w \geq 4$, most round functions cannot be individually analyzed this way. Further, while π is a permutation of the order of the bits (and thus does not change the values of any bits), certain instances in $w = 2$ where $x \neq y$ produce runtimes in excess of an hour. This is surprising as the model is trivially SAT if and only if $x == y$ (and the model merely involves changing the orders of variables).

The benchmarks in `output-margins` consider the effects of the sponge function. Since all of the round functions are bijective, if two inputs differ, the interal state of Keccak must also differ. However, to produce a collision, only the first y bits of the output must be the same. Thus we can create a model for a given number of differences $x, z \leq 25w$, security margin $y \leq 25w$, and round functions f:

$$\#(1, axorb) = x$$
$$\#(1, (f(a)xorf(b))[0 : y]) = z$$

If such a witness a and b exist for $z = 0$, then the input difference x is possible of producing collisions at a security margin m. Our provided benchmarks sample the space for small values of w and relatively few round functions; for $w \geq 16$ and for any set of functions including θ, the runtimes become exceedingly long.

The benchmarks in `xof-state` attempt to recreate the internal state of Keccak given a series of outputs from the XOF (extensible output function at a given margin). In general, this is possible for either small values of w or small numbers of rounds (for larger values of w). However, care must be selected in choosing the base seed, otherwise, there can possibly be multiple satisfying seeds. However, for a set of inputs with unique solution, these benchmarks can be extended to contain the output from several rounds of Keccak. After a threshhold dependent on the margin, these are redundant information and thus test the SAT solvers to work with larger models which overfit to the solution.

IV. Thanks

A special thanks to Mate Soos and his CryptoMiniSat solver [7] for suggesting these problems be submitted as benchmarks to the conference.

This work was completed as part of an Undergraduate Honors Research project at Iowa State University under the guidance of Eric W. Davis and as part of MATH 490: Independent Study of SHA-3 under the guidance of Clifford Bergman.

References

[1] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: The keccak reference - version 3.0 (2011), https://keccak.team/files/Keccak-reference-3.0.pdf

[2] Jovanović, D., Janičić, P.: Logical Analysis of Hash Functions, pp. 200–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), http://csl.sri.com/users/dejan/papers/jovanovic-hashsat-2005.pdf

[3] Mironov, I., Zhang, L.: Applications of SAT Solvers to Cryptanalysis of Hash Functions, pp. 102–115. Springer Berlin Heidelberg, Berlin, Heidelberg (2006), https://doi.org/10.1007/11814948_13, https://eprint.iacr.org/2006/254.pdf

[4] Rivest, R., RSA Data Security, I.: The MD4 Message-Digest Algorithm. RFC 1320, IETF (April 1992), https://tools.ietf.org/html/rfc1320

[5] Scheel, A.: Sat competition 2018 submission. GitHub (2018), https://github.com/cipherboy/sat/tree/master/sat-competition-2018

[6] Schläffer, M., Oswald, E.: Searching for Differential Paths in MD4, pp. 242–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2006), https://doi.org/10.1007/11799313_16, https://link.springer.com/content/pdf/10.1007%2F11799313_16.pdf

[7] Soos, M.: Cryptominisat sat solver. GitHub (2017), https://github.com/msoos/cryptominisat

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

Benchmark: Assigning Time-Slots to Students with Optimal Preference Satisfaction

Markus Iser
Institute for Theoretical Computer Science,
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: markus.iser@kit.edu

***Abstract*—This family contains benchmarks from a tool that assigns time-slots to students by given individual preferences with local and global cost reduction. The problem is solved by encoding a sequence of SAT problems with decreasing local and global bounds.**

I. Introduction

During each semester of our lecture, a number m of tutors $S = \{s_0, \dots, s_m\}$ is helping their fellow students to comprehend the subject matter through lessons in small tutorial groups. The trainer allocates n free time-slots $T = \{t_0, \dots, t_n\}$ $(n \geq m)$ in seminar rooms, in order to assign exactly one time-slot to each tutor. Before an assignment $z : S \rightarrow T$ can be determined, each tutor may order the available time-slots by their preference, such that prefered time-slot come first in their list. A cost function $c : S \times T \rightarrow \mathbb{N}$ can be deduced such that the cost of the assignment corresponds to its postion in the ordered list.

The goal is to find an assignment z of time-slots to tutors, such that their preferences are met as good as possible. In order to achieve this, we use two subsequent optimization rounds: one local and one global optimization round. In the local optimization round, we search the smallest k_1 and an assignment z that satisfies $\max_{s \in S} c(s, z(s)) \leq k_1$. When the optimal k_1 is found, a second optimzation round begins. In the global optimization round we search for the smallest k_2 and an assignment z that satisfies $\sum_{s \in S} c(s, z(s)) \leq k_2$, and that still satisfies the local bound constraint with the previously determined k_1.

We use a tool [1] that generates a sequence of SAT problems to find an optimal solution for this problem. It follows a quick overview on the encoding.

II. Encoding

So the input of our tool is a set of dates $T = \{t_0, \dots, t_n\}$, a set of students $S = \{s_0, \dots, s_m\}$ and the set of preferences $c : S \times T \rightarrow \mathbb{N}$.

The goal in the local optimization round is to find an assignment $z : S \rightarrow T$ and the minimal local bound k_1 such that $\max_{s \in S} c(s, z(s)) \leq k_1$ holds.

The goal in the subsequent global optimization round is to find an assignment $z : S \rightarrow T$ and the minimal global bound k_2 such that $\sum_{s \in S} c(s, z(s)) \leq k_2$ holds and the optimal local bound k_1 is still satisfied.

A. Boolean atoms

In our encoding we use $n \times m$ Boolean atoms z_{st} for the assignment which shall be true iff date t gets assigned to student s. For the costs we generate $n \times n \times m$ Boolean atoms c_{sti} such that for each student s and date t we have n cost atoms that form a unary representation of the cost of the assignment.

B. Constraints

Two basic contraints ensure that (A) z assigns there is exactly one date to each student and (B) each assignment produces the costs given by the preferences c.

(A) $\forall s \in S, \sum_{t \in T} z_{st} = 1$.

(B) $\forall s \in S, \forall t \in T, z_{st} \implies \bigwedge_{i \leq c(s,t)} c_{sti}$.

C. Minimization

For optimization we use a cardinality encoding based on parallel counters [2] on the bound variables. Optimization is then realized incrementally calling a SAT solver with decreasing bounds.

For the local optimization round we encode m cardinality constraints, one for each individual students cost.

$\forall s \in S, \sum_{t,i} c_{sti} \leq k_1$, find smallest $k_1 \in [1, n]$

In the global optimization round we use 1 cardinality constraint over the sum of all costs.

$\sum_{s,t,i} c_{sti} \leq k_2$, find smallest $k_2 \in [m, n * m]$

III. Benchmarks

We submitted a set of problems generated from real data with 34 students with individual preferences on 35 available dates. We sampled a sequence of 10 problems from the first optimization round with different local bounds and a sequence of 10 problems from the second optimization round with fixed local bound and different global bounds.

References

[1] Iser, M.: Student date assignment by preference, encoding tool, git repository. https://git.scc.kit.edu/fv2117/terminvergabe, accessed: 2017-04-17

[2] Sinz, C.: Towards an optimal cnf encoding of boolean cardinality constraints (2005)

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

SAT-Encodings of Tree Decompositions

Max Bannach
Institute for Theoretical Computer Science
Universität zu Lübeck
Email: bannach@tcs.uni-luebeck.de

Sebastian Berndt, Thorsten Ehlers and Dirk Nowotka
Department of Computer Science
University of Kiel
Email: {seb,the,dn}@informatik.uni-kiel.de

***Abstract*—We suggest some benchmarks based on a propositional encoding of tree decompositions of graphs.**

I. Introduction

The treewidth of a graph is a fundamental property, often used e.g. in the field of fixed-parameter tractability (FPT). Intuitively speaking, it describes how tree-like a graph is. Many NP-hard problems on graphs can be solved efficiently if the input instance has bounded treewidth.

Recently, there have been two competitions seeking for efficient methods for computing the tree decomposition of graphs [1], seeking for both exact and heuristic algorithms.

Although the most successful exact approaches were based on a combination of dynamic programming and decomposition, we submit some SAT encodings of tree decompositions. Each of these formulas encodes the existence of a tree decomposition of a certain width. They were generated using Jdrasil [2]. The encoding basically describes an elimination ordering on a preprocessed version of the graph [3], from which a tree decomposition can be derived efficiently. This encoding is combined with some symmetry breaking constraints [2].

II. Graph Selection

For the generation of these formulas, we used graph instances which are publicly available at https://people.mmci.uni-saarland.de/~hdell/pace17/ex-instances-PACE2017-public-2016-12-02.tar.bz2. Out of these graphs, we selected those for which Jdrasil can compute an optimum tree decomposition within 1800 seconds. As the size of the SAT encoding is cubic with respect to the number of vertices, we only selected graphs with at most 200 nodes (after preprocessing). For each of them, we then created formulas which describe the existence of a tree decomposition of width $opt \pm 3$, where opt is the treewidth of the respective graph.

III. Benchmark Selection

We generated 448 SAT formulas from 64 graphs (192 unsatisfiable, 256 satisfiable). For each of these formulas, we ran Glucose 3.0 for 5 hours, and partitioned the formulas into three categories:

- Easy: Solvable within 20 minutes.
- Medium: Solvable with 60 minutes.
- Hard: Solvable within 300 minutes.
- Very Hard: Not solved within 5 hours.

In order to provide formulas of different hardness, we then chose 5 satisfiable and unsatisfiable formulas from each of the first 3 categories. Furthermore, we chose the 5 smallest and largest formulas from the last category, as those might be especially interesting for the parallel track. This yields overall 40 formulas.

These formulas are available at http://www.informatik.uni-kiel.de/~the/sat_benchmarks_2018_ehlers_nowotka.tar.gz.

We are curious to see the difference of solver strengths on this kind of formulas.

Acknowledgment

This work is funded by the German Federal Ministry of Education and Research, combined project 01IH15006A.

References

[1] H. Dell, C. Komusiewicz, N. Talmon, and M. Weller, “The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration,” in *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), D. Lokshtanov and N. Nishimura, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 30:1–30:12.

[2] M. Bannach, S. Berndt, and T. Ehlers, “Jdrasil: A modular library for computing tree decompositions,” in *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, ser. LIPIcs, C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, Eds., vol. 75. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, pp. 28:1–28:21. [Online]. Available: https://doi.org/10.4230/LIPIcs.SEA.2017.28

[3] J. Berg and M. Järvisalo, “Sat-based approaches to treewidth computation: An evaluation,” in *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*. IEEE Computer Society, 2014, pp. 328–335. [Online]. Available: https://doi.org/10.1109/ICTAI.2014.57

Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, volume B-2018-1 of Department of Computer Science Series of Publications B, University of Helsinki 2018.

CBMC Formulae

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

Michael Tautschnig
michael.tautschnig@qmul.ac.uk
Queen Mary University of London, UK

***Abstract*—SAT-based bounded model checking translates reachability questions over input programs into propositional satisfiability. This submission of benchmarks presents formulae generated by the C Bounded Model Checker (CBMC) on input programs from the TACAS Software Verification Competition (SV-COMP). CBMC uses MINISAT 2.2 as its default back-end, and can optionally print the problems in DIMACS format. Profiling on SV-COMP benchmarks reports that from 10 to more than 50% of CBMC's CPU time is spent in the SAT solver. Improvements in SAT solvers directly translate to reduced program analysis time.**

I. THE CBMC TOOL

CBMC [1] is a bounded model checker for C, C++, and Java programs. It supports C89, C99, most of C11 and most compiler extensions provided by GCC and Visual Studio. It also supports SystemC using Scoot. More recently, support for Java bytecode has been added.

CBMC verifies array bounds (buffer overflows), pointer safety, arithmetic exceptions and user-specified assertions. Furthermore, it can check C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding loops in the program and passing the resulting equation to a decision procedure.

CBMC has built-in support to translate bit-vector equation systems to gate level, and by default links in MINISAT 2.2 [2] as a decision procedure. Interfaces to other SAT solvers are implemented as well, as is an IPASIR interface. As an alternative, CBMC supports external SMT solvers. Finally, CBMC can also dump the formula in DIMACS format instead of running a solver. This feature has been used to generate the benchmark set.

II. SAT COMPETITION BENCHMARK

The collection of formulae in this benchmark submission were generated from input problems of the TACAS Software Verification Competition (SV-COMP) [3]. Each problem is the output of CBMC for a given software benchmark, a bit-width (32 or 64 bits), and a loop unwinding bound. To determine these configuration parameters for the purpose of generating SAT benchmarks, we provide a script `run.sh` that first downloads the SV-COMP logs, extracts unwind values and bit width for CBMC, and finally runs CBMC and gzips the CNF. The script that is used to assemble the benchmark is maintained at
https://github.com/tautschnig/sv-comp-sat.

To make the benchmarks more challenging, the unwind bound has been increased by 2. The generation time for CBMC has been limited to 30 seconds. Furthermore, any generated CNF formula is dropped as soon as MINISAT 2.2 can solve it within 10 seconds.

III. AVAILABILITY

The source of CBMC can be found at https://github.com/diffblue/cbmc. For more details, please have a look at http://www.cprover.org/cbmc/.

REFERENCES

[1] D. Kroening and M. Tautschnig, "CBMC - C bounded model checker - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 389–391. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_26

[2] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518. [Online]. Available: https://doi.org/10.1007/978-3-540-24605-3_37

[3] D. Beyer, "Software verification with validation of results - (report on SV-COMP 2017)," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10206, 2017, pp. 331–349. [Online]. Available: https://doi.org/10.1007/978-3-662-54580-5_20

Solver Index

Benchmark Index

Author Index