Proceedings of

SAT COMPETITION 2017

Solver and Benchmark Descriptions

Tomáš Balyo, Marijn J. H. Heule, and Matti Järvisalo *(editors)*

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2017-1

Helsinki 2017

PREFACE

Boolean satisfiability (SAT) solver enable tackling instances of various real world problems that seemed completely out of reach 1-2 decades ago. Besides new algorithms and better heuristics, refined implementation techniques have turned out to be vital for this success. To offer further incentives for 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 2017 (SC 2017) was organized as a satellite event of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT 2017). SC 2017 consisted of six tracks, including a (sequential) Main track together with a parallel track using the same benchmarks, and special tracks for incremental solvers, solvers specifically developed for Random SAT, a "No-Limits" track relaxing requirements on open source solvers and allowing any type of solvers—including solver portfolios—to compete, as well as an agile track for solvers specializing in fast-to-solve instances, motivated by iterative applications of SAT solvers.

As for new developments in 2017, the ranking scheme was changed from counting the number of solved instances to the PAR-2 scheme, that is, penalized average runtime, which assigns a runtime of two times the time limit (instead of a "not solved" status') for each benchmark not solved by a solver. Furthermore, every solver submission had to be accompanied by at least 20 benchmark instances, for further widening the range of benchmarks used in the competition.

Contributions to SC 2017 came in the forms for solvers (for competing in one or several of the competition tracks) and benchmark instances (for evaluation the relative performance of submitted solvers). Following the tradition put forth by SAT Challenge 2012, the rules of SC 2017 invited all contributors to submit a short, around 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.

We acknowledge the StarExec initiative, the Texas Advanced Computing Center (TACC, `http://www.tacc.utexas.edu`) at The University of Texas at Austin, and Karlsruhe Institute of Technology for computing resources for running the competition. Finally, we thank all those who contributed to SC 2017 by submitting solvers or benchmarks and the related descriptions.

Tomáš Balyo, Marijn J.H. Heule, & Matti Järvisalo
SAT Competition 2017 Organizers

Contents

Benchmark Descriptions

SOLVER DESCRIPTIONS

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Improving abcdSAT by Weighted VSIDS Scoring Schemes and Various Simplifications

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

***Abstract*—We improve abcdSAT by a new various scoring Scheme. AbcdSAT is a SAT solver built on the top of Glucose 2.3 [5]. The existing Glucose has no inprocess. Nevertheless, here we add various inprocesses to AbcdSAT. According to the rule of different tracks at the SAT Competition 2017, we develop multple versions of abcdSAT, which are submitted to agile, main, no-limit, incremental library and parallel track.**

I. Introduction

The abcdSAT solver submitted to the SAT Competition 2017 is an improved version of the previous version. Although in the old abcdSAT, we used simplification techniques such as lifting, probing, distillation, elimination, hyper binary resolution etc, they were used only as a preprocessing. In the new version, we use them as an inprocess. In this version, the main technique is a weighted VSIDS (Variable State Independent Decay Sum) scoring scheme. In addition, we add a symmetry breaking as a new preprocessing.

II. Weighted VSIDS Scoring Schemes

the most popular variable scoring scheme is Currently VSIDS, which was first proposed in the Chaff paper [1]. VSIDS consists of two phases: decay and bump. In the bump phase, the scores of variables in a learnt clause are increased by 1. When determining branching variables, variables with the largest score are always preferred. In the decay phase, all variable scores are divided by 2, the goal of which is to protect scores from overflow.

Ref. [2] proposed a normalized VSIDS (NVSIDS for short), which is an exponential moving average of occurrence frequency. The score s of a bumped variable is computed as $s' = f \cdot s + (1-f)$, where $0 < f < 1$. The scores of variables that are not bumped have to recomputed by $s' = f \cdot s$. Notice, it is impossible to implement efficiently NVSIDS, since at each conflict, it requires to update the scores of all variables. A practical version is exponential VSIDS (EVSIDS), which was first proposed by MiniSAT [3], and is identical to NVSIDS. EVSIDS re-computes only scores of bumped variables by $s' = s + g^i$ at the i-th conflict, where $g = 1/f$, thus $g > 1$.

Our weighted VSIDS (WVSIDS) is a variant of EVSIDS. Unlike EVSIDS, WVSIDS computes the score of a variable by replacing $s' = s + g^i$ with $s + w_k \cdot g^i$, where w_k is the weight value of the k-th bumped variable. In our implementation, $w_k = 1 - \frac{k}{10^4}$. If the k-th bumped variable occurs in the last, w_k is set to a constant 0.5. Suppose that the learnt clause at the i-th conflict is $x_1 \vee x_2 \vee \cdots \vee x_n$. Then the score of the variable represented by x_k is computed by

$$s' = \begin{cases} s + (1 - \frac{k}{10^4})g^i & 1 \leq k < n \\ s + 0.5g^i & k = n \end{cases}$$

III. Symmetry Breaking Preprocessor

In the new abcdSAT, we add symmetry exploitation, and use BreakID [7] as a symmetry breaking preprocessor. The main track must be able to output certificates for UNSAT instances. However, BreakID cannot produce any certificate, so abcdSAT submitted to the main track does not contain BreakID. BreakID uses Saucy [8] to output symmetry generators for an input CNF formula. For the agile track, Saucy is limited to 15 seconds to detect symmetry generators. For other tracks, it is limited to 100 seconds. In addition, the number of searches in Saucy is limited to 1000. The number of auxiliary variables introduced by a symmetry breaking is limited to 50 auxiliary variables.

IV. Various Simplifications

We use simplifications similar to Lingeling 587f [6]. Except for the no-limit track, our simplification do not use the source code of Lingeling. Because the implementation mechanism is different, the performance of our simplification is also different from that of Lingeling. Our simplification consists of lifting, probing, distillation, elimination, hyper binary resolution, equivalent literal search, unhiding redundancy, XOR Gaussian elimination etc. For the main track, the XOR Gaussian elimination is limited. Our elimination includes blocked clause elimination, forced resolution of clauses, hidden tautology elimination. The equivalent literal search is done by Tarjan's strongly connected components algorithm. Our lifting is to probe failed literals in a double depth way. Lifting and hyper binary resolution are used as an inprocessing. The cost of other simplifications are expensive, so our inprocessing gives up them.

V. LBD or Clause-sized based Learnt Clause Maintenance

In general, the Glucose-style solver uses the literal block distance (LBD) to maintain learnt clause database. Ehlers et al. [9] found that replacing LBD with clause size could improve

astonishingly their solver. Whether to use LBD or clause size is better? So far there is no clear answer. Here we mix them to maintain learnt clause database. In most cases, LBD is used to reduce learnt clause database. In rare cases, a clause-size measure is used. For example, only when the maximal size of input clauses is less than 10, and their sizes are the same, we use the clause size measure to delete learnt clauses with large size first.

VI. ABCDSAT *r17*

This *r17* version is an improved version of abcdSAT *drup*. In general, it produces DRUP proofs for UNSAT instances. However, in the tree-based search, it produces DRAT proofs in the following way. Suppose that an input formula is F, and branch literals are $x_1, x_2, \ldots, x_n$. A subproblem F' with depth n is defined as $F' = F \cup \{x_1 \wedge x_2 \wedge \cdots \wedge x_n\}$. If $I = y_1 \vee y_2 \vee \cdots \vee y_m$ is a learnt clause on F', abcdSAT *r17* outputs $\bar{z} \vee y_1 \vee \cdots \vee y_m$, where z is an auxiliary variable, which is defined as $z = x_1 \wedge x_2 \wedge \cdots \wedge x_n$. Before beginning to solving F', abcdSAT *r17* outputs the following $n+1$ RAT proofs:

$$\begin{cases} z \vee \overline{x_1} \vee \overline{x_2} \vee \cdots \vee \overline{x_n} & \\ \bar{z} \vee x_i & 1 \leq i \leq n \end{cases}$$

VII. ABCDSAT *a17*

This is submitted to the agile track. Compared to abcdSAT *r17*, abcdSAT *a17* adds a XOR Gaussian solving and a symmetry breaking preprocessor with 15 seconds limit. In addition, it removes a recursive splitting solving that is a tree-based search. The variable branching heuristic based on blocked clause decomposition is abandoned also.

VIII. ABCDSAT *i17*

This solver is submitted to the incremental library track. Compared with the previous version abcdSAT *inc*, abcdSAT *i17* adds simplifications such as lifting and hyper binary resolution as a preprocessing or inprocessing. Sometimes it divides the original problems into some subproblems. When solving each subproblem, we apply all the simplifications of abcdSAT *r17*, and the XOR Gaussian solving technique.

IX. ABCDSAT *n17*

This is the version submitted to the no-limit track. Except for the symmetry breaking preprocessing, the simplification technique used in abcdSAT *n17* is the same as one used in the last year's version abcdSAT *lim*. Here the symmetry breaking preprocessor is limited to 100 seconds. This solver divides the whole solving process into three phases. In the first phase, we use a glucose-style solver to find a solution. This phase is limited to 250000 conflicts. In the second phase, we simplify the formula generated in the first phase, using various simplification technique including XOR and cardinality constraint simplification. If the average LBD is less than 16, the third phase uses splitting and merging (reconstructing) strategy.

X. ABCDSAT *p*

This is submitted to the parallel track. We extract 32 variables with the highest occurrence frequency as pivots. Let the pivots be $p_1, p_2, \ldots, p_{32}$, and input formula F. We use the i-th thread to solve the subproblem $F \wedge p_i$. The master-thread solve the original problem. Once $F \wedge p_i$ is found to be UNSAT, we set $\neg p_i$ to a unit literal. After this, the i-th thread continues to solve $F \wedge G$, where G is learnt clauses generated by the master-thread. Learnt clauses with the size at least 2 are not shared, but unit clauses are shared among all solvers. Each solver adopts different solving strategies. These strategies can be done by different parameter configures. Parameters used in the configure include whether LBD or clause size is applied to database maintenance, whether WVSID scoring schemes, bit-encoding phase selection strategies [4], symmetry breaking preprocessor and and inprocessing techniques such as Lifting and hyper binary resolution are used. Of course, variable decay ratio, database first reduction size and random policies etc used in different solvers are also different.

REFERENCES

[1] M.W. Moskewicz and C.F. Madigan and Y. Zhao and L.T. Zhang and S. Malik: Chaff: engineering an efficient SAT solver, In *Proceedings of the 38th Design Automation Conference, (DAC 2001)*, ACM, Las Vegas, 2001, pp. 530–535.

[2] A. Biere, Adaptive restart strategies for conflict driven SAT solvers, SAT 2008, LNCS, vol. 4996, 2008, pp. 28–33.

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

[4] 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.

[5] G. Audemard and L. Simon, Glucose 2.3 in the sat 2013 competition,in *Proceedings of the SAT Competition 2013*, pp. 40–41.

[6] A. Biere: Lingeling, plingeling and treengeling. [Online]. Available: http://fmv.jku.at/lingeling/

[7] J. Devriendt and B. Bogaerts: BreakID, a symmetry breaking preprocessor for sat solvers, bitbucket.org/krr/breakid, 2015.

[8] H. Katebi, K. A. Sakallah, and I. L. Markov: Symmetry and satisfiability: An update, SAT 2010, pp. 113–127.

[9] T. Ehlers, D. Nowotka: Sequential and parallel Glucose hacks, in *Proceedings of the SAT Competition 2016*, p. 39.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

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], [3] 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. In what follows, we present the optimizations we implemented in Candy and describe two variants of the solver.

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. Gate Structure Analysis and Applications

Candy provides modules for gate extraction [4] and random simulation [5]. Detemerminized random simulation is used on gate structure extracted from SAT problems to generate conjectures about literal equivalences and backbone variables. The Candy solver variant Candy/RSAR uses these conjectures to compute and iteratively refine under-approximations of the SAT problem instance [6]. The Candy solver variant Candy/RSIL uses branching heuristics based on implicit learning [6], [7] to stimulate clause learning by violating extracted conjectures about variable equivalencies where possible, otherwise using the VSIDS branching heuristic. Candy/RSIL includes the sub-variants Candy/RSILv, with which the probability of implicit learning being used is successively halved after a fixed amount of decisions, and Candy/RSILi, with which usage budgets are assigned to each of the two implications represented by a conjecture. If budgets are assigned, such an implication can be used for implicit learning only if its budget is nonzero, and an implication's budget is decreased every time it is used for implicit learning.

Employing implicit learning has proven particularly efficient for solving miter problems with general-purpose SAT solvers [6]. Candy includes fast miter problem detection heuristics enabling implicit learning to be enabled only for SAT problem instances detected to be miter encodings within a given time limit. For a general description of miter problems, see e.g. [2].

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] Biere, H., Järvisalo, M.: Equivalence checking of HWMCC 2012 circuits. In: In Proceedings of SAT Competition 2013. vol. B-2013-1, p. 104. Department of Computer Science, University of Helsinki (2013)

[3] 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)

[4] Iser, M., Manthey, N., Sinz, C.: Recognition of nested gates in CNF formulas. In: Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. pp. 255–271 (2015)

[5] Krohm, F., Kuehlmann, A., Mets, A.: The use of random simulation in formal verification. In: 1996 International Conference on Computer Design (ICCD '96), VLSI in Computers and Processors, October 7-9, 1996, Austin, TX, USA, Proceedings. pp. 371–376. IEEE Computer Society (1996), http://dx.doi.org/10.1109/ICCD.1996.563581

[6] Kutzner, F.: Exploiting Gate-Structure to Direct CDCL Search via Variable Selection and Approximation. Diplomarbeit, Karlsruhe Institute of Technology (June 2016)

[7] Lu, F., Wang, L., Cheng, K.T., Moondanos, J., Hanna, Z.: A signal correlation guided circuit-sat solver. J. UCS 10(12), 1629–1654 (2004), https://doi.org/10.3217/jucs-010-12-1629

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

COMiniSatPS Pulsar and GHackCOMSPS

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 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 [3] and then into SWDiA5BY. Subsequently, the patches implement new techniques described in [4], [2], and [5] to generate the current form of COMiniSatPS.

COMiniSatPS is a base solver of MapleCOMSPS solver series [6], [7] that participated in SAT Competition 2016 and 2017.

II. COMiniSatPS Pulsar

Differences from the last year's COMiniSatPS the Chandrasekhar Limit [8] are as follows:

- Fast and simple Gaussian elimination [9] as one-time pre-processing. The implementation is small and straightforward, as we just need to extract XOR clauses from an input CNF, thanks to M4RI [10] which we used to perform actual Gaussian elimination (as in CryptoMiniSat 5 [11]). However, this feature is disabled for the Main competition track due to its inability to provide UNSAT proof.
- Performs part of the on-the-fly literal elimination and probing techniques introduced in GlueMiniSat [12]. However, most of these techniques are disabled for the Main competition track, as the support for UNSAT proof will increase both code complexity and proof size.
- Minor code cleanup.

III. GHackCOMSPS

This year's solver is identical to the last year's solver [5]. 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 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] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI*, 2009.
[4] C. Oh, "Between SAT and UNSAT: The fundamental difference in CDCL SAT," in *SAT*, 2015.
[5] ——, "COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS," in *SAT Competition*, 2017.
[6] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Pascal, "MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *SAT Competition*, 2016.
[7] ——, "MapleCOMSPS_LRB_VSIDS, MapleCOMSPS_LRB_VSIDS_2, MapleCOMSPS_CHB_VSIDS," in *SAT Competition*, 2017.
[8] C. Oh, "Patching MiniSat to deliver performance of modern SAT solvers," in *SAT-RACE*, 2015.

[9] M. Soos, K. Nohl, and C. Claude, "Extending SAT Solvers to Cryptographic Problems," in *SAT*, 2009.
[10] M. Albrecht and G. Bard, "The M4RI Library – Version 20121224," http://m4ri.sagemath.org, The M4RI Team, 2012, [Online; accessed 10-June-2017].
[11] M. Soos, "The CryptoMiniSat 5 set of solvers at SAT Competition 2016," in *SAT Competition*, 2016.
[12] H. Nabeshima, K. Iwanuma, and K. Inoue, "GlueMiniSat 2.2.7," in *SAT Competition*, 2013.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2017

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

***Abstract*—This paper serves as a first solver description for our new SAT solver CaDiCaL and documents the versions of our other solvers submitted to the SAT Competition 2017, which are Lingeling, its two parallel variants Treengeling and Plingeling, and our local search solver YalSAT.**

Lingeling,Plingeling, Treengeling,YalSAT

Our focus in the SAT Competition 2016 was on our new SAT solver Splatz [1]. It tried to simplify the Lingeling design, and further implemented a first inprocessing [2] version of blocked clause decomposition for SAT sweeping to detect equivalences. In the same spirit, we focus on a new solver called CaDiCaL in the SAT Competition 2017.

We submitted to the *agile*, *main*, and *no-limit* tracks of the SAT Competition 2017 the Lingeling version *bbe*, which except for some minor bug fixes in the code for picking random decisions is the same as the version entering the SAT Competition 2016 [1]. Its parallel extensions Plingeling and Treengeling submitted to the *parallel* track have the same version *bbe* accordingly. They were were marking the state-of-the-art in the parallel track of the SAT Competition 2016 [3] and have not changed at all.

The same applies to our local search solver YalSAT which also did not really change and was submitted as version *03s* to the *random* track only.

CaDiCaL

The goal of the development of CaDiCaL was to obtain an inprocessing solver [2], which is easy to understand and change, while at the same time not being much slower than other state-of-the-art solvers. Originally we also wanted to radically simplify the design and internal data structures. But that goal was only achieved partially, for instance compared to Lingeling. On the other hand, after adding, what we believe, are essential ingredients of a state-of-the-art solver, the solver *did* become competitive with other state-of-the-art solvers, for instance surpassing Lingeling on many instances. The name of the solver has its roots in "radical(ly)" and "CDCL" [4].

The main search loop interleaves inprocessing [2] and CDCL [4] search. The inprocessing part consist of three individually scheduled inprocessing methods: probing, subsumption, and (bounded) variable elimination.

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

During (failed literal) **probing** only roots of the binary implication graph are probed and binary clauses are learned through hyper binary resolution [5]. These are used to eliminate equivalent literals after decomposing the binary implication graph into strongly connected components, which is scheduled right before and after probing. Hyper binary clauses tend to be generated many and thus will only survive at most one clause reduction. We also explicitly remove duplicated binary clauses before probing.

As in Splatz we also remove subsumed learned clauses during **subsumption** in regular intervals. Due to a new much faster subsumption algorithm than in previous solvers [6] we can afford to apply subsumption checking to redundant learned clauses with small glucose level [7] too, which might otherwise be kept forever. Of course, we also perform self-subsuming resolution [6] to strengthen clauses. We also have a second propagation based subsumption check, similar to vivification [8], in each subsumption phase, which however is restricted to irredundant clauses only. For binary clauses there is a specialized transitive reduction algorithm for the binary implication graph at the end of each subsumption phase.

As in other solvers (bounded) **variable elimination** [6] is one of the most effective inprocessing techniques and is carefully scheduled as in Lingeling [9], except that we wait for an initial interval (of 1000) conflicts before being triggered. Variable elimination is interleaved with subsumption for a bounded number of rounds and as long something changes.

More precisely, the solver carefully monitors variables which occurred in removed irredundant clauses or in added (arbitrary redundant or irredundant) clauses. Removed variables trigger variable elimination attempts, while added variables trigger subsumption checks. This information is kept persistent across CDCL search and inprocessing phases and allows the solver to restrict the effort in subsumption checking and variable eliminations to those part of the formula which changed.

On the **search** side, we incorporated the idea of saving the position of the last replaced watch in large clauses [10], use VMTF instead of VSIDS as explained in [11], schedule restarts based on exponential moving averages [12] and alternate and reset the default phase of decisions, starting with picking true as initial phase of yet unassigned decision variables.

The version *sc17* of CaDiCaL, which in essence is identical to our internal version *058*, was submitted to the *agile*, *main* and *no-limit* track. In contrast to Lingeling generating proofs for the *main* track should not change how the solver works and in our experiments adds negligible overhead. Currently only Drup proofs are generated.

We have also made some minor effort to come up with parameter settings, which improve CaDiCaL on the *agile* track compared to its otherwise used default configuration. From the benchmarks of the *agile* track of the SAT Competition 2016 substantially more could be solved, if the base restart interval is increased from 6 to 400 conflicts and flipping and resetting the default decision phase is disabled. We have also submitted this "agile" version to the *agile*, *main* and *no-limit* tracks.

The solver is implemented from scratch in a modular way in C++. There is also an API interface for C, but the core library is not ready for incremental usage yet, since it is lacking assumption handling. The source code strives to be carefully documented and consists of roughly 10 000 lines of code.

License

The default license of YalSAT, Lingeling, Plingeling and Treengeling did not change in the last two 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. For the new solver CaDiCaL we use an MIT style license which is far less restrictive.

References

[1] 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.

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

[3] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT competition 2016: Recent developments," in *AAAI*. AAAI Press, 2017, pp. 5061–5063.

[4] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 131–153.

[5] M. Heule, M. Järvisalo, and A. Biere, "Revisiting hyper binary resolution," in *CPAIOR*, ser. Lecture Notes in Computer Science, vol. 7874. Springer, 2013, pp. 77–93.

[6] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT*, ser. Lecture Notes in Computer Science, vol. 3569. Springer, 2005, pp. 61–75.

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

[8] 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.

[9] A. Biere, "Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling," in *POS@SAT*, ser. EPiC Series in Computing, vol. 27. EasyChair, 2014, p. 88.

[10] I. P. Gent, "Optimal implementation of watched literals and more general techniques," *J. Artif. Intell. Res. (JAIR)*, vol. 48, pp. 231–251, 2013.

[11] 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.

[12] ——, "Evaluating CDCL restart schemes," in *Proceedings POS-15. Sixth Pragmatics of SAT workshop*, 2015, to be published.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Glucose and Syrup in the SAT'17

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

Laurent Simon
Univ. Bordeaux
LABRI
lsimon@labri.fr

***Abstract*—Glucose is a CDCL solver heavily based on Minisat, with a special focus on removing useless clauses as soon as possible, and an original restart scheme. Syrup is the parallel version of Glucose, with a lazy clauses exchanges policy. In the 2015 version of these solvers, we proposed a genuine version and an "adaptative" version of each of these solvers. The adaptative versions use a set of particular parameters and techniques to adress some outliers benchmarks that can be found in typical competitions sets.**

I. Introduction

Since 2009, Glucose enters SAT competitions/races [1], [2]... Glucose is based on minisat [3] and depends heavily on the concept of Literal Block Distance, a measure that is able to estimate the quality of learnt clauses [4]. 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 7 years old.

This year Glucose continue to adapt its strategy depending the kind of instances solved. Furthermore, we also propose a new phase saving strategy that focus on conflicting variables when restarting.

II. Adaptative solver

Selected benchmarks of the SAT competition come from many distinct domains. For example, in 2014, industrial benchmarks can be assigned in (at least) nine families like argumentation, io, crypto, diagnosis... It seems unrealistic to design one strategy that will be efficient on all the benchmarks. For instance, Glucose is known to perform better on UNSAT than on SAT instances. On the other side, it is known that long runs (without restarts) are efficient in case of some families of SAT instances.

A. Adaptation in Glucose

A number of recent solvers includes, directly or not, automatic adaptations to benchmarks. In our approach, we used our set of experimental data to classify some strategies adapted to outliers benchmarks. We took 2632 benchmarks from all the competition, and selected only 1164 interesting ones (benchmarks that needed at least one minute to be solved). We ran a set of Glucose "hacks" on this set of problems and tried to detect some simple measures that identified families of problems. We tried to consider only some "semantic" measure instead of syntactic measures on the initial formula. Glucose is run during 10,000 conflicts with its default parameters, then we may switch to some particular behavior if our indicators say so. We searched for simplicity. We identifed 4 outliers signatures.

- The number of decisions divided by the number of conflicts. This allows us to identify 123 problems over the 1164, containing bivium, hitag, gss, homer, ctl and longmult series of problems. If this number is low, we switch the reduction learnt clauses strategy by using the one proposed by Chanseok Oh [5].
- The number of conflicts without decision (when a conflict is directly reached after a conflict analysis). If this number is low, this is typically a nossum crypto problem. We identified 66 problems from the 1164 ones like that. For these problems, we used a Luby restart policy, and a much less agressive var decay. In the contrary, if this number is important, then we use the Chanseok Oh policy [5] to reduce the learnt clause database, a much less agressive var decay, and a limited randomization on the first descent after each restart [6]. In this last case, we typically identified vmpc problems.
- The number of "pure" glue clauses (glue clauses of size > 2). A large number is a typical signature of SAT_dat problems (we identified 31 of them with that, over the 1164). In this case, we observed that a much more aggressive var decay may pay.

We observed an important increasing of Glucose performances on the last competitions by using this. In the SAT competition 2014, among the 300 instances of the application category, glucose adjust its parameters on 58 instances and benefits are clears.

III. Playing with the phase

Phase saving is an essential component of a SAT solver. We refine this notion by saving in a different data-structure the phase of propagated variables that effectively participate to conflict. Then, on restart, until the next conflict, we use this polarity. The main goal is to reach a conflict as soon as possible. Combined with the online modifications of Glucose, this tecnhique reveals efficient [7].

IV. Specificities of the parallel version

We use the 24 cores available this year. Adaptive versions of Glucose is enabled on half of cores.

V. Incremental track

Glucose also entered the incremental part of the SAT-Race. In this case, it uses dedicated data-structures and techniques introduced in [8]. Unfortunately, in the incremental track, the rules were not in favor of our specialized data structure. It was not possible to know the initial variables and the variables added for the search (commonly called the "assumptions", for example variables added to simulate clauses removals). Thus, all the strategies proposed in [8] are useless here.

VI. Algorithm and Implementation details

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

References

[1] G. Audemard and L. Simon, "Glucose: a solver that predicts learnt clauses quality," *SAT Competition*, pp. 7–8, 2009.

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

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

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

[5] C. Oh, "gluh: Modified version of gluclose 2.1," *SAT COMPETITION 2013*, p. 48, 2013.

[6] J. Chen, "A bit-encoding phase selection strategy for satisfiability solvers," in *Theory and Applications of Models of Computation - 11th Annual Conference, TAMC 2014, Chennai, India, April 11-13, 2014. Proceedings*, 2014, pp. 158–167.

[7] G. Audemard and L. Simon, "Extreme Cases in SAT," in *19th International Conference on Theory and Applications of Satisfiability Testing(SAT'13)*, 2013, p. To appear.

[8] G. Audemard, J.-M. Lagniez, and L. Simon, "Improving glucose for incremental sat solving with assumptions: Application to mus extraction," in *16th International Conference on Theory and Applications of Satisfiability Testing(SAT'13)*, 2013, pp. 309–317.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Glulu

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

***Abstract*—Glulu is a SAT solver based glucose 4.1, and inherits the glucosePLE [1] that focuses upon the special strategy of pure literal elimination. Furthermore, an enhance of CDCL is considered in our solver.**

I. Introduction

Glucose [2] is an open-source CDCL-based SAT solver [3] that has achieved numerous excellent performance in past SAT Competition. In the major solving procedure of glucose based solvers, variable assignment essentially happens in two different situations: the one consists of the identification of unit clauses and the creation of the associated implications which carries out *Unit Propagation* (UP) [4]; the other one is decision assignment which picks an unassigned literal by a decision strategy, called *Variable State Independent Decaying Sum* (VSIDS) heuristic [5]. Let $p\,$(assign) as the priority evaluation of variable assignment, we can obviously know that $p\,$(UP) $> p\,$(VSIDS). We consider that pure literal elimination (PLE) [6] as the third situation of variable assignment which has a higher priority than decision assignment, but lower than UP. In general, we have $p\,$(UP) $> p\,$(PLE) $> p\,$(VSIDS).

In UP, the head of trail queue of literal assigned 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. If we denote the pair type of conflict by a pair of literals, e.g. $\{a, \neg a\}$, which are in two different clauses; denote the multi-pair type of conflict by a set of literals, e.g. $\{a, \neg a, \neg a\}$, which are in three different clauses; denote the independent pairs type of conflict by some sets of literals, e.g. $\{a, \neg a\}$, $\{b, \neg b, \neg b\}$, which are in five different clauses. Glucose based solvers usually stop searching watch list when a pair type of conflict is detected, like $\{a, \neg a\}$. However, there may be more conflicts (multi-pair type or independent pair type) occur, but not be detected under current assignment.

II. Main Techniques

With a standard structure of occurrence vector for each literal, which is recorded by the ID of clauses where the literal occurs, we introduce an algorithm that can be easily implemented to extract the pure literals within linear time in number of variables. We perform pure literal extraction for variable decision. If we succeed in extracting pure literals, we manipulate them as decision variables and give each literal a value of `1`. The extended solver carries out PLE by setting a new decision level for each of extracted pure literals, then skip the VSIDS heuristic and run the decision assignment. However, we are well aware of that pure literal extraction will keep a high execution frequency in solving process, which might reduce the efficiency of solving. A dynamic adjustment approach is introduced into glulu that can effectively utilize the features of instance and current states to set an optimal frequency. For example, assume that in unit time ($\Delta time$) the *Decision Level* increases ($\Delta DecisionLvl$) by 100, without considering the circumstances under *backtracking* [7], we should make the extraction frequency to be less than or equal to 100. Let the *Difficulty Coefficient* be (*nVars* $*$ *nClauses*), and the *Computing Coefficient* be ($\Delta propagation$ / $\Delta time$).

$$\begin{aligned} frequency &= \frac{\textit{Computing Coefficient}}{\textit{Difficulty Coefficient}} * \Delta DecisionLvl \\ &= \frac{(\Delta propagation / \Delta time)}{(nVars * nClauses)} * \Delta DecisionLvl \end{aligned}$$

In this competition, we set unit time to one CPU second ($\Delta time = 1$).

Furthermore, glulu enhance the CDCL by detecting the multi-pair type and independent pair type of conflicts in UP procedure. We also stop unit propagating (assigning), but keep searching watch list when first conflict occurs, and all detected conflicts stored in a vector of clauses. The $analyze()$ function will generate corresponding learnt clause for each conflict in this vector. Finally, we evaluate and filter some effective learnt clauses, whose LBD ≤ 2 and the number of literal contained ≤ 2, and add them into the learnt clause database.

Acknowledgment

I wish to express my gratitude to M. Koshimura and H. Fujita for valuable advices and comments.

References

[1] Z. Aolong, "Glucoseple." in *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, vol. B-2016-1 of Department of Computer Science Series of Publications, 2016, p. 42.
[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers." in *IJCAI*, vol. 9, 2009, pp. 399–404.
[3] A. Biere, M. Heule, H. van Maaren, and T. Walsh, "Conflict-driven clause learning sat solvers," *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pp. 131–153, 2009.
[4] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.
[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.
[6] J. Johannsen, "The complexity of pure literal elimination," in *SAT 2005*. Springer, 2006, pp. 89–95.
[7] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Glu_vc: Hacking Glucose by Weighted Variable State Independent Decay Sum Branching Policy

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

***Abstract*—Glu_vc is a SAT solver submitted to the hack track of the SAT Competition 2017. It updates Glucose 3.0 in the following aspects: phase selection, learnt clause database reduction and decision variable selection. Here we focus on decision variable selection, which is considered as a crucial element of CDCL (Conflict Driven, Clause Learning) solvers. So far, almost all CDCL solvers adopt the VSIDS (Variable State Independent Decay Sum) heuristic or its exponential version EVSIDS to select branching variables. We present a new variant of VSIDS to achieve higher efficiency of a SAT solver.**

I. Introduction

Remarkable achievements have been made in SAT solvers. Nevertheless, many SAT problems remain open yet. New heuristics are needed imperatively. By carefully observing the existing various heuristics, we found that there is still room for improvement on the VSIDS (Variable State Independent Decay Sum) heuristic [1].

II. Weighted Variable State Independent Decay Sum Branching Policy

CDCL (Conflict Driven, Clause Learning) solvers need to pick a variable in every branch of a search tree. A heuristic used in this process is called branching heuristic. The most popular branching heuristic is VSIDS, which was first introduced in the Chaff paper [1]. The original VSIDS in Chaff may be described briefly as follows. Variables with the largest score are preferred as branching ones. Every time a learnt clause is generated, the score of its variables is increased by 1. This process is called bump. In addition, every 256 conflicts, all variable scores are divided by 2. This is called decay. It is used to protect scores from overflow during bumping.

A smoothing version of VSIDS called normalized VSIDS (NVSIDS for short) was proposed in [2]. It is an exponential moving average of occurrence frequency. The score s of a bumped variable is computed as $s' = f \cdot s + (1 - f)$, where f is a decay factor with $0 < f < 1$. The scores of variables that are not bumped have to recomputed by $s' = f \cdot s$. Clearly, this version cannot be efficiently implemented, since at each conflict, it requires to update the score of all variables. A version identical to NVSIDS is exponential VSIDS (EVSIDS), which was first proposed by MiniSAT [3]. It re-computes only scores of bumped variables by $s' = s + g^i$ at the i-th conflict, where $g = 1/f$, thus $g > 1$.

Our WVSIDS (Weighted VSIDS) is a variant of EVSIDS. The score computation between them is different. WVSIDS transforms $s' = s + g^i$ to $s + w_k \cdot g^i$, where w_k is the weight value of the k-th bumped variable. In our implementation, $w_k = 1 - \frac{k}{10^4}$. If the k-th bumped variable occurs in the last, w_k is set to a constant 0.5. Suppose that the learnt clause at the i-th conflict is $x_1 \vee x_2 \vee \cdots \vee x_n$. Then the score of the variable denoted by x_k is computed by

$$s' = \begin{cases} s + (1 - \frac{k}{10^4})g^i & 1 \leq k < n \\ s + 0.5g^i & k = n \end{cases}$$

III. Other various heuristics

Like the previous hack version Glue_alt [6], this hack version uses also the bit-encoding phase selection strategy given in [5]. This strategy has been shown to be beneficial to solve satisfiable instances. But it has a negative impact on other instances. To tradeoff its advantages and disadvantages, this strategy is restricted to be used in the case where the number of conflicts is less than 10^5. Compared to Glue_alt, this hack version reduces the application of the bit-encoding phase selection strategy.

Glucose uses the literal block distance (LBD) to maintain learnt clause database and decide when to restart. Ehlers et al. [4] improved astonishingly Glucose by replacing LBD with clause size. Here we use still LBD to maintain learnt clause database in most cases, but use a clause-size measure in a few cases, e.g., the maximal size of input clauses is less than 10. That is, we sort learnt clauses sometimes according to their size, and do them sometimes according to their LBD.

References

[1] M.W. Moskewicz and C.F. Madigan and Y. Zhao and L.T. Zhang and S. Malik: Chaff: engineering an efficient SAT solver, In *Proceedings of the 38th Design Automation Conference, (DAC 2001)*, ACM, Las Vegas, 2001, pp. 530–535.

[2] A. Biere, Adaptive restart strategies for conflict driven SAT solvers, SAT 2008, LNCS, vol. 4996, 2008, pp. 28–33.

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

[4] T. Ehlers, D. Nowotka: Sequential and parallel Glucose hacks, in *Proceedings of the SAT Competition 2016*, p. 39.

[5] 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, vol. 8402, 2014, pp. 158–167.

[6] J.C. Chen: Glue_alt: hacking Glucose by applying at-least-one recently used rule to learnt clause management, in *Proceedings of the SAT Competition 2016*, pp. 12–13.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS

Jia Hui Liang*‡, Chanseok Oh†‡, Vijay Ganesh*, Krzysztof Czarnecki*, Pascal Poupart*
* 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].

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

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]. However, we changed VSIDS slightly for our MapleCOMSPS solvers: during conflict analysis, if decision levels of variables are greater (resp., less) than the computed backtrack level, such variables get more (resp., less) bumps to activity scores. MapleCOMSPS_CHB_VSIDS also disables on-the-fly failed literal detection [9].

A. MapleCOMSPS_LRB_VSIDS

The difference from COMiniSatPS is that it regularly switches between LRB and VSIDS, in the almost same manner that COMiniSatPS switches between the no-restart phase and the Glucose-restart phase [10], [11]. However, unlike the original COMiniSatPS, we allocate equal amounts of time to LRB and VSIDS. The solver employs Luby restarts and Glucose-style restarts for LRB and VSIDS, respectively. LRB's locality extension (i.e., decaying unplayed arms) is disabled.

When comparing to the last year's MapleCOMSPS_LRB [12], the only difference is the updated base

solver COMiniSatPS. (Unfortunately, however, due to our porting error in upgrading the base solver, the on-the-fly probing techniques of COMiniSatPS does not work properly.)

B. MapleCOMSPS_LRB_VSIDS_2

Unlike the MapleCOMSPS_LRB_VSIDS version above, this solver does not upgrade the base COMiniSatPS. This solver differs from the last year's MapleCOMSPS_LRB [12] only in the following three minor ways:

1) Implements the one-line hack of tb_glucose [13]: after learning a clause from each conflict, the solver bumps activity scores of the variables in the learnt clause. The bump amount is inversely proportional to the clause LBD.
2) The intervals of LRB-VSIDS cycles double (i.e., 100% increase) each time a cycle completes (up from 10% increase).
3) The interval of the first LRB-VSIDS cycle is of length 20,000 conflicts (up from 200 conflicts).

C. MapleCOMSPS_CHB_VSIDS

The difference from COMiniSatPS is that it regularly switches between CHB and VSIDS, in the similar manner as MapleCOMSPS_LRB_VSIDS. The solver employs Glucose-style restarts for both CHB and VSIDS.

When comparing to the last year's MapleCOMSPS_CHB [12], the only difference is the updated base solver COMiniSatPS.

V. SAT Competition 2017 Specifics

1) The three solvers are participating in the Main and No-Limits tracks.
2) We used the same LRB and CHB parameters presented in our papers [1], [2], which have not changed from the last year's versions.

VI. 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.

Acknowledgment

We thank Gilles Audemard and Laurent Simon, the authors of Glucose.

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 *SAT Competition*, 2017.

[9] M. Heule, M. Järvisalo, and A. Biere, "Efficient CNF Simplification based on Binary Implication Graphs," in *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'11, 2011.

[10] C. Oh, "Between SAT and UNSAT: The fundamental difference in CDCL SAT," in *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'15, 2015.

[11] ——, "Patching MiniSat to Deliver Performance of Modern SAT Solvers," in *SAT Race*, 2015.

[12] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Pascal, "MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *SAT Competition*, 2016.

[13] S. Moon and I. Mary, "CHBR_glucose," in *SAT Competition*, 2016.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width in SAT Competition 2017

Fan Xiao[1], Mao Luo[1], Chu-Min Li[2], Felip Manyà[3], Zhipeng Lü[1]
[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@u-picardie.fr, corresponding author
[3]Artificial Intelligence Research Institute (IIIA-CSIC), Barcelona, Spain
felip@iiia.csic.es

***Abstract*—This document describes the SAT solvers MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width, which participate in the SAT Competition 2017.**

1. Introduction

Maple_LCM and MapleLRB_LCM are CDCL SAT solvers obtained by implementing the recent learnt clause minimization approach of [1] on top of the CDCL solvers MapleCOMSPS_DRUP and MapleCOMSPS_LRB_DRUP [2], [3], respectively. They correspond to solvers Maple+ and MapleLRB+ in Tables 1 and 2 of [1]. LCM stands for Learnt Clause Minimization. Maple_LCM_Dist is Maple_LCM with a new branching heuristic called *Distance*. MapleLRB_LCMoccRestart is MapleLRB_LCM with a new restart strategy based on the number of conflicts in which every variable is involved. Glucose-3.0+width is Glucose-3.0 [4] using a new learnt clause quality measure called *Width* in the clause management, instead of using LBD. The new techniques are described in the subsequent sections.

2. Learnt Clause Minimization

The learnt clause minimization based on unit propagation (UP) can be described as follows. For each learnt clause $C = l_1 \vee l_2 \vee \cdots \vee l_k$ with small LBD (i.e. in the subset $CORE$ or $TIER2$ in MapleCOMSPS_DRUP or MapleCOMSPS_LRB_DRUP), if UP($F \cup \{\neg l_1, \neg l_2, \ldots, \neg l_i\}$) derives an empty clause, let $\{\neg l'_1, \neg l'_2, \ldots, \neg l'_{i'}\}$ be the subset of literals in $\{\neg l_1, \neg l_2, \ldots, \neg l_i\}$ that are responsible of the conflict. We replace C by $\{l'_1 \vee l'_2 \vee \ldots \vee l'_{i'}\}$. Actually, unit propagation is implemented incrementally for efficiency reasons: $\neg l_i$ is propagated in the formula returned by UP($F \cup \{\neg l_1, \neg l_2, \ldots, \neg l_{i-1}\}$) after checking that l_i or $\neg l_i$ is not asserted.

The above minimization cannot be executed at every restart, because it is costly. Following [1], in Maple_LCM, Maple_LCM_Dist, MapleLRB_LCM and MapleLRB_LCMoccRestart, it is executed before a restart if the number of clauses learnt since the last 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. Furthermore, each learnt clause is minimized at most once. See [1] for more details, and the references therein for related work.

3. *Distance*: A New Branching Heuristic Based on Implication Graph

Standard decision heuristics such as VSIDS and LRB [2] in CDCL solvers select a branching variable based on the behaviour of this variable in the past. These heuristics are not accurate at the beginning of search, because few things have happened with that variable. Maple_LCM_Dist uses, instead of VSIDS or LRB, a new heuristic at the beginning of search.

Let $distAct[v]$ denote a score of a variable v, which is initialised to 0. When Maple_LCM_Dist derives an empty clause C, it constructs the complete implication graph involving all the branching decisions responsible of the conflict. Let V be the set of variables occurring in the graph. Maple_LCM_Dist calls Algorithm 1 to update $distAct(v)$ for each $v \in V$, where $dist[v]$ denotes the number of variables in the longest path from v to a variable of C; $maxDistance = \max_{v \in V} dist[v]$; inc is a global variable, whose initial value is 1 in the first call of Algorithm 1 and whose final value will be its initial value in the next call of Algorithm 1; and $dist\_Decay = 0.6$.

The intuition behind the new heuristic is that a variable v with higher $distAct[v]$ probably has a bigger capacity to imply other variables. Note, however, that $distFactor$ is an integer array (instead of real number in double precision) in the current implementation, meaning that its real value does not increase monotonically because of integer overflow (for integers bigger than 2^{31}), introducing a diversification in the score $distAct[v]$.

Algorithm 1: updateDistanceScore(V)

Input: V: set of variables occurring in the complete implication graph of a conflict

```
begin
    for d := 1 to maxDistance do
        distFactor[d] ← inc;
        inc ← inc/dist_Decay;
    Let V = {v_1, v_2, ..., v_|V|} in the inverse order of being assigned a truth value;
    for i := 1 to |V| do
        distAct[v_i] ←
        distAct[v_i] + dist[v_i] * distFactor[dist[v_i]];
end
```

Since it is time-consuming to construct the complete implication graph, Maple_LCM_Dist computes $distAct[v]$ and branches on the variable v that has maximum $distAct[v]$ only for the first 50000 conflicts. It behaves like Maple_LCM after the first 50000 conflicts.

4. OccRestart: A New Restart Mechanism

The restart mechanism was first introduced in Satz to prevent heavy-tailed phenomena [5]. Usual restart mechanisms in modern CDCL solvers include glucose-style restart, Luby restart and geometric length restart. MapleLRB_LCM follows MapleCOMSPS_LRB_DRUP and interleaves Glucose-style restart phases with Luby restart phases. MapleLRB_LCMoccRestart is MapleLRB_LCM but replaces the Luby restart phases by the OccRestart restart phases described as follows:

Let V_i be the set of variables involved in at least one conflict in the i^{th} glucose-style restart, and let $nbOcc[v]$ be the number of conflicts in which variable v is involved. MapleLRB_LCMoccRestart computes $meanNbOcc_i = \sum_{v \in V_i} nbOcc[v]/|V_i|$ and then $r_i = \max_{v \in V_i} nbOcc[v]/meanNbOcc_i$, which is the ratio of the maximum number of conflicts in which a variable is involved to the mean number of conflicts in which a variable is involved. Let $meanRatio = \sum_{i=1}^{n} r_i/n$, where n is the total number of glucose-style restarts so far. In each non glucose-style restart, at every conflict, let V be the set of variables involved in at least one conflict so far, MapleLRB_LCMoccRestart computes $meanNbOcc = \sum_{v \in V} nbOcc[v]/|V|$ and then $r = \max_{v \in V} nbOcc[v]/meanNbOcc$, and stops the search as soon as $r > meanRatio$.

In a word, MapleLRB_LCMoccRestart computes the average ratio $meanRatio$ of glucose-style restarts and uses $meanRatio$ to control the non glucose-style restarts. The intuition is the following: If a variable v is involved in many more conflicts than other variables, then the search probably falls in a heavy-tail and should be restarted to branch on v before other variables. MapleLRB_LCMoccRestart interleaves glucose-style restart phase and OccRestart phase during 2500 seconds before switching to pure glucose-style restarts.

5. Width: A New Learnt Clause Quality Measure

A learnt clause in a CDCL solver is obtained by a sequence of resolution steps. We call the length of the largest resolvent in these steps the *width* of the learnt clause. While Glucose-3.0 uses LBD to measure the usefulness of a learnt clause C, Glucose-3.0+Width uses the width of C to measure its usefulness and removes the half of learnt clauses with greater width in every clause database reduction, breaking ties first by the LBD measure, and then by the length of the clauses. All other things remain unchanged.

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] 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.

[2] 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.

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

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

[5] C. P. Gomes, B. Selman, and K. Henry, "Boosting combinatorial search through randomization," in *Proceedings of AAAI*, 1998.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

bs_glucose, tch_glucose

Seongsoo Moon
Graduate School of Information Science and Technology,
The University of Tokyo, Japan

Inaba Mary
Graduate School of Information Science and Technology,
The University of Tokyo, Japan

***Abstract*—We briefly introduce our solver bs_glucose and tch_glucoses submitted to SAT-Competition 2017. All solvers are implemented on glucose 3.0. A bs_glucose includes a hybrid branching heuristic model using random forest method. Solvers tch_glucoses break ties in branching heuristics.**

I. Introduction

Decision heuristic is one of the most important elements in modern SAT solvers. The most prominent method is VSIDS[1]. There were lots of attempts to surpass VSIDS [2] [3] [4], but VSIDS is still most popular decision heuristic because of its robustness.

Recently new branching heuristic CHB[5] was provised and it showed significant improvements for some benchmarks. However, single branching heuristic cannot cover all instances.

Several studies have attempted to integrate different strategies in a sequential solver. For example, the multi-solver SATzilla [6] builds an empirical model using machine learning techniques and chooses an adequate solver for each problem based on its feature values. A deep learning approach has also been attempted [7]. This approach converts a CNF into a grayscale image and builds a classifier using a convolutional neural network. However, these methods require several state-of-the-art solvers. They achieve higher performance than single solvers, but are unsuitable as base solvers for other solvers because they already include several base solvers.

Noting that a slow Luby restart policy is superior to rapid restart policies for SAT problems, [8] designed a hybrid restart strategy. Such approaches could be combined with other approaches.

In our program, we implemented CHB and applied *tie-breaking* method for VSIDS and CHB. A tch_glucose uses a simple policy for selection between VSIDS and CHB. A bs_glucose includes eight branching heuristics of CHB, VSIDS and their variant using *tie-breaking*.

II. Details of random forest model for bs_glucose

We trained our model on 1400 benchmarks of SAT Competitions from 2014 to 2016 in both the Crafted and Application Tracks. The random forest model was constructed from 13 features: vars (number of variables), clauses (number of clauses), vars/clauses, and variable-clause graph features (mean, variation coefficient, min, max, and entropy for both variable and clause node degrees). These features can be extracted within a short time without requiring a specific algorithm. The calculation time of 1400 benchmarks was under 1,000 s. This is important because when solving a formula using a SAT solver, our model must extract features as a preprocessing step. Therefore, time-consuming feature extraction is undesirable. We implemented *Tie-breaking of VSIDS* (TBVSIDS) and *Tie-breaking of CHB* (TBCHB) by applying *tie-breaking* method for VSIDS and CHB, respectively. Total of eight diffrent branching heuristics were used as classes, which are VSIDS, CHB, TBVSIDSs, and TBCHBs. Details of our model and branching heuristics will be discussed later in [9], currently in press.

Trained model is applied in a solver and work as a preprocessing. When a input formula is given to a solver, 13 features are extracted and a branching heuristic is chosen for a solver. We submitted the best model from our experiments. A model only use 4 features (vars, variable nodes: variation coefficient, entropy, clause nodes: min). Maximum depth of decision trees is set to 5.

III. tch_glucoses

We submit 3 tch_glucoses for SAT Competition 2017.

A tch_glucose1 apply *tie-breaking* in VSIDS. To break ties in VSIDS, we update VSIDS scores after we obtain learned clauses. After a clause is obtained, we add 1 / (LBD of a clause $\times r$). A value r is set to 1000 to as an attempt to reduce additional scores obtained through *tie-breaking*s to some extent.

A tch_glucose2 apply *tie-breaking* in CHB. To adjust scale, we update a maximum value MV added by CHB and reset on each restart. After a clause is obtained, we add MV / (LBD of a clause $\times r$). A value r is set to 1000 for the same reason in tch_glucose1.

A tch_glucose3 is a hybrid one of tch_glucose1 and tch_glcose2. It utilizes tch_glucose1 / tch_glucose2 when the number of variables in an SAT instance is over / under 15000, respectively. This version is prepared for Glucose-Hack.

References

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

[2] N. Dershowitz, Z. Hanna, and A. Nadel, "A clause-based heuristic for sat solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 46–60.

[3] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," *Discrete Applied Mathematics*, vol. 155, no. 12, pp. 1549–1561, 2007.

[4] H. Zhang, "Sato: An efficient prepositional prover," in *International Conference on Automated Deduction*. Springer, 1997, pp. 272–275.

[5] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Exponential recency weighted average branching heuristic for sat solvers." in *AAAI*, 2016, pp. 3434–3440.
[6] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla-07: the design and analysis of an algorithm portfolio for sat," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2007, pp. 712–727.
[7] A. Loreggia, Y. Malitsky, H. Samulowitz, and V. A. Saraswat, "Deep learning for algorithm portfolios." in *AAAI*, 2016, pp. 1280–1286.
[8] C. Oh, "Between sat and unsat: the fundamental difference in cdcl sat," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2015, pp. 307–323.
[9] S. Moon and M. Inaba, "Boost sat solver with hybrid branching heuristic," in *Tenth Annual Symposium on Combinatorial Search*, 2017, p. in press.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

painless-maplecomsps

Ludovic Le Frioux*†, Souheib Baarir*†‡, Julien Sopena†, Fabrice Kordon†
*LRDE, EPITA, Kremlin-Bicêctre, France
†Sorbonne Universités, UMPC Univ Paris 06, UMR 7606, LIP6, Paris, France
CNRS, UMR 7606, LIP6, Paris, France
‡Université Paris Nanterre, France

***Abstract*—This paper describes the `painless-maplecomsps` solver submitted to the parallel track of the SAT Competition in 2017. It is a parallel solver instantiated with PArallel INstantiabLE Sat Solver (`PaInleSS`) framework and using `MapleCOMSPS` as core sequential solver.**

I. Introduction

`painless-maplecomsps` is a parallel SAT solver built by instantiating components of the `PaInleSS` parallel framework. It is a Portfolio based solver implementing a diversification strategy, fine control of learnt clause exchanges, and using `MapleCOMSPS` [1] as a core sequential solver.

Section II gives an overview on `PaInleSS` framework. Section III details the implementation of `painless-maplecomsps` using `PaInleSS` and `MapleCOMSPS`.

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 are depicted in Fig. 1, and form the global architecture of `PaInleSS`.

A. Sequential Engines

The core element that we consider in our framework is a sequential SAT solver (called *sequential engine*). 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, bump activities, interrupt, add classes*, etc.

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

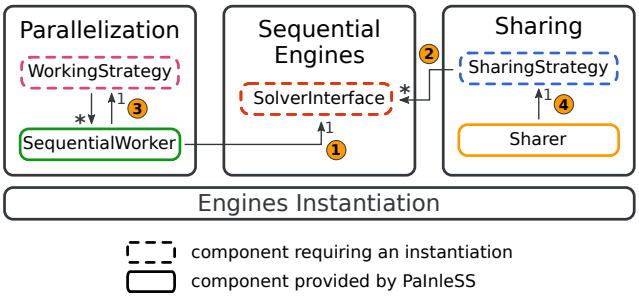

Fig. 1. Architecture of `PaInleSS`.

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 [2]. 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.

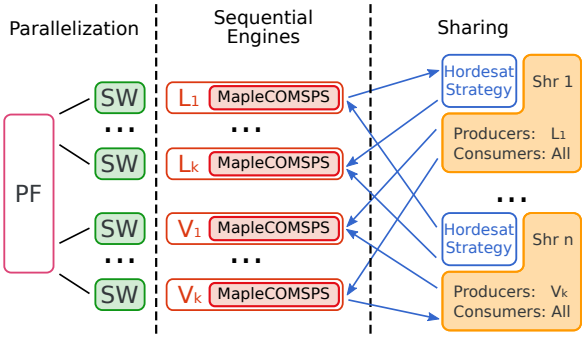

Fig. 2. Architecture of `painless-maplecomsps`.

III. PAINLESS-MAPLECOMSPS

This section describes the overall behaviour of our competing instantiation, namely `painless-maplecomsps`. Its architecture is highlighted in Fig. 2.

A. Sequential Engines: *MapleCOMSPS*

`MapleCOMSPS` is the winner sequential solver of the main track of the SAT Competition 2016. It is based on `MiniSat` [3], and uses as decision heuristics the classical Variable State Independent Decaying Sum (VSIDS) [4], and newly defined Learning Rate Branching (LRB) one [5]. These heuristics are used in one-shot phases: first LRB, then VSIDS.

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. The export is parametrized according to a Literal Block Distance (LBD) [6] threshold.

B. Parallelization: Portfolio and Diversification

`painless-maplecomsps` 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 [7]. 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.

C. Sharing: Controlling the Flow of Shared Clauses

In `painless-maplecomsps`, the sharing strategy is inspired from the one used by [7]. 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 1.5 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 (¡ 1200) are dispatched.

ACKNOWLEDGMENT

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

REFERENCES

[1] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maplecomsps, maplecomsps lrb, maplecomsps chb," *SAT COMPETITION 2016*, p. 52.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

CBPeneLoPe2017 and CCSPeneLoPe2017 at the SAT Competition 2017

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: PeneLoPe2017 and CCSPeneLoPe2017. PeneLoPe2017 and CCSPeneLoPe2017 are based on the parallel SAT solver PeneLoPe. We slightly changed the last versions by adding a memory management function and changing some parameters.**

I. PeneLoPe2017

PeneLoPe2017 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, and SAT Competition 2016. PeneLoPe2017 implements *community branching* [6], a diversification [4] technique using community structure of SAT instances [1]. The 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, PeneLoPe2017 implements *community-based learnt clause sharing (CLCS)*. The CLCS conducts the community detection algorithm on the VIG of target SAT instance. Then, this method 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 the community branching, the 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 the community branching.

The differences between PeneLoPe2017 and previous versions are as follows.

- Memory management by reducing the number of threads
- Changes of some parameters
- Refactoring for some parts of the program

Our new memory management detects memory usage explosion during the search, and then reduces the number of running threads.

II. CCSPeneLoPe2017

CCSPeneLoPe2017 is a parallel portfolio solver based on PeneLoPe. The features of CCSPeneLoPe2017 are as follows.

- Conflict history-based branching heuristic (CHB) [5] for some workers
- CLCS prioritizing high VSIDS or CHB scores

In CCSPeneLoPe2017, some workers use this heuristic with different sets of parameters. For the 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". The 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] 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.

[5] 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.

[6] 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 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Riss 7

Norbert Manthey

***Abstract*—The sequential SAT solver RISS combines the 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. Most recent changes focus mainly in incremental solving.**

I. INTRODUCTION

The CDCL solver RISS is a highly configurable SAT solver based on MINISAT [1] and GLUCOSE 2.2 [2], [3]. Besides many search algorithm extensions, RISS is equipped with the preprocessor COPROCESSOR [4]. The solvers supports, emitting DRAT proofs for many techniques, enumerating more than a single model, and incremental solving.

This document mentions only the differences to RISS 6 that has been submitted to SAT Competition 2016. Most differences are motivated by axiom pinpointing with SAT [5] that involves incremental SAT solving with many calls to the solver, where each single call has many assumption literals, and only a few conflicts.

II. MODIFICATIONS OF THE SEARCH

RISS 6 used automatic configuration based on formula features. This capability has been dropped.

III. INCREMENTAL SAT SOLVING WITH RISS

RISS 6 used simplification that have been applied lazily during calls to the solver. These simplifications have been disabled.

A. Persistent Incremental Solving

From a generic point of view, the idea of persistent incremental calls is to follow the following rules:

1) do not clear the trail after stopping search
2) in case new clauses are added, integrate them after making sure there are two literals that are not falsified
3) when being called with a new set of assumptions, extract the common prefix, and re-use this part of the trail

This way, parts of the search can be saved, because the trail has not to be revisited each time. Together with the IPASIR interface, benefits of this technique are not very strong, as new assumptions have to be passed for each call to the solver each time. When RISS is tightly integrated into an application, e.g. SATPIN [5], the effect is much more visible.

B. Applying Techniques Partially

If assumptions are given, RISS performs restarts only to the decision level where the last assumption was used as decision. This saves redundant work, as the first part of the trail in the solver stays fixed with assumptions (assuming the order of assumptions is not switched during search).

The original implementation of MINISAT 2.2 executed search under assumptions until the decision literal that should be assigned next is falsified already. In RISS we implement *Early Refined Cores* [6], which starts conflict analysis already if a conflict is found on a decision level that was reached based on assumptions only. This will trigger conflict analysis faster, but might produce longer conflict clauses.

C. Reverse Core Refinement

After a set of assumptions A has been found to be infeasible, a conflict clause C is produced. Reverse core refinement [6] tries to produce a smaller clause $D \subseteq C$ by re-running a call to incremental search and using $\neg C$ as assumptions – most importantly in the reverse order of the initial assumptions.

IV. SAT COMPETITION SPECIFICS

RISS and COPROCESSOR are implemented in C++. RISS is submitted to all sequential tracks, except the *random SAT* track.

V. AVAILABILITY

All tools in the solver collection are available for research. The source of RISS will be made publicly available under the LGPL v2 license at https://github.com/nmanthey/riss-solver.

ACKNOWLEDGMENT

The author would like to thank the developers of GLUCOSE 2.2 and MINISAT 2.2.

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] N. Manthey, R. Peñaloza, and S. Rudolph, "Efficient axiom pinpointing in EL using SAT technology," in *Proceedings of the 29th International Workshop on Description Logics, (DL 2016)*, ser. CEUR Workshop Proceedings, M. Lenzerini and R. Peñaloza, Eds., vol. 1577. CEUR-WS.org, 2016. [Online]. Available: http://ceur-ws.org/Vol-1577/paper_33.pdf

[6] N. Manthey, "Refining unsatisfiable cores in incremental SAT solving," TU Dresden, Tech. Rep., September 2015.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

satUZK-seq and satUZK-ddc: Solver description

Alexander van der Grinten
University of Cologne

***Abstract*—We describe the current version of our sequential CDCL solver and a new distributed SAT solver based on it. This distributed SAT solver uses a parallel lookahead technique to partition the search space until there are at least as many subproblems as parallel computation resources. Those subproblems are solved by CDCL.**

I. Sequential solvers

We submitted four configurations of our sequential *satUZK-seq* solver to the SAT Competition 2017. The preceding version of satUZK-seq is described in [1]. As the solver makes use of C++14 features it could not be compiled on the StarExec cluster that hosts the competition and therefore had to be submitted to the No Limit Track.

The four configurations differ in their clause reduction and restart strategies and in whether they apply preprocessing before CDCL search. The configurations `sm` and `sg` apply preprocessing to the input formula while `m` and `g` do not run any CNF simplification algorithms. Configurations `sm` and `m` use a MiniSat-like exponential clause reduction heuristic [2] as well as Luby restarts while `sg` and `g` use a Glucose-like aggressive clause reduction strategy [3] and LBD-based restarts.

None of those configurations uses any novel algorithms. Instead they rely on well-known heuristics and implementation techniques. Our primary motivation to submit them to the SAT Competition is to measure their performance relative to the performance of our parallel *satUZK-ddc* solver.

II. Distributed divide-and-conquer solver

Our main contribution to the SAT Competition 2017 is the submission of our *satUZK-ddc* configuration to the Parallel Track of the competition.

The solver is based on a divide-and-conquer algorithm. Specifically the solver constructs a divide-and-conquer tree that partitions the search space. Each vertex of this tree corresponds to a CNF formula that is derived from the input formula by fixing additional variables [1]. We attach groups of threads to the vertices of this tree. They operate on leaves of this tree by one of the following two steps:

- **Divide-step** The threads perform a parallel lookahead on the leaf vertex. This results in the expansion of the leaf vertex by multiple child vertices. Additionally the lookahead returns a set of failed literals. We fix those failed literals by extending the formula with conflict clauses.
- **Conquer-step** The threads perform a CDCL search to directly solve the leaf vertex. If the vertex could not be solved within 10000 conflicts we interrupt the CDCL search and perform a divide-step instead.

[1]This is sometimes called a *guiding path* approach in the literature

After either a divide-step or a conquer-step has completed, the participating threads have to be routed through the divide-and-conquer tree until they reach a new unsolved leaf. The algorithm terminates when all vertices have been solved by one of those two steps or if a solution is found.

This strategy is related to other search space partitioning approaches like [4] and [5]. However compared to [4] we use a more complex lookahead-based branching strategy. In contrast to parallel Cube&Conquer [5] solvers, we do not use work stealing but we route threads through the divide-and-conquer tree instead.

Notable features of our distributed solver include:

a) Routing: Our routing algorithm simply routes threads to their parent vertex until they reach a vertex with unsolved child vertices. If such a vertex is found, the threads are distributed evenly to all unsolved child vertices. This routing strategy ensures that a maximal number of learned clauses is still relevant after routing.

b) Incremental solving: We use assumptions (as in MiniSat) to fix variables at the current vertex. We adapted our clause reduction heuristics to this situation by maintaining two LBD scores per clause: The *global LBD* score counts each assumption as a different decision level while the *local LBD* does not count assumptions at all. When a thread is routed to a different vertex we reset the local LBD of each clause to its global LBD. As usual, the local LBD can then be decreased when the clause participates in a conflict. This idea of not counting assumptions is due to [6], however the reset mechanism is not implemented in other solvers to the best of the author's knowledge.

c) Load balancing: The lookahead of different variables is done in parallel. We employ a load balancer to ensure that no thread idles for extended periods of time during lookahead. Our load balancing algorithm uses a simple *dimension exchange* method on hypercubes.

d) Lookahead scores: Our lookahead heuristic evaluates variables based on the number of literals that are fixed by unit propagation. In contrast to more complex lookahead heuristics this information can be computed quickly even for large application instances. In order to further reduce computation time we preselect the 10000 variables that occur most frequently in the CNF formula and only compute lookahead scores for these variables.

e) Clause sharing: Our solver implements a limited form of clause sharing where each thread shares learned unit clauses with all other threads. We use clause freezing [7] to avoid interrupting active CDCL searches. Unit clauses are unfrozen during clause database reduction.

f) Diversification: Previous research has shown that portfolio techniques are very effective in SAT solving. We take advantage of this fact by varying the solver configuration among different threads. Half of our threads use MiniSat-like clause reduction and restart heuristics (as in our satUZK-seq `m` configuration) while the other half uses Glucose-like heuristics (as our satUZK-seq `g` configuration).

g) Inprocessing: When a thread is idle (e.g. because it is waiting for other threads to complete their divide-steps) it performs inprocessing. We apply subsumption as our only inprocessing technique.

III. Preprocessing

Our preprocessing heuristics are similar to that of SatELite but also include more modern extensions. We apply multiple preprocessing passes to the input formula. Each pass runs unhiding [8], blocked clause elimination (BCE) [9], subsumption, resolution subsumption and bounded variable elimination (BVE) [10] to the formula. Similar to SatELite, our preprocessing halts if BCE, subsumption, resolution subsumption and BVE reach a fix point.

During unhiding we perform five randomized traversals of the binary implication graph. Subsumption and resolution subsumption skip binary clauses as we expect unhiding to be able to approximate those techniques on binary clauses.

As our preprocessing algorithms can be quite expensive on large formulas, we expect them to perform significantly better in the Parallel Track (that is run on a more powerful execution environment) than in the Main Track.

References

[1] A. van der Grinten, A. Wotzlaw, and E. Speckenmeyer, "satUZK: Solver description," in *Proceedings of SAT Competition 2014*, 2014, p. 75.

[2] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, ser. SAT '03. Springer, 2004, pp. 502–518.

[3] 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.

[4] G. Audemard, J.-M. Lagniez, N. Szczepanski, and S. Tabary, "An Adaptive Parallel SAT solver," in *Principles and Practice of Constraint Programming*, ser. CP '16. Springer, 2016, pp. 30–48.

[5] 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*. Springer, HVC '12, pp. 50–65.

[6] G. Audemard, J.-M. Lagniez, and L. Simon, "Improving glucose for incremental sat solving with assumptions: Application to mus extraction," in *Theory and Applications of Satisfiability Testing - SAT 2013*. Springer, 2013, pp. 309–317.

[7] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs, "On freezing and reactivating learnt clauses," in *Theory and Applications of Satisfiability Testing - SAT 2011*, ser. SAT '11. Springer, 2011, pp. 188–200.

[8] 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.

[9] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '10. Springer, pp. 129–144.

[10] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing SAT 2015*. Springer, 2005, pp. 61–75.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

ScaLoPe : A scalable parallel SAT Solver based on PENELOPE

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

***Abstract*—In this paper, we provide a short description of our solver ScaLoPe built on the top of PENELOPE which aim to improve scalability of portfolio-based SAT solvers. Concretely this solver uses a particular collaboration scheme by organizing threads in teams. Each team behaves like a classical portfolio and clause sharing among threads of the same team is performed as usual. However, only unit clauses are shared among teams.**

I. INTRODUCTION

Portfolio-based SAT solving is the current dominating approach for parallel SAT solving. It simply consists of running different solvers (or the same solver with different configurations) on the same instance until one of them find a solution. Sharing learnt clauses among threads can help improve the runtime of the resolution process. However, sharing too many learnt clauses can degrade the performances of the solver. Therefore it is important for threads to select and share only useful clauses. Unfortunately, determining which clauses are useful to a particular threads is not an easy task. Many SAT solvers heuristically choose clauses to share taking into account some of their characteristics such as the *size*, the LBD [1], PSM [2] etc.

As mentioned in [3] dealing with clauses imported from other threads leads to additional problems such as : imported clauses that are already subsumed by those already present in the database, the increasing number of clauses that have to be managed by each thread, a long time of uselessness of some imported clauses before becoming suddenly useful. To cope with these problems, [3] proposed a clause sharing strategy relying on both import and export policies where some clauses are frozen [2] and reactivated later. This strategy has been implemented in the solver PENELOPE [4] (built on the top of MINISAT [5] and MANYSAT [6]) on which our solver ScaLoPe is based. One problem remains: since it is difficult to detect all irrelevant clauses, threads continue to receive a lot of clauses coming from others that can be redundant or subsumed; even if some of them are frozen. This issue can be amplified especially with a great number of threads and therefore limit the scalability of the solver.

The aim of our solver ScaLoPe is to increase scalability while keeping the strength of learnt clauses sharing by implementing a different collaboration model.

II. COLLABORATION MODEL

In the most of the state-of-the-art parallel SAT solvers, clauses judged relevant by a particular thread are shared with all the other threads. So if some of these clauses are not useful at all, they will contribute to slowdown the entire solver due to large learnt clauses databases full of useless clauses. To cope with this issue, our approach organizes threads in teams as depicted in figure 1 and authorizes strong sharing (represented by line connections among threads $T_0, \cdots, T_3$ in the figure) only with threads that are in the same team. By strong sharing we mean sharing clauses with larger size, LBD, PSM etc. Furthermore, collaboration is allowed among teams where only unit clauses are exchanged: this is represented by dashed lines among teams in the figure. This model has some advantages: for instance, if each thread (among 12 as in figure 1) shares only one useless clause with the others, it will result after exchanges with the classical approach to 11 useless new clauses per thread whereas with our scheme each thread will receive only 3 useless new clauses to add in its learnt clauses database. This can improve memory usage and prevent threads from being rapidly overwhelmed by clauses coming from others.

In that manner, useless clauses shared by a thread can only slowdown threads in its team since the learnt clauses database of each thread do not increase very quickly: this is due to the fact that each of them receives clauses only from a limited number of other threads (only from threads in its team).

With this scheme, we can conserve the power of clauses sharing while increasing the scalability of the portfolio solver.

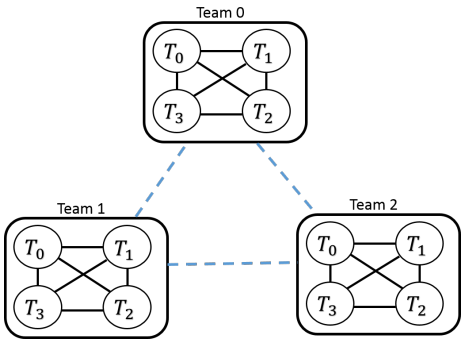

Fig. 1. Collaboration model

III. IMPLEMENTATION DETAILS

The collaboration among teams was implemented as follow: Only 0^{th} thread of each team is allowed to communicate with other teams. It is responsible for exporting units of its own

team and importing units coming from other teams. Once units are imported by thread number 0, the others can retrieve them during classical exchanges within the team. Teams are homogeneous in term of configurations apart for the random seed which is different for every thread regardless to the team it belong to. This random seed is deduced from the global ID which is computed as follow:

$$globalThreadID = teamSize \times teamID + localThreadId$$

where $teamSize$ is the number of thread per team and $localThreadID$ is the Id of the thread in its team. Figure 2 shows an overview of some properties in the configuration file (`configuration.ini`) of PENELOPE in which we added *teamSize* to control the number of threads per team and tuned all threads to use only LBD restart policy [7] (with sometimes different size of the LBD queue). See [4] for more details about these properties.

```
[global]
;specify the number of cores that will be used for solver
ncores = 36/48;
;specify the number of threads per team
teamSize=12
[default]
;choose between the different restart policies
restartPolicy = avgLBD
;choose between the different export policies
exportPolicy = lbd;
;this set the initial number of conflict allowed before the first reduceDB
initialNbConflictBeforeReduce = 500;
;the maximum value allowed for clauses in order to not be deleted in the
    reduceDB process
maxLBD = 10;
;the incremental factor for the limit in number of conflict before the
    reduceDB
nbConflictBeforeReduceIncrement = 100;
;the initialization policy for the phase
initPhasePolicy = random;
;the lenght of the historic we compute the average of for the avglbd
    restart
historicLength = 100;
[solver0]
initPhasePolicy = true;
historicLength = 50;
[solver1]
nbConflictBeforeReduceIncrement = 30
[solver2]
historicLength = 150;
[solver3]
nbConflictBeforeReduceIncrement = 30
[solver4]
maxFreeze=4
maxLBDExchange = 2
[solver5]
initPhasePolicy = false
[solver6]
initialNbConflictBeforeReduce = 150
nbConflictBeforeReduceIncrement = 10
maxFreeze = 5
[solver7]
initPhasePolicy = true
historicLength = 50;
[solver8]
historicLength = 150;
[solver9]
restartPolicy = avgLBD
maxLBD = 2
[solver10]
restartPolicy = avgLBD
maxLBD = 7
[solver11]
restartPolicy = avgLBD
historicLength = 200;
```

Fig. 2. Partial view of `configuration.ini` file

IV. SUBMITTED SOLVERS

We submitted two solvers in the SAT Competition 2017 : `scaLoPe36` and `scaLoPe48` . `scaLoPe48` uses 48 threads with 12 threads per team while `scaLoPe36` uses 36 threads with 12 threads per team.

V. ACKNOWLEDGEMENT

We would like to thanks authors of MINISAT , MANYSAT and PENELOPE for the great work they have done.

REFERENCES

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

[2] 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.

[3] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette, "Revisiting clause exchange in parallel sat solving," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2012, pp. 200–213.

[4] ——, "Penelope in sat competition 2014," *SAT COMPETITION 2014*, p. 58.

[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.

[7] G. Audemard and L. Simon, "Refining restarts strategies for sat and unsat," in *Principles and Practice of Constraint Programming*. Springer, 2012, pp. 118–126.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Score$_2$SAT Solver Description

Shaowei Cai*, Chuan Luo†
*Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
†Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{shaoweicai.cs, chuanluosaber}@gmail.com

***Abstract*—This document describes local search SAT solver *Score$_2$SAT*.**

I. Introduction

The second level score, which was first introduced in CCASat [1], has proved effective in solving random SAT with long clauses (with length greater than 3). After its introduction, the second level scoring functions have been used in various successful solvers for SAT, including *DCCASat*, *WalkSATlm*, *CScoreSAT* and Sattime2014r.

The second level scoring functions depend on the notion of satisfaction degree, which captures the degree of a clause being satisfied, as defined below.

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

Formally, the second level scoring functions are defined as follows.

Definition 2: For a variable x,

(1) its second level $make$, denoted by $make_2(x)$, is the number of 1-satisfied clauses that would become 2-satisfied by flipping x;

(2)its second level $break$, denoted by $break_2(x)$, is the number of 2-satisfied clauses that would become 1-satisfied by flipping x;

(3) its second level $score$, denoted by $score_2(x)$, equals $make_2(x) - break_2(x)$.

The *DCCASat* solver [2] uses $score_2$, and it shows efficiency in solving random k-SAT instances at phase transition. The The *WalkSATlm* solver improves the original *WalkSAT* algorithm [3] by incorporating the $make_2$ property [4]. An improved version of *WalkSATlm* also use an efficient implementation for computing $break$, $make$ and $make_2$ [5]. The *WalkSATlm* solver has been shown to be efficient and robust on large scale random k-SAT instances with various k and the ratios near the phase transition.

The *Score$_2$SAT* solver is a combination of *DCCASat* and *WalkSATlm*. The main procedures of *Score$_2$SAT* can be described as follows. For solving an SAT instance, *Score$_2$SAT* first decides the type of this instance. Then based on the properties of the instance, *Score$_2$SAT* calls either *DCCASat* or *WalkSATlm* to solve the instance. For *Score$_2$SAT*, we adopt the version of *DCCASat* in [2], and the improved version of *WalkSATlm* in [5].

II. Special Algorithms, Data Structures, and Other Features

The notation r denotes the clause-to-variable ratio of an SAT instance. The procedures of *Score$_2$SAT* are described as follows. For random 3-SAT with $r \leq 4.24$, *WalkSATlm* is called; for random 3-SAT with $r > 4.24$, *DCCASat* is called. For random 4-SAT with $r \leq 9.35$, *WalkSATlm* is called; for random 4-SAT with $r > 9.35$, *DCCASat* is called. For random 5-SAT with $r \leq 20.1$, *WalkSATlm* is called; for random 5-SAT with $r > 20.1$, *DCCASat* is called. For random 6-SAT with $r \leq 41.2$, *WalkSATlm* is called; for random 6-SAT with $r > 41.2$, *DCCASat* is called. For random 7-SAT with $r \leq 80$, *WalkSATlm* is called; for random 7-SAT with $r > 80$, *DCCASat* is called.

III. Implementation details

The *Score$_2$SAT* solver is implemented in programming language C/C++, and is developed on the basis of *DCCASat* and *WalkSATlm*.

IV. SAT Competition 2017 Specifics

The *Score$_2$SAT* solver is submitted to Random SAT track, SAT Competition 2017. The command line of *Score$_2$SAT* is described as follows.

```
./Score2SAT <instance> <seed>
```

References

[1] S. Cai and K. Su, "Local search for boolean satisfiability with configuration checking and subscore," *Artificial Intelligence*, vol. 204, pp. 75–98, 2013.

[2] C. Luo, S. Cai, W. Wu, and K. Su, "Double configuration checking in stochastic local search for satisfiability," in *Proc. of AAAI 2014*, 2014, pp. 2703–2709.

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

[4] S. Cai, K. Su, and C. Luo, "Improving WalkSAT for random k-satisfiability problem with k > 3," in *Proc. of AAAI 2013*, 2013, pp. 145–151.

[5] S. Cai, C. Luo, and K. Su, "Improving WalkSAT by effective tie-breaking and efficient implementation," *The Computer Journal*, vol. 58, no. 11, pp. 2864–2875, 2015.

BENCHMARK DESCRIPTIONS

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Generating the Uniform Random Benchmarks

Marijn J. H. Heule

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

***Abstract*—The uniform random k-SAT instances described here, together with the hard satisfiable random instances described on pages XXX of this compilation, constitute the benchmark set of the Random Track of SAT Competition 2017.**

INTRO

This description explains how the benchmarks were created of the uniform random categories of the SAT Competition 2017. These categories consists of uniform random k-SAT instances with $k \in 3, 4, 5, 6, 7$ – Boolean formulas for which all clauses have length k. For each k the same number of benchmarks have been generated.

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	$r = 4.267$ $n \in \{5000, 5200, \ldots, 12800\}$	$r \in \{3.86, 3.88, \ldots, 4.24\}$ $n = 1,000,000$
5	$r = 21.117$ $n \in \{200, 210, \ldots, 590\}$	$r \in \{16, 16.2, \ldots, 19.8\}$ $n = 250,000$
7	$r = 87.79$ $n \in \{90, 92, \ldots, 168\}$	$r \in \{55, 56, \ldots, 74\}$ $n = 50,000$

[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.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Deep Bound Hardware Model Checking Instances, Quadratic Propagations Benchmarks and Reencoded Factorization Problems Submitted to the SAT Competition 2017

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

***Abstract*—In this benchmark description we describe our three set of benchmarks submitted to the SAT Competition 2016. The first contains bounded model checking problems from the deep bound track of the hardware model checking competition. The second crafted set of benchmarks has the sole purpose to show that the standard watch list implementation has a quadratic corner case. As third set of benchmarks we submitted factoring problems of products of medium sized primes, which seem to be hard for standard SAT solvers, but become trivial if the solution is reencoded back into the CNF by flipping literals appropriately.**

Deep Bound Hardware Model Checking Instances

The Hardware Model Checking Competition (HWMCC) [1] has a deep bound track, in which only safety model checking benchmarks are considered, and which remained unsolved in the main track. Model checkers participating in this track are supposed to print bounds k, as soon they were able to show that a bad state violating the given safety property can not be reached in k steps from an initial state. This track was inspired by the need to run bounded model checking for big industrial models which are too hard to be solved completely. In this setting a model checker is superior to another one if it goes deeper, i.e., it reaches a higher bound k.

For this benchmark set HWMCC15DEEP we used the 135 model checking problems of the deep bound track of the HWMCC'15, see http://fmv.jku.at/hwmcc15, which consists of 123 industrial and 12 academic instances (the latter all from the BEEM family). The deep bound track is dominated by plain bounded model checkers, which after some optimizations, unroll the circuit, and then use a SAT solver directly. In this track our own BLIMC model checker, which is based on Lingeling [2] and runs hors concurs in the competition, performs best. It uses SAT preprocessing to simplify the transition relation once before copying it and running an incremental SAT check for each new bound following [3].

In order to generate non-incremental problems instead, we took the deepest bound k reached by BLIMC on these benchmarks and unrolled the model up to the bounds $k-2$, $k-1$, k, $k+1$, $k+2$ and all the powers of two 2^i with $2^i < k-2$, as well the bound 0 which checks whether an initial state is bad.

The unrolling process is based on functional substitution [4] as implemented in the AIGUNROLL tool, which comes with the AIGER distribution (see http://fmv.jku.at/aiger). However, if the bound reached by BLIMC in the given time limit of one hour is 100 or more, then the benchmark was not included. This removed 24 models. One of them was as BEEM model. The remaining 109 models are further split in two sets. The first set contains 55 "small" models, where the original sequential model (before unrolling) has less than 100 000 AND gates. The rest makes up the set of 54 "big" models.

The resulting AIGs after unrolling the models are translated into CNF with AIGTOCNF, which yields 433 small CNFs and 330 big CNFs, after removing trivial ones, where the unrolled AIG is constant false. There are 134 non-interesting benchmarks in the small set and 67 non-interesting benchmarks in the big set which can all be solved by MINISAT [5] in less than a minute. There are additional 97 small and 82 big benchmarks which are solved by all five test solvers in 5000 seconds (LINGELING, GLUCOSE, MINISAT, MAPLECOMSPSLRB from the SAT Competition 2016 and CADICAL). At the end we obtain 202 interesting small benchmarks and 181 interesting big benchmarks. Note, that we kept 33 small and 21 big CNFs, which were not solved by any test solvers.

Crafted Quadratic Propagations Benchmarks

The standard implementation of watch lists in MINISAT and its descendants is suboptimal and in some situation might lead to a quadratic overhead. This observation occurs in an JAIR article by Ian Gent [6] from 2013. For some benchmarks from the SAT Competition 2016, we have seen severe slowdown in propagation speed for an earlier version of our new SAT solver CADICAL, which were due to exactly the observation made by Ian Gent. The solution we implemented, which was suggested in this article, is to save the position of the replaced literal and start searching from that position instead of from the beginning of the clause, the next time a watch in that clause has to be replaced.

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

The article does not really have convincing experimental evidence that this scheme is beneficial in practice and also failed to provide benchmarks, where this quadratic behavior can be observed. The purpose of our BCPSQR benchmark set is to provide exactly such parameterized set of crafted instances, where propagation in MINISAT is quadratic. The basic idea is to have a very long clause, say $x_1 \vee x_2 \vee \ldots \vee x_{1000}$ and force the solver to assign and thus watch all the literals in turn, e.g., assign $x_1 = 0$, $x_2 = 0$, ..., $x_{1000} = 0$, in this order.

However, since MINISAT sorts clauses to remove duplicate literals in increasing variable index order, and then, due to how the binary heap for decision ordering works, decides on largest variable indices first, actually except for the first decision which is always the first variable, this is hard to achieve, e.g., the default decision assignments in MINISAT would be $x_1 = 0$, $x_{1000} = 0$, $x_{999} = 0$, ..., $x_2 = 0$ but the clause $x_1 \vee x_{1000} \vee x_{999} \ldots \vee x_2$, which would trigger the intended bad behavior, becomes $x_1 \vee x_2 \vee \ldots \vee x_{1000}$ after sorting during parsing.

This can be addressed by adding the following binary clauses $(\bar{x}_2 \vee x_{1999})$, $(\bar{x}_3 \vee x_{1998})$, ..., $(\bar{x}_{1000} \vee x_{1001})$ and would result in quadratic propagation.

However, the whole input also has to go through variable elimination untouched. To achieve that, the long clause of n variables is replaced by m copies, adding one new variable (positively too) each time. Then appropriate binary clauses are added which turn these new variables into the output of NAND gates over the old variables.

Further, those new variables are restricted by a size m parity constraint, encoded with 2^{m-1} clauses, which has the all zero assignment as solution. For the submitted benchmarks we use $m = 4, 5, 6$, which all make variable elimination ineffective.

Finally, a binary clauses $(\bar{a} \vee c)$ as above is split into

$$(\bar{a} \vee \bar{b}_1 \vee \bar{b}_2 \vee \bar{b}_3) \quad \begin{array}{ll} (a \vee b_1) & (b_1 \vee c) \\ (a \vee b_2) & (b_2 \vee c) \\ (a \vee b_2) & (b_3 \vee c) \end{array} \quad (\bar{b}_1 \vee \bar{b}_2 \vee \bar{b}_3 \vee \bar{c})$$

with new variables b_1, b_2, b_3 ordered after a and before c. Adding less than 3 variables per implication would allow variable elimination to eliminate all additional variables.

These problems are satisfied by MINISAT without producing any conflict, but require substantial propagation overhead starting with $n > 100\,000$.

REENCODED FACTORIZATION PROBLEMS

In MINISAT the heuristic for assigning a decision variable the first time before it was ever assigned during propagation is to assign it to false. Indeed, it seems that solving many real-world instances benefits from this choice of phase decision heuristic, even though there are cases where the opposite is much better, e.g., for miters between correct and incorrect large multipliers [7]. The organizers of the SAT Competition 2014 selected certain benchmarks, which are very hard unless the simple phase heuristic of MINISAT is used, as for instance discussed in [8]. In essence there is the danger of using artificially trivial benchmarks.

In order to stress this point we generated CNFs which model factoring the product of primes. For each benchmark we picked random primes with 9 to 11 decimal digits, computed their product and generated an SMT benchmark in bit-vector logic, which forces the output of a multiplier to the concrete product. One has to make sure that the bit-length of the output is big enough. The inputs are zero-extended, different from one and ordered. Then the SMT benchmark is bit-blasted by BOOLECTOR [9] into an AIG and translated to CNF with AIGTOCNF.

This procedure generates medium to hard satisfiable instances for today's SAT solver, and, of course, can be made arbitrarily hard, by increasing the number of digits. However, since we know the primes, we can easily construct an assignment to the input bits which then satisfies the CNF after unit-propagation. With this knowledge and using the assumption mode of PICOSAT [10] we generated a complete satisfying assignment for each generated CNF. This assignment is then used to flip literals in the CNF as follows. If a variable is assigned to true the variable is flipped (replaced by its negation). The resulting CNF is trivially satisfiable by assigning all variables to false and thus trivial to solve by say MINISAT.

Beside generating the original hard instance, and then the all zero instance, we repeated the procedure, but flip variables which are assigned to false. Now the instance becomes trivially satisfiable by assigning all variables to true. It turns out these instances are also easy to solve for solvers which detect this situation, e.g., LINGELING, or assign to true first, as phase decision heuristic, but they seem to be as hard as the original instance for solvers which assign to false first, e.g., MINISAT.

REFERENCES

[1] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, and K. Heljanko, "Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 135–172, 2014 (published 2016).

[2] 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.

[3] S. Kupferschmid, M. D. T. Lewis, T. Schubert, and B. Becker, "Incremental preprocessing methods for use in BMC," *Formal Methods in System Design*, vol. 39, no. 2, pp. 185–204, 2011.

[4] T. Jussila and A. Biere, "Compressing BMC encodings with QBF," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 3, pp. 45–56, 2007.

[5] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 502–518.

[6] I. P. Gent, "Optimal implementation of watched literals and more general techniques," *J. Artif. Intell. Res. (JAIR)*, vol. 48, pp. 231–251, 2013.

[7] A. Biere, "Collection of Combinational Arithmetic Miters Submitted to 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. 65–66.

[8] 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.

[9] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0 system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 53–58, 2014 (published 2015).

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

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Crafted Combinational Equivalence Instances

William Klieber
Carnegie Mellon University
Pittsburgh, PA, USA
{wklieber}@cmu.edu

I. Background

In formal verification, one often wants to check whether two propositional formulas or combinational logic circuits are equivalent. Given two propositional formulas ϕ_1 and ϕ_2, we can check equivalence by taking the exclusive-or (XOR) of these formulas, denoted "$\phi_1 \oplus \phi_2$", and querying whether this formula is satisfiable. We present a generator for creating problems of this nature, available at:
https://www.cs.cmu.edu/%7Ewklieber/bench-sat2017/benchgen.py

II. Overview

This benchmark suite contains two classes of benchmarks: satisfiable and unsatisfiable. The unsatisfiable formulas are created as follows. First, a propositional formula ϕ is randomly created, as detailed in section III. This formula is then refactored, as detailed in section IV, to produce a logically equivalent but syntactically very different formula ϕ'. Finally, we take the XOR of ϕ and ϕ' and encode it in DIMACS.

For the satisfiable instances, we proceed as follows. We first generate a formula ϕ, mostly in the same way as for an unsatisfiable instance, but with a slight complication explained in section V. Then, before refactoring it, we first slightly modify ("tickle") it to produce a formula that is guaranteed to not be logically equivalent, as described in section VI. Let ϕ' be the refactored tickled formula. So, $\phi \oplus \phi'$ is satisfiable. However, it turns out that modern SAT solvers can easily solve even large instances of this form. So, to make things more challenging, we instead create k formulas $(\phi_1, \ldots, \phi_k)$ for some small k. (In particular, we use $k = 12$.) We randomly generate a single assignment A and then tickle and refactor the original formulas to produce modified formulas $(\phi'_1, \ldots, \phi'_k)$ in such a way that $\phi_i|_A \neq \phi'_i|_A$ for $i \in \{1, ..., k\}$, where "$\psi|_A$" denotes the truth value that ψ evaluates to under A. The final formula is the conjunction:

$$(\phi_1 \oplus \phi'_1) \wedge \cdots \wedge (\phi_k \oplus \phi'_k)$$

III. Generation Of Random Formulas

We generate formulas with the following BNF grammar:

$$\begin{aligned}
\mathit{AndFmla} &::= \texttt{AND(}\mathit{XorFmla}\texttt{,}\ \mathit{XorFmla}\texttt{)} \mid \mathit{Lit} \\
\mathit{OrFmla} &::= \texttt{OR(}\mathit{XorFmla}\texttt{,}\ \mathit{XorFmla}\texttt{)} \mid \mathit{Lit} \\
\mathit{XorFmla} &::= \texttt{XOR(}\mathit{AndFmla}\texttt{,}\ \mathit{OrFmla}\texttt{)} \\
&\quad \mid \texttt{XOR(}\mathit{OrFmla}\texttt{,}\ \mathit{AndFmla}\texttt{)} \mid \mathit{Lit} \\
\mathit{Lit} &::= \mathit{Var} \mid \neg \mathit{Var}
\end{aligned}$$

In other words: Each gate has two children. Each child of an AND or OR gate is either an XOR gate or a literal. Each XOR gate has one AND child and one OR child, unless one or both of these children are literals instead.

The formula, viewed as tree, is a balanced tree. In a subtree with 8 or fewer leafs, each leaf has a distinct variable. This avoids degenerate cases such as $\texttt{AND}(x, \neg x)$ and helps avoid producing such subformulas during refactoring (section IV).

IV. Refactoring Of Formulas

First all the gates of the formula are converted to *if-then-else* (ITE) gates, as follows:

$$\begin{aligned}
\texttt{AND}(x, y) &= \texttt{ITE}(x, y, \texttt{false}) \\
\texttt{OR}(x, y) &= \texttt{ITE}(x, \texttt{true}, y) \\
\texttt{XOR}(x, y) &= \texttt{ITE}(x, \neg y, y)
\end{aligned}$$

Negations are pushed inwards so that they occur only directly in front of variables. Then, some subformulas of the form

$$\texttt{ITE}(\texttt{ITE}(\mathit{sel}, t_{in}, f_{in}), t_{out}, f_{out})$$

are refactored to the following logically equivalent form:

$$\texttt{ITE}(\mathit{sel},\ \texttt{ITE}(t_{in}, t_{out}, f_{out}), \texttt{ITE}(f_{in}, t_{out}, f_{out}))$$

Specifically, we define a recursive procedure $\mathit{Refactor}$ as follows:

$\mathit{Refactor}\big(\texttt{ITE}(\texttt{ITE}(\mathit{sel}, t_{in}, f_{in}), t_{out}, f_{out})\big)$ returns either

$$\mathit{Refactor}\big(\texttt{ITE}(\mathit{sel}, \mathit{Refactor}(\texttt{ITE}(t_{in}, t_{out}, f_{out})), \mathit{Refactor}(\texttt{ITE}(f_{in}, t_{out}, f_{out}))\big)$$

or

$$\texttt{ITE}\big(\mathit{Refactor}(\texttt{ITE}(\mathit{sel}, \texttt{true}, \texttt{false})), \mathit{Refactor}(\texttt{ITE}(t_{in}, t_{out}, f_{out})), \mathit{Refactor}(\texttt{ITE}(f_{in}, t_{out}, f_{out}))\big)$$

with the choice of these two options determined partially at random. As the base case, $\mathit{Refactor}\big(\texttt{ITE}(\mathit{lit}, t_{out}, f_{out})\big) = \texttt{ITE}(\mathit{lit}, t_{out}, f_{out})$, where lit is a literal.

V. Pretickling Of Formulas

When creating satisfiable instances, there is an additional step in randomly generating a formula. After the steps in section III are completed, the formula is *pretickled* to produce a semantically different (i.e., not logically equivalent) formula that is suitable for input to the Tickle function described

in section VI. The purpose of this is to ensure that, for a predetermined randomly generated assignment A, $Tickle$ can flip the truth value of the formula ϕ by flipping the polarity of one of its leafs. Let L_{flip} be the leaf whose polarity we will flip. Let P be the path from the root of ϕ to L_{flip}. Then, for each gate G of the form $\texttt{AND}(x, y)$, $\texttt{AND}(y, x)$, $\texttt{OR}(y, x)$, or $\texttt{OR}(x, y)$, where G and x are on the path P (and therefore y is not), we must ensure that y does not control the output of G. If G is an $\texttt{AND}$ gate and $y|_A = \texttt{false}$, then $Pretickle$ replaces y with its negation. Likewise, if G is an $\texttt{OR}$ gate and $y|_A = \texttt{true}$, $Pretickle$ replaces y with its negation.

VI. Tickling Of Formulas

Given an assignment A and a formula ϕ produced by $Pretickle_A$, the $Tickle_A$ function flips the polarity of a single leaf node (literal) of ϕ such that $\phi|_A \neq Tickle_A(\phi)|_A$. As in section V, let L_{flip} be the leaf whose polarity we will flip, and let P be the path from the root of ϕ to L_{flip}. We define the $Tickle_A$ function as follows, where $op \in \{\texttt{AND}, \texttt{OR}, \texttt{XOR}\}$:

$$Tickle_A(op(x, y)) = \begin{cases} op(Tickle_A(x), y) & \text{if } x \text{ is on } P \\ op(x, Tickle_A(y)) & \text{if } y \text{ is on } P \end{cases}$$

$$Tickle_A(lit) = \neg lit \quad \text{for a literal } lit$$

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

The LLBMC Family of Benchmarks

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

***Abstract*—This family contains benchmarks from software bounded model checking generated by the tool LLBMC[1].**

I. Introduction

LLBMC (the low-level bounded model checker) is a static software analysis tool for finding bugs in C (and, to some extent, in C++) programs. It is mainly intended for checking low-level system code and is based on the technique of Bounded Model Checking.

The files in this benchmark have been generated from small sample programs and a more realistic embedded C code device driver by extracting the SAT formulas from proof attempts. In LLBMC, checks on programs are converted to SMT formulas in the logic of bit-vectors and arrays (QF_ABV), which are in turn converted to SAT.

The benchmarks have been generated using the SMT solver STP[2].

QPR-VERIFY[3] is an extension of LLBMC, which is intended for commercial use, and has been used to generate the SAT instances "BMP280 Driver".

II. Benchmarks

A. BMP280 Driver

This benchmark is based on the Bosch Sensortec MEMS pressure sensor driver [4], consisting of two C files (`bmp280.c`, version V2.0.5, and `bmp280_support.c`, version V1.0.6) with 1963 lines of code combined.

Each function of the device driver is considered as an entry point for checking for violations of run-time properties such as integer overflows, or array index out-of-bounds.

The benchmark contains nine files corresponding to nine functions of the device drivers that have been checked by QPR-VERIFY in that way.

B. Array Average

The Array-Average subcategory contains instances generated by checking equivalence of two functions (`avg_l` and `avg_i`) computing the average of the first N array elements with LLBMC, where the parameter N is chosen to be between 2 and 10.

[1]http://llbmc.org
[2]https://github.com/stp/stp
[3]http://qpr-technologies.de
[4]https://github.com/BoschSensortec/BMP280_driver

The main function (`__llbmc_main`) runs both implementations on a non-deterministically initialized array and checks that they always return equal values.

Instances where N is a power of two should be simpler to solve due to simplifications performed by LLBMC and the SMT solver. All instances are unsatisfiable.

```
extern int t[N]; // size parameter N set by define

static int avg_l(int *a, int n) {
   long s = 0, i;

   for (i = 0; i < n; i++)
      s += a[i];
   if (s < 0 && (s % n != 0)) {
      // avoid round-towards-zero
      return (s / n) - 1;
   } else {
      return s / n;
   }
}

static int avg_i(int *a, int n) {
   int i, p = 0, s = 0;

   for (i = 0; i < n; i++) {
      s += a[i] / n;
      p += a[i] % n;
      if (p >= n) { p -= n; s++; }
      else if (p < 0) { p += n; s--; }
   }

   return s;
}

void __llbmc_main(void) {
   int a_l, a_i;

   a_l = avg_l(t, N);
   a_i = avg_i(t, N);

   __llbmc_assert(a_l == a_i);
}
```

C. Modmul

Modmul is a simple performance test for LLBMC to check that if $x = z \cdot n$, then $x \equiv 0 \bmod n$ (for machine integers x, z, and n). The different instances of this subcategory have been generated by adapting the SMT formula generated by LLBMC manually for different bit-widths b of the integers ($b \in \{8, 10, 12, 14, 16, 32\}$). Formulas for smaller bit-width should be easier to solve, all instances are unsatisfiable.

```
void __llbmc_main(int x, int n, int z) {
   __llbmc_assert(x != (long)z*n || x % n == 0);
}
```

D. Division-by-5

The Division-by-5 subcategory checks the equivalence of two different ways to compute $x/5$ for a positive, 32-bit

machine integer x. The instance is unsatisfiable.

```
void __llbmc_main(unsigned int x) {
  unsigned int a, b, c, d;

  a = x / 5;

  b = (x >> 3) + (x >> 4);
  b += (b >> 4);
  b += (b >> 8);
  b += (b >> 16);
  c = x - 5*b;
  d = b + ((13*c) >> 6);

  __llbmc_assert(a == d);
}
```

E. Fermat-3

The file from this subcategory tests a restricted version of Fermat's Last Theorem for an exponent n of 3, i.e. that there are no solutions of the equation $x^3 + y^3 = z^3$ for suitable machine integers x, y, and z. The instance is unsatisfiable.

```
void fermat(int32_t x, int32_t y, int32_t z) {
   __llbmc_assume(x > 0 && y > 0);

   __llbmc_assume(!__llbmc_ovfl_mul_int32_t(x, x));
   __llbmc_assume(!__llbmc_ovfl_mul_int32_t(x*x, x));
   __llbmc_assume(!__llbmc_ovfl_mul_int32_t(y, y));
   __llbmc_assume(!__llbmc_ovfl_mul_int32_t(y*y, y));
   __llbmc_assume(!__llbmc_ovfl_mul_int32_t(z, z));
   __llbmc_assume(!__llbmc_ovfl_mul_int32_t(z*z, z));
   __llbmc_assume(!__llbmc_ovfl_add_int32_t(x*x*x, y*y*y));

   __llbmc_assert(x*x*x + y*y*y != z*z*z);
}
```

F. Magic

This instance checks an assertion in a C program containing shifts and multiplication. The origin of the program is unknown, it might be related to some optimized computations involving remainder in signed division by 100.

```
#define lshrl(a,b) (int64_t)(((uint64_t)a) >> ((uint64_t)b))
#define lshri(a,b) (int32_t)(((uint32_t)a) >> ((uint32_t)b))
#define trim(n) (int32_t)((uint32_t)((uint64_t)n))
#define zext(n) (int64_t)((uint64_t)((uint32_t)n))

void __llbmc_main(int32_t input) {
   int64_t magic = (int64_t)1374389535;
   char result;

   int32_t e = trim(lshrl(magic * ((int64_t)input), 32));
   int64_t a = (e >> 5) + lshri(e, 31);

   result = (char)(lshrl(-zext(input - (100 * a)), 31) & 1);

   if (trim(lshrl(100 * ((int64_t)a), 32)) != 0) {
      __llbmc_assert(result == 0);
   }
}
```

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Polynomial Multiplication

Chu-Min Li[1], Fan Xiao[2], Mao Luo[2], Felip Manyà[3], Zhipeng Lü[2]
[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:

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

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:

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

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,

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

3. Set of Submitted Instances

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

- $n=5$, $t \in \{6, 7, 14, 15, 16\}$
- $n=6$, $t \in \{6, 7, 25, 26, 27, 28, 29, 30\}$
- $n=7$, $t \in \{7, 8, 42, 43, 44, 45, 46\}$

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 E7-8837 of 2.66GHz and 10GB of memory under Linux. The cutoff time is 5000 seconds.

TABLE 1. INFORMATION ABOUT THE GENERATED INSTANCES.

Instance	#Variables	#Clauses	Satisfiability	Time
pol5-06	2139	9000	UNSAT	75.9447
pol5-07	2608	10800	unkown	timeout
pol5-14	5891	23400	unkown	timeout
pol5-15	6360	25200	SAT	387.488
pol5-16	6829	27000	SAT	90.1176
pol6-06	3702	15840	UNSAT	59.2317
pol6-07	4517	19008	unkown	timeout
pol6-25	19187	76032	unkown	timeout
pol6-26	20002	79200	SAT	172.943
pol6-27	20817	82368	SAT	2534.04
pol6-28	21632	85536	SAT	391.544
pol6-29	22447	88704	SAT	811.887
pol6-30	23262	91872	SAT	425.223
pol7-07	7196	30576	unkown	timeout
pol7-08	8497	35672	unkown	timeout
pol7-42	52731	208936	unkown	timeout
pol7-43	54032	214032	unkown	timeout
pol7-44	55333	219128	SAT	3058.01
pol7-45	56634	224224	unkown	timeout
pol7-46	57935	229320	SAT	113.903

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 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

SHA-1 Preimage Instances for SAT

Saeed Nejati, Jia Hui Liang, Vijay Ganesh, Catherine Gebotys and Krzysztof Czarnecki
University of Waterloo, Waterloo, ON, Canada

***Abstract*—A brief description of the instances we submitted to the SAT Competition 2017 encoding SHA-1 preimage attacks.**

I. Introduction

Cryptographic hash functions are vital to many security-related applications. *Preimage resistance* is one the key properties of a cryptographic hash function to ensure good security. In other words, given a sequence of bits H (also called the *hash value*), it must be computationally infeasible to find a message M such that $H = HashFunction(M)$. Trying to find such a M is called the *preimage attack*. Many security protocols and primitives rely on the assumption that the preimage attack is difficult on cryptographic hash functions. If the assumption were to fail, then these protocols and primitives will be rendered insecure.

SHA-1 is a popular cryptographic hash function designed by NSA [1]. The full version of SHA-1 essentially applies a step function 80 times (also called 80 rounds), but preimage attack on full SHA-1 is still far out of reach for modern SAT-solvers which is good news for SHA-1 users. Our benchmark submission encodes SHA-1 with fewer than 80 rounds, in which case the hash function is called step reduced SHA-1. The fewer the rounds, the easier it is to find preimages.

Recent work by Google researchers successfully found SHA-1 collisions in February 2017, a related but different problem to preimage attack.

II. Benchmark

We submit 10 instances each of 21/22/23/24 step reduced SHA-1 preimage attack. More precisely, we initialize H to be an array of 160 random bits. Then we encode $H = StepReducedSha1(M, rounds)$ into CNF where M is an array of 512 Boolean variables. The SAT solver must solve for M such that a model for M would hash to H, hence a preimage attack. We repeat this process ten times for $rounds = 21/22/23/24$ for a total of 40 instances. The best SAT-based preimage attack can invert 23 rounds [2]. These instances come from the experimental section of that paper [2].

References

[1] NIST, "Secure Hash Standard," *Federal Information Processing Standard, FIPS-180-1*, 1995.

[2] S. Nejati, J. H. Liang, V. Ganesh, C. H. Gebotys, and K. Czarnecki, "Adaptive Restart and CEGAR-based Solver for Inverting Cryptographic Hash Functions," *CoRR*, vol. abs/1608.04720, 2016. [Online]. Available: http://arxiv.org/abs/1608.04720

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Encoding Rubik's Cube Puzzle to a SAT Problem

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

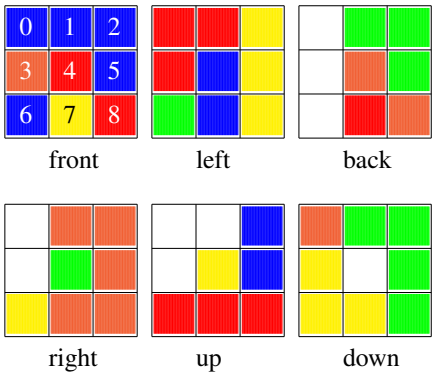

Fig. 1. A state of Rubik's Cube

***Abstract*—Rubik's Cube is an easily-understood puzzle, which is originally called the "magic cube". It is a well-known planning problem, which has been studied for a long time. This document describes how to to translate Rubik's Cube puzzle to a SAT problem.**

I. Introduction

Rubik's Cube puzzle is a well-known planning problem, which is originally called the "magic cube". The puzzle game was invented in 1974[1] by Ernö Rubik. So far,yet many simple properties remain unknown.

With respect to this puzzle, one of the most natural questions is how many moves are required to solve Rubik's Cube in the worst case. In 1995, Reid proved that the lower bound on the number of moves and the upper bound is 20 and 29, respectively [1], [2], [3]. In 2010, this open problem was settled. Rokicki, Kociemba, Davidson and Dethridge [4], [5] proved that God's number (i.e. the upper bound) for the Cube is exactly 20. They spent about 35 CPU-years of idle computer donated by Google to solve all 43,252,003,274,489,856,000 positions of the Cube.

Rubik's Cube is a 3-D mechanical cube. The center cubies on each face and the core of Rubik's Cube forms a fixed frame. Other 20 cubies move around them. A full face of the larger cube is divided into 9 facelets, each of which is a face of a distinct cubies, where each face of the cubies is colored one of six colors. A state of Rubik's Cube can be considered as a permutation on 48 facelets, since 6 center facelets are fixed. In general, it may be described by six faces: "front", "left", "back", "right", "up" and "down" face, each with 3×3 facelets. Figure 1 presents a sate (position) of Rubik's Cube. A state of Rubik's Cube is said to be the home state (position) if all facelets of each face in that state are the same color.

II. SAT encoding of the Rubik's Cube puzzle

Rubik's Cube has eighteen 18 moves, which are denoted by $U, U', U2, D, D', D2, L, L', L2, R, R', R2, F, F', F2, B, B'$, and $B2$. Each clockwise 90 degree move is specified by just the face with no suffix, and each counterclockwise 90 degree move and each 180 degree move are specified by the face followed by a prime symbol ($'$), and 2. So U here denotes a clockwise quarter turn of "up" face, and similarly, D, L, R, F and B denote "down", "left", "right", "front" and "back". A solution can be denoted by a move sequence. For example, the move sequence $F2U2B'U'B2D'U2F'U2LDR2B2U2F'U2F'U2B2L2$ is a solution to the state shown in Figure 1. That is, performing each move in this sequence can restores that state to the home state. We denote the set of 18 moves mentioned above by S_{18} , and $A_{10} = \{U, U', U2, D, D', D2, L2, R2, F2, B2\}$.

The Rubik's Cube puzzle may be described by initial state, move sequence, map relation of each move and the solved state. A Rubik's Cube has a total of six colors. A color corresponds a Boolean variable. Thus, representing each color on each facelet requires six Boolean variables. In fact, six colors contains only $\log 6 \approx 2.6$ bits. So the number of Boolean variables can be reduced. Let $b_1b_2b_3$ be the binary representation of $k(0 \leq k \leq 5)$. The Boolean variable representation of the k-th color is $x_1(b_1), x_2(b_2), x_3(b_3)$, where $x_i(b_i)$ is x_i if b_i is 1, and $\overline{x_i}$ otherwise. For example, the Boolean variable representation of the second color is $\overline{x_1}, x_2, \overline{x_3}$. Therefore, 3 Boolean variables suffice for the colors of each facelet. We divide states into two categories: general state and H-state. A state is said to be H-state if it can be transformed into the solved state by a sequence of the moves in A_{10} mentioned above. In the two-phase algorithm, each state in Phase two is H-state. For general states, we represent each color on each facelet with three Boolean variables. For H-states, we represent each color on each facelet with two Boolean variables. In the 2-variable scheme, we represent the colors of the front, left, back, right face in the solved state by 00, 01, 10 and 11, respectively, and then re-use 00 and 01 to represent the colors of the other two (top and down) faces. Notice, any move in in A_{10} cannot transform any facelet on top and down faces to somewhere on the other four faces. H-

states are allowed to use only moves in A_{10}. Therefore, under H-states, the 2-variable scheme does not yield any confusing.

Let $c(i,j,m)$ be the color of the j-th facelet in the i-the face under the m-th ($m \geq 1$) state, $c(i,4,1)$ the center facelet color of the i-th face under the initial state. If the m-th state is the solved state, it may be represented by

$$\bigwedge_{1\leq i\leq 6, 0\leq j\leq 8} c(i,j,m) = c(i,4,1)$$

Using 3-variable scheme, $c(i,j,m) = c(i,4,1)$ is translated into $c(i,j,m,1) = c(i,4,1,1) \wedge c(i,j,m,2) = c(i,4,1,2) \wedge c(i,j,m,3) = c(i,4,1,3)$, where $c(\ldots 1)$, $c(\ldots 2)$ and $c(\ldots 3)$ are literals that denote the 1st, 2nd and 3rd bit of a color. Formula $c(i,j,m,1) = c(i,4,1,1)$ can be translated into the following clauses: $(c(i,j,m,1) \vee \neg c(i,4,1,1)) \wedge (\neg c(i,j,m,1) \vee c(i,4,1,1))$.

An initial state of a cube is considered as State 1, which is interpreted as

$$\bigwedge_{1\leq i\leq 6, 0\leq j\leq 8, k=1,2,3} B(c(i,j,1,k))$$

where $B(c(i,j,1,k))$ is defined as $c(i,j,1,k)$ if the value of the k-th bit color of facelet$(i,j,1)$ is 1, and $\neg c(i,j,1,k)$ otherwise.

Assume we take at most $n-1$ moves to solve Rubik's Cube, and associate a Boolean variable s_t with each state $t (1 \leq t \leq n)$. " $s_t =$ true " mean the t-th state is the solved state. Then, this constraint can be represented by

$$\bigwedge_{1\leq i\leq 6, 0\leq j\leq 8} (\neg s_t \vee c(i,j,t) = c(i,4,1))$$

This formula can be converted easily into clauses.

At any time, among $S = \{s_1, s_2, \ldots, s_n\}$, we must ensure that exactly one s_t is true. The *exactly-one* constraint can be formalized by the *at-least-one* (ALO) and *at-most-one* (AMO) constraint. That is, exactly-one$(S) \equiv$ ALO $(S) \wedge$ AMO (S). The standard SAT encodings of constraints ALO and AMO are the following.

$$\mathrm{ALO}(S) \equiv s_1 \vee s_2 \vee \cdots \vee s_n$$
$$\mathrm{AMO}(S) \equiv \{\overline{s_i} \vee \overline{s_j} | s_i, s_j \in S, i < j\}$$

The ALO constraint ensures that a variable is true. And the AMO constraint ensures that no more than one variable is true. The standard AMO encoding requires much more clauses. To reduce the number of clauses, we can apply a two-product AMO encoding [6], which is recursively defined as

$$\mathrm{AMO}(S) \equiv \mathrm{AMO}(U) \wedge \mathrm{AMO}(V) \wedge \bigwedge_{1\leq k\leq n, k=(i-1)q+j, 1\leq i\leq p, 1\leq j\leq q} ((\overline{x_k} \vee u_i) \wedge (\overline{x_k} \vee v_j))$$

where $p = \lceil \sqrt{n} \rceil, q = \lceil \frac{n}{p} \rceil, U = \{u_1, u_2, \ldots, u_p\}, V = \{v_1, v_2, \ldots, v_q\}$, each element u_i in U and each element v_j in V are auxiliary variables. Here, $\mathrm{AMO}(U)$ and $\mathrm{AMO}(V)$ apply the standard AMO encoding.

To encode efficiently the constraints on the turns, we classify the turns of Rubik's Cube into six classes: u, d, l, r, f and b. Let $u_k (1 \leq k \leq n)$ be a Boolean variable that is associated with the up turn of step k. We perform either U-, or U'- or $U2$-type up turn at step k when u_k is true, and do the other turn otherwise. The meaning of d_k, l_k, r_k, f_k and b_k is similar. At any step, we have a unique turn. This constraint can be formalized by exactly-one$(u_k, d_k, l_k, r_k, f_k, b_k)$ for $1 \leq k \leq n$. Each u_k corresponds actually three different turn: $U, U', U2$. We denote the $U, U', U2$ of step k by Boolean variables U_k, U'_k, U_k2. Clearly, these Boolean variables should satisfy $\neg u_k \vee$ exactly-one(U_k, U'_k, U_k2). Similarly, we have the following constraint conditions:

$\neg d_k \vee$ exactly-one(D_k, D'_k, D_k2)
$\neg l_k \vee$ exactly-one(L_k, L'_k, L_k2)
$\neg r_k \vee$ exactly-one(R_k, R'_k, R_k2)
$\neg f_k \vee$ exactly-one(F_k, F'_k, F_k2) and
$\neg b_k \vee$ exactly-one(B_k, B'_k, B_k2).

A move can be considered as a mapping that maps each facelet $c(i,j,k) (1 \leq i \leq 6, 0 \leq j \leq 8, 1 \leq k \leq n)$ at State k to a facelet $c(i',j',k-1)$ at State $k-1$. Let M_k be a Boolean variable denoting one of 18 different moves at step k. The corresponding mapping is denoted by f_{M_k}. Then we have the following constraint condition.

$$\neg M_k \vee \bigwedge_{1\leq i\leq 6, 0\leq j\leq 8} c(i,j,k) = f_{M_k}(c(i,j,k))$$

For the clockwise up turn U_k, the mapping relationship of f_{U_k}: $c(i,j,k) \to c(i',j',k-1)$ is the following.

$c(i \bmod 4+1, 0, k) = c(i, 0, k-1)$,
$c(i \bmod 4+1, 1, k) = c(i, 1, k-1)$,
$c(i \bmod 4+1, 2, k) = c(i, 2, k-1)$ for $1 \leq i \leq 4$
$c(5,0,k) = c(5,6,k-1)$, $c(5,1,k) = c(5,3,k-1)$,
$c(5,2,k) = c(5,0,k-1)$, $c(5,3,k) = c(5,7,k-1)$,
$c(5,5,k) = c(5,1,k-1)$, $c(5,6,k) = c(5,8,k-1)$,
$c(5,7,k) = c(5,5,k-1)$, $c(5,8,k) = c(5,2,k-1)$

The other facelets keep unchanged.

Some two-move sequences will yield the same result. For example, two-moves UD and DU have the same result states. To speed up the search, we remove the search on two-move sequences such as DU. The removing of such a search can be done by adding the following constraint clauses to the SAT encoding of Rubik's Cube.

$$\bigwedge_{1\leq k\leq n} ((\overline{u_k} \vee \overline{d_{k+1}}) \wedge (\overline{l_k} \vee \overline{r_{k+1}}) \wedge (\overline{f_k} \vee \overline{b_{k+1}}))$$

To find efficiently a solution, we need encode Kociemba's algorithm, which is a two-phase algorithm. it splits the problem into two almost equal subproblems, each of which can use a lookup table to search for exhaustively a solution. Here is the pseudo-code of Kociemba's algorithm.

Kociemba's Algorithm

```
d ← 0, t ← ∞
while d < t do
  for s ∈ S^d_18, ps ∈ H do
    if d + D(ps) < t then
      find a better solution, using moves in A_10
      t ← d + D(ps)
    end if
  end for
  d ← d + 1
end while
```

This algorithm assumes that the original state is p, and applies some move sequence $s \in S^d_{18}$ of length d to the original cube yielding ps which lies in H. This search process

is called phase one. Here H is a subset of states that is composed of all patterns with following characteristics:

1) The orientation of all corner cubies and edge cubies is correct.
2) The edge cubies that should be in the middle layer are now located in the middle layer.

These characteristics are preserved by moves in the set A_{10}. The search process from the new state ps to the fully solved state is called Phase two. In this phase, each move is in A_{10}. $D(ps)$ returns the distance from the state ps to the solved state using moves in A_{10}.

It is impossible to encode directly the entire Kociemba's algorithm into a SAT problem, because it contains lookup tables. However, it is possible to encode the basic idea of Kociemba's algorithm with a CNF formula. Let Cube_CNF(n) denote a SAT encoding of Rubik's Cube with a total of n states. Assume that the k-th state s_k reaches a state in H. A SAT encoding of Rubik's Cube based on Kociemba's algorithm can be described as

Cube_CNF(n) $\wedge$ Hstate(s_k) $\wedge$ A_{10}_move(k, n)

where Hstate(s_k) is true if s_k is in H, and A_{10}_move(k, n) is used to restrict moves from step k to step n to be moves in A_{10}. Hstate(s_k) is defined as

$$\bigwedge_{1\le i\le 4, j=3,5} (c(i,j,k) = c(i,j,4) \vee c(p(i),j,k) = c(i,j,4)) \wedge$$
$$\bigwedge_{i=5,6 \wedge 0\le j\le 8} (c(i,j,k) = c(i,j,4) \vee c(p(i),j,k) = c(i,j,4))$$

where $p(i)$ denotes the opposite face of the i-th face, i.e., the mapping relationship of p is: $1 \leftrightarrow 3$, $2 \leftrightarrow 4$, $5 \leftrightarrow 6$.

Recall that L, L', R, R', F, F', B and B' all are not in A_{10}. So A_{10}_move(k, n) may be described by the following logic formula.

$$\bigwedge_{k<m\le n} \neg(L_m \vee L'_m \vee R_m \vee R'_m \vee F_m \vee F'_m \vee B_m \vee B'_m)$$

It means that after step k, neither clockwise nor counter clockwise 90 degree turn of any face except for the up and down face is allowed. In general, k is set to less than 12. That is, the length of Phase one is limited to 12.

REFERENCES

[1] Rokicki, T.: In search of: 21f*s and 20f*s; a four month odyssey, 2006, http://cubezzz.homelinux.org/drupal/?q=node/view/56

[2] Reid, M.: Superflip requires 20 face turns, 1995, http://www.math.rwth-aachen.de/ Martin.Schoenert/Cube-Lovers/ michael_re%id_superflip_requires_20_face_turns.html

[3] Reid, M.: New upper bounds, 1995, http://www.math.rwth-aachen.de/ Martin.Schoenert/Cube-Lovers/ michael_re%id_new_upper_bounds.html http://cubezzz.homelinux.org/drupal/?q=node/view/53.

[4] Flatley, J.F.: Rubik's Cube solved in twenty moves, 35 years of CPU time, Engadget, 2010.

[5] Rokicki,T., Kociemba, H., Davidson, M., Dethridge, J.: God's Number is 20, 2010, www.cube20.org.

[6] Chen, J.C.: A new SAT encoding of the at-most-one constraint, Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation, St. Andrews, Scotland, UK, 2010.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Description of Popularity-Similarity SAT Instances

Jesús Giráldez-Cru
KTH, Royal Institute of Technology
Stockholm, Sweden
giraldez@kth.se

Jordi Levy
Artificial Intelligence Research Institute (IIIA-CSIC)
Barcelona, Spain
levy@iiia.csic.es

***Abstract*—This document describes the Popularity-Similarity random SAT instances submitted to the SAT Competition 2017. For a more detailed description of this model, we address the reader to the original reference [8].**

I. Introduction

It is well accepted that *application* or *real-world* SAT instances are characterized by a big variability in the number of variable occurrences. In particular, it has been shown that in most of these instances, the number of variable occurrences x follows a power-law distribution $P(x) \sim x^{-\gamma}$ [2]. In many cases, the clauses size also shows this kind of distribution. The *scale-free* model [3] has been proposed to generate random SAT instances where the number of occurrences of variables follows a power-law distribution, as observed in real-world SAT instances. Power-law distributions implies the existence of very frequent or *popular* variables in the formula. The scale-free model has been used to show that VSIDS prefers branching on those popular variables. However, it has been observed that VSIDS also focuses on some *local area* of the formula during the search, suggesting that popularity is not the only crucial feature of real-world SAT formulas. In fact, sometimes it is preferable to assign variables *close*, or *similar*,[1] to other recently assigned variables, rather than assigning popular but *distant* variables [9].

Another common feature found in a vast majority of real-world SAT instances is modularity [4], [5], [1]. The Community Attachment model [6], [7] has been proposed to generate random SAT instances with clear community structure (or high modularity). Using this model, it has been observed that VSIDS tends to focus its decisions inside *communities* of variables closely connected by clauses.

To the best of our knowledge, the Popularity-Similarity model of random SAT formulas [8] is the first random model that captures both properties at once: a power-law distribution in the number of variable occurrences and community structure. Moreover, it also allows the generation of SAT formulas with clause size following a (different) power-law distribution.

[1]Similar with respect to some metric, e.g., distance in a graph representation of the formula.

II. The Popularity-Similarity Model

In this section, we briefly summarize the main features of the Popularity-Similarity model. For a more detailed description, we address the reader to the original reference [8].

In order to generate a Popularity-Similarity (PS) random SAT instance over n variables with m clauses of average size k, we first assign a random angle $\theta_i \in [0, 2\pi]$, to every variable $i \in \{1, \ldots, n\}$, and a random angle $\theta'_j \in [0, 2\pi]$, to every clause $j \in \{1, \ldots, m\}$, with uniform probability distributions.

Then, we construct a bi-partite random graph with n variable-nodes and m clause-nodes, where every possible edge $i \leftrightarrow j$ between a variable-node i and a clause-node j, is selected with probability:

$$P(i \leftrightarrow j) = \min\left\{1\,,\, \left(\frac{R}{i^{\beta} \cdot j^{\beta'} \cdot \theta_{ij}}\right)^{1/T}\right\}$$

where β, β' and T are parameters of the model, θ_{ij} is the minimal distance between angles θ_i and θ'_j:

$$\theta_{ij} = \pi - \left|\pi - |\theta_i - \theta'_j|\right|$$

and R is a normalizing constant ensuring that, on average, the number of selected edges is $k\,m$.

Finally, we construct a random SAT formula from the graph as follows. For every edge $i \leftrightarrow j$ in the bi-partite graph, we add to the clause C_j the literal x_i with probability $1/2$, or the literal $\neg x_i$ otherwise.

In this model, the number of variable occurrences x follows a power-law distribution $P(x) \sim x^{-\gamma}$, with $\gamma = 1 + 1/\beta$. This distribution is obtained as a consequence of *popularity*, being popular variables more likely to be chosen in each clause. On the other hand, *similarity* (i.e., the angular distance θ_{ij}) provokes that non-popular but similar variables to already selected variables in the clause are also likely to be chosen in such a clause. This results into a formula with high clustering and hence, with community structure. Finally, the clause size x' also follows a power-law distribution $P(x') \sim x'^{-\gamma'}$ with $\gamma' = 1 + 1/\beta'$.

When the temperature T is low, variables are (mostly) chosen according to their popularity and similarity. In contrast, when T is high enough, all variables have almost the same probability to be chosen at a certain step. Therefore, the

temperature *regulates* how close the PS model behaves w.r.t. the classical random model.

It is also important to remark that, although the average clause size in the model is k, the existence of (very few) short clauses *dominates* the model, making formulas (almost) trivial. In order to avoid this situation, we force the model to create clauses with at least K literals, being $(K+k)$ the new average clause size of the generated formula.

In summary, the model is characterized by the following parameters:

- n: number of variables.
- m: number of clauses.
- K: minimum clause size.
- $(K + k)$: average clause size.
- $\gamma = 1 + 1/\beta$: exponent of the power-law distribution for number of variable occurrences.
- $\gamma' = 1 + 1/\beta'$: exponent of the power-law distribution for clause size.
- T: temperature.

III. Submitted Benchmarks

In order to generate the instances submitted to the SAT Competition 2017, we use the following parameter values:

- $n = 5000$ and $m = 21250$ ($m/n = 4.25$)
- $K = 3$, $k = 0$, and $\beta' = 0$
- $\beta = \{0.8, 0.7, 0.6\}$
- $T = \{1.45, 1.50, 1.55, 1.60, 1.70, 1.80\}$

Notice that using $K = 3$, $k = 0$ and $\beta' = 0$, all clauses in the generated formulas have exactly 3 literals.

For each family of instances, we generate 10 distinct instances (with 10 distinct random seeds).

References

[1] C. Ansótegui, M. L. Bonet, J. Giráldez-Cru, and J. Levy, "Community structure in industrial SAT instances," *arXiv*, vol. 1606.03329, 2016.

[2] C. Ansótegui, M. L. Bonet, and J. Levy, "On the structure of industrial SAT instances," in *Proc. of the 15th Int. Conf. on Principles and Practice of Constraint Programming (CP'09)*, 2009, pp. 127–141.

[3] C. Ansótegui, M. L. Bonet, and J. Levy, "Towards industrial-like random SAT instances," in *Proc. of the 21st Int. Joint Conf. on Artificial Intelligence (IJCAI'09)*, 2009, pp. 387–392.

[4] C. Ansótegui, J. Giráldez-Cru, and J. Levy, "The community structure of SAT formulas," in *Proc. of the 15th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'12)*, 2012, pp. 410–423.

[5] C. Ansótegui, J. Giráldez-Cru, J. Levy, and L. Simon, "Using community structure to detect relevant learnt clauses," in *Proc. of the 18th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'15)*, 2015, pp. 238–254.

[6] J. Giráldez-Cru and J. Levy, "A modularity-based random SAT instances generator," in *Proc. of the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI'15)*, 2015, pp. 1952–1958.

[7] J. Giráldez-Cru and J. Levy, "Generating SAT instances with community structure," *Artif. Intell.*, vol. 238, pp. 119–134, 2016.

[8] J. Giráldez-Cru and J. Levy, "Locality in random SAT instances," in *Proc. of the 26th Int. Joint Conf. on Artificial Intelligence (IJCAI'17)*, 2017, p. accepted.

[9] G. Katsirelos and L. Simon, "Eigenvector centrality in industrial SAT instances," in *Proc. of the 18th Int. Conf. on Principles and Practice of Constraint Programming (CP'12)*, 2012, pp. 348–356.

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Railway Interlocking System Safety Proof

Damien Ledoux
SNCF Reseau
Saint-Denis, France
damien.ledoux@reseau.sncf.fr

I. INTRODUCTION

SNCF Reseau owns, and has overall responsibility for the management of the French rail network. In particular, it oversees the circulation of all trains. The operation of this transport system relies on critical systems that interact in an indeterministic environment consisting of qualified operators who apply operational rules.

The overall safety of the railway system is therefore based on both:

- the correctness and suitability of operational rules; and
- the absence of failures in critical systems (interlocking, centralized control systems, etc.).

To continue to ensure a high level of safety in the French rail network SNCF has, for many years, drawn upon formal methods. Technology based on the Boolean satisfiability problem (SAT) has become a very active field of academic research. Interest in these techniques lies in their effectiveness in solving industrial problems and their ease of use. Proof of a safety property can be largely automated using a proof engine. If the proof is found, the property is valid. Alternatively, a counter example is found, in the form of a system execution leading to the violation of a property. These counter examples are very useful for the ongoing development of the system.

II. BENCHMARK

This benchmark is an extension of the benchmark [1] submitted during the SAT COMP2016, which confirmed the ability of current SAT solvers to rapidly solve models by induction. Furthermore, when the models are manipulated in Bounded Model Checking (BMC) mode to find a counter example, our results showed that the greater the depth, the harder it is for the SAT solver to find a solution. A major drawback of the models that were presented at the SAT COMP2016 is that they use brute force, rather than drawing upon the intelligence of an incremental solver. Therefore, in order to compare new, theoretical models from the academic world with the models that are currently manipulated in the industry, we developed the following two models from an IXL of 100 routes, 25 signals and 35 points.

- model_proof_cex: a model containing a single safety property that is falsified to a depth of 8.
- model_proof_all_to_1PO: a model containing approximately 18000 safety properties that are all valid.

The models were first generated in AIGER format [2], in order to be able to test several model checkers (e.g. aigbmc, iimc, tip, abc, …). The XCNF model [3] was generated from this format, using a modified aigbmc tool.

The models were run on the following solvers:

- AIGER format: aigbmc + lingeling-aqw [2], iimc-2.0 [4], nuXmv-1.0.1 [5], tip-2014 [6]
- XCNF format: all incremental solvers presented at the SAT COMP2016 [7] (except for riss-600 which returned an error for model_proof_all_to_1_PO at a depth of 4).

The results are summarized in the tables below. The models were run on an Intel(R) Core(TM) i7, 3820 CPU @ 3.60GHz, with a time limit of 3600 seconds. Where no time is given, the time limit was exceeded before the solver had finished.

Format	Solver (ipasir-incremental)	Depth							
		1	2	3	4	5	6	7	8
		Calculation time (seconds)							
XCNF	abcdsat_inc	0	0	1	1	53	214	97	310
	cominisatps2sun	0	1	1	1	5	24	87	134
	icominisatps2sun_nopre	0	0	0	1	7	31	20	152
	cryptominisat4	0	1	1	1	13	13	44	257
	cryptominisat4auto	0	1	1	1	13	13	44	257
	glucose4	0	0	0	1	16	16	41	231
	riss_521	0	0	0	2	4	4	17	228
AIGER	aigbmc + lingeling-aqw	0	0	1	6	27	111	278	832
	iimc-2.0	21	23	25	27	44	59	52	189
	nuXmv-1.0.1	1435	2984						
	tip-2014	0	0	0	0	1	2	13	41

Table 1- Solver performance on model_proof_cex

Format	Solver (ipasir-incremental)	Depth						
		1	2	3	4	5	6	7
		Calculation time (seconds)						
XCNF	abcdsat_inc	34	114	401	1316			
	cominisatps2sun	27	110	462	1351			
	cominisatps2sun_nopre	39	127	515	1556			
	cryptominisat4	30	88	397	1157	2679		
	cryptominisat4auto	30	88	390	1171	2702		
	glucose4	45	130	621	2268			
	riss_521	51	186	1174				
AIGER	aigbmc + lingeling-aqw	71	410					
	iimc-2.0	87	413	2142				
	nuXmv-1.0.1							
	tip-2014	7	32	452	1325			

Table 2 - Solver performance on model_proof_all_to_1_PO

In view of these results, we selected the following models to present at SAT COMP2017 (in XCNF format):

- model_proof_cex.8.xcnf: model_proof_cex to a depth of 8. This model should not be a problem for solvers. It will allow us to see the progress made and compare it to last year's competition.
- model_proof_all_to_1PO.4.xcnf: model_proof_all_to_1_PO to a depth of 4. This model should not be a problem for solvers. It will allow us to see the progress made and compare it to last year's competition.
- model_proof_all_to_1PO.5.xcnf: model_proof_all_to_1_PO to a depth of 5. This model is closer to the limits of some of the solvers presented at SAT COMP2016, and it will be interesting to see if all of the current solvers are able to handle it.
- model_proof_all_to_1PO.6.xcnf: model_proof_all_to_1_PO to a depth of 6. This model has been designed to challenge the best SAT solvers, notably cryptominisat4, which obtained the best results in our benchmark tests.

[1] « An Interlocking Safety Proof Applied to the French Rail Network », SAT COMP2016

[2] http://fmv.jku.at/aiger/

[3] http://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=incremental

[4] http://iimc.colorado.edu

[5] https://nuxmv.fbk.eu

[6] https://github.com/niklasso/tip

[7] http://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=solvers

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Balanced random SAT benchmarks

Ivor Spence
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
University Road
Belfast BT7 1NN Email: i.spence@qub.ac.uk

***Abstract*—Empirically, the most difficult SAT benchmarks for their size measured by number of literals, have been generated by `sgen6` [3]. These benchmarks included 2-clauses and a method for solving them quickly has now been identified. We describe here a method for generating difficult benchmarks in which every clause has the same size and for which there is no quick solution method known.**

I. Introduction

The authors benchmark generator `sgen` has created CNF instances which, for the number of literals, have proved to be amongst the most difficult for any known solver. The most recent version was sgen6. However, Knuth has noted [2] that these benchmarks were equivalent to a matching problem which can be solved quickly by the dancing links algorithm [1] and although no known solver exploits this, in principle such a solver could be written. We describe here a generator for a revised series of benchmarks which cannot be solved in this way. For *existing* solvers the new benchmarks are easier than `sgen6` but unlike `sgen6` there is no known way of solving these new benchmarks quickly.

The construction of the `balancedsat` benchmarks is less constrained than for `sgen` and it is debatable whether these belong in the crafted or random benchmark categories. However, the method of construction does guarantee that each variable occurs the same number of times (± 1) and the number of positive and negative occurrences is the same (again ± 1). We also ensure that, so far as possible, the same two variables do not occur in two difference clauses and these constraints together suggest the name `balancedsat`. We submit a generator and some suggestions for the scaling values to be used in the SAT 2017 Competition.

II. Overall Structure of a Benchmark

Each clause in a benchmark has the same number of literals (k) for which the suggested and default value is 3. The clauses are generated in rows and within each row every variable occurs exactly once. The assignment of variables to clauses within a row is done by a simple greedy algorithm which attempts to minimize the number of clauses in the whole benchmark which contain the same two variables.

For each variable, the signs of the corresponding literals are constrained so that the number of positive and negative occurrences are almost equal. The sign of the first occurrence of each variable (which will be on row 0) is chosen randomly but the second occurrence is then given the opposite sign.

cnf file	rows (ignoring signs)
p cnf 9 12	
1 -7 -5 0 -6 -9 -4 0 2 -8 3 0	(1 7 5)(6 9 4)(2 8 3)
4 8 7 0 5 9 -3 0 -1 -2 6 0	(4 8 7)(5 9 3)(1 2 6)
7 2 9 0 5 4 -3 0 -8 1 6 0	(7 2 9)(5 4 3)(8 1 6)
-2 -5 -4 0 -1 -9 3 0 -6 -7 8 0	(2 5 4)(1 9 3)(6 7 8)

Fig. 1. Output generated by balancedsat 9 12 0

Every occurrence on an even numbered row is given a random sign and the occurrence on the subsequent row is given the opposite sign from the previous row.

III. Example

For example Fig. 1 lists a benchmark with 9 variables and 12 clauses. Each row uses every variable exactly once and so contains 3 clauses. The signs of the variable 1 in successive rows are: 1 (random), -1, 1 (random), -1 whereas for 3 the signs are 3 (random), -3, -3 (random), 3. The assignment of variables to clauses has not been able to prevent 7 and 8 from being in the same clause twice, i.e. in (4 8 7) and (-6 -7 8) but this gets easier with large problems.

IV. Satisfiability and SAT2017 benchmarks

Empirical results suggest that `balancedsat` benchmarks exhibit a phase change where the number of clauses is approximately 3.6 times the number of variables. With fewer clauses the benchmarks are likely to be satisfiable and easier. With more clauses they are likely to be unsatisfiable and also slightly easier. The benchmarks suggested for SAT2017 were determined by using the smallest number of clauses leading to an unsatisfiable benchmark for a given number of variables. They are listed in Fig. 2, where the seed was always 0, the benchmarks are all unsatisfiable, and the times for solution are for the solver `clasp`.

vars	clauses	clauses/vars	time(s)
180	648	3.6	33
192	689	3.6	97
201	707	3.5	483
210	739	3.5	984
222	777	3.5	5994

Fig. 2. benchmarks suggested for SAT2017

V. Generator

The generator consists of a single C program which can be compiled with the command

```
gcc -o balancedsat balancedsat.c
```

There are four possible arguments when running `balancedsat`. These are

numVars The number of variables to be used. This should be a multiple of the number of literals in each clause (k).

numClauses The number of clauses to be used. If numClauses is less than approximately 3.7*numVars the generated benchmark is likely to be satisfiable and easy to solve. Above 3.7*numVars the benchmark is likely be unsatisfiable.

seed A seed for the random number generation.

k The number of literals in each clause. This is optional and if not specified a default of 3 is used.

VI. Conclusion

It has not been found possible to generate benchmarks where every clause is of length 3 which are as difficult for available solvers as those generated by `sgen6` which contain 2-clauses. However there is a known method (as opposed to an actual solver) for solving `agen6` quickly whereas there is no known method for solving `balancedsat` quickly.

References

[1] D. E. Knuth, "Dancing links," in *Millennial Persectives in Compute Science*. Palgrave, 2000.

[2] ——, *The Art of Computer Programming Fascicle 6 Volume 4B: Satisfiability*. Pearson Education, 2005.

[3] I. Spence, "Weakening cardinality constraints creates harder satisfiability benchmarks," *J. Exp. Algorithmics*, vol. 20, pp. 1.4:1–1.4:14, may 2015. [Online]. Available: http://doi.acm.org/10.1145/2746239

Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, volume B-2017-1 of Department of Computer Science Series of Publications B, University of Helsinki 2017.

Difficult Regular Graph Coloring Theorems

Daniel Pehoushek
Self Organized
Email: pehoushek1@gmail.com

***Abstract*—Regular graph coloring is an abundant resource for sat solving, with instances ranging over the spectrum, from trivial to exponentially difficult. The submitted benchmarks focus on one known theorem, that Almost all 5 regular graphs are 3 colorable(C3D5), and two new theorems, about C4D9 and C5D13. The 71 benchmarks are mostly satisfiable, with a satisfying assignment in the preamble comments. Asymptotically, as N grows, all graphs in the theorem categories will be satisfiable. A search is underway for the C4D9 "Golden Point". See related SAT17 paper for many graph coloring results.**

The submission has 71 sat competition benchmarks. Source code, sat, and qbf benchmarks are within the directory submitted to both competitions, along with a paper with a very nice qbf result. A random instance generator is also in the submission, as a suggestion for a community wide shared library of possible instances, in a single cpp source file with zero header files.

1. Introduction

These three theorems are thought to be tight, in the sense that incrementing the degree by one makes them unsatisfiable at the given number of colors. The Golden Point (N out of N random instances are satisfiable, for a "high" probability of $(1-1/N)$) for C3D5 has been located at $N = 180$ vertices. The golden points for the two new theorems may take at least one year to locate and verify, and is in progress. So far, the 70 percent threshold for the two new theorems has been found, at C4D9N180, and C5D13N117. There is some gold mining effort at higher degrees and number of colors, mostly "gold panning" for single satisfiable instances. Single satisfiable random instances are sufficient to prove the "almost all" result for that category of graphs, but "tight" success gets difficult to achieve with higher degrees. For easy degrees and colors, and for finding various levels of high probability, some of the author's experimental files contain hundreds of thousands of instances. But, for these two difficult theorems, while searching for golden points, there only a few instances.

2. The Graphs

The graphs are included in the preamble of the instance files, in case other translations to sat are desired. The C3D5 instances are all satisfible, with an incredibly high probability. When satisfiable, the C4D9 and C5D13 instances contain an assignment or coloring of the graph. When unsatisfiable, that is noted also. There is one C4D9 instance in the "unknown" category, and is probably unsatisfiable.

The C4D9 instances are 4 cnfs (lg of number of colors variables per vertex), and the C5D13 are 6-cnfs. The C4D9 instances are the hardest, most compact 4-cnfs ever encountered by the author in over 25 years of effort. The majority of these have 360 variables (at the 70 percent satisfiable threshold), and there are a few with 400 variables; the 200 vertex C4D9 graphs are part of the search for the C4D9 golden point. There are a handful of easy C4D9 instances, at N=132, that are used to generate qbf benchmarks. Eventually, the 8 coloring problems, near degree 25, will be very hard and compact 6-cnfs, but there is little hope of finding many 8 coloring golden points.

The C5D13 instances are 6 cnfs. Note that D10 D11 and D12 are all 5 colorable, and the author has found the golden points for those degrees with 5 colors. Most of the C5D13 graphs have 117 vertices, and 351 variables. There is one instance with 390 variables, and sat solvers are encourage to solve that instance at home and report their time by email. It is in the million second category at present. There may be several in the "easy" category, for C5D13N104, but easy is relative.

Next year, if C4D9N200 turns out to be the golden point, I will submit 200 randomly generated satisfiable instances. For hard problems that take alot of time, I believe satisfiable formulas are the only ones with significant value, as the answer is so trivial to check. This philosophy has guided my search for the various levels of Golden Points in regular graph coloring, for many years. Asymptotically, for some degree and number of colors, the "almost all" theorem is either true or false, but until an example is successfully solved, the answer is unknow.

3. Comparison with theoretical result

In the paper "On the chromatic number of random regular graphs", by Coja-Oghlan Efthymiou and Hetterich, they have the following theorem, a fine general result about regular graph coloring:

Theorem 1.1 There is a sequence (ek)k=3 with limk?8 ek = 0 such that the following is true.

1. If d = (2k - 1) ln k - 2 ln 2 - ek, then G(n, d) is k-colorable w.h.p.

2. If d = (2k - 1) ln k - 1 + ek, then G(n, d) fails to be k-colorable w.h.p.

(We have not attempted to explicitly extract or even optimize the explicit error term ek.)

Plugging into the equations for k we get the following bounds on degree d, using zero for the error term:

```
C4D9:   8.32   8.70
C5D13: 13.10  13.48
C6D17: 16.82  18.71
C7D21: 23.91  24.30
```

So, the experimental results are in close agreement. I have a couple C6D17N102 graphs, and am gold panning for any C6D18 graph. There is some need to work on the error term for 4 coloring, as both of the theoretical raw numbers are less than 9. The difficulty of degree 9 graphs is indicated by the closeness to the theoretical limit. As the degree increases, the size of solvable graphs also necessarily increases, and the graphs become quite difficult. Perhaps the authors of the theory paper should offer a prize for the first randomly generated degree 23 to be 7 colored, which would thoroughly confirm their theorem. Such a graph would require many cpu weeks to solve. Presently, the author is gold panning for any 7-colorable 22-regular graph.

4. Conclusion

The author plans to search for the golden points for these two difficult theorems, plus some higher degree effort, to fill out the table published in a related paper. The goal is to generate and successfully satisfy N out of N graphs of size N, for C4D9 and C5D13.

I have four suggestions for improving the structure of the sat competition.

First, for entry level purposes, I suggest multiple intances per file, for maintaining neat directorys with well organized sets of instances. I have used hundreds of thousands of instances per file to find the cubic super high probability points for regular graphs. These make great regression suites, but would not have been possible without the basic multiple instances per file feature. I am sure other areas of research would be greatly improved with multiple formulas per file.

Second, the sat competition may need new time limit categories for certain classes of benchmarks and solvers, perhaps for "playoff levels" among the best solvers; at least 50,000 seconds, maybe 500,000 seconds; perhaps a few solvers want to submit single 5,000,000,000 second instances, for a yearly competition that is self organized among competitors. I suggest the higher time limits always be satisfiable instances, for certainty of checking results.

Third, I do not really want to spend my effort on supporting inevitably infeasible methods for checking long winded unsatisfiable formulas; thus, the Golden Points agenda gradually came into being... where correctness is probabilistically verified over large quantitys of formulas per file, and theoretically increasing probabilities of satisfiability on regular graphs as N grows, with correctness verified over hundreds of thousands of formulas per file. The disasterous idea that unsatisfiability proofs are a good idea has probably wasted many man decades of effort, and is diametrically opposite to the value of satisfiability proofs. The result is a small time limit that admits only easy problems, whose unsatisfiability is then verifiable by someone's proof system, by examining a trace of the program's execution. Some of my present benchmarks take two weeks of cpu time, with a trace that is roughly twenty trillion lines. And I know I am planning harder formulas. So, proofs of unsatisfiability should be eliminated from the entrance requirements, because they are stifling, costly, wasteful, and theoretically unnecessary to prove correctness of a solver. The competition constraints appear to be designed by a committee.

Fourth, somehow, formula size should be part of the score, with smaller formulas being more significant than large formulas. New programmers that can solve hundreds of thousands of trivial formulas in reasonable time would then recieve the positive and encouraging feedback of a decent score. At present, the large number of megasized but otherwise easy and theoretically insignificant formulas makes many of the solvers in the competition simple derivative implementations, that use some minor variation of random assignments, hoping for good luck. By making the importance and score for a formula inversely proportional to Log of the Size, new implementations will never be dismayed by failure on million variable problems, inspiring many more students to enter the field of satisfiability. The inversely proportional suggestion is so that new entrants do not spend futile effort on larger instances. Important work on an infinite number of small interesting formulas has largely been ignored, due to the Size Error in the satisfiability competition scoring system. Without this particular scoring change, the Satisfiability Conference and competition may gradually disappear into oblivion from self imposed constraints, that preclude new participants from even making an attempt. In the present system, the scoring values are linearly related to how much time a solver takes on an instance, with a two*limit penalty for failure to succeed within the time limit. The benchmarks are sverely limitted to being artificially easy for at least one solver. The large formulas satisfy some egos, but, are essentially theoretically insignificant to the core NP complete problem. I suggest muliple instances per file, where correctness is primary, and total time taken on the file be the score. This allows some solver variations on instances, without a crushingly constraining time limit per instance. There could be of course some time limit on the set of instances. Successful solvers on easy sets of instances get promoted to the next level. This will admit new competitors and allow students to get decent scores, inspiring further research. The committee should admit the exponential nature of the core NP complete problem; that is vital to continuation of the competition.

I have included some goldenpoint files for other degrees, and some smaller C4D9 instances. While they are not benchmarks, they may be interesting. The goldenpoint files contain multiple instances per file, where N out of N are satisfiable, in reasonable time.

References

[1] Amin Coja-Oghlan Charilaos Efthymiou Samuel Hetterich: On the chromatic number of random regular graphs. arXiv:1308.4287v1 [math.CO] (20 Aug 2013)

Solver Index

Benchmark Index

Author Index