!!! DRAFT !!! DRAFT !!!

***Proceedings of***

# SAT COMPETITION 2022

## Solver and Benchmark Descriptions

**Tomáš Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda** *(editors)*

# PREFACE

The area of Boolean satisfiability (SAT) solving keeps on making progress. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for the success story of SAT solving. To keep up the driving force in improving SAT solvers, SAT solver competitions provide opportunities for solver developers to present their work to a broader audience and to objectively compare the performance of their own solvers with that of other state-of-the-art solvers.

SAT Competition 2022 (SC 2022, `https://satcompetition.github.io/2022/`), a competitive event for SAT solvers, was organized as a satellite event of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). SC 2022 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held 2002-2005, biannually during 2007-2013, 2014, 2016–2018, and 2020–2021; the SAT Races held in 2006, 2008, 2010, 2015, and 2019; and SAT Challenge 2012.

SC 2022 consisted of a total of four tracks: Main Track (with CaDiCaL 1.4 Hack and No Limits sub-tracks), Parallel Track, Cloud Track and a special Anniversary Track in celebration of the 20th anniversary of this series of SAT Competitions (as well as the 30th anniversity of the first competition). The benchmarks for the anniversary track comprise of *all* benchmark instances which have been used in Application, Crafted, and Main Tracks of previous SAT competitions.

There were two ways of contributing to SC 2022: by submitting one or more solvers to participate in the competition and by submitting interesting benchmark instances on which the submitted solvers could be evaluated in the competition. The rules of SC 2022 required all contributors to submit a short, 1-2 page long description as part of their contribution. This book contains these non-peer-reviewed descriptions in a single volume, providing a way of consistently citing the individual descriptions and finding out more details on the individual solvers and benchmarks.

Successfully running SC 2022 would not have been possible without active support from the community at large. We would like to thank the StarExec initiative (`http://www.starexec.org`) for the computing resources needed to run SC 2022. Many thanks go to Aaron Stump for his invaluable help in setting up StarExec to accommodate for the competition's needs. Furthermore, we thank Amazon for providing the resources and support to develop parallel and distributed solvers on the AWS cloud and for executing the Cloud and Parallel tracks.

Finally, we would like to emphasize that a competition does not exist without participants: we thank all those who contributed to SC 2022 by submitting either solvers or benchmarks and the related description.

*Tomáš Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, & Martin Suda*
*SAT Competition 2022 Organizers*

# Contents

**Benchmark Descriptions**

# SOLVER DESCRIPTIONS

# GIMSATUL, ISASAT, KISSAT Entering the SAT Competition 2022

Armin Biere Mathias Fleury
University of Freiburg, Germany

***Abstract*—This system description explains the features of our new multi-threaded SAT solver GIMSATUL submitted to the parallel track of the SAT Competition 2022, as well as updates to our sequential SAT solvers ISASAT, and KISSAT, submitted to the corresponding sequential tracks of the competition.**

## IMPROVED SWEEPING IN KISSAT

Already in version "KISSAT SC2021 SWEEP" submitted to the SAT Competition 2021 we supported SAT based sweeping [1] which relies on the internal embedded SAT solver KITTEN to find backbones and equivalent literals extracted from the *environment* clauses of a candidate variable in which it occurs and some of its neighbouring clauses. The algorithm was improved by eagerly substituting literals determined to be equivalent already during sweeping and more careful scheduling and rescheduling of candidate variables, particularly within the same sweeping phase.

Furthermore we sped up the common case of satisfiable queries to the embedded SAT solver KITTEN, by adding to KITTEN the following *model flipping* API function:

```
bool kitten_flip_literal (kitten *, unsigned lit);
```

It is inspired by non-recursive *model rotation* [2] used in MUS extraction. Our new model-flipping tries to flip the value of the specified literal in the last model returned by KITTEN and succeeds if the resulting new assignment still satisfies the formula. On success the model is updated. Otherwise if flipping falsifies the formula the last model is not touched.

In our application we further require that the model does not change for other literals. Thus the implementation of *model flipping* is straightforward and simply consists of just traversing the clauses watched by the literal to be flipped and checking whether there are "one-satisfied" clauses with only that literal satisfying the watched clause (assuming it was assigned to true in the last model).

We make use of this new API function by trying to flip during sweeping all literals of either the remaining backbone candidates if there are any left or the literals in candidate equivalent literal classes. If all flipped literal attempts failed we have to fall back to a more expensive actual SAT solver call to KITTEN. If flipping succeeds, which happens actually surprisingly often, we refine the backbone candidate list or the candidate equivalent literal class as usual.

In [1] we described how we use randomization of saved phases before KITTEN queries to reduce the number of necessary refinements in the common case that sweeping is mostly unsuccessful for a candidate variable. It turns out that for some benchmarks the old version "KISSAT SC2021 SWEEP" spent a substantial percentage of time during sweeping in just generating random bits for this purposes. By using all 64 bits produced by our random number generator each time instead of just one (while dropping 63 bits) and updating saved phases in a bit-parallel fashion we could remove that bottle-beck.

Supported by Austrian Science Fund (FWF) project W1255-N23 and the Inst. of Formal Methods and Verification, Johannes Kepler University Linz.

## KISSAT SC2022 BULKY

Improved sweeping above is used in all our three versions of KISSAT submitted to the SAT competition 2022. The version "KISSAT SC2022 BULKY" submitted in 2022 inherits most features of version "KISSAT SC2021 SWEEP" [1] submitted in 2021 but includes the following changes:

- added ACIDS [3] branching variable heuristics (disabled)
- added CHB [4] variable branching heuristic (but disabled by default) inspired by the success of 'kissat_mab' [5]
- faster randomization of phases in the Kitten sub-solver
- literal flipping for faster refinement during sweeping
- disabled priority queue for variable elimination (elimination attempts follow the given fixed variable order)
- disabled by default reusing the trail during restarts
- disabled by default hyper ternary resolution
- initial local search through propagation (similar to "warmup" runs of Donald Knuth [6] and how local search is initialized in "ReasonLS" solvers by Shaowei Cai [7])
- actual watch replacement of true literals during unit propagation instead of just updating the blocking literal (as suggested by Norbert Manthey [8])
- fixed clause length and variable occurrences limits during variable elimination instead of dynamically increasing

## KISSAT SC2022 LIGHT AND KISSAT SC2022 HYPER

In order to focus on the most important features of KISSAT, we removed those that did not substantially improve performance on the last three competitions benchmarks. As a result of these experiments we removed the following features:

- autarky reasoning
- eager forward and backward subsumption during variable elimination (global forward subsumption only)
- caching and reusing of minima during local search
- failed literal probing
- transitive reduction of the binary implication graph
- eager subsumption of recently learned clauses
- XOR gate extraction during variable elimination

- delaying of inprocessing functions based on formula size
- vivification of *irredundant* clauses
- keeping untried elimination, backbone and vivification candidates for next inprocessing round (removed options)
- initial focused mode phase limited only by conflicts now (not as before also by ticks)

The light version also removes hyper binary resolution, enabling the use of more variables ($2^{29}-1$ instead of $2^{28}-1$).

## GIMSATUL SC2022

Our new SAT solver GIMSATUL is a parallel multi-threaded SAT solver written from scratch in C in six weeks. Its core engine follows the architecture of "KISSAT SC2022 LIGHT", even though it is missing non-chronological backtracking, on-the-fly subsumption, advanced shrinking, binary implication graph backbones, advanced definition extraction and sweeping.

The main new feature is to aggressively exchange learned clauses by sharing and reference counting instead of copying, reviving an old line of research. The solver is built on top of pthreads, but also uses C11 atomic operations as well as several lock-less fast-paths. For original clauses this already gives substantial memory savings which extends to learned clauses too and allows to generate compact DRUP proofs.

The simplification procedure implements bounded variable elimination, subsumption and equivalent literal substitution and is run up-front as preprocessing in single threaded mode and further in regular intervals after synchronizing all threads and handing over control and clauses to one single simplification thread. During search each solver thread also performs inprocessing in form of vivification and failed literal probing.

References to learned clauses of low glucose level (LBD) are immediately put in thread local pools to be exported. All exporting and importing thread combinations have exactly one pool and each pool has several slots ordered by glucose level. Threads import clauses from the slots of their pool of a randomly chosen thread, prioritized by glucose level. Except for units, which are always eagerly and completely imported, at most one clause is imported before making a decision.

For more details on "GIMSATUL SC2022" and particularly extensive experimental results on scalability and other aspects of our new solver we refer to our presentation at the workshop on *Pragmatics of SAT (POS'22)* [9].

## ISASAT

This is the first submission of the fully verified SAT solver ISASAT to the SAT Competition (and to the best of our knowledge, the first submission of a fully verified SAT solver). Since the submission to the EDA Challenge 2021 [10], we implemented only few new features, namely pure literal detection and resolution and deduplication of binary clauses. The first features is our first non-equivalence preserving transformation.

The main work went into updating the Isabelle version we are using and the version of the LLVM-based library of synthesis [11]. With the update to Isabelle2021-1, synthesis started to take *hours* for even the simplest function, so we had to replace the formalization of the solver state by a proper structure and reorganize our entire development around that.

Unrelated to our verification, we added proof logging to our solver. Remark that there is absolutely no proof of correctness of the generated proofs: The correctness theorem does not mention the proofs (and it happened during development that we forgot to print some of them leading to incorrect proofs–but the result was always correct).

The submitted sources of the SAT solver contain only files generated by Isabelle in the intermediate representation used by clang. For the complete sources (including correctness theorem and comments), refer to the `sc2022` tag in the IsaFOL repository https://bitbucket.org/isafol/isafol/src/sc2022/.

## LICENSE

All our solvers are licensed under an MIT license, with GIMSATUL available at https://github.com/arminbiere/gimsatul KISSAT at https://github.com/arminbiere/kissat and further ISASAT at https://m-fleury.github.io/isasat/isasat-release/.

## REFERENCES


[1] A. Biere, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021," in *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, ser. Dept. of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2021-1. Univ. of Helsinki, 2021, pp. 10–13.

[2] J. P. M. Silva and I. Lynce, "On improving MUS extraction algorithms," in *Theory and Applications of Satisfiability Testing - 14th International Conference, SAT 2011*, ser. LNCS, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 159–173.

[3] A. Biere and A. Fröhlich, "Evaluating CDCL variable scoring schemes," in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, SAT 2015*, ser. LNCS, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 405–422.

[4] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Exponential recency weighted average branching heuristic for SAT solvers," in *Proc. 13th AAAI Conf. on Artificial Intelligence, AAAI 2016*, D. Schuurmans and M. P. Wellman, Eds. AAAI Press, 2016, pp. 3434–3440.

[5] M. S. Cherif, D. Habet, and C. Terrioux, "Combining VSIDS and CHB using restarts in SAT," in *27th International Conference on Principles and Practice of Constraint Programming, CP 2021*, ser. LIPIcs, L. D. Michel, Ed., vol. 210. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:19.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*, 1st ed. Addison-Wesley Professional, 2015.

[7] S. Cai, C. Luo, X. Zhang, and J. Zhang, "Improving local search for structured SAT formulas via unit propagation based construct and cut initialization (short paper)," in *27th International Conference on Principles and Practice of Constraint Programming, CP 2021*, ser. LIPIcs, L. D. Michel, Ed., vol. 210. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 5:1–5:10.

[8] N. Manthey, "CaDiCaL modification – Watch Sat," in *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2021-1. University of Helsinki, 2021, pp. 28–29.

[9] M. Fleury and A. Biere, "Scalable proof-producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses," in *Pragmatics of SAT 2022*, D. L. Berre and M. Järvisalo, Eds., 2022.

[10] M. Fleury, "CaDiCaL, Kissat, Paracooba entering the EDA Challenge 2021," 2021, submitted to the EDA Challenge 2021.

[11] P. Lammich, "Generating verified LLVM from Isabelle/HOL," in *10th International Conference on Interactive Theorem Proving, ITP 2019*, ser. LIPIcs, J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 22:1–22:19.

# BREAKID-KISSAT and BREAKID-KISSAT-WITHUNSATCERTIFICATES in SAT Competition 2022 (System Description)

Bart Bogaerts
*Vrije Universiteit Brussel*
Brussels, Belgium
ORCID: 0000-0003-3460-4251

Jakob Nordström
*University of Copenhagen*
Copenhagen, Denmark
and *Lund University*
Lund, Sweden
ORCID: 0000-0002-2700-4285

Andy Oertel
*Lund University*
Lund, Sweden
email address or ORCID

Çağrı Uluç Yıldırımoğlu
*Vrije Universiteit Brussel*
Brussels, Belgium
cagriuluc96@gmail.com

***Abstract*—BREAKID-KISSAT and BREAKID-KISSAT-WITHUNSATCERTIFICATES combine the symmetry breaking preprocessor BREAKID with the SAT solver KISSAT.**

## I. INTRODUCTION

For several years, participation in the main tracks of the SAT competition has required solvers to output proofs in the DRAT format [11]. Unfortunately, this means that several state-of-the-art solving techniques are de facto excluded from participation in these tracks. One prime example of such a technique is *symmetry breaking*: while for limited types of symmetries, breaking constraints can be derived in DRAT [8], for the general case, no techniques are known. Our two solvers employ symmetry breaking and hence only participate in the no-limit track of the competition.

Our solvers are a combination of the symmetry breaker BREAKID [3] and the SAT solver KISSAT [10].

The difference between BREAKID-KISSAT and BREAKID-KISSAT-WITHUNSATCERTIFICATES is that BREAKID-KISSAT-WITHUNSATCERTIFICATES *does* provide UNSAT certificates. For the reason mentioned above, the certificates are not in the DRAT format, but in the VERIPB format and can be verified by VERIPB. VERIPB [4]–[7] was originally designed as a proof checker for pseudo-Boolean satisfiability and was recently extended to *pseudo-Boolean optimization* [1], making it not just a viable candidate for certification of SAT techniques, but also for MaxSAT. The underlying proof format is a strict generalization of DRAT. Morover, since it is based on the cutting planes proof system [2], it also naturally facilitates proof logging for advanced techniques such as XOR and cardinality reasoning [7].

## II. MAIN TECHNIQUES

The workflow of our two solvers is as follows:

- First, the instance to a colored graph in such a way that syntactic symmetries of the problem correspond to automorphisms of the graph.
- Next, BREAKID uses SAUCY [9] to detect automorphisms of the constructed graph.
- Next, BREAKID optimizes the set of detected symmetries to ensure complete breaking of certain subgroups. For each of the resulting symmetries, it creates symmetry breaking clauses.
- Subsequently, the original instance, together with the symmetries is passed to KISSAT, which solves the resulting instnace.

For BREAKID-KISSAT-WITHUNSATCERTIFICATES, BREAKID and KISSAT each produce a part of the resulting proof.

## III. AVAILABILITY

The source code of BREAKID is available at https://bitbucket.org/krr/breakid/src. Our modified version of KISSAT to output proofs in the VERIPB format rather than DRAT is available at https://gitlab.com/MIAOresearch/kissat_fork.

## IV. ACKNOWLEDGEMENT

We would like to thank everyone who contributed to SAUCY, BREAKID, and KISSAT for their efforts and for making their tools publicly available.

## REFERENCES

[1] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of AAAI*, 2022. accepted.

[2] William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.

[3] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016.

[4] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[5] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[6] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[7] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[8] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, pages 591–606. Springer, August 2015.

[9] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *LNCS*, pages 113–127. Springer, 2010.

[10] Kissat SAT solver. http://fmv.jku.at/kissat/.

[11] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th Internatjuional Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

# Kissat_MAB: Upper Confidence Bound Strategies to Combine VSIDS and CHB

Mohamed Sami Cherif, Djamal Habet and Cyril Terrioux
Aix-Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
{mohamed-sami.cherif, djamal.habet, cyril.terrioux}@univ-amu.fr

***Abstract*—This document describes two solvers, Kissat_MAB_UCB and Kissat_MAB_MOSS, submitted to the 2022 SAT competition. The solvers are based on Kissat, the winner of the 2020 SAT competition, which we augment with a Multi-Armed Bandit (MAB) framework. The submitted solvers rely on two different Upper Confidence Bound strategies, namely UCB1 and MOSS, to adaptively choose a relevant heuristic among VSIDS and CHB at each restart.**

***Index Terms*—Branching Heuristics, Multi-Armed Bandit, Upper Confidence Bound**

## I. Introduction

Conflict Driven Clause Learning (CDCL) [12] solvers are known to be efficient on structured instances and manage to solve ones with a large number of variables and clauses. An important component in such solvers is the branching heuristic which picks the next variable to branch on. The Variable State Independent Decaying Sum (VSIDS) [13] has been the dominant heuristic since its introduction two decades ago. Recently, Liang and al. devised a new heuristic for SAT, called Conflict History-Based (CHB) branching heuristic [10], and showed that it is competitive with VSIDS. In the last years, VSIDS and CHB have dominated the heuristics landscape as practically all the CDCL solvers presented in recent SAT competitions and races incorporate a variant of one of them.

Recent research has shown the interest of machine learning in designing efficient search heuristics for SAT [9]–[11] as well as for other decision problems [6], [14]–[16]. One of the main challenges is defining a heuristic which can have high performance on any considered instance. Indeed, a heuristic can perform very well on a family of instances while failing drastically on another. To this end, we use reinforcement learning under the Multi-Armed Bandit (MAB) framework to pick an adequate heuristic among CHB and VSIDS for each instance [7]. The MAB takes advantage of the restart mechanism in modern CDCL algorithms to evaluate each heuristic and choose the best one accordingly. The MAB uses Upper Confidence Bound [1] strategies to select an arm at each restart.

## II. Combining VSIDS and CHB through MAB

Let $A = \{VSIDS, CHB\}$ be the set of arms for the MAB containing different candidate heuristics. The proposed framework selects a heuristic $a \in A$ at each restart of the backtracking algorithm. To choose an arm, MAB relies on a reward function calculated during each run to estimate the performance of the chosen branching heuristic. We choose a reward function that estimates the ability of a heuristic to reach conflicts quickly and efficiently. If $t$ denotes the current run, the reward of arm $a \in A$ is calculated as follows:

$$r_t(a) = \frac{log_2(decisions_t)}{decidedVars_t}$$

$decisions_t$ and $decidedVars_t$ respectively denote the number of decisions and the number of decision variables, i.e. variables which were branched on at least once, in the run $t$. This reward function is adapted from the explored sub-tree measure introduced in [14].

## III. Upper Confidence Bound Strategies for MAB

We use two upper confidence bound strategies to select an arm at each restart, namely UCB1 [3] and MOSS [2]. The following parameters are maintained for each candidate arm $a \in A$:

- $n_t(a)$ is the number of times the arm $a$ is selected during the $t$ runs,
- $\widehat{r}_t(a)$ is the empirical mean of the rewards of arm $a$ over the $t$ runs.

UCB1 and MOSS select the arm $a \in A$ that respectively maximizes $UCB(a)$ and $MOSS(a)$, defined as follows :

$$UCB(a) = \widehat{r}_t(a) + c\sqrt{\frac{ln(t)}{n_t(a)}}$$

$$MOSS(a) = \widehat{r}_t(a) + c\sqrt{\frac{1}{n_t(a)} ln\left(max\left(\frac{t}{K.n_t(a)}, 1\right)\right)}$$

The left-side term of $UCB(a)$ and $MOSS(a)$ is similar and aims to put emphasis on arms that received the highest rewards. Conversely, the right-side term ensures the exploration of underused arms. The parameter $c$ can help to appropriately balance the interchange between the exploitation and exploration phases in the MAB framework.

## IV. Implementation and Submissions

We implement this MAB framework in Kissat [5], which won first place in the main track of the SAT Competition 2020. Note that this solver is a condensed and improved reimplementation of the reference and competitive solver CaDiCaL [4] in C. We submit two solvers to the Main and Anniversary tracks of the 2022 SAT competition. The

first, called Kissat_MAB_UCB, uses the UCB1 strategy and corresponds to the solver Kissat_MAB [8] which we submitted to the 2021 SAT competition. This solver turned out to be highly competitive as it won the Main and SAT Main tracks of the previous competition. Our study in [7] also shows that the MOSS strategy can be highly competitive with respect to UCB1 as it outperformed it in terms of the number of solved instances and solving time on the 2018, 2019 and 2020 benchmarks. Therefore, we submit our second solver, called Kissat_MAB_MOSS, which relies on the MOSS strategy to choose a heuristic at each restart.

Note that we maintain the VSIDS variant already implemented in Kissat which is similar to Chaff's where all analyzed variables are bumped after every conflict [13]. We also augment the solver with the heuristic CHB as specified in [10]. In addition, we set the parameter $c$ to 2. The rewards are initialized by launching each heuristic once during the first restarts. It is important to note that the only modified components of the solver are the decision component and the restart component, i.e., all the other components as well as the default parameters of the solver are left untouched.

## REFERENCES


[1] R. Agrawal. Sample mean based index policies by o (log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995.

[2] J.-Y. Audibert and S. Bubeck. Minimax Policies for Adversarial and Stochastic Bandits. In *COLT 2009 - The 22nd Conference on Learning Theory, Montreal, Quebec, Canada, June 18-21, 2009*, 2009.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[4] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In T. Balyo, M. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[5] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[6] M. S. Cherif, D. Habet, and C. Terrioux. On the Refinement of Conflict History Search Through Multi-Armed Bandit. In M. Alamaniotis and S. Pan, editors, *Proceedings of 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 264–271. IEEE, 2020.

[7] M. S. Cherif, D. Habet, and C. Terrioux. Combining VSIDS and CHB Using Restarts in SAT. In L. D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[8] M. S. Cherif, D. Habet, and C. Terrioux. Kissat_MAB: Combining VSIDS and CHB through Multi-Armed Bandit. In *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Series of Publications B*, page 15. University of Helsinki, 2021.

[9] V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro. Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. *CoRR*, 2019.

[10] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3434–3440, 2016.

[11] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140, 2016.

[12] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

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

[14] A. Paparrizou and H. Wattez. Perturbing Branching Heuristics in Constraint Solving. In H. Simonis, editor, *Principles and Practice of Constraint Programming*, pages 496–513, Cham, 2020. Springer International Publishing.

[15] H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou, and S. Tabary. Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. In *Proceedings of the European Conference on Artificial Intelligence*, 2020.

[16] W. Xia and R. H. C. Yap. Learning Robust Search Strategies Using a Bandit-Based Approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 6657–6665, 2018.

# CDCL Solvers based on Bounded Exploration and the Glue Bumping method

Md Solimul Chowdhury
*School of Computer Science*
*Carnegie Mellon University*
Pittsburgh, Pennsylvania, USA
mdsolimc@cs.cmu.edu

***Abstract*—This document describes 5 CDCL SAT solvers: kissat_mab_gb, ekissat_mab_be-v1, ekissat_mab_be-v2 and kissat_mab_gb_be and cadical_hack_gb which are entering to the SAT Competition-2022. These solvers are based on the following 2 ideas: 1) Bounded randomized exploration amid conflict depression phases and 2) Activity score bumping of variables that appear in the glue clauses.**

## I. Bounded Exploration amid a CD Phase

This approach is based on our observation that search in Conflict Directed Clause Learning (CDCL) entails clear patterns of bursts of conflicts followed by longer phases of *conflict depression (CD)* [1]. During a CD phase, for a consecutive number of decisions, a CDCL solver is unable to generate conflicts, from which the search could learn clauses to prune the search space. To correct the course of such a search, we propose to use random exploration to combat conflict depression. In this approach, when the search enters into a *substantially long CD phase*, instead of using the currently active decision heuristic, we employ a uniform random strategy for selecting decision variables. The goal of this random exploration is to find conflicts amid a substantially long CD phase, in which the currently active decision heuristic is unable to find a conflict. This random selection continues, until the search finds a conflict or takes a maximum of $s > 0$ random steps. We call this approach *bounded exploration* (BE).

Fig. 1 shows how this approach works.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| heu | heu | heu | heu | heu | heu | heu | heu | heu | heu | heu | heu | heu | rand | rand | rand | rand | heu | heu | heu |

Fig. 1: Assume that a CDCL solver is running a given instance. This figure shows 20 consecutive decisions taken by that solver. The top row shows decision indexes which starts at 0 and ends at 19. In the second row, the grey cells depict decisions with no conflict and green cells depict decisions with non-zero conflicts. For a decision, the text inside a colored cell of the bottom row denotes type of decision strategy (**heu**: heuristic decision, **rand**: random decision) used at that decision. In this search snapshot, a long CD phase starts at the 5th decision. Amid this long CD phase, the search decides to perform random decisions from decision 13. At decision 16, with a random decision the search finds a conflict. This results in the end of the current CD phase.

## II. Glue Variable Bumping

Let a CDCL SAT solver $M$ is running a given SAT instance $\mathcal{F}$ and the current state of the search is $S$. We call the variables that appeared in at least one glue clause up to the current state $S$ *Glue Variables*. We design a structure-aware variable score bumping method named *Glue Bumping* (GB) [2], based on the notion of *glue centrality (gc)* of glue variables. Given a glue variable $v_g$, glue centrality of $v_g$ dynamically measures the fraction of the glue clauses in which $v_g$ appears, until the current state of the search. Mathematically, the glue centrality of $v_g$, $gc(v_g)$ is defined as follows:

$$gc(v_g) \leftarrow \frac{gl(v_g)}{ng}$$

, where $ng$ is the total number of glue clauses generated by the search so far. $gl(v_g)$ is the glue level of $v_g$, a count of glue clauses in which $v_g$ appears, with $gl(v_g) \leq ng$.

### A. The GB Method

The GB method modifies a CDCL SAT solver $M$ by adding two procedures to it, named *Increase Glue Level* and *Bump Glue Variable*, which are called at different states of the search. We denote by $M^{gb}$ the GB extension of the solver $M$.

**Increase Glue Level:** Whenever $M^{gb}$ learns a new glue clause $g$, before making an assignment with the first UIP variable that appears in $g$, it invokes this procedure. For each variable $v_g$ in $g$, its glue level, $gl(v_g)$ is increased by 1.

**Bump Glue Variable:** This procedure bumps a glue variable $v_g$, which has just been unassigned by backtracking. First a bumping *factor (bf)* is computed as follows:

$$bf \leftarrow activity(v_g) * gc(v_g)$$

, where $activity(v_g)$ is the current activity score of the variable $v_g$ and $gc(v_g)$ is the glue centrality of $v_g$. Finally, the activity score of $v_g$, $activity(v_g)$ is bumped as follows:

$$activity(v_g) \leftarrow activity(v_g) + bf$$

## III. Solvers Description

We have submitted five CDCL SAT solvers to the SAT Competition-2022, which are based on combinations of the two approaches described in the previous sections. Our solvers are implemented on top of the solver kissat_mab (winner of

SAT competition-2022) [3] and CaDiCaL1.4.1 (base solver for the CaDiCaL hack track) [4]. In the following, we describe our solvers:

*a)* ***kissat_gb****:* This solver implements the GB method on top of kissat_mab. kissat_mab employs three branching heuristics: VSIDS, CHB and VMTF. In kissat_mab, the GB scheme is kept active only when VSIDS and CHB are active.

*b)* ***ekissat_be_v1****:* The solver ekissat_be_v1 implements the BE strategy on top of kissat_mab, only when VSIDS and CHB are active.

*c)* ***ekissat_be_v2****:* This solver same as ekissat_be_v1, except that ekissat_be_v2 does not perform any exploration, if a small fraction (10%) of variables remains to be assigned.

*d)* ***ekissat_be_gb****:* This system implements the GB method on the top of ekissat_be_v1.

*e)* ***cadical_hack_gb****:* This solver implements the GB method on top of cadical-rel-1.4.1.

## REFERENCES


[1] Md Solimul Chowdhury and Martin Müller and Jia You, Guiding CDCL SAT Search via Random Exploration amid Conflict Depression. In Proceedings of AAAI-2020:1428-1435.

[2] Md. Solimul Chowdhury, Martin Müller, Jia-Huai You, Exploiting Glue Clauses to Design Effective CDCL Branching Heuristics. In Proceedings of CP 2019: 126-143.

[3] Mohamed Sami Cherif, Djamal Habet and Cyril Terrioux. Kissat MAB: Combining VSIDS and CHB through Multi-Armed Bandit, In Proceedings of SAT Competition 2021:15-16.

[4] CaDiCal 1.4.1, https://github.com/arminbiere/cadical/tree/rel-1.4.1, access date: 15-May-2022.

# Kissat-ELS and its friends at the SAT Competition 2022

Fei Geng, Lei Yan and ShuCheng Zhang
*TCS Lab, Huawei Technologies, Beijing, China*
{gengfei12, david.yan, zhangshucheng}@huawei.com

***Abstract*—This paper describes our sat solvers are for SAT Competition 2022. Our solvers based on Kissat with enhanced local search (Kissat-ELS), including four different version submitted to SAT Competition 2022. All the four solvers participate in the Main Track, while the Kissat-ELS-v1 and Kissat-ELS-v2 also participate in the Anniversary Track.**

## I. Introduction

Four different versions of our solvers are all based on Kissat-sc2021-sweep [1], which wins the second place in Main Track in SAT Competition 2021. Local search procedure in Kissat is a subroutine of rephase, and the solution of local search will be saved in decision phases. Improving local search not only provides help for solving random instances, but also affects the search tree in stable mode of Kissat. We experimented with a number of strategies to enhance local search to improve the solver's ability to solve satisfiable instances. The main methods to imporve local search are as follows:

- Extend the execution time of local search.
- Use Unit Propagation to get a better initial solution for local search algorithm.
- Import CCAnr [2] algorithm to replace the original local search algorithm in Kissat.

## II. The Effect of Local Search Effort

In order to solve more satisfiable instances by stochastic local search, it is effective to simply extend the execution time of local search. We uniformly sampled 360 instances from the competition benchmarks of the past two decades as our training benchmark. We experimented with the change of the number of solved instances as the local search effort increases and got some interesting experimental conclusions. In Fig.1, x-axis represents the parameter in Kissat (with *--walkeffort*) which control the execution time of local search, y-axis represents the number of solved instances, the dark red and light red areas represent the number of solved SAT instances by local search and CDCL, respectively. The green area stands for UNSAT part.

With the increase in local search effort, the local search solved number increases while the CDCL solved SAT parts decreases, due to the fact that some instances which should have been solved by CDCL are replaced by local search. Unexpectedly, the UNSAT part increases slightly and then decreases, and the maximum number of UNSAT instances solved reached at the same time as the total solved number.

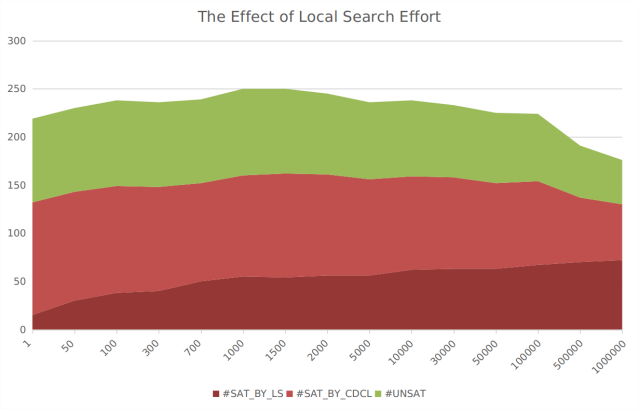



Fig. 1. The Effect of Local Search Effort on SAT Solving.

We conjecture that rephase by local search has some positive effects on solving UNSAT cases. We think that the optimal parameters are positively correlated with the number of random instances in training benchmark, and the relation should be a unimodal function.

## III. Unit Propagation for Local Search Initialization

The initial value is significant for local search engine, different initialization will lead to different search subspaces for local search. Kissat imports decision phases or takes over the previous local search result as the initial value. Inspired by *lstech_maple* [3], we found that unit propagation might be a more suitable initialization method for local search, which would cause less conflict and be more closer to the satisfiable result. During implementation, we randomly pick an unassigned variable and assign it by decision phase, then do unit propagation similar to Kissat search mode propagation without dealing with conflicts until all active variables are assigned.

## IV. Kissat-ELS-v1 and Kissat-ELS-v2

In Kissat-ELS-v1, we set the configuration to:

- *--walkeffort=1000, --vivifyeffort=1, --sweep=false*

In Kissat-ELS-v2, we implement the unit propagation without dealing with conflict as *--walkup* option, and set *vivifyeffort* and *sweep* to default.

## V. KISSAT-ELS-V3 AND KISSAT-ELS-V4

We import the ccnr part from $lstech\_maple$ [3] as the base version of CCAnr's implementation, and adapt the ccnr part to be compatible with Kissat. Kissat will set some variables to inactive and fix these variables during searching, so we use a filter to delete those variables and some clauses which will be passed to ccnr engine. Different from the original implementation, we only choose irredundant clauses as clause database to ccnr algorithm instead of original clauses and some learnt clauses. We also made some minor modifications, such as the ccnr reward ratio (the conflicts from ccnr to modify the VSIDS score in CDCL). The major difference between Kissat-ELS-v3 and Kissat-ELS-v4 is that the former use the decision phases as initial values for the local search engine, while the latter uses the unit propagate initialization described before.

## REFERENCES


[1] A. Biere, M. Fleury and M. Heisinger. CADICAL, KISSAT, PARACOOBA Entering the SAT Competition 2021. in Proc. of SAT Competition 2021 - Solver and Benchmark Descriptions.

[2] Cai S, Luo C, Su K. CCAnr: A configuration checking based local search solver for non-random satisfiability. International Conference on Theory and Applications of Satisfiability Testing. Springer, Cham, 2015: 1-8.

[3] Zhang X, Cai S, Chen Z. Improving CDCL via Local Search. in Proc. of SAT Competition 2021 - Solver and Benchmark Descriptions.

# Combining Hybrid Walking Strategy with Kissat_MAB, CaDiCaL, and LStech-Maple

Jiongzhi Zheng[1] Kun He[1] Zhuo Chen[1] Jianrong Zhou[1] Chu-Min Li[†2]
[1]School of Computer Science and Technology,
Huazhong University of Science and Technology, China
[2]MIS, Université de Picardie Jules Verne, France
[†]Chu-Min Li does not participate in solvers LStech-Maple-FPS and CaDiCaL-HyWalk.

***Abstract*—This document describes our five SAT solvers, LStech-Maple-BandSAT, LStech-Maple-FPS, LStech-Maple-HyWalk, Kissat_MAB-HyWalk, and CaDiCaL-HyWalk, submitted to the SAT Competition 2022. CaDiCaL-HyWalk is submitted to the hack track, and the others are submitted to the main track.**

## I. Introduction

Recently we propose two kinds of local search algorithms for MaxSAT problems, called BandMaxSAT [1] and FPS [2]. We propose their two variants for the SAT problem, and replace the CCAnr algorithm [3] with them in the LStech-Maple solver [4], which participated in SAT Competition 2021. The resulting solvers are called LStech-Maple-BandSAT and LStech-Maple-FPS.

Moreover, we propose a Hybrid Walking (HyWalk) strategy that combines BandSAT, FPS, and some other local search algorithms with different random walking or say local optimal escaping strategies, and obtain another solver LStech-Maple-HyWalk. In LStech-Maple-HyWalk, a decision tree is applied to help the solver decide which walking strategy can solve the input instance well.

Finally, we apply a similar hybrid mechanism to the Kissat_MAB [5] and CaDiCaL solvers, and yield Kissat_MAB-HyWalk and CaDiCaL-HyWalk.

## II. LStech-Maple-BandSAT

BandSAT is a variant of BandMaxSAT [1]. It associates a multi-armed bandit with all the clauses. Each arm corresponds to a clause. BandSAT uses the same method as CCAnr to select the variable to be flipped when the algorithm does not reach a local optimum (the process before updating the clause weights in CCAnr). When falling into a local optimum, BandSAT selects to pull an arm that corresponds to a falsified clause, which indicates satisfying the clause by flipping the variable with the highest score in the clause. The bandit model in BandSAT can help the algorithm select a better search direction than CCAnr, which randomly selects the clause to be satisfied when falling into a local optimum. When selecting the are to be pulled, BandSAT first randomly samples 20 arms which are all corresponding to falsified clauses, then selects the arm according to the Upper Confidence Bound of the sampled arms. We also apply the delayed reward method in BandMaxSAT to update the estimated values of the arms.

Replacing CCAnr in LStech-Maple with BandSAT results in LStech-Maple-BandSAT.

## III. LStech-Maple-FPS

The Farsighted Probabilistic Sampling (FPS) [2] strategy combines the look-ahead strategy with the probabilistic sampling strategy in an effective way. FPS for SAT also applies the same method as CCAnr when the algorithm does not reach a local optimum for the CCAnr. When a local optimum is reached. FPS first randomly samples 10 falsified clauses, then tries to look-ahead from a random variable of each sampled clause, to check whether flipping a pair of variables can improve the current solution. If FPS fails to improve the current solution by flipping a pair of variables, it will select to flip the best among the best sampled single variable and the best sampled pair of variables.

With the help of the look-ahead strategy, FPS can improve the local optima for the CCAnr, so as to find higher-quality solutions. While the probabilistic sampling strategy can help the algorithm improve its efficiency.

Replacing CCAnr in LStech-Maple with FPS results in LStech-Maple-FPS.

## IV. LStech-Maple-HyWalk

Different walking strategies such as those in BandSAT and FPS are suitable for different kinds of instances. Therefore, to help the LStech-Maple solver decide to select an appropriate walking strategy to explore the solution space, we use a decision tree that trains on all the instances from the main tracks of the last three years of SAT Competition. The features include the number of variables $V$, the number of clauses $C$, the ratio of $C$ to $V$, the minimum number of clause lengths, the average number of clause lengths, and the maximum number of clause lengths.

LStech-Maple-HyWalk contains a total of six walking strategies. They are CCAnr, BandSAT, FPS, SimpleWalk (first randomly sample 10 falsified clauses, then randomly sample 5 variables in each sampled clause, finally select the variable with the highest score), FPS+SimpleWalk (FS, first randomly

sample clauses and variables as SimpleWalk does, then look-ahead from the sampled variable with the highest score in each sampled clause), and BandSAT+FPS (BF, first select the falsified clause to be satisfied as BandSAT does, then randomly sample 10 variables in the selected clause to look-ahead).

## V. KISSAT_MAB-HYWALK AND CADICAL-HYWALK

Kissat_MAB-HyWalk and CaDiCaL-HyWalk use a similar hybrid strategy as LStech-Maple-HyWalk. However, these two solvers use very simple basic walking strategies. The basic idea is also very simple, that is, walking is useful for some instances but not for all the instances. So there are some instances need not the walking process. Others need to spend more resources on walking. Therefore, Kissat_MAB-HyWalk combines three methods, i.e., Kissat_MAB itself, Kissat_MAB without the walking phase, and Kissat_MAB with the rounds of the walking phase multiplied by 5. CaDiCaL-HyWalk is the same. CaDiCaL-HyWalk only changes the *walk.cpp* file of CaDiCaL 1.4.1.

## REFERENCES


[1] J. Zheng, J. Zhou, K. He, "Farsighted Probabilistic Sampling based Local Search for (Weighted) Partial MaxSAT," arXiv preprint arXiv:2108.09988, 2021.

[2] J. Zheng, K. He, J. Zhou, Y. Jin, C. M. Li, F. Manyà, "BandMaxSAT: A Local Search MaxSAT Solver with Multi-armed Bandit," IJCAI 2022.

[3] S. Cai, C. Luo, K. Su, "CCAnr: A Configuration Checking Based Local Search Solver for Non-random Satisfiability," SAT 2015: 1-8.

[4] X. Zhang, S. Cai, Z. Chen, "Improving CDCL via Local Search," SAT COMPETITION 2021, 2021: 42.

[5] M. S. Cherif, D. Habet, C. Terrioux, "Kissat MAB: Combining VSIDS and CHB through Multi-Armed Bandit," SAT COMPETITION 2021, 2021: 15.

# Descriptions for CadicalReorder SAT Solver

Junhua Huang
Xiamen University

Hui-Ling Zhen, Wanqian Luo, Mingxuan Yuan
Noah's Ark Lab, Huawei, China

***Abstract*—Here is the brief description for modified CaDiCal solver which submitted to the cadical hack track, according to the necessary information for submission.**

## I. Introduction

This is the brief description for modified CaDiCal SAT solver, according to the necessary information for submission.

### *A. Author Information*

There are three main authors for the modifications of SAT solvers:

(a) Junhua Huang, Xiamen University, China.
(b) Hui-Ling Zhen, Noah's Ark Lab, Huawei, Hong Kong, China
(c) Wanqian Luo, Noah's Ark Lab, Huawei, Hong Kong, China
(d) Mingxuan Yuan, Noah's Ark Lab, Huawei, Hong Kong, China

### *B. Descriptions for Algorithms*

We focus on the different searching strategies and heuristic algorithms for SAT and UNSAT instances. In this solver, we have considered the instance structure from clustering view, based on which we reorder the initial branch queue. The corresponding reference is given in Refs. [1-2]. This modification is based on our observations on the structural characteristics's effect on CDCL.

We also have tried different scoring method in UNSAT instances, such as bandit score and bump score in branching for literals. Furthermore, we have a more aggressive conflict record method for restart, backtrack and propagate. However, considering the limit from the characteristics numbers, this performance is unstable and we will not provide these codes and descriptions here.

## References

[1] Hireche, C., Drias, H. and Moulai, H., 2020. Grid based clustering for satisfiability solving. Applied Soft Computing, 88, p.106069.
[2] Boltenhagen, Mathilde, Thomas Chatain, and Josep Carmona. "Optimized SAT encoding of conformance checking artefacts." Computing 103.1 (2021): 29-50.

# MapleLCMDistChronoBT-DL-v3, the duplicate learnts heuristic-aided solver at the SAT Competition 2022

Stepan Kochemazov, Oleg Zaikin, Victor Kondratiev and Alexander Semenov
Email: veinamond@gmail.com, zaikin.icc@gmail.com, vikseko@gmail.com, axelvonemes@gmail.com

***Abstract*—This document describes the `MapleLCMDistChronoBT-DL-v3` solver which is based on the SAT Competition 2018 winner, the `MapleLCMDistChronoBT` solver, augmented with duplicate learnts heuristic.**

## I. Duplicate Learnts

During the CDCL inference, some learnt clauses can be generated multiple times. It is reasonable to assume that they deserve special attention. In particular, the simple rule for their processing can look as follows: if a learnt clause was repeated at least $k$ times ($k \geq 2$) during the derivation, then this clause should be permanently added to the conflict database. It can be naturally implemented for solvers based on `COMiniSatPS` [1], since they store learnt clauses in three tiers: *Core*, *Tier2* and *Local*, where the learnts in *Core* are not subject for `reduceDB`-like procedures. Thus we basically can put duplicate learnts into *Core* when they satisfy the conditions outlined below.

In the submitted solver we track the appearances of duplicate learnts using a hashtable-like data structure and process them based on several parameters. The hashtable is implemented on top of C++ 11 `unordered_map` associative container. The goal of parameters is to ensure that the hashtable does not eat too much memory, that the learnt clauses are filtered based on their LBD, and that the learnts repeated a prespecified number of times are added to *Tier2*/*Core*.

- `lbd-limit` – only learnt clauses with `lbd` $\leq$ `lbd-limit` are screened for duplicates.
- `min-dup-app` – learnt clauses that repeated `min-dup-app` times are put into *Tier2*, and the ones repeated `min-dup-app`+1 times – to *Core* tier.
- `dupdb-init` – the initial maximal number of entries in the duplicate learnts hashtable.

The duplicates database is purged as soon as its size exceeds `dupdb-init`. Only the entries corresponding to learnt clauses repeated at least `min-dup-app` times are preserved. With each purge, the value of `dupdb-init` is increased by $10\%$.

Additionally, we limit `core_lbd_cut` parameter of the solver to 2 since duplicate learnts can provide a lot of additional clauses to store in *Core*.

## II. MapleLCMDistChronoBT-DL-v3 [2]

`MapleLCMDistChronoBT-DL-v3` is based on the SAT Competition 2018 main track winner, `MapleLCMDistChronoBT` [3], which in turn is based on `Maple_LCM_Dist` [4], the successor of `MapleCOMSPS` [5].

The solver employs `lbd-limit`=12, `min-dup-app`=3 (e.g. only learnts repeated 4 times are added to *Core*), and `dupdb-init`=500000. It also uses a deterministic LRB-VSIDS switching strategy: it starts with LRB [5] and switches between LRB and VSIDS [6] each time the number of propagations since the last switch exceeds a specific value. This value starts at 30000000 propagations and is increased by 10% with each switch.

This version of the solver is the same as in SAT Competition 2020 (and SAT Race 2019 with several small typos fixed).

## References

[1] C. Oh, "Between SAT and UNSAT: The fundamental difference in cdcl SAT," in *SAT*, ser. LNCS, vol. 9340, 2015, pp. 307–323.
[2] S. Kochemazov, O. Zaikin, A. Semenov, and V. Kondratiev, "Speeding up CDCL inference with duplicate learnt clauses," in *ECAI*, 2020, pp. 339–346.
[3] A. Nadel and V. Ryvchin, "Chronological backtracking," in *SAT*, ser. LNCS, vol. 10929, 2018, pp. 111–121.
[4] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *IJCAI*, 2017, pp. 703–711.
[5] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *SAT*, ser. LNCS, vol. 9710, 2016, pp. 123–140.
[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01, 2001, pp. 530–535.

# HKIS, HCAD and PAKIS in the SAT Competition 2022

Rodrigue Konan Tchinda[1,2], Clémentin Tayou Djamegni[1]
[1]*University of Dschang, Dschang, Cameroon*
[2]*University of Bamenda, Bamenda, Cameroon*
{rodriguekonanktr, dtayou}@gmail.com

***Abstract*—This document describes the sequential solvers HKIS, HCAD and the parallel solver PAKIS submitted to the SAT Competition 2022.**

## I. HCAD AND HKIS

The solvers HKIS and HCAD [1] are "hacks" of KISSAT and CADICAL [2], [3] respectively that integrate the PSIDS heuristic [4] for choosing the polarities of selected branching variables.

We submitted two versions of the HCAD solver to the SC22, namely HCAD_V1 and HCAD_V2. The source code of HCAD_V1 is the same as the one submitted to the SC21 but we changed the configuration to the following:

- `psids` where the options are: `--psids=1 --target=2 --walk=false` and `--chrono=true`;

As for HCAD_V2, it is built on top of version 1.4.1 of CADICAL. It was submitted to the CADICAL Hack Track of the SC22 with the following configuration:

- `default` where the options are: `--target=0 --walk=false`;

The source code of HKIS is also identical to the one submitted to SC21, but for this year we used the following three configurations:

- `psids` where the options are: `--unsat` and `--psids=true`;
- `sat` where the options are `--sat` and `--walkinitially=true`;
- `unsat` where the options are `--target=1 --walkinitially=true` and `--chrono=true`.

## II. PAKIS

We submitted two versions of the parallel solver PAKIS to the SAT Competition 2022. The first version is identical to the one submitted to the SC 2021 [1] with the only difference that the number of threads has been reduced from 24 to 12. The second version was obtained by replacing the worker solver KISSAT of PAKIS with the KISSAT_MAB [5] solver, winner of the sequential track of SC21. For the latter, we set the number of threads to 24.

## III. ACKNOWLEDGMENTS

We would like to thank the developers of PAINLESS [6], KISSAT, CADICAL [2] and Kissat_MAB [5].

## REFERENCES

[1] R. K. Tchinda and C. T. Djamegni, "HKIS, HCAD, PAKIS and PAINLESS_ExMapleLCMDistChronoBT in the SC21," *SAT COMPETITION 2021*, p. 26, 2021.

[2] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[3] A. Biere, "CaDiCaL at the SAT Race 2019," in *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.

[4] R. K. Tchinda and C. T. Djamegni, "PADC MapleLCMDistChronoBT, PADC Maple LCM Dist and PSIDS MapleLCMDistChronoBT in the SR19," *SAT RACE 2019*, p. 33.

[5] M. S. Cherif, D. Habet, and C. Terrioux, "Kissat MAB: Combining VSIDS and CHB through Multi-Armed Bandit," *SAT COMPETITION 2021*, p. 15, 2021.

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

# MergeSat, Merge-Mallob and Mallob-MergeCadLing

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

***Abstract*—The sequential SAT solver MERGESAT is a fork of the 2018 SAT competition winner, and adds known as well as novel improvements. MERGESAT is setup to simplify merging solver contributions into one solver, to motivate more collaboration among solver developers. MERGESAT has been integrated into the parallel HORDESAT, which in turn is used in MALLOB. Additionally, MERGESAT offers a deterministic parallel mode, with the ability to generate unsatisfiability proofs.**

## I. INTRODUCTION

The CDCL solver MERGESAT is based on the competition winner of 2018, MAPLE_LCM_DIST_CHRONOBT [15], and adds several known techniques, fixes, and some novel ideas around reasoning as well as parallel solving. MERGESAT uses git to combine changes, and comes with continuous integration to simplify extending the solver further.

## II. DEVELOPMENT TENETS

When given a sequential compute resource, the CDCL algorithm [16] is assumed to be the most efficient way to solve SAT. To avoid duplicating implementation effort, MERGESAT is setup to easily incorporate modifications to other solvers. This setup allows to keep up with the state-of-the-art and research. Automated testing as well as extended internal checks and proof validation help to spot merge issues early.

For parallel computing resources, portfolio solvers are assumed to be limited with respect to scalability in proof generation [11]. MERGESAT's parallel variant allows to use search space partitioning in an experimental mode. Partitioning is currently handled via assumption literals, similarly to the *cube-and-conquer* [6] approach. The key difference is that MERGESAT dynamically and recursively re-partitions the search space again if compute resources become available again [8], [9]. The heuristic is to keep the sequential algorithm running as long as possible on the largest possible portion of the search space. Thanks to using assumption literals, the used base-solver does not need to implement *dependency-tracking* [12], as done in PCASSO [10]. Learnt clauses can be shared across all solver instances, and unsatisfiability proofs can be generated as done in parallel portfolio solvers [7].

MERGESAT is not tuned for a specific application or benchmark. Solver additions try to stay as close to the original behavior as possible, and can be enabled by configuration. Behavior-changing modifications are automatically detected.

Most algorithms in MERGESAT can be configured. The parameter specification can be printed to a file, to be used by tools to automatically configure the solver. Furthermore, when using MERGESAT as a library, the parameters can be configured – and tuned – via environment variables.

To improve solver maintenance, the solver is implemented in a deterministic way. Algorithms are limited or switched based on step counters instead of measured run time, as the later is highly platform specific. The parallel execution is based on *barriers* similar to MANYSAT [4], to obtain a deterministic parallel solver execution. Cross-platform determinism is work in progress: MERGESAT already replaces some of the math-library functions like *exp*, to become independent of the implementation differences for different platforms.

While CDCL, as well as variable elimination [2], use resolution as the main reasoning, other simplification techniques exist that do not follow the obvious resolution pattern. *Learnt Clause Minimization* [13] is such an example. Similarly, MERGESAT implements *look-ahead* [5], which can be used to create search decisions, as well as to partition the formula. The implemented look-ahead uses double-look ahead for the second assessed polarity, as ternary clauses are collected after propagating the first polarity.

The sequential and parallel MERGESAT can emit unsatisfiability proofs in the *DRAT* format [17]. In both modes, the generation of the proof can be verified during runtime.

The sequential solver supports incremental solving with assumption literals. Incremental solving is not yet compatible with the search space partitioning, so that the parallel solver falls back to portfolio mode with sharing.

## III. IMPROVEMENTS SINCE COMPETITION 2021 VERSION

Besides the extension to deterministic parallel solving with clause sharing, proof generation and search space partitioning, MERGESAT received some updates in the used heuristics.

The solver CADICAL initially assigns the free variables in order. MERGESAT assigned the smallest free variable first, and the greatest variable next; and then continue in reverse order – as also done in MINISAT 2.2 or GLUCOSE 2.2. This order is caused by the initialization of variable activities to the same value, in combination with the implementation of the used ordering data structure. To stay in order, the activities of the variables are not assigned the same value, but instead are assigned the value $\frac{1000}{x}$ for each variable x.

To get access beyond the usual search and conflict analysis with learnt clause minimization, *necessary assignments* [14] based on the literals of binary clauses can be detected during

decisions on the top level. Heuristically, this search happens for every fourth decision on the top-level.

Changes to the solver are tracked in a *CHANGELOG* file. Updates to this file are enforced via automated checks.

## IV. Submitted Solvers

### A. Sequential Solvers

MergeSAT is submitted in three different configurations.

*1) Default Configuration (4.0-rc1):* This configuration uses the setup as described above.

*2) No Platform (4.0-rc2):* This configuration differs to (4.0-rc1) in the fact that the systems math library implementation is used for the functions *exp* and *log*.

*3) No SLS (4.0-rc3):* This configuration differs to (4.0-rc1) in the fact that the CCnr SLS engine is not used during search. However, *rephasing* [1] is still used.

### B. Parallel Solvers

The more recent variant of MergeSAT has been submitted in a stand-alone fashion as determinism portfolio solver with sharing. Furthermore, variants of HordeSat and Mallob have been submitted, using improvements from MergeSAT, as well as more recent solvers that also use insights from MergeSAT.

The docker image used for the parallel MergeSAT is based on Fedora 36. This version already uses glibc 2.35, which contains the modifications to memory allocation to easily use huge page memory management, which has been shown to improve solver performance [3]. Furthermore, prefetching is disabled in the solver backends for parallel solvers.

*1) Merge-Mallob:* For the cloud-track, the solver Mallob, which is based on HordeSat, has been extended to use the latest variant of MergeSAT. In this configuration, MergeSAT is the only used solving engine.

*2) Mallob-MergeCadLing:* All other backend solvers (CaDiCaL, lingeling, YalSAT) have been bumped to their most recent version. To simplify future updates, these solvers are added via git submodules. Prefetching is not used in the parallel mode, to not overload load the memory subsystem with avoidable memory accesses. The used CaDiCaL backend furthermore received the *watch-sat* modification from the hack-track of the SAT competition 2021.

## V. Availability

The source of MergeSAT is publicly available under the MIT license at https://github.com/conp-solutions/mergesat. The version with the git tag "sat-comp-2022" is used for all MergeSAT-related submissions. The submitted starexec package can be reproduced by running "./scripts/make-starexec.sh" on this commit.

Merge-HordeSat is available under the MIT license at https://github.com/conp-solutions/hordesat. Merge-Mallob is available under the LGPL license at https://github.com/conp-solutions/mallob, with the tag "sat-comp-2022".

The parallel variant of MergeSAT has a few open issues: tuning the default configuration, improving handling of special cases like lazily initializing and synchronizing parallel solvers during incremental solving, as well as combining incremental solving with search space partitioning and proof generation.

## Acknowledgment

The author would like to thank the developers of all predecessors of MergeSAT, and all the authors who contributed the modifications that have been integrated. Furthermore, we thank the ZIH of TU Dresden for making compute resources available that have been used to develop earlier versions of MergeSAT. Adhemerval Zanella made transparent huge pages easily accessible via his modifications to glibc.

## References

[1] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[2] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT 2005*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. Heidelberg: Springer, 2005, pp. 61–75.

[3] J. K. Fichte, N. Manthey, J. Stecklina, and A. Schidler, "Towards faster reasoners by using transparent huge pages," 2020. [Online]. Available: https://arxiv.org/abs/2004.14378

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

[5] M. Heule and H. van Maaren, "Look-ahead based SAT solvers," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. Amsterdam: IOS Press, 2009, pp. 155–184.

[6] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding cdcl sat solvers by lookaheads," in *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, ser. HVC'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 50–65. [Online]. Available: https://doi.org/10.1007/978-3-642-34188-5_8

[7] M. J. H. Heule, N. Manthey, and T. Philipp, "Validating unsatisfiability results from clause sharing parallel sat solvers," 2014, submitted.

[8] A. Hyvärinen, T. Junttila, and I. Niemelä, "A distribution method for solving SAT in grids," in *SAT 2006*, ser. LNCS, A. Biere and C. P. Gomes, Eds., vol. 4121. Springer, 2006, pp. 430–435.

[9] A. E. Hyvärinen and N. Manthey, "Designing scalable parallel SAT solvers," in *Theory and Applications of Satisfiability Testing – SAT 2012*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer Berlin Heidelberg, 2012, pp. 214–227. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31612-8_17

[10] A. Irfan, D. Lanti, and N. Manthey, "PCASSO – a parallel cooperative SAT solver," 2014.

[11] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon, "Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, M. desJardins and M. L. Littman, Eds. AAAI Press, 2013. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6421

[12] D. Lanti and N. Manthey, "Sharing information in parallel search with search space partitioning," in *Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION 7)*, ser. LNCS, G. Nicosia and P. Pardalos, Eds., vol. 7997, 2013.

[13] M. Luo, C.-M. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for cdcl sat solvers," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 703–711. [Online]. Available: https://doi.org/10.24963/ijcai.2017/98

[14] I. Lynce and J. P. Marques-Silva, "Probing-based preprocessing techniques for propositional satisfiability," in *ICTAI 2003*. IEEE Computer Society, 2003, pp. 105–110.

[15] V. Ryvchin and A. Nadel, “Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking,” in *Proceedings of SAT Competition 2018*, 2018. [Online]. Available: http://hdl.handle.net/10138/237063
[16] J. P. M. Silva and K. A. Sakallah, “GRASP - a new search algorithm for satisfiability,” in *ICCAD 1996*. Washington: IEEE Computer Society, 1996, pp. 220–227.
[17] N. Wetzler, M. Heule, and W. A. H. Jr., “Drat-trim: Efficient checking and trimming using expressive clausal proof,” in *SAT*, 2014, accepted.

# Watch Sat and LTO for CaDiCaL and Kissat

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

***Abstract*—When reading the source code of the solver CaDiCaL, many differences to solvers based on MiniSat 2.2 or Glucose 2.2 can be found. Porting an algorithm detail in unit propagation from CaDiCaL to MergeSAT resulted in a performance degradation. In MergeSAT, when watching a satisfied literal during unit propagation, the clause is moved to the watch list of that literal. In 2021, Kissat and CaDiCaL just update the blocking literal of the clause and keep the clause in the current watch list. MergeSAT's behavior was ported to CaDiCaL and Kissat. Furthermore, link-time-optimization, as used in Riss already, is enabled for the two solvers.**

## I. Unit Propagation Improvements

SAT solvers are used in many fields. Hence, some solvers are heavily tuned to perform well for target applications. Other research focusses on improving the overall solver performance in general. Many heuristic and algorithmic extensions to the core algorithm have been proposed [1]. The overall runtime distributions among the algorithm components still did not change significantly: unit propagation still takes a vast majority of the overall runtime [6], [3].

### A. Watching Clauses in Propagation

The modification presented in this description alters an implementation detail of unit propagation that is different in CaDiCaL when being compared to other MiniSat 2.2-based SAT solvers that participate in competitive events. The two watched literals scheme has been implemented first in [7]. The next major improvement to skip processing clauses early was to move literals, so called *blocking literals*, from the clause into the watch list data structure. MiniSat 2.2 2.1 [2] started to use a blocking literal. When propagating a clause, first the truth value of the blocking literal is checked. In case the blocking literal is satisfied, the related clause is known to be satisfied. Therefore, the clause does not have to be processed further. This technique helps to improve the performance of SAT solvers [6].

In MiniSat 2.2, the blocking literal of a clause is typically the other watched literal. However, any other literal of the clause could be chosen.

### B. How to Handle Satisfied Clauses

When a blocking literal is not satisfied, the clause has to be processed. During this process, each clause of the watch list for the current literal has to be iterated. For each clause, the truth value of all literals has to be checked, in case we find a *conflict clause* or *unit clauses* that force the extension of the current truth assignment. For satisfied clauses, we only need to process the literals until we find a satisfied clauses.

One difference between CaDiCaL and MiniSat 2.2 based solvers is the way how they treat these satisfied clauses. MiniSat 2.2 based solvers watch the satisfied literal. CaDiCaL skip updating watch lists. Instead, CaDiCaL implements further extensions, like memorizing the literal in a clause that was tested when last processing the clause [4].

*a) Always Watching the Satisfied Literal:* When a satisfied literal is detected in a clause during propagating a literal, the clause is removed from the current watch list. As a next step, solvers append the clauses to the watch list of the satisfied literal. Both operations are constant time, but require accessing the other watch list, which can lead to a cache miss [6] and TLB miss [3]. The watch list of the other literal can be higher in the search tree, so that the clause will be touched less frequent in the remainder of the search. Restarts might reduce the saving, on the other hand solver today use *partial restarts* [9], *chronological backtracking* [8] as well as *trail saving* [5]. All these technique give this saving back partially.

*b) Just Update the Blocking Literal:* As an alternative, CaDiCaL keep watching the current literal, which is now falsified, but updates the blocking literal to the satisfied literal. While this breaks the assumption that falsified literals are only watched for *conflict clauses* or *unit clauses*, we still know that the clause is satisfied. Hence, breaking this assumption does not have consequences. The positive effect is that the clause does not have to be removed from the current watch list. This results in no cache miss, nor a TLB miss. However, when the search progresses, after backtracking, the same clause might need to be processed again. In case the satisfied literal is still satisfied, only the blocking literal has to be processed. Otherwise, backtracking also removed the assignment for the blocking literal, so that the whole clause needs to be processed again.

*c) Watching the Satisfied Literal in CaDiCaL and Kissat:* Preliminary testing with MergeSAT when just updating the blocking literal of a clause resulted in a performance degradation. Hence, removing this technique for CaDiCaL might result in a performance improvement. The solver CaDiCaL-watch-sat and Kissat-watch-sat implements this modification.

Not processing a satisfied clause during propagation soon again can result in a different order of propagated literals, as well as different conflicts, and consequently in different heuristic updates and many different follow-up search steps of the solver. Hence, performance differences can not only be attributed to lower or higher compute resource utilization.

## II. Generic Improvements

Besides modifying the algorithm directly, other parameters of the environment can be influenced as well. Helping the CPU to access likely-to-be-accessed memory early with *prefetching* [6], as well as using *(transparent) huge pages* to reduce the paging overhead of a program [3] have been discussed already. Another area to investigate is compiler parameters. By default, compilers optimize code per compilation unit, which usually translates to source files. Optimizations across source files, so called *link time optimization* (LTO), has to be enabled explicitly. Besides spotting programming errors during compile time, LTO also allows to improve the performance of a solver slightly. LTO can be enabled by adding `-flto` to the compiler invocation.

This compile time flag has been added to the build files for both KISSAT and CADICAL.

## III. Availability

The source of the modified CADICAL is publicly available at https://github.com/conp-solutions/cadical/tree/watch-sat-flto. The used version of the tool is "rel-1.4.1-4-g8de178e". This solver has been submitted to the CADICAL hack track.

The source of the modified KISSAT is publicly available at https://github.com/conp-solutions/kissat/tree/SC2022-watchsat-flto. The used version of the tool is "sc2021-sweep-9-gba4bc9e".

## References


[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. Amsterdam: IOS Press, 2009.

[2] 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.

[3] J. K. Fichte, N. Manthey, J. Stecklina, and A. Schidler, "Towards faster reasoners by using transparent huge pages," in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 304–322.

[4] I. P. Gent, "Optimal implementation of watched literals and more general techniques," *J. Artif. Intell. Res.*, vol. 48, pp. 231–251, 2013. [Online]. Available: https://doi.org/10.1613/jair.4016

[5] R. Hickey and F. Bacchus, "Trail saving on backtrack," in *Theory and Applications of Satisfiability Testing – SAT 2020*, L. Pulina and M. Seidl, Eds. Cham: Springer International Publishing, 2020, pp. 46–61.

[6] S. Hölldobler, N. Manthey, and A. Saptawijaya, "Improving resource-unaware SAT solvers," ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Heidelberg: Springer, 2010, pp. 519–534.

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

[8] A. Nadel and V. Ryvchin, "Chronological backtracking," in *Theory and Applications of Satisfiability Testing – SAT 2018*, O. Beyersdorff and C. M. Wintersteiger, Eds. Cham: Springer International Publishing, 2018, pp. 111–121.

[9] P. van der Tak, A. Ramos, and M. Heule, "Reusing the assignment trail in cdcl solvers," *JSAT*, vol. 7, no. 4, pp. 133–138, 2011.

# SEQFROST at the SAT Race 2022

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

## I. INTRODUCTION

This paper presents a brief description of our solver SEQFROST which stands for *Sequential Formal ReasOning about SaTisfiability* in 3 different configurations. SEQFROST is a new solver mostly rewritten from scratch based on our last year submission [1] with efficient data structures and many code optimizations. This year, we observed a large amount of time is spent on function calls in Boolean Constraint Propagation (BCP) and data sorting especially in Multiple-Decision Making (MDM) and inprocessing. Thus, we resorted to inlined code and pointer prefetching in BCP and replaced the standard *sort* procedure with faster modern alternatives, e.g., *pdqsort*. Further, we augment the Multi-Arm Bandit (MAB) rewarding scheme as implemented in the last year winner KISSAT-MAB to MDM strategy [2], and implement functional dependency extraction to enhance the effectiveness of variable elimination. Finally, we extend our elimination method *eager redundancy elimination* (ERE) [1], [3] with clause strengthening to remove redundant literals.

## II. DECISION MAKING

The decision-making step in SEQFROST switches periodically from the standard single-decision procedure as originally introduced in CDCL search to our MDM procedure previously presented in [2]. Both single and multiple decisions are chosen according to VSIDS, VMTF, and CHB [4] branching heuristics. This year, we add the latter to our solver decision heuristics to alleviate the quality of the picked decisions in MDM. SEQFROST decides whether to use VSIDS or CHB based on MAB restarts [5]. The decision phases of multiple decisions are still improved via local search but only once at the initial MDM call.

## III. VARIABLE ELIMINATION

In gate-equivalence reasoning, we substitute eliminated variables with deduced logical equivalent expressions. Combining gate equivalence reasoning with the resolution rule tends to result in smaller formulas compared to only applying the resolution rule [1], [3], [6]–[9]. Let $G_\ell$ be the gate clauses having $\ell$ as the gate output and $H_\ell$ the non-gate clauses, i.e., clauses not contributing to the gate itself. For regular gates (e.g. AND), substitution can be performed by resolving non-gate with gate clauses as follows: $R_x = \{\{G_x \otimes H_{\neg x}\}, \{G_{\neg x} \otimes H_x\}\}$, omitting the tautological and the redundant parts $\{G_x \otimes G_{\neg x}\}$ and $\{H_x \otimes H_{\neg x}\}$, respectively [6].

In this submission, we focus on finding definitions for irregular gates by checking the unsatisfiability of the *co-factors* formula $\{\mathcal{S}_x|_{\neg x} \cup \mathcal{S}_{\neg x}|_x\}$, that is, the formula obtained by removing all occurrences of $x$ from $\mathcal{S}_x$ and $\neg x$ from $\mathcal{S}_{\neg x}$. In [10], a BDD-based approach is used to solve the co-factors. In this work, we replace the BDD structure with a function table (bit-vector) encoding the clausal core of the co-factors. The clausal core is mapped back to the original gate clauses $G_x$ and $G_{\neg x}$ by adding back $x$ and $\neg x$, respectively. Then, the set of resolvents $R_x = \mathcal{S}_x \otimes \mathcal{S}_{\neg x}$ is reduced to $\{\{G_x \otimes G_{\neg x}\}, \{G_x \otimes H_{\neg x}\}, \{G_{\neg x} \otimes H_x\}\}$, dropping the redundant part $\{H_x \otimes H_{\neg x}\}$. In contrast to gate substitution, the resolvents $\{G_x \otimes G_{\neg x}\}$ are not necessarily tautological.

## IV. EAGER REDUNDANCY ELIMINATION

ERE was designed originally to target and remove redundant equivalences after a resolution step. It repeats the following until a fixpoint has been reached: for a given formula $\mathcal{S}$ and clauses $C_1 \in \mathcal{S}, C_2 \in \mathcal{S}$ with $x \in C_1$ and $\bar{x} \in C_2$ for some variable $x$, if there exists a clause $C \in \mathcal{S}$ for which $C \equiv C_1 \otimes_x C_2$, then let $\mathcal{S} := \mathcal{S} \setminus \{C\}$ iff ($C$ is $learnt$ $\vee$ ($C_1$ is $original \wedge C_2$ is $original$)). The clause $C$ in this case is called a *redundancy* and can be removed without altering the original satisfiability. In addition to the redundancies removal, we observed that if the resolvent $C_1 \otimes_x C_2$ is not equivalent to any clause, it can still subsume many others in $\mathcal{S}$. However, to preserve correctness, subsumed clauses are only strengthened via the generated resolvents. Suppose that $C = (C' \cup C'')$. Extended-ERE (i.e. as we call it in this submission) may strengthen $C$ by removing the redundant literals $C'$ (resp. $C''$) if $C'' = C_1 \otimes_x C_2$ (resp. $C' = C_1 \otimes_x C_2$).

## V. CODE OPTIMIZATIONS

As mention earlier in the introduction section, all pointers of vector-type variables are prefetched to save the time spent in calling the overloaded indexing operator `[]`. Additionally, all functions repeatedly called in unit propagation and conflict analysis are replaced with macros as inlining is not always guaranteed by the compiler. Lastly, the bytes generated by DRAT proof are now stored in a 1-MB buffer. Once, the buffer is full, the data is written to the output file via a single call to `fwrite` (i.e. writes data in burst mode). Compared to previous submissions and other solvers, `putc_unlock` was being called to write on disk byte by byte which, of course, adds unnessary overhead to the proof generation.

## VI. Submissions

The solver instance SeqFROST comprises all configurations described in the previous sections, in which MDM with local search, CHB decision heuristic, and all simplifications are enabled with Extended ERE to strengthen original clauses only (e.g. the option `redundancyextend=1` is set). The second configuration SeqFROST-ere-all extends ERE with both original and learnt clause strengthening (e.g. `redundancyextend=2`). The third configuration SeqFROST-no-extend disables Extended ERE (e.g. `redundancyextend=0`). The initial settings of the SeqFROST have been tuned and tested on the DAS-5 cluster [11] and the Dutch national supercomputer Snellius[1].

## References


[1] M. Osama and A. Wijs, "ParaFROST at the SAT Race 2021," in *Proc. of SC (2021)*, ser. Report Series B, vol. B-2021-1. University of Helsinki, 2021, pp. 32–34. [Online]. Available: http://hdl.handle.net/10138/333647

[2] ——, "Multiple Decision Making in Conflict-Driven Clause Learning," in *Proc. of ICTAI (Nov. 2020), Baltimore, USA*. IEEE, 2020, pp. 161–169.

[3] M. Osama, A. Wijs, and A. Biere, "SAT Solving with GPU Accelerated Inprocessing," in *Proc. of TACAS (Mar. 2021), Luxembourg*, ser. LNCS, vol. 12651. Springer, 2021, pp. 133–151.

[4] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Exponential recency weighted average branching heuristic for sat solvers," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 3434–3440.

[5] M. S. Cherif, D. Habet, and C. Terrioux, "Combining VSIDS and CHB Using Restarts in SAT," in *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), L. D. Michel, Ed., vol. 210. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:19.

[6] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing Rules," in *Proc. of IJCAR (Jun. 2012), Manchester, UK*, ser. LNCS, vol. 7364. Springer, 2012, pp. 355–370.

[7] M. Osama and A. Wijs, "GPU Acceleration of Bounded Model Checking with ParaFROST," in *Proc. of CAV (Jul. 2021), USA*, ser. LNCS, vol. 12760. Springer, 2021, pp. 447–460.

[8] ——, "Parallel SAT Simplification on GPU Architectures," in *Proc. of TACAS (Apr. 2019), Prague, Czech Republic*, ser. LNCS, vol. 11427. Springer, 2019, pp. 21–40.

[9] ——, "SIGmA: GPU Accelerated Simplification of SAT Formulas," in *Proc. of IFM (Dec. 2019), Bergen, Norway*, ser. LNCS, vol. 11918. Springer, 2019, pp. 514–522.

[10] A. Biere, "Lingeling, Plingeling and Treengeling Entering the Sat Competition 2013," in *Proc. of SC (2013)*, ser. Report Series B, vol. B-2013-1. University of Helsinki, 2013, pp. 51–52. [Online]. Available: http://hdl.handle.net/10138/40026

[11] H. E. Bal, D. H. J. Epema, C. de Laat, R. van Nieuwpoort, J. W. Romein, F. J. Seinstra, C. Snoek, and H. A. G. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016.



[1]This work was carried out on the Dutch national e-infrastructure with the support of SURF cooperative.

# SLIME SAT Solver

1st Oscar Riveros
Santiago, Chile
oscar.riveros@gmail.com

***Abstract*—A "on the fly" version of HESS algorithm for Multi-armed bandit style selection of rephasing, dual search algorithm, full remove of randomness on sequential and cloud version, based on SLIME sc2021 [1].**

## I. Introduction

We create an on the fly version of HESS [1] algorithm, this allow an optimization of the rephasing states. We create a dual search phase of the solver, one more simple and other more complex, this are called according if the VSIDS heuristic is running. The DISTANCE heuristic is now parametric, on the experiments, DISTANCE work well with cryptographic instances, but not with generics. For the cloud version the initial polarities are different for every node, on deterministic way, also the rank of the node is influencing on the execution. On this version is added an optional parameter for sharing learnt clauses between nodes, alternate sharing clauses, and a filter for the size of learnt shared. Several optimizations, and simplifications.

## II. Methods

### A. HESS black-box algorithm

HESS black-box algorithm [1] of º2 order to approximate values from an Oracle, In this case a "On The Fly" oracle as execution of SLIME and the sequence used to maximize the selection of rephase heuristics like Multi-armed bandit algorithm.

### B. HESS º2 order (On The Fly)

- Create an initial sequence array $\rho(1, 2 \dots\ n)$
- Repeat to a final state.
- Set the current value to $\infty$.
- Set $i$ to 0 and $j$ to 0.
- increment $j$, if $j == n$ increment $i$ and put $j = 0$, if $i == n$, put $i = 0$ and $j = 0$. (two for loops)
- invert the array from $min(i, j)$ to $max(i, j)$
- Get $oracle()$ (call the search algorithm, and use the curren order of the array for select the bandit)
  1) less than current value, reassign and retain the current assignment, and continue with next increment.
  2) if greater, change the array to original state, and continue with next increment.
  3) if equal, continue with next increment.
- Continue with execution and repeat from step 2.

### C. Experimental Evaluation

The default version of SLIME solve the 80% of the entire Crypto Track 2021 (a set of cryptography instances) at 18000 seconds at www.starexec.org cluster.

## III. SLIME CLOUD

Consist on a MPI implementation of SLIME where all nodes compete for the solution, can generate certificates for UNSAT.

## References

[1] Balyo , T , Froleyks , N , Heule , M , Iser , M , Järvisalo , M Suda , M (eds) 2021 , Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions . Department of Computer Science Report Series B , vol. B-2021-1 , Department of Computer Science, University of Helsinki , Helsinki .

# Solvers Cadical_ESA and Kissat_MAB_ESA in 2022 SAT competition

1st Shuolin Li
*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
shuolin.li@etu.univ-amu.fr

2nd Jordi Coll
*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
jordi.coll@lis-lab.fr

3rd Chu-Min Li
*Université de Picardie Jules Verne*
Amiens, France
*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
chu-min.li@u-picardie.fr

4th Mao Luo
*School of Computer Science,*
*Huazhong Univ of Science and Technology*
Wuhan, China
maoluo@hust.edu.cn

5th Djamal Habet
*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
Djamal.Habet@univ-amu.fr

6th Felip Manyà
*Artificial Intelligence Research Institute*
CSIC,Bellaterra, Spain
felip@iiia.csic.es

***Abstract*—This document describes the solvers Cadical_ESA and Kissat_MAB_ESA submitted to the 2022 SAT competition.**

As a technology that can reduce the number of variable in a CNF formula, and thus reduce the search space, variable elimination is essential for modern SAT solvers. In MiniSat [1] and its derived solvers, variable elimination is used in preprocessing, and in Kissat [2] and other Kissat-based solvers, it is used in inprocessing.

The main work that variable elimination does is: choose a variable $x$ to eliminate, resolve $x$, for each clause $c$ contains literal $x$ and each clause $d$ contains literal $\neg x$, combine $c$ and $d$ into a new clause $resolvent$, after that, add all of the new non-tautology $resolvents$ into clause set and delete all clauses contain literal $x$ or $\neg x$.

Each time a variable elimination procedure starts, there is a question that needs to be answered: which variables should we delete? Usually, when solver delete a variable $x$ whose literal $x$ occurs in $pos$ clauses and literal $\neg x$ occurs in $neg$ clauses, it can add $pos \times neg$ new clauses, and the increase in the total number of clauses can be $pos \times neg - pos - neg$ in the worst case. When $pos$ and $neg$ are large, this increase appears to be too expensive. So, one would naturally prefer to eliminate those variables that won't cause too big increase in total number of clauses.

With this consideration, the usual variable scoring formula used in the modern solvers is:

$$score(x) = a \times (pos \times neg) + b \times (pos + neg) \qquad (1)$$

In Equation (1), $a$ and $b$ are two parameters chosen experimentally, which are used to control the contribution of the two parts. Each time the solver selects a variable to eliminate, it just selects the variable with the lowest score.

But does it really matters eliminating these variables? Admittedly, eliminating these variables prevents the number of clauses from growing too fast. However, these easy-to-eliminate (low-scoring) variables only appear in few clauses, and eliminating them would just remove some unimportant branches of the whole search tree.

In a solving process, different variable owns different importance. Some of them decide the result of solving, but others only disturb the search process. Eliminating those trivial variables helps the solver focus on the core search space and then speeds up solving time.

There are many measurements to estimate variables importance, e.g., the number of times a variable occurs in conflicts, the number of times a variable is selected as a decision variable, etc. Here we use a experimentally effective measurement: the variable activity in VSIDS heuristic, to estimate variable importance. The idea of variable activity in VSIDS is to give higher scores to those variables occurring frequently in recent conflict analysis. The variables with high activity in VSIDS can be considered to critical for search, a solver should focus on these variables. And low-activity variables are trivial and should be eliminated as soon as possible, allowing to prevent the solver from unimportant variables and help it focus on critical search space.

We implement this idea in Cadical and Kissat_MAB, resulting in two new solvers Cadical_ESA and Kissat_MAB_ESA, respectively, where ESA is short for Eliminate heap Sorted by Activity. With this idea, Cadical_ESA and Kissat_MAB_ESA sort the variables in the increasing order of their VSIDS activity and eliminate them in this order until the same stop conditions as in Cadical and Kissat_MAB are satisfied.

Since the change is very small, Cadical_ESA is submitted to the Cadical hack track. Kissat_MAB_ESA is submitted to

This work has been partially funded by the French Agence Nationale de la Recherche, 35 reference ANR-19-CHIA-0013-01, and the Spanish AEI project PID2019-111544GB-C2. We also thank the MATRICS platform of the university of Jules Verne.

the normal Main track.

## REFERENCES


[1] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75. [Online]. Available: https://doi.org/10.1007/11499107_5

[2] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

# Kissat-MAB-rephasing and Kissat_relaxed

Xinyan Chen‡, Wenxuan Guo‡, Wanqian Luo†, Hui-Ling Zhen†,
Xijun Li†, Mingxuan Yuan† and Junchi Yan‡
†Huawei Noah's Ark Lab
‡Shanghai Jiao Tong University
{luowanqian1, zhenhuiling2, xijun.li, yuan.mingxuan}@huawei.com
{moss_chen, arya_g, yanjunchi}@sjtu.edu.cn

***Abstract*—We introduce two Kissat-based SAT solvers in this report, i.e., Kissat-MAB-rephasing and Kissat_relaxed. The idea of these solvers falls into two categories: 1) devising a multi-armed bandit mechanism to select rephasing schemes, and 2) embedding local search into the CDCL solving process to explore solutions near promising branches in the search space. The solvers are based on Kissat and Kissat-cf, respectively.**

## I. INTRODUCTION

Phase selection has been a key heuristic in the design of CDCL SAT solvers, which determines the polarity of a branched variable. As proposed in [1], phase saving can alleviate the problem of work repetition for non-chronological solvers. The concept of "rephasing" first appeared in [2], which replaces the saved phases with new values generated by four different schemes (initial, inverted initial, flipped and random) in arithmetically increasing conflict intervals. The following work [3] introduced local search and best phases for rephasing. Currently, there are two types of rephasing schemes. In CaDiCaL and its variant, the candidate schemes are selected in turn. Meanwhile, this decision in [4] is made according to a predefined probability. Based on Kissat-MAB [5], we devise a multi-armed bandit mechanism for rephasing in our proposed solver *Kissat-MAB-rephasing*.

It is known that CDCL and local search are two main paradigms for traditional SAT solvers. Recently, there have been some attempts to equip CDCL solvers with a local search module to boost rephasing and branching parts [6]. Our version, *Kissat_relaxed*, focuses on the combination of local search and relaxed CDCL [6]. If the partial assignment satisfies some proper conditions, a "relaxed" CDCL without backtracking will proceed until all variables are assigned and the conflicts during the propagation will be ignored. Then the solver calls the local search procedure from this full assignment. If the local search finds a solution, it would return a verdict of "satisfiable". Otherwise, the solver backtracks to the level before entering the relaxed CDCL and recovers to the original CDCL mode. This design facilitates the solver to find possible solutions when it gets close to a satisfiable assignment. The implementation of [6] is based on Glucose [7] and MapleLCMDistChronoBT-DL [8] rather than the winning solver Kissat [9] in 2020, but it still ranked very high last year. We try to transplant this idea to Kissat-cf and hope for a greater performance boost. The local search solver in Kissat-cf is YalSAT [10].

## II. IMPLEMENTATION

### A. MAB for Rephasing

Following [5], we use the Upper Confidence Bound (UCB) [11] policy in Kissat-MAB-rephasing to choose from the arm set:

$$A = \{best\ phases, inverted, original, local\ search\}$$

inheriting from Kissat 2021. The reward function (Eq. 1) estimates the ability of current heuristic to quickly reach conflicts:

$$r_t(a) = \frac{\log_2(decisions_t)}{decidedVars_t}, \tag{1}$$

where $t$ denotes the current run, $decisions_t$ denotes the number of decisions already made and $decidedVars_t$ denotes the decided variable reached at least once in the current run.

### B. Local Search with Relaxed CDCL

This method enters the relaxed mode during original CDCL process when one of these conditions [6] is satisfied:

- $\frac{|\alpha|}{|V|} > p$, where $\alpha$ is the current conflict-free partial assignment; $V$ is size of variables; $p$ is a hyperparameter set as $0.75$ in our solver.
- $\frac{|\alpha|}{|\alpha_{max}|} > q$, where $\alpha_{max}$ is the past largest partial assignment and $q$ is a parameter set to $0.9$ in our solver.

Once entering the relaxed mode, the solver will ignore $50$ subsequent hits on the relaxed conditions. In this way, the local search module is able to avoid repeated walks in neighbouring solution space. The restart time of local search is set to 3 and walk step in each round is limited to $500$.

In Kissat_relaxed, the condition is checked when no conflict has been detected by current propagation. In the relaxed mode, Kissat extends the branch without analysis. Once reaching a full assignment, the solver state would be saved before calling the walker. If the local search walker fails to find a solution, the solver state is restored to the saved one and then backtracks to the level before calling relaxed propagation.

## REFERENCES

[1] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *International conference on theory and applications of satisfiability testing*, pp. 294–299, Springer, 2007.
[2] A. Biere, "Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018," *Proceedings of SAT Competition*, vol. 14, 2017.

[3] S. D. QUEUE, "Cadical at the sat race 2019," *SAT RACE 2019*, p. 8, 2019.
[4] X. Zhang and S. Cai, "Relaxed backtracking with rephasing," *Proceedings of SAT competition*, pp. 15–15, 2020.
[5] M. S. Cherif, D. Habet, and C. Terrioux, "Kissat mab: Combining vsids and chb through multi-armed bandit," *SAT COMPETITION 2021*, p. 15, 2021.
[6] S. Cai and X. Zhang, "Deep cooperation of cdcl and local search for sat," in *Theory and Applications of Satisfiability Testing – SAT 2021*, pp. 64–81, Springer, 2021.
[7] L. Simon and G. Audemard, "Predicting learnt clauses quality in modern sat solver," *Proc. IJCAI-2009*, vol. 38, no. 4, pp. 399–404, 2009.
[8] S. Kochemazov, O. Zaikin, V. Kondratiev, and A. Semenov, "Maplelcmdistchronobt-dl, duplicate learnts heuristic-aided solvers at the sat race 2019," *Proceedings of SAT Race*, pp. 24–24, 2019.
[9] A. B. K. F. M. Fleury and M. Heisinger, "Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020," *SAT COMPETITION*, vol. 2020, p. 50, 2020.
[10] A. Biere, "Yet another local search solver and lingeling and friends entering the sat competition 2014," *Sat competition*, vol. 2014, no. 2, p. 65, 2014.
[11] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

# CDCL Solvers with Improved Local Search Cooperation and Pre-processing

Zhihan Chen[1,2], Xindi Zhang[1,2], Shaowei Cai[1,2,*] , Pinyan Lu[3,4]

[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{chenzh, zhangxd, caisw}@ios.ac.cn
[3]Shanghai University of Finance and Economics, China
[4]Huawei TCS Lab, China
lu.pinyan@mail.shufe.edu.cn

## I. Introduction Of Solvers Submitted to SC22

Our solvers submitted to SC22 are summarized in Table I, In this document, we introduce the sequential solvers, and the parallel solvers can be found in our another document. Noting that all the submitted solvers follow the standard building and running rules of SC22.

TABLE I: Solvers Submitted to SC22

| Track | Solver | Key Techniques |
|---|---|---|
| Main | LSTech-Maple | Deep**er** cooperating of CDCL and LS |
| | LSTech-Kissat | LSTech style rephasing on Kissat |
| | Kissat-Inc | LSTech's depth updating trick to Kissat |
| | Kissat-Pre | Pre-processing before Kissat-Inc |
| CaDiCaL | LSTech-CaDiCaL | LSTech style rephasing on cadical |
| Parallel | ParKissat-RS | Random Shuffle with clause sharing |
| | ParKissat-Pre | Pre-processing, Diversification |
| Anniversary | Kissat-Inc | Same as above |
| | LSTech-Maple | Same as above |
| | ParKissat-Pre | Same as above |

## II. LSTech-Maple

LSTech-Maple is an improved version of the SC21 version [3], [6]. We list the main features of the cooperation between CDCL and local search below.

- Local search is called after a certain number of backtrackings, which is dynamically adjusted [6].
- Rules for Generating Local Search Initial Assignment: Let $p$ be the size of non-conflict trail that allowing the algorithm enter the non-backtracking stage. $p = 0$ at the beginning. If the CDCL process reaches a longer no-conflict trail with size $p'$, then it enters the non-backtracking stage, and $p \leftarrow p'$ accordingly. Each time the solver enters the non-backtracking stage, it produces a new complete assignment, which serves as the initial assignment for next local search call. Note that, $p$ is be updated by multiplying 0.9 after each local search call.
- Best From Multiple Selection (BMS) [1] is added into CCAnr. This BMS strategy is used to sample candidate variables from a large set of candidate variables (such as configuration changed decreasing variables) when choosing a variable to flip.
- A filtering mechanism that blocks a local search entrance if the difference on the initial solution with that of the preceding local search entrance is less than 0.1%.
- Dynamically adjusted the time limit for each local search process based on memory access numbers [3].
- A probabilist phasing strategy for local search, which mainly depends on the local search assignments [3].
- Using conflict frequency of variables in local search to enhance the branching heuristics in CDCL [3].

We also use two preprocessing techniques, one of which is Equivalent-literal substitution, and the other is proposed in this document.

- Equivalent-literal substitution (ELS): We generalize a disjoint-set that is suitable for variable with polarity, and we use it to save the equivalence relationship between literals. We also use a hash map for fast-searching clauses. Noting that ELS is only turned on when the clauses are less than 1.5 million.
- Resolution Checking (RC): We use $s(l)$ denote the clauses set that literal $l$ shows. For each variables $x$, if $|s(x)| \times |s(\neg x)| \leq |s(x)| + |s(\neg x)|$, we will do resolution between the two clause sets $s(x)$ and $s(\neg x)$, and checking whether all resulting clauses are always true. If so, the clauses related to variable $x$ can be removed. The neighbor variables of success resolved variables are set to be checked again in the next turn. RC is a variant of the Bounded Variable Elimination, with stricter condition.

Noting that ELS and RC are pre-processing methods and can be used interactively until a fixpoint.

This work is supported by NSFC Grant 62122078.
* Corresponding author
- Zhihan Chen and Xindi Zhang are co-first authors, which are considered to have equal contributions.

## III. LSTech-Kissat and LSTech-CaDiCaL

Both LSTech-Kissat and LSTech-CaDiCaL employ the probabilistic rephasing method of LSTech-Maple, and their base solvers are Kissat and CaDiCaL. Moreover, LSTech-Kissat uses CCAnr [2] to gain phases and adopts the RC technique; LSTech-CaDiCaL watches the satisfied literal as [5].

## IV. Kissat-Inc and Kissat-Pre

Kissat-Inc is modified from Kissat-MAB [4], by using the technique for generating local search initial assignments used in LSTech-Maple. Specifically, we use the strategy to manage *target* phase, and $p$ is updated for each rephasing operation. The resulting solver is named Kissat-Inc.

Kissat-Pre extends the pre-processing techniques of Kissat-Inc with RC and Fourier-Motzkin Variable Elimination (FME).

## References


[1] S. Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 747–753. AAAI Press, 2015.

[2] S. Cai, C. Luo, and K. Su. Ccanr: A configuration checking based local search solver for non-random satisfiability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 1–8, 2015.

[3] S. Cai and X. Zhang. Deep cooperation of cdcl and local search for sat. In *SAT 2021*, pages 64–81, 2021.

[4] M. S. Cherif, D. Habet, and C. Terrioux. Combining vsids and chb using restarts in sat. In *CP 2021*, pages 1–19, 2021.

[5] N. Manthey. Cadical modification–watch sat. *SAT COMPETITION 2021*, page 28.

[6] X. Zhang, S. Cai, and Z. Chen. Improving cdcl via local search. *SAT COMPETITION 2021*, page 42, 2021.

# Kissat_Adaptive_Restart, Kissat_Cfexp: Adaptive Restart Policy and Variable Scoring Improvement

Yang Li‡, Yuqi Jia‡, Wanqian Luo†, Hui-Ling Zhen†,
Xijun Li†, Mingxuan Yuan† and Junchi Yan‡
†Huawei Noah's Ark Lab
‡Shanghai Jiao Tong University
{luowanqian1, zhenhuiling2, xijun.li, yuan.mingxuan}@huawei.com
{yanglily, jiayuqi001023, yanjunchi}@sjtu.edu.cn

***Abstract*—This report describes two CDCL SAT solvers: Kissat_adaptive_restart, Kissat_cfexp. The improvements are two-fold: 1) The design of the adaptive restart strategy selection for SAT solvers based on Multi-Armed Bandit algorithm; 2) The introduction of expSAT with conflict frequency. All solvers are based on the state-of-the-art SAT solver Kissat.**

## I. INTRODUCTION

**Kissat_adaptive_restart** focuses on the restart strategy, which is one of the most critical strategies of CDCL solvers, playing a crucial role in maintaining the searching efficiency during instance solving [1]–[4]. The restart strategy of the existing mainstream solvers basically follows the alternation of stable (e.g. Luby series restarting [5]) and unstable (e.g. Glucose restarting [6]) modes in rotation [1], and does not involve efficient heuristic-guided mode selection strategies. Inspired by Kissat_mab [7], the first-place solver from main track of 2021 SAT Solver Competition, we devise a Multi-Armed Bandit mechanism for simultaneously acquiring new knowledge about the potential distribution of different restart strategies' rewards and optimizing mode selecting decisions based on existing knowledge. The MAB mechanism is enforced among Luby series restarting and EMA restarting [1] adopted in the current Kissat solver, utilizing Upper Confidence Bounds (UCB) [8] strategy at each restart.

**Kissat_cfexp** focuses on the variable scoring strategy. In each branching decision of a CDCL solver, the variable with the highest activity score tends to be selected (VSIDS). It is essential to pick the variable that makes the search process the most effective. One promising idea is to two methods called conflict frequency (CF) [9] and expSAT [10], both of which try to find the branching variables that are most likely to cause conflict. Based on Kissat solver, we modify the variable scoring mechanism, which is employed to determine branching variables during the branching decision.

## II. IMPLEMENTATION

### A. MAB for Restarting

We adopt the Upper Confidence Bounds (UCB) strategy to decide on a preferred arm in the arm set $A = \{Luby, EMA\}$ at each restart of the backtracking algorithm. The decision-making process relies on the solver's performance during the restart phase in the prior process, and the specific reward function is presented in Eq. (1), following [7].

$$r_t(a) = \frac{\log_2(decisions_t)}{decidedVars_t} \tag{1}$$

where $t$ denotes the current run, $decisions_t$ denotes the number of decisions already made and $decidedVars_t$ denotes the decided variable reached at least once in the current run. The value of the function estimates the searching efficiency characterized by the width of the search tree. The UCB algorithm is formulated as follows:

$$UCB(a) = \hat{r}_t(a) + c \cdot \sqrt{\frac{\ln(t)}{n_t(a)}} \tag{2}$$

where $n_t(a)$ denotes the number of times arm $a$ is selected and $c$ is a weight hyper-parameter controlling the extent of exploration.

For implementation details, we integrate the search mode switching logic into the $kissat\_restarting$ function, which determines the timing for restarting. Prior to the restart execution, the restart strategy is adaptively selected based on UCB.

### B. ExpSAT with Conflict Frequency

There are two new terms added to the original score of one variable $x$, i.e., the score obtained from the random exploration process of expSAT (i.e. expScore) and the score representing the conflict frequency of the variable $x$ during local search (namely CF score).

The expScore of $x$ is defined as Eq. (3):

$$expScore(x) = avg(walkScore_w(v)) \tag{3}$$

for all random walk $w$, where

$$walkScore_w(v) = \begin{cases} 0 & \text{if no conflict in } w \\ \frac{\omega^d}{lbd(c)} & \text{otherwise} \end{cases} \tag{4}$$

with decay factor $\omega$, decision distance $d$ and the learned clause $c$. And the function to obtain the CF score of variable $x$ is presented in Eq. (5):

$$Score_{cf}(x) = \frac{STEP_{unsat}}{STEP_{total}} \tag{5}$$

where $STEP_{unsat}$ denotes the number of steps that at least one unsatisfied clause contains $x$, and $STEP_{total}$ denotes the number of all steps of the local search.

## REFERENCES


[1] Biere A, Fröhlich A. Evaluating CDCL restart schemes[J]. Proceedings of Pragmatics of SAT, 2015: 1-17.

[2] Oh C. Between SAT and UNSAT: the fundamental difference in CDCL SAT[C]//International Conference on Theory and Applications of Satisfiability Testing. Springer, Cham, 2015: 307-323.

[3] Haim S, Heule M. Towards ultra rapid restarts[J]. arXiv preprint arXiv:1402.4413, 2014.

[4] Van Der Tak P, Ramos A, Heule M. Reusing the assignment trail in CDCL solvers[J]. Journal on Satisfiability, Boolean Modeling and Computation, 2011, 7(4): 133-138.

[5] Luby M, Sinclair A, Zuckerman D. Optimal speedup of Las Vegas algorithms[J]. Information Processing Letters, 1993, 47(4): 173-180.

[6] Audemard G, Simon L. Refining restarts strategies for SAT and UNSAT[C]//International Conference on Principles and Practice of Constraint Programming. Springer, Berlin, Heidelberg, 2012: 118-126.

[7] Cherif M S, Habet D, Terrioux C. Kissat MAB: Combining VSIDS and CHB through Multi-Armed Bandit[J]. SAT COMPETITION 2021, 2021: 15.

[8] Auer P, Cesa-Bianchi N, Fischer P. Finite-time analysis of the multiarmed bandit problem[J]. Machine learning, 2002, 47(2): 235-256.

[9] Cai S, Zhang X. Deep Cooperation of CDCL and Local Search for SAT[C]//International Conference on Theory and Applications of Satisfiability Testing. Springer, Cham, 2021: 64-81.

[10] Chowdhury M S, You J. Guiding CDCL SAT search via random exploration amid conflict depression[C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2020, 34(02): 1428-1435.

# CaDiCaL-DVDL

Zhenjiang Zhao*, Takahisa Toda*, Takashi Kitamura†
*Graduate School of Informatics and Engineering, University of Electro-Communications, Tokyo, Japan
{zhenjiang, toda}@disc.lab.uec.ac.jp
†National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan
t.kitamura@aist.go.jp

***Abstract*—This document describes the SAT solver CaDiCaL-DVDL. Kochemazov et al. presented an efficient clause management method for CDCL solvers that extracts duplicate conflict clauses and stores them indefinitely. Inspired by this work, we present a similarity-based clause management method and implement it on top of CaDiCal 1.4.1.**

## I. Introduction

The conflict-driven clause learning (CDCL) is a standard algorithmic framework on which almost state-of-the-art SAT solvers are based [1]. During the solving process, many learnt clauses are generated, and those turned out to be useless are removed. The decision for the clause deletion commonly relies on heuristics, and the same clauses can be learnt and removed multiple times. Hence, these heuristics have a significant impact on the solver's performance. The recently proposed DL heuristic [2] determines the utility of conflict clauses in terms of the number of times clauses are recorded as conflict clauses. To improve this, we focus on the similarity between conflict clauses. We consider a method for determining the clause similarity with hashing and implement it on top of CaDiCal 1.4.1.

## II. DVDL Heuristic And Implementation

We call two learned clauses that have both the same decision variables and the same order in the decision levels of those decision variables as similar clauses. For instance, the following two similar clauses are learned during CDCL:

$$\begin{aligned} a &: (\underline{x1@4}, \underline{x2@3}, x3@5, \underline{x5@6}) \\ b &: (\underline{x1@3}, \underline{x2@2}, \underline{x5@4}, x7@5) \end{aligned} \tag{1}$$

The decision variables for both $a$ and $b$ are $x1, x2, x5$, and the decision levels for those decision variables of $a$ and $b$ are $4, 3, 6$ and $3, 2, 4$ respectively. When arranging decision variables $x1, x2, x5$ in increasing order of decision levels, we will get the same string "x2x1x5". A set of mutually similar clauses can be represented by an identical string. Similar clauses share the number of occurrences. We believe that the more times similar clauses occur, the more valuable they are. Therefore, those clauses shouldn't be deleted when the occurrences reach a certain number.

For implementation, $unordered\_map$ in C++ Standard Library is uses to count the number of occurrences of similar clauses. The $key$ of $unordered\_map$ represents the string of similar clauses just like the above-mentioned string "x2x1x5", the $value$ of $unordered\_map$ refers to the number of occurrences of similar clauses represented by the $key$.

4 files $subsume.cpp$, $analyze.cpp$, $internal.hpp$, $vivify.cpp$ of CaDiCaL 1.4.1 are modified, and DVDL heuristic is implemented within 1000 non-space characters.

In SAT Competition 2022, we submit two solvers $CaDiCaL\_DVDL\_V1$ and $CaDiCaL\_DVDL\_V2$, which differ only in parameters.

## References

[1] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, 2009.

[2] S. Kochemazov, O. Zaikin, A. A. Semenov, and V. Kondratiev, "Speeding up CDCL inference with duplicate learnt clauses," in *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, ser. Frontiers in Artificial Intelligence and Applications, G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds., vol. 325. IOS Press, 2020, pp. 339–346. [Online]. Available: https://doi.org/10.3233/FAIA200111

# Paracooba Enters SAT Competition 2022

Maximilian Levi Heisinger
*Institute for Symbolic Artificial Intelligence*
*Johannes Kepler University Linz*
Linz, Austria
maximilian.heisinger@jku.at

PARACOOBA is a parallel, distributed, Cube-and-Conquer (CnC) SAT solver based on KISSAT [1] and CADICAL [2]. It tries to split problems into sub-problems by recursively setting variables to concrete values, i.e. applying assumptions to the original formula. These sub-problems are then solved with the incremental version of CADICAL, combined with an (optional) timer to stop the solving process and re-split the formula again. Next to this parallel solving process, an instance of KISSAT is also running sequentially to stop bad splits from having too large impacts. The PARACOOBA solver can be found on GitHub using the URL below.

github.com/maximaximal/paracooba

## I. FORMULA SPLITTING

Splitting formulas is based on our implementation of tree-based lookahead [3] part of CADICAL. If the splitting with lookahead takes too long, we fall back to the number of occurrences of variables. The chosen variable $x$ is then deemed to be the most decicive one and used to split the formula $\phi$ into $\phi \wedge \neg x$ and $\phi \wedge x$. Now, if both sub-formulas are unsatisfiable, the whole formula is also unsatisfiable. If one is satisfiable, the whole formula is also satisfiable with the same assignment. This process is repeated until a predefined cube-tree-depth is reached, which defaults to saturate the locally available cores. Sub-problems usually vary greatly in their hardness and are then solved in individual solver threads or offloaded to other worker nodes in the network.

## II. COMMUNICATION AND TASK SCHEDULING

The communication between nodes uses a custom binary protocol transported via TCP. After starting, a fully interconnected network of workers is created. Every worker receives utilization information of all other worker nodes and is thus able to decide locally, whether to offload work from the local queue to other nodes in the network. The dynamic offloading of tasks ensures that the highly different task hardness is mitigated by always saturating as many cores as possible.

To start, paracooba does not need the addresses of it's workers. The main node can start on it's own and either immediately start solving (if local worker threads were enabled) or idle until worker nodes connect to it. The main node listens on a port for incoming connections from workers (both the port and listen address are configurable). After connecting, workers receive all other known peers from the peer they connected to, in turn forming the fully connected network explained above. Workers may solve multiple problems at once, problems sharing the available worker threads on a given node.

## III. EXTENDING PARACOOBA

PARACOOBA is built to be highly modular, also offering a QBF solving module. The software is MIT-licensed and can be found on GitHub. More detauls about the implementation, the employed algorithms and the expandability can be found in [4] and [5]. Possible extension points are the communication stack, the solver module, the offloading / scheduling mechanism, and the local task runner. Extensions may be loaded from provided shared object files at start-up. Automated testing of modules is done using integration tests, also provided in the repository.

## REFERENCES

[1] A. Biere, K. Frazekas, M. Fleury, M. Heisinger, CaDiCaL, Kissat, Paracooba,
[2] A. Biere, CaDiCaL at the SAT Race 2019, SAT Race 2019
[3] M. Heule, M. Järvisalo, A. Biere, Revisiting hyper binary resolution, International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Springer, Berlin, Heidelberg, 2013
[4] M. Heisinger, M. Fleury, A. Biere, Distributed Cube and Conquer with Paracooba, SAT2020
[5] M. Heisinger, Distributed SAT & QBF Solving: The Paracooba Framework, 2021 Plingeling and Treengeling Entering the SAT Competition 2020, SAT 2020

# DPS-KISSAT

Hidetomo Nabeshima Tsubasa Fukiage Yuto Obitsu Xiao-Nan Lu
*University of Yamanashi*
Yamanashi, JAPAN
{nabesima,xnlu}@yamanashi.ac.jp

Katsumi Inoue
*National Institute of Informatics*
Tokyo, JAPAN
inoue@nii.ac.jp

***Abstract*—DPS is a framework for easily constructing efficient deterministic parallel SAT solvers, providing the delayed clause exchange technique introduced in MANYGLUCOSE. We applied DPS to KISSAT to construct a portfolio parallel SAT solver DPS-KISSAT.**

## I. INTRODUCTION

DPS is a framework for easily implementing deterministic portfolio parallel SAT solvers for shared memory multicore environment, that guarantee reproducible behavior. Reproducibility means that the execution result (the running time and a found model if satisfiable ) does not change across runs. DPS is a successor to the deterministic parallel SAT solver MANYGLUCOSE [1], from which it extracts and generalizes the mechanisms necessary to achieve reproducible behavior. We applied DPS to KISSAT [2], one of the state-of-the-art sequential SAT solvers, to construct a portfolio parallel SAT solver DPS-KISSAT.

## II. DELAYED CLAUSE EXCHANGE

In parallel SAT solvers, reproducibility is lost when learnt clauses are exchanged asynchronously. Synchronous clause exchange ensures reproducibile behavior, but increases latency. The delayed clause exchange introduced in MANYGLUCOSE allows a certain delay in the timing of clause exchanges, thereby absorbing fluctuations in the exchange interval and can reduce reducing the waiting time. However, implementing delayed clause exchange requires expert knowledge of concurrent programming, so introducing it into existing sequential SAT solvers is a time-consuming task. We have extracted the delayed clause exchange method from MANYGLUCOSE and developed a framework DPS with a generic interface to facilitate its integration into existing sequential solvers.

DPS-KISSAT is a deterministic parallel SAT solver that applies the delayed clause exchange provided by our framework to KISSAT.

## III. PORTFOLIO STRATEGY

The diversity strategy of DPS-Kissat consists of the following three elements:

1) random variable selection until the first conflict occurs except for the first thread. The random seeds use different values for each thread.
2) 24 different search strategy settings shown in the portfolio parallel SAT solver PAKIS [3].
3) disabled elimination in half of threads.

The first strategy was introduced in MANYSAT 2.0 [4], the first deterministic parallel SAT solver. Clause exchange in non-deterministic parallel SAT solvers is one of the causes of search diversity due to its asynchronous nature, but this is not expected in deterministic solvers, so strategies such as random decision are necessary to ensure diversity. PAKIS executes 24 Kissat processes with different strategies in parallel without clause exchange, and has won the parallel SAT track in the SAT 2021 competition. The last strategy was introduced because there were some instances where a lot of time was spent on in-processing.

## IV. IMPLEMENTATION

DPS-KISSAT parallelizes KISSAT-SC2021 [5], which required about 400 lines of modification to KISSAT and about 400 lines for the wrapper class to incorporate KISSAT into DPS. The version submitted to SAT Competition 2022 launches 32 threads. The results are guaranteed to be reproducible. DPS supports non-deterministic mode, which is also entered in the competition as NPS-KISSAT.

## ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Numbers JP20H05963, JP20K11934. In this research work we used the supercomputer of ACCMS, Kyoto University.

## REFERENCES

[1] H. Nabeshima and K. Inoue, "Reproducible efficient parallel SAT solving," in *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020), LNCS 12178*, 2020, pp. 123–138.
[2] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING entering the SAT competition 2020," http://hdl.handle.net/10138/318450, 2020, SAT Competition 2020 Solver Description.
[3] R. K. Tchinda and C. T. Djamegni, "HKIS, HCAD, PAKIS and PAINLESS_EXMAPLELCMDISTCHRONOBT in the SC21," http://hdl.handle.net/10138/333647, 2021, SAT Competition 2021 Solver Description.
[4] Y. Hamadi, S. Jabbour, C. Piette, and L. Sais, "Deterministic parallel DPLL," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 4, pp. 127–132, 2011.
[5] A. Biere, M. Fleury, and M. Heisinger, "CADICAL, KISSAT, PARACOOBA entering the SAT competition 2021," http://hdl.handle.net/10138/333647, 2021, SAT Competition 2021 Solver Description.

# ITMO-ParSAT, the parallel solver utilizing probabilistic backdoors at the SAT Competition 2022

Ibragim Dzhiblavi, Daniil Chivilikhin, Stepan Kochemazov and Alexander Semenov
Email:dzhiblavi@gmail.com, chivdan@gmail.com, veinamond@gmail.com, axelvonemes@gmail.com

***Abstract*—This document describes the `ITMO-ParSAT` parallel solver. It utilizes the `Painless` framework together with a novel idea on finding probabilistic backdoors to SAT to combine portfolio and divide-and-conquer approaches in a single solver.**

## I. Introduction

`ITMO-ParSAT` is a novel parallel solver, that was implemented on top of `P-MCOMSPS`, the winner of the parallel track of SAT Competition 2021 [1]. It combines two main parts: the first part is similar to the aforementioned `P-MCOMSPS` solver. The second part of the solver utilizes a recently proposed idea of *probabilistic backdoors* for SAT [2] to find promising decompositions of a formula in a manner reminiscent of Cube and-Conquer [3] solvers, such as e.g. `Treengeling` [4]. Essentially, it employs a metaheuristic optimization algorithm to find a special set called a *probabilistic backdoor* which is then used to split the original SAT instance into subproblems that can be solved independently. To solve them, it employs `MapleCOMSPS`-based [5] subsolvers similar to the ones used in `Painless` [6].

## II. Probabilistic backdoors

The idea of backdoors goes back to the paper [7] by Ryan Williams, Carla Gomes and Bart Selman. They introduced the concept of the so-called strong backdoor sets. If $C$ is a SAT instance, then a Strong Backdoor Set (SBS) is such a set of variables from $C$ that the substitution of any of their assignments into $C$ results in a formula for which SAT is solvable by a polynomial algorithm (e.g. continuous application of the unit propagation rule).

The attractiveness of strong backdoors is easy to see: if we have an SBS and it is small, then the hardness of the problem is defined by the size of the SBS. Unfortunately, small strong backdoors are exceedingly rare and also very hard to find. This fact led to the development of other variants of backdoors to SAT, see e.g. [8]. They are usually formed by lifting some of the restrictions imposed on SBS.

The probabilistic backdoors employed by the proposed solver are among the most novel variants of backdoors. They have been proposed in the recent paper [2].

Let $C$ be a CNF formula over Boolean variables $X$, $B$ be a subset of $X$ ($B \subseteq X$), and $A$ be some polynomial algorithm. Next, denote the set of all assignments of variables from $B$ by $\{0,1\}^{|B|}$. By $C[\beta/B]$ we denote the CNF formula obtained from $C$ by substituting the values $\beta \in \{0,1\}^{|B|}$ to variables from $B$. The notation $C[\beta/B] \in S(A)$ means that SAT for $C[\beta/B]$ is solvable by algorithm $A$.

**Definition.** The set $B \subseteq X$ is called a $\rho$-backdoor ($\rho \in [0,1]$) w.r.t. $A$ if the fraction of CNF formulas $C[\beta/B]$ that belong to $S(A)$ over all possible $\beta \in \{0,1\}^{|B|}$ is at least $\rho$.

It is clear that $\rho$-backdoors can be viewed as a straightforward probabilistic generalization of SBS [7], because the value of $\rho$ represents the probability that the polynomial algorithm $A$ will solve $C[\beta/B]$ for a randomly chosen $\beta \in \{0,1\}^{|B|}$.

We use the Unit Propagation rule as algorithm $A$. In [2] it was shown that it is possible to accurately approximate $\rho$ for a given set $B$ via the Monte Carlo method. In practice it makes sense to find small sets $B$ with large $\rho$. It can be done e.g. via a metaheuristic optimization algorithm, since it is quite cheap to construct the estimation of $\rho$ for any given set $B$.

### A. Using probabilistic backdoors to solve SAT

Assume that we have a small $\rho$-backdoor $B$ (say, $|B| \leq 16$), with $\rho = 0.99$. How can we use it to solve SAT for the CNF $C$? The basic scheme is as follows: we use unit propagation to traverse all $\beta \in 2^{|B|}$ and filter out the ones that are solved by UP. The remaining portion of no more than $0.99 \cdot 2^{|B|}$ subproblems may be significantly simpler compared to the original SAT instance, since the values of many variables are fixed in the corresponding formula $C[\beta/B]$. Regardless of the complexity, they can be solved in parallel using assumptions and incremental SAT, as it is done in Cube-and-Conquer [3]. Particularly hard subproblems $C[\beta/B]$ can be additionally (recursively) decomposed via probabilistic backdoors.

### B. Using several probabilistic backdoors at once

As it was noted in [2], one can combine several $\rho$-backdoors with sufficiently high $\rho$ into a single large backdoor. For example, if we have backdoors $B_1, \ldots, B_5$, we can form a single backdoor $B^* = \cup_{i=1}^{5} B_i$. The main advantage of such a combination is that we can separately construct the sets of subproblems that can not be resolved by UP for each $B_i$ and the Cartesian product of such sets forms the set of such subproblems for $B^*$. Note, that if $|B^*| > 50$ it is basically impossible to construct such a set by naively traversing all $\beta \in \{0,1\}^{|B^*|}$.

## III. `ITMO-ParSAT`

As we noted above, the solver is based on the `Painless` framework [6], in particular, on its most recent incarnation

from 2021, P-MCOMSPS [1]. The original P-MCOMSPS occupies 27 out of 64 threads. An additional thread is occupied by the lstech_maple solver [9] in more or less pristine condition. The remaining 36 threads are engaged in finding probabilistic backdoors, and solving the original instance using them. The reasoning behinds this is that one cannot expect that solving with backdoors will be fast for all instances, so using other solvers in parallel seems to be a good choice.

### A. Finding backdoors

The backdoor finding stage in ITMO-ParSAT is organized similar to the procedure described in [2] with several additional tweaks and optimizations.

We employ metaheuristic algorithms (e.g. $(1+1)$ evolutionary algorithm) to find probabilistic backdoors. To reduce the search space we use the following heuristic. For each variable $x_i \in X$ we separately unit propagate both $x$ and $\neg x$ (just like in look-ahead solvers or in Failed Variable Probing) and compute the size of the union of sets of assumed and propagated literals. Then, we pick 300 variables with the largest value of this statistic. After this, we launch metaheuristic search which is allowed to work only with the chosen 300 *most important* variables.

As the metaheuristic algorithm we employ the evolutionary algorithm in the configuration identical to the one described in [2]. All 36 threads are engaged in this process using varying degrees of parallelism. The process stops when one of the specific conditions is satisfied, e.g. the time limit is exceeded, or the number of iterations with no improvements reaches a certain limit.

An important optimization we added in comparison to [2] is an efficient algorithm for calculating $\rho$ using unit propagation. Suppose we want to check if for some $\beta \in \{0,1\}^{|B|}$ the corresponding CNF $C$ is solved by the polynomial subsolver $A$: $C[\beta/B] \in S(A)$, i.e. a conflict is generated by the solver as a result of applying UP with assumptions $\beta$. But in many cases, some prefix of $\beta$ already leads to a conflict. Based on this idea, we implemented an algorithm that uses a tree of assumptions $\beta$, detects conflicts early and does not descend into subtrees that already correspond to a conflict. In fact, it closely resembles DPLL repurposed to compute the number of leaves satisfying a certain criterion.

### B. Solving SAT with backdoors

When solving SAT instances using a backdoor $B$, we need to solve the "hard" subproblems $C[\beta/B] \notin S(A)$. If the instance is satisfiable, then as soon as one of these subproblems is found to be satisfiable, the solver generates a satisfying assignment and terminates.

### C. Communication between different parts

The clause sharing is performed in the following manner: the portfolio part (27 threads of Painless) and the backdoor part (36 threads of MapleCOMSPS) follow the rules set in P-MCOMSPS. Other than that, the portfolio part, the backdoor part, and the lstech_maple solver additionally share clauses in a round-robin fashion.

## IV. Submitted configurations

We submit the solver in two configurations which differ only in the way the probabilistic backdoors are employed.

### A. Configuration A

The first configuration (to which we refer as configuration A) employs a single probabilistic backdoor. The subproblems that are not resolved by Unit Propagation are solved by MapleCOMSPS subsolvers with the time limit of three minutes. If a subproblem $C[\beta/B]$ is not solved within this time limit, then it is recursively decomposed (i.e. we find another probabilistic backdoor treating this subproblem as the original one). The depth of recursion is limited to one.

### B. Configuration B

The second configuration (Configuration B) employs multiple backdoors in the manner described in Section II-B. In particular, it finds at most 10 probabilistic backdoors, and then combines them in such a way that the number of resulting subproblems does not exceed $8 \cdot 10^6$. The recursive decomposition is disabled in this configuration.

## V. Availability

The source code of the solver is available online [1]

## References


[1] V. Vallade, L. L. Frioux, R. Oanea, S. Baarir, and J. Sopena, "New Concurrent and Distributed Painless solvers: P-MCOMSPS, P-MCOMSPS-COM, P-MCOMSPS-MPI, and P-MCOMSPS-COM-MPI," in *Proc. of SAT Competition 2021*, 2021, pp. 40–41.

[2] A. Semenov, A. Pavlenko, D. Chivilikhin, and S. Kochemazov, "On Probabilistic Generalization of Backdoors in Boolean Satisfiability," in *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence, (AAAI-22), in press*. AAAI Press, 2022.

[3] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads," in *Hardware and Software: Verification and Testing*, 2012, pp. 50–65.

[4] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2017," in *Proc. of SAT Competition 2017*, vol. B-2017-1, 2017, pp. 14–15.

[5] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *Proc. of SAT Competition 2016*, vol. B-2016-1, 2016, pp. 52–53.

[6] L. L. Frioux, S. Baarir, J. Sopena, and F. Kordon, "PaInleSS: A framework for parallel SAT solving," in *SAT*, ser. LNCS, vol. 10491, 2017, pp. 233–250.

[7] R. Williams, C. P. Gomes, and B. Selman, "Backdoors to typical case complexity," in *IJCAI*, 2003, pp. 1173—1178.

[8] A. A. Semenov, O. Zaikin, I. V. Otpuschennikov, S. Kochemazov, and A. Ignatiev, "On cryptographic attacks using backdoors for SAT," in *AAAI*, 2018, pp. 6641–6648.

[9] X. Zhang, S. Cai, and Z. Chen, "Improving CDCL via Local Search," in *Proc. of SAT Competition 2021*, 2021, pp. 42–43.


[1] https://github.com/dzhiblavi/itmo-parsat

# Mallob in the SAT Competition 2022

Dominik Schreiber
*Institute of Theoretical Informatics*
*Karlsruhe Institute of Technology*
Karlsruhe, Germany
dominik.schreiber@kit.edu

***Abstract*—We describe our submissions to the parallel and cloud tracks of the SAT Competition 2022. Notable differences over last year's submission include a reworked clause sharing mechanism with a new approach to distributed clause filtering; further solvers and updated solver configurations; and an additional kind of memory awareness.**

***Index Terms*—Parallel SAT solving, distributed SAT solving**

## I. Introduction

In this report we describe the configurations of our system **Mallob** which we submit to this year's International SAT Competition. Mallob (**Mal**leable **Lo**ad **B**alancer / **M**ultitasking **A**gi**l**e **Lo**gic **B**lackbox) is a decentralized job scheduling platform capable of prioritizing, balancing, and processing many SAT instances at once [1]. However, due to the rules of the competition, we configure our system to immediately schedule a single instance (i.e., the problem input) with full demand of resources and to quit after its processing.

## II. System and Solver Setup

As in last years [2], [3], we subdivide each physical compute node into groups of four hardware threads each and run one MPI process on each such group. Each MPI process then deploys four core solvers. Contrary to last years where we run solvers as separate threads within each MPI process, this year each MPI process spawns a separate subprocess which runs four solver threads. This has two advantages: First, from a fault-tolerance perspective, individual solvers crashing (e.g., due to pathological inputs or internal errors) do not break the entire system but trigger a clean restart of the concerned subprocess. Secondly, we can deliberately restart individual solver processes for the purpose of memory awareness (see IV.). Despite these benefits, we acknowledge that this approach incurs some overhead for Inter-Process Communication, especially for transferring the formula and for periodic clause sharing.

In its current state, Mallob features four full-featured solver interfaces, namely for Lingeling [4], Glucose [5], CaDiCaL [6], and Kissat [6]. Most recently, we modified Kissat's codebase to support import and export of redundant clauses (only triggered at decision level 0 and every 500 conflicts) as well as setting initial phases for individual variables.

For the Parallel Track we submit a version with a portfolio purely consisting of Kissat configurations. We refer to this version as *Mallob-Ki*. In addition, we submit the most diverse portfolio Mallob can currently employ to the Cloud Track: We mix Kissat, CaDiCaL, Lingeling, and Glucose solvers roughly weighted according to their relative base performance and memory efficiency (eight parts Kissat, six parts CaDiCaL, four parts Lingeling, and two parts Glucose). We refer to this version as *Mallob-Kicaliglu*.

For both of these versions, we have identified strong solver configurations by running each SAT solver in various different configurations on the benchmarks of the International SAT Competition 2020.

## III. Clause Exchange

We have reimplemented and overhauled large portions of Mallob's clause sharing strategy. Most significantly, we introduce a new approach to clause filtering, i.e., the problem of deciding for a shared clause $c$ and a solver $S$ whether $S$ has received or produced $c$ before and should therefore not receive $c$ (again). The previous clause filtering mechanism of Mallob (inherited from HordeSat [7]) featured multiple large Bloom Filters at each solver process which occasionally result in erroneous rejection of unseen clauses. The probability for such *false positives* grows with the number of clauses registered in the filters, which may become noticeable in large distributed systems with millions of clauses being shared.

Our new clause filtering mechanism is exact and requires memory proportional to the set of "potentially good" clauses produced by a given solver process. We use two local datastructures: First, a hash table $H$ of clauses maps each produced clause to a small bundle (32 bits) of meta data, including its LBD score, which local solver(s) produced it, and whether it was shared before. Secondly, a buffer structure $B$ maintains a space-limited selection of the best clauses ready for export, discarding some of the worst clauses if better clauses arrive. Clause quality is determined by clause length and (secondarily) by LBD score. Our approach functions as follows:

- A clause $c$ learnt by a solver which meets a basic quality criterium (length $\leq 20$) is checked against $H$. If $c \notin H$ and if $c$ fits into $B$, then $c$ is inserted into $B$ and $H$.
- At clause exchange time, each process flushes the highest priority clauses from $B$ up to a certain total length.
- A buffer $b$ of globally best clauses is aggregated and then shared among all processes as described in [1].
- Each process iterates over each clause $c_i \in b$ and checks whether $q_i := [c_i \in H \text{ and } c_i \text{ is marked as shared}] = 1$. A bit vector $\tilde{v}$ is constructed: $\tilde{v}[i] := q_i$ for each $c_i$. If $c \in H$ and $c$ was not shared before, $c$ is marked as shared.

- All created bit vectors $\tilde{v}$ are reduced to a single *filter vector* $v$ via bitwise OR operations. $v$ is aggregated and then shared among all processes just like $b$.
- Each process iterates over $b$ and $v$ simultaneously and only considers clauses $c_i$ for which $v[i] = 0$. Each such clause $c$ is forwarded to all local solvers which have not produced $c$ yet according to $H[c]$.

The described approach ensures that a clause $c$ shared in a given epoch $e \in \mathbb{N}$ will not be re-shared in a later epoch $e' > e$, since there is at least one process where $c$ was produced and where, consequently, it was marked as shared in epoch $e$. At epoch $e'$, this status is propagated to all processes via $v$, hence $c$ is filtered. We can still allow for clauses to be reshared after a certain period of time elapsed: We can store in $H[c]$ the epoch $e$ where $c$ was last shared, and we re-admit $c$ for sharing if $e$ is sufficiently old. Likewise, we can allow re-sharing a clause if its LBD score improved since the last sharing. However, we did not find re-sharing upon improved LBD to be promising, and we set the minimum period until a clause is re-shared to a conservative 500 s.

We implemented $B$ as an array of buckets, one bucket for each clause length $1 \leq l \leq 20$. Each bucket features a list of clauses which can be added to or removed from. A global *budget* integer represents the remaining number of literals which can still be inserted until $B$ is full. If this budget is insufficient for inserting a given clause $c$ of length $l$, an attempt is made to discard clauses from a bucket $l' > l$ in order to "steal" space for $c$. If this is unsuccessful, $c$ is discarded. With this flexible buffering structure, we account for the observation that different solvers export differently sized clauses at different points in time during the solving procedure: For this reason, the available space is balanced dynamically among the different clause quality levels.

We employ the same data structure as $B$ for the *import buffer* $B_S$ of each solver $S$. This way, the buffering of incoming clauses is robust towards solvers which may not import clauses for a long period of time and therefore necessitate dropping some buffered clauses.

## IV. Memory Awareness

Last year we introduced a rudimentary kind of memory awareness to Mallob: When starting to solve a formula, the number of contained literals is used to decide on how many threads to spawn in each MPI process. While this measure can be effective for some inputs, it does not address all issues. Memory usage which is initially acceptable but then grows to unsustainable levels is not accounted for. Furthermore, for extreme inputs even spawning a single solver thread for each MPI process may require too much memory.

This year we introduce an additional measure to counteract excessive memory usage. At program start, we create one communicator for the MPI processes at each physical machine. In other words, we identify groups of MPI processes with a shared RAM budget. Each group periodically checks the current memory usage of its machine and exchanges certain diagnostics for each MPI process. If a certain memory limit is exceeded ($> 90\%$ of RAM used), one or multiple MPI processes are chosen to trigger a memory panic. The heuristic which decides on the particular process(es) considers the memory used by each process as well as the importance of its role in the portfolio. A memory panic at a process which currently runs $t$ solver threads triggers an immediate restart of the SAT solving process with $t-1$ solver threads. For extreme cases this can go as low as $t = 0$, i.e., no more solvers are executed on this MPI process. The decision heuristic ensures that at least one active solver thread remains on each machine.


## Acknowledgment

The author expresses his heartfelt thanks to Laurent Simon and Armin Biere for allowing the use of Glucose, Kissat, CaDiCaL, and Lingeling in the competition. Furthermore, the author thanks Maximilian Schick for implementing initial CaDiCaL bindings for Mallob which we built upon. The author gratefully acknowledges the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). Moreover, some preparation for this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).



## References


[1] D. Schreiber and P. Sanders, "Scalable SAT solving in the cloud," in *International Conference on Theory and Applications of Satisfiability Testing*, 2021. In review.

[2] D. Schreiber, "Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track," in *Proc. of SAT Competition*, pp. 45–46, 2020.

[3] D. Schreiber, "Mallob in the SAT competition 2021," *SAT COMPETITION 2021*, p. 38.

[4] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2018," *Proc. of SAT Competition*, pp. 13–14, 2018.

[5] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[6] A. B. K. F. M. Fleury and M. Heisinger, "CaDiCaL, kissat, paracooba, plingeling and treengeling entering the sat competition 2020," *SAT COMPETITION*, vol. 2020, p. 50, 2020.

[7] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio SAT solver," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 156–172, Springer, 2015.

# P-KISSAT-MAB: Painless based parallel SAT solvers

Anissa Kheireddine*¶, Souheib Baarir*§, Etienne Renault¶,
*Sorbonne Université, LIP6, CNRS, UMR 7606, Paris, France
‡Université Paris Nanterre, France
¶ EPITA, LRDE, Kremlin-Bicêtre, France

***Abstract*—This paper describes the solver P-KISSAT-MAB submitted to the parallel track of the SAT Competition 2022. P-KISSAT-MAB is LBD-based.**

## I. Introduction

P-KISSAT-MAB is a concurrent SAT solver built using the Painless framework [1]. It is a portfolio-based [2] solvers implementing a diversification strategy based on the work in [3], fine control of learnt clause exchanges [4] based on LBD [5]. P-KISSAT-MAB uses Kissat-MAB [6] as a core engine.

Section II details the implementation of P-KISSAT-MAB using Painless and Kissat-MAB.

## II. P-KISSAT-MAB

### A. Kissat-MAB

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

Kissat-MAB [6] has been adapted for the parallel context as follows: (1) we parameterized the solver to use either LRB [7], or VSIDS [8] (resp. L and V); (2) we disabled the walk SAT procedure used to generate the initial phase of the variables; (3) we disabled the compacting garbage collection option used to remove original variables of the problem from the solver, in order to allow the communication between different core engines (4) we added callbacks to export and import clauses.

### B. Portfolio and Diversification

As depicted in Fig. 1, P-KISSAT-MAB implements a 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 [3].

### C. Controlling the Flow of Shared Clauses

In P-KISSAT-MAB, the sharing strategy ControlFlow is inspired by the one used in [3] and [4]. As highlighted in Fig. 1, we instantiate one sharer, each of the solvers are both producers and consumers. The sharer gets clauses from these producers and exports some of them to all others (the consumers).

The exchange strategy is defined as follows: each solver exports clauses having an LBD value under a given threshold (it started with LBD = 2). Every 1.5 seconds, 1500 literals

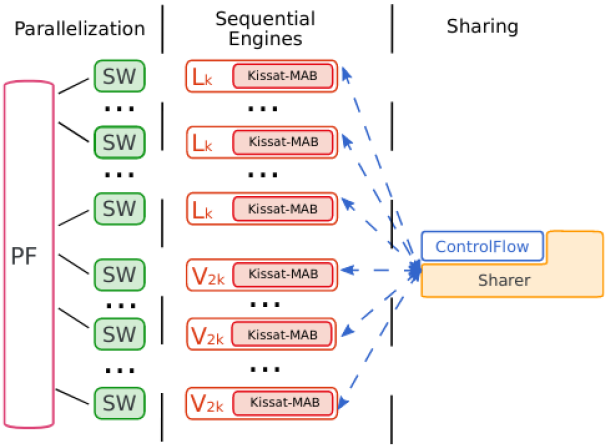



Fig. 1. Architecture of P-KISSAT-MAB

(the sum of the size of the shared clauses) are selected from each producer by the sharers and dispatched to consumers. The LBD threshold of the concerned solver is increased (resp. decreased) if an insufficient (resp. a too big) number of literals are dispatched (75% and 98%).

## References

[1] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Painless: a framework for parallel sat solving," in *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 233–250, Springer, 2017.

[2] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

[3] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 156–172, Springer, 2015.

[4] Y. Hamadi, S. Jabbour, and J. Sais, "Control-based clause sharing in parallel sat solving," in *Autonomous Search*, pp. 245–267, Springer, 2011.

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

[6] M. Sami Cherif, D. Habet, and C. Terrioux, "Un bandit manchot pour combiner CHB et VSIDS," in *Actes des 16èmes Journées Francophones de Programmation par Contraintes (JFPC)*, (Nice, France), June 2021.

[7] 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 (SAT)*, pp. 123–140, Springer, 2016.

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

# P-MCOMSPS: a parallel SAT solver with asynchronous clause strengthening

Vincent Vallade*, Souheib Baarir*‡, Julien Sopena*†,
Etienne Renault¶, Saeed Nejati‖, Vijay Ganesh§
*Sorbonne Université, LIP6, CNRS, UMR 7606, Paris, France
†INRIA, Delys Team, Paris, France
‡Université Paris Nanterre, France
§University of Waterloo, Waterloo, ON, Canada
¶ EPITA, LRDE, Kremlin-Bicêtre, France
‖ Amazon Web Services, Seattle, USA

***Abstract*—This paper describes the solver P-MCOMSPS submitted to the parallel track of the SAT Competition 2022.**

## I. Introduction

P-MCOMSPS is a concurrent portfolio-based [1] solver built using the Painless framework. The sequential engines spawned in this solver are instances of MapleCOMSPS, a solver that uses the LBD heuristics [2] to evaluate the quality of the learnt clauses both for garbage collection and as a filter for sharing. This version of P-MCOMSPS improves one particular component of the framework which is the one dedicated to the strengthening of learnt clauses called Reducer [3]. In the version submitted in the previous competition [4], this component was completely detached from the clauses sharing pipeline. If we describe the clauses shared by a particular solver as a stream, then the strengthening component added a parallel stream of reduced clauses. This as for consequence that the original clauses and the reduced ones were shared to all the solvers. It will be more efficient if each solvers only receives the clauses after reduction. This new implementation mitigates this problem, by placing the Reducer in a more central role in the sharing mechanism. The next section describes the overall behaviour of our competing instantiation named P-MCOMSPS. Its architecture is highlighted in Figure 1.

## II. P-MCOMSPS

### A. Setting-up MapleCOMSPS

As it is necessary to introduce significant diversification in the portfolio to ensure that each instance of the sequential algorithm has a different execution, MapleCOMSPS has been adapted for the parallel context as follows:

1) We parametrized the solver to use either LRB [5], or VSIDS [6] (resp. L and V in Figure 1) as branching heuristics.
2) We initialize the polarity values of the variables (a vector that decides whether a branching variable is assigned true or false) with a random value.
3) One thread activate Gaussian elimination preprocessing.
4) We parametrized the solver to use as a variable score comparator either $<$ or $<=$ (resp. head: H and tail: T). This only results in the initial branching order being reversed for one in relation to the other.

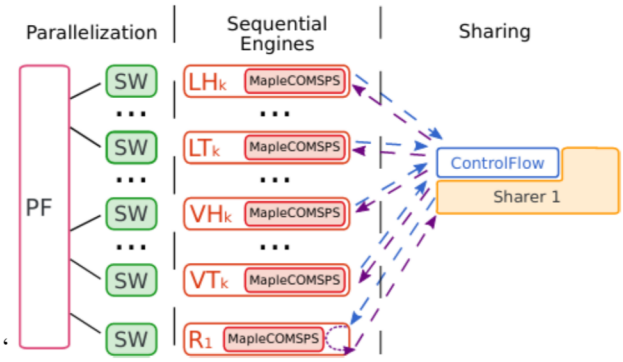



Fig. 1. Architecture of P-MCOMSPS

Each instance of the solver is executed by a thread called SequentialWorker (SW).

### B. Reducer

In the portfolio, two Reducer engines (R in Fig. 1) implement the algorithm introduced in [7]. This algorithm relies on a CDCL engine, in the present case a MapleCOMSPS solver. Since all clauses pass through this component before being distributed, the Reducer component will output a clause no mather if the clause has been reduced or not. We have taken advantage of the fact that our new sharing architecture puts all clauses through the strengthening mechanism to introduce a bloom filter that will prevent sharing clauses produced by this component that have already been seen in the past.

### C. Sharing strategy

In P-MCOMSPS, the sharing strategy is inspired by the one used in [8] and [9]. The exchange strategy is defined as follows: First, we divide the solvers into two groups of producers, each associated with a sharing thread called Sharer. A producer selects the clauses to be exported following a dynamic LBD threshold. At the beginning of the solving process, the threshold is LBD = 2. , if the Sharer feels

that a specific solver is sharing too few or too many clauses, then it can increase or decrease this threshold in consequence. In Figure 1, only one group of producers with its associated `Sharer` is represented. Every 0.75 seconds, the `Sharer` selects 1500 literals (the sum of the size of the shared clauses) from each producer and dispatch them to consumers.

As presented in the introduction, the `Reducer` takes a more central place in the strategy. Each regular CDCL solver sends their clauses to the `Sharer` (blue arrows). After the selection describes above, the `Sharer` sends these clauses to the `Reducer`. The `Reducer` sends back either the original or the reduced clauses depending on the succes of the strengthening algorithm. These clauses are then dispatch to the consumer solvers (purple arrows). Every solver in the Portfolio is a consumer.

## REFERENCES


[1] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

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

[3] V. Vallade, L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "On the usefulness of clause strengthening in parallel sat solving," in *Proceedings of the 12th NASA Formal Methods Symposium (NFM)*, Springer, 2020.

[4] V. Vallade, L. Le Frioux, R. Oanea, S. Baarir, J. Sopena, F. Kordon, S. Nejati, and V. Ganesh, "New concurrent and distributed painless solvers: P-mcomsps, p-mcomsps-com,p-mcomsps-mpi, and p-mcomsps-com-mpi," in *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*, p. 40, Department of Computer Science, University of Helsinki, Finland, 2021.

[5] 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 (SAT)*, pp. 123–140, Springer, 2016.

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

[7] S. Wieringa and K. Heljanko, "Concurrent clause strengthening," in *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 116–132, Springer, 2013.

[8] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 156–172, Springer, 2015.

[9] Y. Hamadi, S. Jabbour, and J. Sais, "Control-based clause sharing in parallel sat solving," in *Autonomous Search*, pp. 245–267, Springer, 2011.

# ParKissat: Random Shuffle Based and Pre-processing Extended Parallel Solvers with Clause Sharing

Xindi Zhang[1,2], Zhihan Chen[1,2], Shaowei Cai[1,2,*]

[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{zhangxd,chenzh,caisw}@ios.ac.cn

## I. Introduction

In this document, we introduce two parallel solvers called ParKissat-RS and ParKissat-Pre, where ParKissat is short for Parallel Kissat. The main novelty of ParKissat-RS is to randomly shuffle the initial variable branching order for each thread, which shows surprisingly good performance on satisfiable instances. Additionally, random shuffle cooperates well with clause sharing, and also gains improvements on UNSAT instances. The main novelty of ParKissat-Pre is adding some pre-processing techniques and using both VMTF and LRB in the focused mode.

## II. Literal Assumption and Random Shuffle

Current parallel solvers in SAT Competitions can mainly be divided into the portfolio-based method and the partitioning-based method, and the first one is usually more powerful in SAT Competitions. By preliminary experiments, we noticed that there are some **crucial variables**. Given a formula $F$, a crucial variable $x$ is a partition pivot such that solving sub-formulae $F \wedge x$ and $F \wedge \neg x$ in two threads separately gains at least $2\times$ acceleration. Sometimes, we can even find a crucial variable with $10\times$ acceleration. In practice, it is hard to accurately find such a crucial variable heuristically. As a result, we design a simple parallel method called Literal Assumption (LA) that randomly assumes one literal for each thread. With the advantages of parallel, the more sampling, the more possibility to gain the crucial variable.

The LA method shows surprisingly good performance on satisfiable instances but is weak on the unsatisfiable instances. The completeness of LA on unsatisfiable instances needs to be guaranteed by two threads solving the paired sub-formulae. LA can be seen as random fixing a variable as the first branching variable, so we designed a generalized version called Random Shuffle (RS), which shuffles the initial branching order randomly. Each thread of RS is a normal CDCL procedure, and thus preserves the completeness of the CDCL solver. Additionally, we use two common techniques, clause sharing, and diversification, to improve the overall performance.

This work is supported by NSFC Grant 62122078.
* Corresponding author
- Xindi Zhang and Zhihan Chen are co-first authors, which are considered to have equal contributions.

## III. Solvers Submission

All solvers submitted to SC22 are summarized in Table I, this document mainly introduces the parallel solvers, and the sequential solvers can be found in our another document. Noting that all the submitted solvers follow the standard build and running rules of SC22.

TABLE I: Solvers Submitted to SC22

| Track | Solver | Key Techniques |
|---|---|---|
| Main | LSTech_Maple | Deep**er** cooperating CDCL and LS |
| | LSTech_Kissat | LSTech style rephasing on Kissat |
| | Kissat-Inc | LSTech's depth updating trick to Kissat |
| | Kissat-Pre | Pre-processing before Kissat-Inc |
| CaDiCaL | LSTech_CaDiCaL | LSTech style rephasing on cadical |
| Parallel | ParKissat-RS | Random Shuffle with clause sharing |
| | ParKissat-Pre | Pre-processing, Diversification |
| Anniversary | Kissat-Inc | Same as above |
| | LSTech_Maple | Same as above |
| | ParKissat-Pre | Same as above |

### A. ParKissat-RS

ParKissat-RS is a parallel solver built upon Kissat-MAB [2]. It uses the PaInLeSS [4] framework to implement the clause sharing method, and each thread shares at most 1500 literals from clauses with LBD$\leq$ 2 every $0.5s$. Diversification is also used to improve the robustness of instances from different categories. In detail, ParKissat-RS uses three parameters of Kissat [3] called *stable* (0/1/2), *target* (0/1/2) and *phase* (0/1) and decides whether using CCAnr [1]. Meanwhile, ParKissat-RS also uses a simplification method based on equivalent-literal substitution (ELS) and resolution checking (RC) to preprocess the CNF files, and the detailed information can be found in our sequential document this year.

### B. ParKissat-Pre

ParKissat-Pre is a parallel solver aiming to obtain with a balanced ability for both SAT and UNSAT. ParKissat-Pre removes the RS technique from ParKissat-RS and replaces the base CDCL solver with Kissat-bonus [6], which bumps the variable branching

score according to LBD. Different from ParKissat-RS, ParKissat-Pre changes the *stable*, *target* and *chrono* (0/1) parameters for diversification. Further, It allows to use the learning rate-based branching methods (LRB) [5] in the focused mode of Kissat-bonus, which in default only allows the Variable Move to Front (VMTF) branching method in this mode. Moreover, ParKissat-Pre extends the simplification method of ParKissat-RS with Fourier-Motzkin Variable Elimination (FME) to handle cardinality constraints and Gaussian Elimination (GE) to handle XOR equations.

## References


[1] S. Cai, C. Luo, and K. Su. Ccanr: A configuration checking based local search solver for non-random satisfiability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 1–8, 2015.

[2] M. S. Cherif, D. Habet, and C. Terrioux. Combining vsids and chb using restarts in sat. In *CP 2021*, pages 1–19, 2021.

[3] A. B. K. F. M. Fleury and M. Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION 2020*, page 50, 2020.

[4] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Painless: a framework for parallel sat solving. In *SAT 2017*, pages 233–250, 2017.

[5] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for sat solvers. In *SAT 2016*, pages 123–140, 2016.

[6] X. Zhang, S. Cai, and Z. Chen. Improving cdcl via local search. *SAT COMPETITION 2021*, page 42, 2021.

# BENCHMARK DESCRIPTIONS

# AWS CBMC Benchmarks

Ronak Fofaliya
*Amazon Web Services*
New York, USA
ronakf@amazon.com

Jim Grundy
*Amazon Web Services*
Austin, USA
jmgruj@amazon.com

Robert Jones
*Amazon Web Services*
Portland, USA
rbtjones@amazon.com

Kareem Khazem
*Amazon Web Services*
London, United Kingdom
karkhaz@amazon.com

Benjamin Kiesl
*Amazon Web Services*
Munich, Germany
benkiesl@amazon.de

Angelo Nakos
*Amazon Web Services*
Boston, USA
anakos@amazon.com

Michael Tautschnig
*Amazon Web Services*
Vienna, Austria
tautschn@amazon.at

Michael W. Whalen
*Amazon Web Services*
Minneapolis, USA
mww@amazon.com

***Abstract*—This document describes the CBMC benchmark formulas submitted by Amazon Web Services to the SAT Competition 2022.**

***Index Terms*—CBMC, formal verification, SAT solving**

## I. Introduction

The benchmarks submitted by Amazon Web Services (AWS) represent some of the hardest SAT problems generated by the bounded model checker CBMC [1] when verifying open-source code at AWS.

CBMC is a bounded model checker for C and C++ that is used to formally verify software correctness. To verify a given program, CBMC generates formulas in propositional logic and sends them to a SAT solver. These benchmarks represent a subset of the hardest formulas generated by CBMC during the verification of open source projects at AWS. The projects use CBMC as part of their continuous-integration pipelines to check properties such as memory safety, absence of undefined C language behavior, preservation of data structure invariants, and pre/post conditions describing some kinds of functional correctness.

## II. Significance of these Benchmarks

Several teams at AWS use CBMC to verify the correctness of their safety-critical source code. CBMC proofs are checked into the codebase and run in continuous-integration pipelines each time a contributor opens a pull request for one of the source-code repositories [2]. CBMC performance is therefore critical to developer productivity. The performance of SAT solvers on formulas produced by CBMC plays an important role since CBMC's runtime is often dominated by the runtime of the SAT solver. For example, when analyzing CBMC's runtime on hard problems of AWS's open-source projects (problems on which CBMC took more than 10 seconds), we found that about 90 percent of the runtime is spent inside the solver.

We have selected some of the hardest CBMC problems as benchmarks for the SAT competition. Improvements in solver technology that improve performance on these benchmarks will directly translate to developer productivity in an industrial setting.

## III. Overview of Source Projects

The formulas in this benchmark suite arise from the following open-source projects:

- FreeRTOS [3]
- AWS Encryption SDK for C [4]
- AWS IoT Over-the-air Update Library [5]
- s2n-tls [6]
- AWS C Common Library [7]

*FreeRTOS.* FreeRTOS is a real-time operating system (RTOS) for microcontrollers and small microprocessors. It includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors [8].

*AWS Encryption SDK for C.* The AWS Encryption SDK is a client-side encryption library designed to make it easy for developers to encrypt and decrypt data using industry standards and best practices. It enables developers to focus on the core functionality of their application, rather than on the details of encryption and decryption [9].

*AWS IoT Over-the-air Update Library.* The AWS IoT over-the-air update library provides extensive functionality for the management of firmware updates on IoT devices. For example, it provides functionality for the managament of notifications for newly available updates, the download of updates, and their cryptographic verification [5].

*s2n-tls.* s2n-tls is a C99 implementation of the TLS/SSL protocols that is designed to be simple, small, and fast. It was implemented with security as a priority [6].

*AWS C Common Library.* The AWS C Common library is the core C99 package for the AWS SDK for C, which simplifies the integration of applications with AWS services. The AWS C Common library includes cross-platform primitives, configuration, data structures, and error handling [7].

## References

[1] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[2] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow at amazon web services," *Softw. Pract. Exp.*, vol. 51, no. 4, pp. 772–797, 2021. [Online]. Available: https://doi.org/10.1002/spe.2949
[3] "FreeRTOS," https://github.com/FreeRTOS/, accessed: 2022-05-09.
[4] "AWS Encryption SDK for C," https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/introduction.html, accessed: 2022-05-09.
[5] "AWS IoT Over-the-air Update Library," https://github.com/aws/ota-for-aws-iot-embedded-sdk, accessed: 2022-05-09.
[6] "s2n," https://github.com/aws/s2n-tls, accessed: 2022-05-09.
[7] "AWS C Common," https://github.com/awslabs/aws-c-common, accessed: 2022-05-09.
[8] "The FreeRTOS kernel," https://www.freertos.org/RTOS.html, accessed: 2022-05-09.
[9] "What is the AWS encryption SDK?" https://github.com/aws/aws-encryption-sdk-c, accessed: 2022-05-09.

# Hardware Model Checking Certificates

Emily Yu Nils Froleyks
Johannes Kepler University Linz, Austria

Armin Biere Mathias Fleury
University of Freiburg, Germany

## I. Certifying model checking results

The proposed benchmark set is obtained from our recent work on certifying $k$-induction-based model checking [1]. The model checking technique $k$-induction [2] is widely used for verification. A safety property is said to be $k$-inductive iff it satisfies the following two cases: it holds for $k-1$ consecutive steps originating from the initial states; if it holds for $k-1$ steps of unrolling, it also holds at the next state after one transition.

The key idea of certifying a model checking result in a generic form is to generate an inductive invariant as a proof certificate which implies the safety property in the given model. The certification framework proposed in [1] reduces the certification problem to five SAT checks and a one-alternation QBF, allowing certification at a low complexity. In a nutshell, the approach extends the given model to a larger circuit (namely $k$-witness circuit) with an inductive invariant. Two SAT checks and one QBF check are used to verify the so-called combinational simulation relation between the original circuit and the $k$-witness circuit. Furthermore, another three SAT checks are generated for checking the inductive invariant in the $k$-witness circuit.

The size of the $k$-witness circuit is in principle linear in the size of the original circuit and the value of $k$. One of the most essential features is that it includes $k$ copies of the original machine, and allows multiple ways of initialisation. The new property in the $k$-witness circuit (the inductive invariant) is composed of five sub-properties.

To verify $\phi(I, L)$ (a formula over the inputs and latches) is indeed an inductive invariant in the $k$-witness circuit, the following formulas are generated:

- Initiation check ($R(L) \Rightarrow \phi(I, L)$): the inductive invariant must hold at all initial states.
- Consistency check ($\phi(I, L) \Rightarrow P(I, L)$): the inductive invariant must hold at all good states.
- Consecution check ($U_1 \wedge \phi(I_0, L_0) \Rightarrow \phi(I_1, L_1)$): the inductive invariant is preserved during one transition.

## II. Generated instances

The certification approach is implemented into a toolkit Certifaiger [1], which takes as inputs a model in AIGER format [3] and a value of $k$ which is usually provided by a model checker (here we used McAiger [4]). The toolkit originally uses Kissat [5] as the underlying SAT solver, however, in this paper we modified it to use MiniSAT instead.

All benchmarks are obtained by running against a subset of HWMCC'2010 [6] benchmarks which McAiger successfully terminated with. We found 7 SAT formulas for which MiniSAT experienced a timeout of 15 minutes. (However, they were originally solved by Kissat and are unsatisfiable.) Among those, one is an initiation check, and the rest are consecution checks. We then add these 7 formulas to our benchmark set.

As the nature of $k$-induction, if a property is m-inductive in a model for some arbitrary m, it is also $n$-inductive in the same model for any n such that $n > m$. Therefore we obtained another 3 benchmarks from the same set by scaling the inductive depths to 500.

As for the satisfiable instances, we added the 4 instances with the same inductive depth 80 to the benchmark set, which originally were timed out on Kissat mentioned in the paper [1] from the pj20 family. We used McAiger to inspect the inductive depths, which confirms that the values of k are greater than 80, making the consecution check fail thus the formulas satisfiable. With the same logic, we obtained further 6 benchmarks by scaling the inductive depths. The names of the benchmarks are composed by following the convention: original benchmark name + "_$k$" + inductive depth. There are two exceptions starting with the name "bobsmdct_init", where "bobsmdct" is the model name and "_init" is for explicitly stating initiation checks.

## References

[1] E. Yu, A. Biere, and K. Heljanko, "Progress in certifying hardware model checking results," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 12760. Springer, 2021, pp. 363–386.

[2] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, ser. Lecture Notes in Computer Science, vol. 1954. Springer, 2000, pp. 108–125.

[3] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 11/2, 2011.

[4] A. Biere and R. Brummayer, "Consistency checking of all different constraints over bit-vectors within a SAT solver," in *FMCAD*. IEEE, 2008, pp. 1–4.

[5] A. Biere, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021," in *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2021-1. University of Helsinki, 2021, pp. 10–13.

[6] A. Biere and K. Claessen, "Hardware model checking competition 2010," 2010, http://fmv.jku.at/hwmcc10/.

# Minimum Disagreement Parity (MDP) Benchmark

Randal E. Bryant
Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Email: Randy.Bryant@cs.cmu.edu

## I. Obtaining Benchmarks

The benchmark generator, files, and documentation are available at:

https://github.com/rebryant/mdp-benchmark

## II. Description

Crawford, Kearns, and Shapire proposed the Minimum Disagreement Parity (MDP) Problem as a challenging SAT benchmark in an unpublished report from AT&T Bell Laboratories in 1994 [1].

MDP is closely related to the "Learning Parity with Noise" (LPN) problem. LPN has been proposed as the basis for public key crytographic systems [2]. Unlike the widely used RSA cryptosystem, it is resistant to all known quantum algorithms [3]. The capabilities of SAT solvers on MDP is therefore of interest to the cryptography community. We provide these benchmarks as a way to stimulate the SAT community to expand beyond pure CDCL, incorporating other solution methods into their SAT solvers.

Crawford wrote a benchmark generator in the 1990s and supplied several files to early SAT competitions with names of the form par$N$-$S$.cnf, where $N$ is the size parameter and $S$ is a random seed. These had values of $N \in \{8, 16, 32\}$. These files are still available online at https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html. SAT solvers of that era were challenged by $N = 16$ and could not possibly handle $N = 32$. Unfortunately, the code for his benchmark generator has disappeared.

We wrote a new benchmark generator for the MDP problem. In doing so, we added more options for problem parameters and encoding methods. We also replaced the binary encoding of at-most-$k$ constraints with a more SAT-solver-friendly unary counter encoding [4].

## III. Problem Description

In the following, let $\mathcal{B} = \{0, 1\}$ and $\mathcal{N}_p = \{1, 2, \ldots, p\}$. Assume all arithmetic is performed modulo 2. Thus, if $a, b \in \mathcal{B}$, then $a + b \equiv a \oplus b$.

The problem is parameterized by a number of solution bits $n$, a number of samples $m$, and an error tolerance $k$, as follows. Let $\boldsymbol{s} = s_1, s_2, \ldots, s_n$ be a set of *solution* bits. For $1 \leq j \leq m$, let $X_j \subseteq \mathcal{N}_n$ be a *sample set*, created by generating $n$ random bits $x_{1,j}, x_{2,j}, \ldots x_{n,j}$ and letting $X_j = \{i | x_{i,j} = 1\}$. Let $\boldsymbol{y} = y_1, y_2, \ldots, y_m$ be the parities of the solution bits for each of the $m$ samples:

$$y_j \quad = \quad \sum_{i \in X_j} s_i \tag{1}$$

Given enough many samples $m$ for there to be at least $n$ linearly independent sample sets, the values of the solution bits $\boldsymbol{s}$ can be uniquely determined from $\boldsymbol{y}$ and the sample sets $S_j$ for $1 \leq j \leq m$ by Gaussian elimination. To make this problem challenging, we introduce "noise," allowing up to $k$ of these samples to be "corrupted" by flipping the values of their parity. That is, let $T \subseteq \mathcal{N}_m$ be created by randomly choosing $k$ values from $\mathcal{N}_m$ without replacement, and define $m$ "corruption" bits $\boldsymbol{r} = r_1, r_2, \ldots, r_m$, with $r_j$ equal to 1 if $j \in T$ and equal to 0 otherwise. We then provide noisy samples $\boldsymbol{y'}$, defined as:

$$y'_j \quad = \quad r_j + \sum_{i \in X_j} s_i \tag{2}$$

and require the correct solution bits $\boldsymbol{s}$ to be determined despite this noise. That is, the generated solution $\boldsymbol{s}$ must satisfy at least $m - k$ of equations (1). For larger values of $k$, the problem becomes NP-hard.

This problem can readily be encoded in CNF with variables for unknown values $\boldsymbol{s}$ and $\boldsymbol{r}$, along with some auxilliary variables. We further parameterize the problem with a value $t \leq k$, indicating the maximum number of corrupted samples accepted in the solution, where the problem should be satisfiable when $t = k$ but may become unsatisfiable for $t = k - 1$. Each of the $m$ equations (2) is encoded using auxilliary variables to avoid exponential expansion. An at-most-$t$ constraint is imposed on the corruption bits $\boldsymbol{r}$.

For $t = k$, the solution $\boldsymbol{s}$ is not guaranteed to be unique, but we allow any solution that satisfies at least $m - k$ of the constraints (1). In addition, setting $t = k - 1$ does not guarantee that the formula is unsatisfiable. Indeed, we found some instances where there was a solution that satisfied $m - k + 1$ constraints.

By analyzing the CNF file, it is possible to discern the sample sets $S_j$ and the values of the noisy samples $\boldsymbol{y'}$. The values of $\boldsymbol{s}$ and $\boldsymbol{r}$, however, remain hidden, except as comments at the start of the file.

Crawford suggests choosing $n$ to be a multiple of 4 and letting $m = 2n$ and $k = m/8 = n/4$. Our benchmark files were all generated under those conditions.

## IV. Provided files

The generator `mdp-gen.py` and an associated README file are located in the `src` subdirectory. This directory also contains a program `mdp-check.py`. Given a `.cnf` file generated by `mdp-gen.py` and the output of a successful run of a SAT solver, it can check that the solution for the input variables representing $\boldsymbol{s}$ indeed satisfies at least $m - k$ of the equations of (1). The supplied benchmark files were generated by running the script `generate.sh` in the `src` subdirectory.

There are 30 files: five satisfiable and five unsatisfiable instances for $n \in \{28, 32, 36\}$, generated using five different random seeds. The seeds were adjusted so that the 15 instances generated with $t = k - 1$ are all unsatisfiable, as is discussed below.

We tested these benchmarks using the KISSAT [5] CDCL solver. Measurements were peformed on a 3.2 GHz Apple M1 Max processor with 64 GB of memory and running the OS X operating system, with a time limit of 5000 seconds per run. For $n = 28$, KISSAT can easily handle both the satisfiable and the unsatisfiable instances, with times ranging from 30 to 900 seconds. For the satisfiable instances with $n = 32$, it can solve some in just 30 seconds, but times out for two of the five runs. It times out on all unsatisfiable instances for $n = 32$, and it times out on all satisfiable and unsatisfiable instances for $n = 36$.

We also tested the benchmarks with CRYPTOMINISAT, a CDCL solver augmented with the ability to perform Gauss-Jordan elimination on parity constraints [6]. It can easily handle all satisfiable instances, never requiring more than 90 seconds. When not required to generate a proof of unsatisfiability, it can also easily handle all of the unsatisfiable instances. That is how we ensured that the instances with $t = k - 1$ are unsatisfiable. Currently, CRYPTOMINISAT cannot generate DRAT proofs of unsatisfiability when it uses Gauss-Jordan elimination, and so it fares no better than KISSAT on the unsatisfiable instances when proof generation is required.

CRYPTOMINISAT can scale to $n = 60$ without difficulty. Nonetheless, the problem is still NP-hard, and so even CRYPTOMINISAT only pushes the boundary before exponential scaling limits feasibility.

## References


[1] J. M. Crawford, M. J. Kearns, and R. E. Schapire, "The minimal disagreement parity problem as a hard satisfiability problem," 1994. [Online]. Available: http://www.cs.cornell.edu/selman/docs/crawford-parity.pdf

[2] J. Katz, "Efficient cryptographic protocols based on the hardness of learning parity with noise," in *Cryptography and Coding*, ser. LNCS, vol. 4887, 2007, pp. 1–15.

[3] K. Pietrzak, "Cryptography from learning parity with noise," in *SOFSEM 2012: Theory and Practice of Computer Science*, ser. LNCS, vol. 7147, 2012, pp. 99–114.

[4] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," in *Principles and Practice of Constraint Programming (CP)*, ser. LNCS, vol. 3709, 2005, pp. 827–831.

[5] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[6] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Proc. of the 12th Int. Conference on Theory and Applications of Satisfiability Testing* (*SAT 2009*), ser. LNCS, vol. 5584, 2009, pp. 244–257.

# Verifying Optimums of Weighted (Partial) Max-SAT Formulas

Mohamed Sami Cherif, Djamal Habet and Cyril Terrioux
Aix-Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
{mohamed-sami.cherif, djamal.habet, cyril.terrioux}@univ-amu.fr

***Abstract*—Checking whether a certain bound holds over a set of relaxation variables is a subproblem which often arises in the context of Maximum Satisfiability (Max-SAT) solving and particularly SAT-based solving. This document describes a collection of SAT instances that have been submitted to the 2022 SAT competition. These instances are derived from weighted (partial) Max-SAT formulas augmented with relaxation variables. An At-Most-K constraint is then set over these variables to check the validity of a provided bound. We use this process to verify known solutions of weighted (Partial) Max-SAT formulas.**

***Index Terms*—SAT, weighted Max-SAT, At-Most-K constraint**

## I. Introduction

The maximum satisfiability (Max-SAT) problem is an optimization extension of the satisfiability (SAT) problem. For a given formula in Conjunctive Normal Form (CNF), it consists in finding an assignment of the variables which maximizes the number of satisfied clauses. In Partial Max-SAT, clauses are divided into hard and soft clauses and the goal is to find an assignment that satisfies all hard clauses and maximizes the number of satisfied soft clauses. Another variant of this problem is weighted (partial) Max-SAT where each clause is associated to a positive weight in the input formula. Hard clauses which must be satisfied can be considered of infinite weight. For a given weighted CNF formula, this variant consists in finding an assignment satisfying all the hard clauses and maximising the sum of weights over satisfied soft clauses.

In recent years, Max-SAT solvers have achieved great breakthroughs by relying on SAT technology. In fact, complete methods for this problem include SAT-based approaches which iteratively call SAT solvers making them particularly efficient on industrial instances [5]. Checking whether a certain bound holds over a set of relaxation variables is a subproblem which often arises in the context of Max-SAT solving and particularly in SAT-based solving. For instance, Linear Search algorithms [3], [4] augment soft clauses with relaxation variables and add a CNF encoding over their sum to specify that the number of falsified soft clauses must be less than a given bound. A SAT solver is then iteratively called and the bound is increased (resp. decreased) until the formula becomes satisfiable (resp. unsatisfiable). Similarly to these algorithms, we rely on the fact that the optimum of a weighted (partial) Max-SAT formula is the threshold in which the formula becomes satisfiable to verify the validity of a given optimum. To this end, given a weighted (partial) Max-SAT formula and an integer value, we encode two SAT instances to check whether the given value is the threshold, i.e. the optimum of the formula.

## II. Verifying weighted (Partial) Max-SAT Optimums

First, we deal with unweighted (partial) formulas as specified in [1] then we show how to extend this encoding to weighted (partial) formulas. Let $\phi = H \cup S$ be a Partial Max-SAT formula where $H$ denotes the set of hard clauses and $S = \{c_1, \ldots, c_m\}$ the set of soft clauses. Let $k$ be an integer value. We define the following formula:

$$\phi_k = H \cup \{c_i \cup \{r_i\} \mid c_i \in S\} \cup CNF(\sum_{1 \leq i \leq m} r_i \leq k)$$

where $r_1, \ldots, r_m$ are new relaxation variables.

To verify that a given value $o$ is the optimum of a CNF formula $\phi$, it is sufficient to check that this value is the threshold in which the formula becomes satisfiable. To this end, we need to encode the formulas $\phi_{o-1}$ and $\phi_o$ and verify that $\phi_{o-1}$ is unsatisfiable and $\phi_o$ is satisfiable.

To deal with weighted formulas in the form $\phi = \{(C_1, w_1), \ldots, (C_m, w_m)\}$, we simply encode them as an unweighted (partial) formula $\phi' = H \cup S$ using a multiset for soft clauses as follows:

$$H = \{C \mid (C, \infty) \in \phi\}$$

$$S = \biguplus_{\substack{1 \leq i \leq m \\ w_i < \infty}} \left( \biguplus_{1 \leq j \leq w_i} \{C_i\} \right)$$

where $\biguplus$ denotes the sum operation on multisets, which maintains the multiplicity of elements after union. Note that a new literal $l_i$ can be added for each non unit soft clause $C_i$ where $1 < w_i < \infty$ in order to avoid duplicating whole clauses as follows:

$$H = \{C \mid (C, \infty) \in \phi\} \cup$$
$$\{l_i \leftrightarrow C_i \mid (C_i, w_i) \in \phi,\ |C_i| > 1 \text{ and } 1 < w_i < \infty\}$$

$$S = \left( \biguplus_{\substack{1 \leq i \leq m \\ w_i = 1}} C_i \right) \biguplus \left( \biguplus_{\substack{1 \leq i \leq m \\ |C_i| = 1 \\ w_i < \infty}} \left( \biguplus_{1 \leq j \leq w_i} C_i \right) \right) \biguplus \left( \biguplus_{\substack{1 \leq i \leq m \\ |C_i| > 1 \\ w_i < \infty}} \left( \biguplus_{1 \leq j \leq w_i} \{l_i\} \right) \right)$$

## III. The Submitted Benchmark

We consider the argumentation framework synthesis [7] (af-synthesis) family in the 2020 Max-SAT Evaluation[1]. We picked the 12 instances which were used in the weighted benchmark. Note that the optimums of these instances are known as they were solved each by at least one of the solvers submitted to the evaluation. Taking these optimum values into account, we apply the previous encoding for each instance. The benchmark we submit to the 2022 SAT competition thus includes 24 instances in total with 12 satisfiable instances and 12 unsatisfiable ones. We maintain the same naming conventions for the instances except that we add '$\_sat$' or '$\_unsat$' to each formula indicating respectively whether it is satisfiable or unsatisfiable. We used the PySAT library [2] to add the cardinality constraints (i.e. At-Most-K constraints) over the relaxation variables. The encoding chosen for these constraints is the modulo totalizer for k-cardinality (mktotalizer) encoding [6]. Finally, it is important to note that, once the constraints added, the clauses in the resulting formulas are shuffled.

## References


[1] M. S. Cherif, D. Habet, and C. Terrioux. Verifying Optimums of (Partial) Max-SAT Formulas. In *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Series of Publications B*, page 49. University of Helsinki, 2021.

[2] A. Ignatiev, A. Morgado, and J. Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[3] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. Qmaxsat: A partial max-sat solver system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8, 01 2012.

[4] D. Le Berre and A. Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability Boolean Modeling and Computation*, 7:59–64, 07 2010.

[5] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18, 10 2013.

[6] A. Morgado, A. Ignatiev, and J. Marques-Silva. MSCG: robust core-guided maxsat solving. *J. Satisf. Boolean Model. Comput.*, 9(1):129–134, 2014.

[7] A. Niskanen, J. P. Wallner, and M. Järvisalo. Synthesizing argumentation frameworks from examples. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence*, ECAI'16, page 551–559, NLD, 2016. IOS Press.


[1] https://maxsat-evaluations.github.io/2020/

# The Graceful Production Problem

Md Solimul Chowdhury
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA.
mdsolimc@cs.cmu.edu

***Abstract*—A production manager at the *FantasyElectronics* corporation is working on tackling a production challenge for their new product *iImagine*, which is a multi-purpose gadget. The manager needs to devise a production plan based on the requirements set by top managements at *FantasyElectronics*. First, he needs to make sure that such a production plan exists which respects these requirements.**

**The company has one production *factory* with $u \geq 1$ production units.**

1. **Each unit produces 0 or more gadgets each day, some of which may have manufacturing defects and are not functioning.**
2. **The capacity of each unit limits a maximum of $m \geq 0$ *functioning* gadgets each day.**

**The production venture of *iImagine* needs to fulfill the following requirements: (i) each day, these $u$ units need to produce at least $u * k$ number of *iImagaine* in total, where $k \geq 1$, (ii) the total number of functioning *iImagaine* produced in a day needs to be higher or equal to its previous day's total, and (iii) For any given day, for these $u$ units, the minimum number of gadgets produced must be greater than the minimum number of gadgets produced in the previous day. Is it possible for these units to run for $D \geq 2$ days, while satisfying these requirements? The manager needs to find answer of this question.**

**We call this problem Graceful Production (GP) Problem. We formulate GP as a SAT problem. We have submitted 20 instances of GP to the SAT Competition-2022.**

## I. SAT ENCODING OF THE GP PROBLEM

### A. GP as a SAT Benchmark

Each one of these $u > 1$ units has maximum production limit of $m \geq 0$ per day. These $u$ units must run for $D \geq 2$ days, where the production venture must respect the following three constraints:

***atleast:*** In any given day, the total number of gadgets produced by $u$ units must be at least $u * k$, where $k > 1$.

***steady:*** For any two consecutive days $d$ and $d'$, where $d' = d + 1$, the total number of gadgets produced in $d'$ is greater or equal to the number of gadgets produced in $d'$.

***robust:*** For any two consecutive days $d$ and $d'$, the minimum number of gadgets produced by $u$ units in day $d'$ must be higher or equal to the minimum number of gadgets produced in day $d$. Fig 1 shows an example of the GP problem.

Here, we encode the GP problem as a SAT benchmark. Let $p_i^d$ be the number of gadgets produced by unit $i$ in day $d$, where $1 \leq d \leq D$ and $1 \leq i \leq u$. Given a GP problem, we encode it as a SAT formula $F_{GP}$ as follows

$$F_{GP} = F_{atleast} \cup F_{steady} \cup F_{robust} \cup F_{ends}$$

| units | | | | day | sum | min |
|---|---|---|---|---|---|---|
| 1 | 11 | 12 | 3 | 1 | 27 | 1 |
| 18 | 2 | 10 | 9 | 2 | 29 | 2 |
| 10 | 5 | 20 | 8 | 3 | 43 | 5 |
| 7 | 13 | 12 | 12 | 4 | 44 | 7 |

Fig. 1: An example of the GP problem. It shows four days of operations for four production units. Number in black in each squared cell in the leftmost grid represents the number of gadgets produced by an unit on a given day. In this example, In each day, `sum` and `min` of productions are higher than the previous day's `sum` and `min`.

, where, $F_{atleast}, F_{steady}, F_{robust}$, and $F_{ends}$ are defined as follows:

$$F_{atleast} : \bigwedge_{d=1}^{D} \sum_{i=1}^{u} p_i^d \geq u * k$$

$$F_{steady} : \bigwedge_{d=1}^{D-1} \sum_{i=1}^{u} p_i^{d+1} \geq \sum_{i=1}^{u} p_i^d$$

$$F_{robust} : \bigwedge_{d=1}^{D-1} \texttt{min}\,(p_1^d \ldots p_u^d) \leq \texttt{min}\,(p_1^{d+1} \ldots p_u^{d+1})$$

$$F_{ends} : \bigwedge_{d=1}^{D} \neg p_0^d \wedge \neg p_{n+1}^d$$

Over $D$ days,

- $F_{atleast}$ encodes the *atleast* constraint.
- $F_{steady}$ encodes the steady constraint.
- $F_{robust}$ encodes the robust constraint.
- $F_{ends}$ encodes the assertion that left unit (resp. right unit) of the leftmost (resp. rightmost) always produces 0 gadgets, which marks the horizon of the factory.

$F_{GP}$ is SATISFIABLE, if the factory can run for $D$ days by conforming to the *atleast*, *steady*, *robust*, and *ends* constraints, otherwise, it is UNSATISFIABLE.

## II. Problem Modeling and Instance Generation for the GP benchmarks

### A. Problem Modeling

**picat** [1] is a CSP solver, which accepts a CSP problem and converts it to a SAT CNF formula, which is inturn solved by a SAT solver hosted by **picat**. Before solving the converted CNF formula, **picat** outputs the CNF formula.

To generate instances for the GP benchmark, we use this CNF generation feature of **picat**. First, we modelled the GP problem in a **picat** program $picat_{GP}$. Then, for a given set of parameter values for $(D, m, k, u)$, we use this $picat_{GP}$ model to generate CNF $F_{GP}$ by exploiting the CNF generation functionality of **picat**.

### B. Instance Generation

We have generated a set of $F_{GP}$ instances with the $picat_{GP}$ by varying the parameters $D$, $k$, and $u$, while setting $m$ (maximum production limit of an unit per day) to a fixed value of 1,000. From this set of instances, we have submitted 20 instances for SAT competition-2022 (CNF file has the the following format $\text{GP}_{D\_u\_k}$).

## References

[1] Picat, http://picat-lang.org/resources.html, Accessed: 2020-04-09

# Bounded Model Checking Instances generated by ABC-BMC

Fei Geng, Lei Yan and ShuCheng Zhang
*TCS Lab, Huawei Technologies, Beijing, China*
{gengfei12, david.yan, zhangshucheng}@huawei.com

***Abstract*—This paper describes our benchmarks for SAT Competition 2022. The formulas are generated by ABC-BMC2 engine with satoko as sat solver engine. The input files for the BMC engine are from the HWMCC (Hardware Model Checking Competition), which use the AIG format.**

## I. Introduction

Sat solver is one of the most important tools for bit-level model checking. BMC [1] (bounded model checking) has been one of the most effective algorithms for model checking since it was proposed by Armin Biere in 2003. Another SAT-based model checking algorithm is IC3 [2] , different from the BMC algorithm, the scale of instances to be solved by IC3 sat solver will be decreased significantly compared to BMC due to the feature of unrolling. Therefore, the performance requirements of IC3 for the sat solver are not as high as those of algorithm.

For a safety property, the BMC algorithm will copy the entire circuit network each time frame. In the $k$-$th$ step, the sequential circuit will be expanded into a connection of $k$ non-sequential circuits, and the non-sequential circuit will be encoded into CNF format and handed over to the sat solver. If the solver gives the UNSAT result, it means that the BAD state(also called unsafe state) is unreachable within $k$ steps. If the solver gives the SAT result, it means that the BMC algorithm has found a witness at time $k$. The BMC algorithm can generally only be used to find counter-example and cannot be used to prove safety.

Since the BMC needs to expand the encoding based on the circuit network of the previous state at each time frame, the circuit scale (also the CNF scale) will increase linearly with iteration. Therefore, it is difficult for the BMC to find out solution when the BAD state is hidden deeply, and the performance requirements of the sat solver are also very high.

## II. Features

- Like the features of the circuit instances, once the circuit input and initial latch values are determined, the values of all nodes will be determined, and more specific circuit characteristics are determined by the original AIG file.
- The entire CNF has sub-structural similarity, and the circuits added in each step are isomorphic but have different index.
- The BMC encoder will return a literal representing the BAD state. This bad state will be given to the sat solver as an assumption for solving, which can be regarded as a unit clause in CNF.

## III. Benchmark Selection

We used the single track benchamrk of HWMCC [3] (Hardware model checking competition) 2017 as the AIG input, and used the BMC2 engine with satoko in the ABC [4] tool to solve the safety property checking problem. For deriving CNF files, we called $satoko\_write\_dimacs$ to dump the CNF file when BMC try to invoke satoko, and add the assumption literal as unit clause in the end of the dumping file. Half of the instances originate from the AIG that cannot be proved or solved by any model checker in the competition. The case naming rule is $name\_IterX.cnf$, $name$ refers to the instance name of the Aiger format source file of HWMCC, $X$ indicates that the BMC engine generates the CNF in step $X$ for the sat solver to solve, the original AIG files can be downloaded here http://fmv.jku.at/hwmcc17/hwmcc17-single-benchmarks.tar.xz

## References

[1] Biere A, Cimatti A, Clarke E M, et al. Bounded model checking[J]. 2003.
[2] Bradley A R. SAT-based model checking without unrolling[C] International Workshop on Verification, Model Checking, and Abstract Interpretation. Springer, Berlin, Heidelberg, 2011: 70-87.
[3] Hardware Model Checking Competition 2017. [Online] http://fmv.jku.at/hwmcc17/
[4] U. Berkeley, ABC: A system for sequential synthesis and verification, Berkeley Logic Synthesis and Verification Group. 2009.

# Unique Reconfiguration Sequence

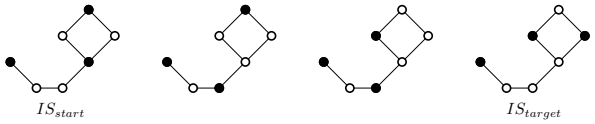
Nils Froleyks, Emily Yu
*Johannes Kepler University Linz*
*Austria*

Armin Biere
*Albert Ludwigs University Freiburg*
*Germany*

The independent Set Reconfiguration Problem (ISP) was the topic of the Core Challenge 2022[2]. For a detailed description, see their website.

Given an undirected graph $G$ and two independent sets $IS_{start}$ and $IS_{target}$ of $G$, find a sequence of independent sets starting with $IS_{target}$ and ending in $IS_{target}$, such that two consecutive sets only differ in one node.

The presented encoding was used to find the longest valid reconfiguration sequence, therefore we added a uniqueness constraint to the problem definition: No independent set in the sequence may appear more than once.




The encoding for a given makespan is straight-forward:

- For each step we have one variable for each node $v$ in $G$, encoding if $v$ is part of the current independent set.
- For each set we add one clause for each edge in $G$ to enforce independence.
- For two adjacent steps we add the constraint that the number of variables $\top$ in one and $\bot$ in the other is exactly one, we do this in both directions. For the at-most-one part, we use a Cartesian-product-encoding [1].
- Lastly, we ensure that each set is different from every previous set in at least one variable.

The benchmark set contains 10 satisfiable and 10 unsatisfiable benchmarks. There are three reasons why an endcoding might be unsatisfiable:

1) The makespan is too low to reach the target set.
2) There are shorter reconfiguration sequences that reach the target, but under the uniquness constraint it is not possible to reach it with the given makespan.
3) The ISP is unsolveable independent of the makespan.

We included examples for all cases.

The benchmarks are named according to the following convention:

`reconf<10/20>_<makespan>_<ISP name>.cnf`

Where a 10 indicates satisfiablity and a 20 unsatisfiablity. The last part is a shortened version of the original ISP instance name from the Core Challenge.

*Acknowledgements:* This work is supported by the Austrian Science Fund (FWF) under projects W1255-N23 and S11408-N23 as well as the LIT AI Lab funded by the State of Upper Austria.

## References

[1] Jingchao Chen. "A New SAT Encoding of the At-Most-One Constraint". In: *Proc. Constraint Modelling and Reformulation* (2010), p. 8.

[2] Takehiro Ito, Yoshio Okamoto, and Takehide Soh. *Core Challenge 2022*. URL: https://core-challenge.github.io/2022/.

# Group ring units in SAT

Giles Gardam
*Mathematical Institute*
*University of Münster*
Münster, Germany
ggardam@uni-muenster.de

***Abstract*—We describe SAT benchmarks encoding the existence of non-trivial units in group rings.**

## I. Group ring units

In algebra, the *group ring* $K[G]$ of a group $G$ over a field $K$ (e.g. the two-element field $\mathbb{F}_2$) consists of the finite formal $K$-linear sums of group elements. A *unit* is an element with a multiplicative inverse; a group is called *torsion-free* if it has no elements of finite order. It was a long-standing conjecture that the units of the group ring of a torsion-free group are all trivial, meaning of the form $kg$ for $k \in K$ and $g \in G$; a counterexample was given by the author in [1]. Finding non-trivial units amounts to solving a system of quadratic equations, which is naturally formulated as an instance of SAT if one works over $K = \mathbb{F}_2$.

## II. Encoding

The submitted benchmarks are a completely naive encoding of the problem. We enumerate the elements $g_1, g_2, \dots$ of the infinite group $G$ according to the shortlex order in some fixed generating set, for example:

$$1, x, x^{-1}, y, y^{-1}, x^2, xy, xy^{-1}, x^{-2}, \dots$$

For some $n$, we consider the first $n$ elements of the group as candidate support for the group ring elements. We thus have $2n$ variables defining the two group ring elements

$$\alpha = \sum_{i=1}^{n} a_i g_i,$$
$$\beta = \sum_{j=1}^{n} b_j g_j.$$

By translation in the group, we can break symmetry and impose unary clauses $a_1$ and $b_1$, and thus the non-triviality is simply asserted by the clauses

$$\bigvee_{i=2}^{n} a_i \quad \text{and} \quad \bigvee_{j=2}^{n} b_j.$$

We introduce the $n^2$ auxiliary variables $c_{ij} := a_i \wedge b_j$ to encode the products and then impose the XOR clause

$$\bigoplus_{\substack{i,j \text{ such that} \\ g_i \cdot g_j = g_k}} c_{ij} = \begin{cases} 1 & \text{if } k = 1 \\ 0 & \text{else} \end{cases}$$

for each group element $g_k$. These equations say precisely that $\alpha\beta = 1$ in $K[G]$. Technically, we also require $\beta\alpha = 1$ for $\alpha$ and $\beta$ to be units, but this extra condition is known to be redundant for all but the most exotic groups. Each XOR is, as is common practice, cut into smaller XORs on at most $\ell = 4$ variables (by introducing auxiliary variables), each of which is then encoded by $2^{\ell-1}$ CNF clauses. The longest sum has length $n$ or thereabouts; the statistics of the other lengths depends a lot on the specific group considered. For the submitted benchmarks, the average length ranges between 10 and 22 and is approximately 13 for the mid-sized problems.

## III. Examples

The specific groups in the benchmarks are the virtually abelian Hantzsche–Wendt group

$$P = \langle\, x, y \,|\, y^{-1}x^2y = x^{-2},\ x^{-1}y^2x = y^{-2} \,\rangle$$

used in [1] and the virtually nilpotent group

$$S = \langle\, x, y \,|\, x^{-1}y^2xy^2, x^{-2}yx^{-2}y^3 \,\rangle$$

introduced by Soelberg [2]. For each group we include the largest unsatisfiable instance and the smallest satisfiable instance, as well as those problem sizes rounded down or up respectively to a ball in the Cayley graph of the group (a natural sequence of values of $n$ to attempt), in this case corresponding to all words of length at most 4 or at most 5 in the generators. The values of $n$ for $P$ are $83, 92, 93, 147$ and for $S$ they are 109, 158, 159 and 223.

## References

[1] G. Gardam, "A counterexample to the unit conjecture for group rings," Ann. of Math., vol. 194, pp. 967–979, November 2021.
[2] L. Soelberg, "Finding torsion-free groups which do not have the unique product property," Master's thesis, Brigham Young University, 2018. Available: https://scholarsarchive.byu.edu/etd/6932.

# Description of CEC Benchmarks

Junhua Huang
Xiamen University, China

Hui-Ling Zhen, Wanqian Luo, Mingxuan Yuan
Noah's Ark Lab, Huawei, China

***Abstract*—All these benchmarks are from Combinational Equivalence Checking (CEC).**

## I. Introduction

CEC problem is from the problem of checking the equivalence of combinational circuits is of key significance in the verification of digital circuits [1]. It is one of the most widely used technologies in the verification of digital circuits. During synthesis process, a register-transfer level (RTL) description, given in e.g. VHDL or given as a set of boolean expression, is translated into a gate-level description.

## II. Encoding

The transformation is followed standard way.

- To make the CNF relatively hard to solve, we make some modifications on the circuit, like hash table and rewrite.
- To make some instance from UNSAT to SAT, we also consider to add certain bugs in some of instance.
- Random shuffling operation is also applied, in order to further increase the complexity of generated CNFs, which includes variable permutation, clause permutation and polarity flip.

Random shuffling operation is also applied, in order to further increase the complexity of generated CNFs, which includes variable permutation, clause permutation and polarity flip. In details, the variable permutation is to change the occurring order of original variables within each clause; and the clause permutation is to change the occurring order of clause within the CNF but the order of variables within each clause remains the same. The polarity flip is to randomly flip the polarity of literals of CNF.

## References

[1] Evguenii I Goldberg, Mukul R Prasad, and Robert K Brayton. Using sat for combinational equivalence checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 114–121. IEEE, 2001.

# Benchmarks encoding logical equivalence checking for sorting algorithms

Ilya Otpuschennikov, Alexander Semenov, Victor Kondratiev, Daniil Chivilikhin, Stepan Kochemazov*
* Email: veinamond@gmail.com

***Abstract*—This document describes the benchmarks encoding logic equivalence checking of different algorithms for sorting integer numbers which were submitted to SAT Competition 2022.**

## I. Introduction

Logic equivalence checking (LEC) [1] is a well-known application of SAT solvers. We consider the problem of proving that the outputs of two algorithms for sorting (the same) $l$ $d$-bit natural numbers are equivalent. It is clear, that as long as the encodings of the sorting algorithms are correct, the corresponding LEC instances are unsatisfiable. While it may seem that the corresponding problems pose no value (since we can usually formally prove for each sorting algorithm that it is correct), they can serve as a good source of unsatisfiable benchmarks with scalable difficulty for SAT solvers.

## II. Formal description

Consider two Boolean circuits $S_1$ and $S_2$ over a complete basis, e.g. $\{\neg, \wedge\}$. Assume that both circuits have $n$ inputs and $m$ outputs, thus defining functions

$$f_1, f_2 : \{0,1\}^n \to \{0,1\}^m$$

The goal is to prove that $f_1 \cong f_2$ (here by $\cong$ we mean pointwise equality). Then the circuits $S_1$ and $S_2$ are equivalent ($S_1 \cong S_2$). A well known fact is that this problem can be efficiently reduced to SAT for CNF $C$ such that $S_1 \cong S_2$ iff $C$ is unsatisfiable. CNF $C$ is constructed from circuits $S_1, S_2$ using Tseitin transformations [2].

In the provided benchmarks we considered circuits $S_1$, $S_2$ that represent two different algorithms for sorting $l$ $d$-bit natural numbers. In the role of sorting algorithms we used bubble sorting, selection sorting, insertion sorting and pancake sorting. To construct the SAT encodings we employed the `Transalg` tool [3].

The attractive feature of this approach to construction of unsatisfiable benchmarks lies in the fact that by varying sorting algorithms to compare and the values of $l$ and $d$ we can produce the benchmarks with almost any desired difficulty.

## III. Instance naming scheme

The benchmarks follow the simple naming convention:

$$Alg1 vs Alg2 Sort\_l\_d.cnf$$

where $Alg1, Alg2 \in \{Bubble, Pancake, Insert, Selection\}$, $l$ is the number of input natural numbers to sort, and $d$ is the number of bits per input number.

## IV. Comments

We submit 32 benchmarks, all of them unsatisfiable. They correspond to different combinations of aforementioned sorting algorithms with varying values of $l$ and $d$.

Overall trend is that given the same two algorithms, for the same $l$ the increase of $d$ by 1 results in the increase of the runtime of a solver by $\approx 2$ times. For the same $d$ the increase of $l$ by 1 results in $\approx 10$ times runtime increase.

To evaluate the hardness of submitted benchmarks we used `Kissat_sc2021_default` with the time limit of 1 hour on a PC with AMD Ryzen 9 3950x. Within the time limit it solved 24 benchmarks with runtime from 5 to 3400 seconds. The `MiniSat 2.2` runtime on these benchmarks is several times larger on average.

## References

[1] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *DAC 97*, 1997, p. 263–268. [Online]. Available: https://doi.org/10.1145/266021.266090

[2] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in Constructive Mathematics and Mathematical Logic, Part II, Seminars in mathematics,*, pp. 115–125, 1970.

[3] A. Semenov, I. Otpuschennikov, I. Gribanova, O. Zaikin, and S. Kochemazov, "Translation of algorithmic description of discrete functions to sat with application to cryptanalysis problems," *Logical Methods in Computer Science*, vol. 16, p. 29:1–29:42, 2020.

# Sports Timetabling SAT Benchmarks

Martin Mariusz Lester
*Department of Computer Science*
*University of Reading*
Reading, United Kingdom
m.lester@reading.ac.uk
0000-0002-2323-1771

***Abstract*—This benchmark consists of SAT encodings of sports timetabling problems for time-constrained double round robin (2RR) tournaments. The majority of instances are based on problems from the International Timetabling Competition 2021 (ITC 2021), but there are are also some more abstract instances based on achieving tight bounds on the number of breaks.**

## I. Introduction

A sports tournament timetable for $n$ teams over $s$ slots lists each pair of teams that will play in each slot. This benchmark encodes the problem of finding a timetable for a *time-constrained double round robin* tournament. This means that: each team plays exactly once in each round; and each team plays against every other team exactly twice, once "home" at the team's own stadium, and once "away" the the opposing team's stadium. Thus $s = 2(n-1) = 2n-2$.

A *break* occurs when the team plays 2 consecutive games at home or 2 consecutive games away. It is often desirable to minimise the number of breaks in a tournament, as they have been shown to disadvantage the affected teams. For the type of tournament we consider, the minimum number of breaks is $2(n-2)$ [1], [2].

This kind of tournament or league is extremely common in reality. However, real-world tournament design problems often come with a wide range of further constraints that must also be satisfied, for example requiring that certain games or sets of games must be timetabled at the same time or at different times.

## II. ITC 2021

The International Timetabling Competition 2021 (ITC 2021) [3], [4] concerned this kind of problem. Problem instances were supplied by the organisers in the XML-based RobinX format [5], expressing a mixture of hard and soft constraints. The goal for entrants was to develop a solver that could produce solutions satisfying all hard constraints and minimising the sum of costs of violated soft constraints.

Our entry to the competition, *Reprobate* [6], [7], used an encoding of sports timetabling problems into pseudoboolean (PB) constraints, which we solved using PB solvers such as *clasp* [8]. This approach was not competitive in terms of quality of solution when compared with other entries, most of which used a custom local search or commercial Mixed Integer Programming (MIP) solvers such as Gurobi and CPLEX. However, it did find feasible solutions to the majority of problems in under 10 minutes, which was much faster than the commercial solvers on an equivalent MIP encoding.

## III. Encoding

Our encoding uses decision variables $M_{t_1,t_2,s}$, which are true just if team $t_1$ plays home against team $t_2$ in slot $s$. Each team must play exactly once in each slot:

$$\forall s, t_1 . \sum_{t_2} (M_{t_1,t_2,s} + M_{t_2,t_1,s}) = 1$$

and each home/away match-up between two teams must occur exactly once:

$$\forall t_1, t_2 . \sum_{s} M_{t_1,t_2,s} = 1$$

Auxiliary variables are used to help encode other constraints, including those on the number of breaks.

## IV. Instances

To produce this benchmark, we used *Reprobate* to generate PB encodings of RobinX format sports timetabling problems, excluding any soft constraints. Then we used *pbencoder* from *pblib* [9] to translate these into SAT instances.

We split the benchmarks into:

- easy — solvable with MiniSAT 2.2.1 in under 1 minute;
- medium — solvable with *clasp* 3.3.5 on *crafty* preset in under 1 hour; and
- hard — not solvable with *clasp* in under 1 hour.

All timings are for an Intel i5-7500 CPU running at 3.40GHz.

The benchmark suite contains encodings of all instances from the ITC 2021, including test instances. These instances do not correspond to any specific real-world sports tournaments. They were generated to test the performance of RobinX solvers on a range of different combinations of constraints. *(23 easy, 11 medium, 19 hard.)*

The suite also contains encodings of instances generated specifically for the SAT competition with only one constraint beyond being a time-constrained double round robin. These instances range from 4 to 20 teams. There are 3 variations:

1) The number of breaks must be at most $k(n-2)$, for $k \in [2,5]$. For $k = 2$, this is the minimal number of breaks. *(24 easy, 1 medium, 11 hard.)*
2) The number of breaks must be at most $k(n-2)$, for $k \in [2,5]$. Furthermore, there must be no sequences of 3 consecutive slots containing breaks; Horbach and others

found this extra constraint improved performance in a SAT formulation of break minimisation [1]. *(19 easy, 8 medium, 9 hard.)*

3) The number of breaks must be at most $k(n-2)-1$. Unlike all the other instances, these are unsatisfiable. *(2 easy, 1 medium, 6 hard.)*

While one would probably expect the tighter bound on the number of breaks to be harder to satisfy, note that for single round robin tournaments, the problem of allocating to an otherwise fixed timetable a home/away pattern with $n-2$ breaks is solvable in polynomial time [10], but conjectured to be NP-complete in general [11]. So it is conceivable that the tighter bound is easier.

## REFERENCES


[1] A. Horbach, T. Bartsch, and D. Briskorn, "Using a sat-solver to schedule sports leagues," *J. Sched.*, vol. 15, no. 1, pp. 117–125, 2012. [Online]. Available: https://doi.org/10.1007/s10951-010-0194-9

[2] D. de Werra, "Geography, games and graphs," *Discret. Appl. Math.*, vol. 2, no. 4, pp. 327–337, 1980. [Online]. Available: https://doi.org/10.1016/0166-218X(80)90028-1

[3] D. Van Bulck, D. R. Goossens, J. Beliën, and M. Davari, "The Fifth International Timetabling Competition (ITC 2021): Sports timetabling," in *Proceedings of MathSport International 2021 Conference*, 2021, pp. 117–122. [Online]. Available: http://www.mathsportinternational.com/MathSport2021Proceedings.pdf

[4] ——, "Itc2021 — sports timetabling problem description and file format," 2020. [Online]. Available: https://www.sportscheduling.ugent.be/ITC2021/images/OrganizationITC2021_V7.pdf

[5] D. Van Bulck, D. R. Goossens, J. Schönberger, and M. Guajardo, "RobinX: A three-field classification and unified data format for round-robin sports timetabling," *Eur. J. Oper. Res.*, vol. 280, no. 2, pp. 568–580, 2020. [Online]. Available: https://doi.org/10.1016/j.ejor.2019.07.023

[6] M. M. Lester, "Reprobate at itc 2021," in *Proceedings of the 13th International Conference on the Practice and Theory of Automated Timetabling-PATAT*, vol. 2, 2021. [Online]. Available: http://www.patatconference.org/patat2020/proceedings/papers/42.ITC2021_paper8.pdf

[7] ——, "Reprobate: Pseudoboolean Optimisation for RobinX Sports Timetabling," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5084254

[8] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "*clasp* : A conflict-driven answer set solver," in *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, ser. Lecture Notes in Computer Science, C. Baral, G. Brewka, and J. S. Schlipf, Eds., vol. 4483. Springer, 2007, pp. 260–265. [Online]. Available: https://doi.org/10.1007/978-3-540-72200-7_23

[9] T. Philipp and P. Steinke, "Pblib – a library for encoding pseudo-boolean constraints into cnf," in *Theory and Applications of Satisfiability Testing – SAT 2015*, ser. Lecture Notes in Computer Science, M. Heule and S. Weaver, Eds. Springer International Publishing, 2015, vol. 9340, pp. 9–16.

[10] R. Miyashiro and T. Matsui, "A polynomial-time algorithm to find an equitable home-away assignment," *Oper. Res. Lett.*, vol. 33, no. 3, pp. 235–241, 2005. [Online]. Available: https://doi.org/10.1016/j.orl.2004.06.004

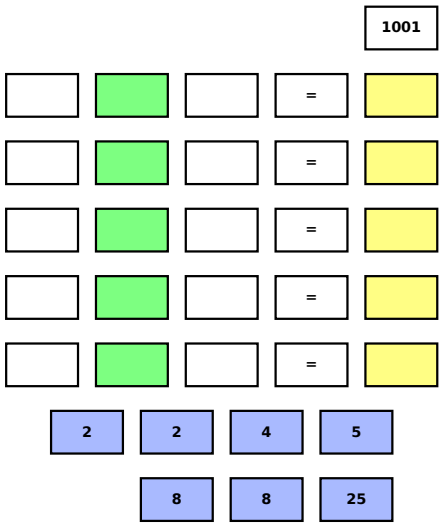
[11] M. Elf, M. Jünger, and G. Rinaldi, "Minimizing breaks by maximizing cuts," *Oper. Res. Lett.*, vol. 31, no. 3, pp. 343–349, 2003. [Online]. Available: https://doi.org/10.1016/S0167-6377(03)00025-7

# Solving Summle.net With SAT

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

## I. Introduction

The simple math puzzle Summle is presented on https://summle.net/. Given a target number $X$, and available input numbers, the goal is to combine the inputs with the math operations addition, subtraction, multiplication and division. Each input, and intermediate results, can be used exactly once. The task is to place the numbers and operations so that the target number is the result of an operation.




in the given example, the input numbers $2, 2, 4, 5, 8, 8, 25$ have to be placed in the empty input boxes (first and third column). Each input can be used at most once. Intermediate results can be used in operations once as well. The allowed operations are addition, subtraction, multiplication and division. The goal is to obtain the value we search for – 1001 – as an intermediate result. [1]

## II. Converting Summle to SAT

Similarly to puzzles like Sudoku, the math puzzle could be converted to a logic, and a reasoner could be applied to a direct translation. To assess the reasoning strength of other tools as well, we chose a different route.

The solution for a given Summle puzzle is encoded in the C programming language. The goal number and input numbers are represented by integer variables. For each position in the puzzle grid, an array of integers stores the position of the selected input number or intermediate result. Similarly, the operations used for the each calculation is selected and stored in an array. These two arrays, together with the number of given steps, form the search space of the puzzle.

The rest of the program verifies whether the chosen position selection and operations result in (1) valid calculations, and (2) that a calculation reaches the goal in a given number of steps. The exit code of the program should indicate whether a solution has been found, or not. This property is useful when using the program in combination with tools that try to generate input to crash the program. Only if a valid solution is reached, an assertion is triggered in the program, otherwise, the program terminates or aborts as usual. As assertions are typically not used in release binaries, we implemented the exit code for a valid solution to be the same as when triggering the assertion. Otherwise, the exit code of the binary will be zero.

The values for the selected-position-array and operations are not defined. Instead, these values are initialized from stdin. This allows tools like fuzzers to select place the numbers and operators in the puzzle and validate the result via the exit code.

Instead of a fuzzer, the bounded model checker CBMC [1] can be used to find a solution. CBMC finds a solution by trying to break the assertion, which is CBMC's default behavior. Additionally, CBMC can emit the formula in conjunctive normal form, which would be solved by the internal SAT solver. We use this feature to generate the submitted benchmark.

The default setup of CBMC take a few seconds to solve the problems presented on the web page of summle. To find more challenging, and unsatisfiable, problems, we extended the number of input numbers, use greater goal values, and experiment with the number of possible steps.

### A. Symmetries in Puzzle

The puzzle provides many symmetries. Addition and multiplication are commutative, so that for each of these operations at least two solutions exist. Symmetry-breaking additions to enforce one operand being less-equal to the other have not been added.

The order of calculations can be mixed, as the only requirement is to calculate intermediate results before the result is needed. Any earlier calculation could be used, multiplying the number of solutions further. No constraints to use intermediate results as late as possible, or similar, have been added.

[1] The given example has a solution, happy hunting.

## III. Using CBMC to solve Summle

The input file main.c has a single assertion. This assertion can only be reached and falsified if the placed numbers in the puzzle describe a valid solution to the puzzle. Hence, this assertion can be targeted by CBMC.

The board to solve has to be defined via compile time parameters. The example puzzle can be solved with the below command. To quickly spot the solution to the problem, you should filter the output for the lines that print the result of the calculation.

```
cbmc --property main.assertion.1
      --trace --trace-hex
      --object-bits 16
      --unwind 10 --depth 2000
          -DTYPE="unsigned short"
          -DGOAL=1001
          -DINPUTS=2,2,4,5,8,8,25
          -DSTEPS=5
      main.c
```

Besides specifying the properties of the input problem, the call shows that we currently use 16 bits to represent numbers – by using the `unsigned short` data type. The difficulty can be increased by lifting this format to 32 or even 64 bits. Furthermore, the number of loop unwinding and execution path depth are limited. With the used input number of steps and input values, these parameters have been good enough to cover the whole program.

## IV. Availability

The source of the tool is publicly available under the MIT license at https://github.com/conp-solutions/summle-solver. The repository also contains a script to generate a first set of not too simple benchmarks.

Note: CBMC can also produce the SMT format. We did not compare the performance of SAT solvers on the CNF files to SMT solvers on the SMT format.

## References

[1] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

# Verifying Linked List Safety Properties in AWS C99 Package with CBMC

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

***Abstract*—In this paper, state-of-the-art proofs are generated with harness using the CBMC bounded model checker for the Amazon Web Services C99 core package. In this submission, we check the safety properties of the *Linked List swap-contents* routine with various *loop unwinding* settings as opposed to the *String compare* submitted last year. The generated proof has proven to be reasonably hard to solve using modern SAT solvers. It has many variable-clause redundancies which are not only challenging for a SAT solver but also useful to assess the performance of different simplification techniques.**

## I. Introduction

Bounded Model Checking (BMC) [1]–[3] determines whether a model $M$ satisfies a certain property $\varphi$ expressed in temporal logic, by translating the model checking problem to a propositional satisfiability (SAT) problem or a Satisfiability Modulo Theories (SMT) problem. The term *bounded* refers to the fact that the BMC procedure searches for a counterexample to the property, i.e., an execution trace, which is bounded in length by an integer $k$. If no counterexample up to this length exists, $k$ can be increased and BMC can be applied again. This process can continue until a counterexample has been found, a user-defined threshold has been reached, or it can be concluded (via $k$-induction [2]) that increasing $k$ further will not result in finding a counterexample. CBMC [4], [5] is an example of a successful BMC model checker that uses SAT solving. CBMC can check ANSI-C programs. The verification is performed by *unwinding* the loops in the program under verification a finite number of times, and checking whether the bounded executions of the program satisfy a particular safety property [6]. These properties may address common program errors, such as null-pointer exceptions and array out-of-bound accesses, and user-provided assertions.

## II. Benchmarks

In this paper, we are interested in verifying the safety properties of the *swap-contents* routine implemented in the Linked List data structure of the Amazon Web Services (AWS) C99 core package. The proof covers the following:

- Memory allocation failure and access violations
- Pointer/floating-point overflow
- Data types conversion

We generated 30 different formulas using a *loop unwinding* upper-bound in the range $[40, 80]$, with an iincremental step. These bounds produce SAT formulas with 100% coverage of all functionalities. All problems are written in this format: `linked_list_swap_contents_safety_unwind<x>` where `x` denotes the unwinding value. The first and the last formulas are solved via MiniSat [7] within 90 and 540 seconds respectively on a machine with AMD EPYC 7H12 64-Core processor operating at base clock of 2.6 GHz. The solving time of the rest of the benchmarks are expected to be monotonically increasing.

## References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proc. of TACAS (Mar. 1999), Amsterdam, The Netherlands*, ser. LNCS, vol. 1579. Springer, 1999, pp. 193–207.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[3] M. Osama and A. Wijs, "GPU Acceleration of Bounded Model Checking with ParaFROST," in *Proc. of CAV (Jul. 2021), USA*, ser. LNCS, vol. 12760. Springer, 2021, pp. 447–460.

[4] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proc. of TACAS (Mar. 2004), Barcelona, Spain*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[5] D. Kroening and M. Tautschnig, "CBMC - C Bounded Model Checker - (Competition Contribution)," in *Proc. of TACAS (Apr. 2014), Grenoble, France*, ser. LNCS, vol. 8413. Springer, 2014, pp. 389–391.

[6] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View, Second Edition*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[7] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Proc. of SAT (May 2003) Santa Margherita Ligure, Italy*, ser. LNCS, vol. 2919. Springer, 2003, pp. 502–518.

# SAT-X Unsolved Problems Benchmarks

1st Oscar Riveros
Santiago, Chile
oscar.riveros@gmail.com

***Abstract*—A set of unsolved and recent solved problems benchmarks for SAT Competition 2022, this problemas are created with SAT-X python library https://github.com/maxtuno/SATX, that is a solver free advanced version of PEQNP python library presented on SAT Competition 2021 [1].**

## I. Introduction

To solve some problems like the sum of three cubes for 33, 42, or 3 (large representation), a large amount of computing power was required, then is very interesting try to solve this with SAT Solvers.

## II. Methods

### A. Descriptions of problems

*1) 3D perfect Euler briks:*

- $a^2 + b^2 = p^2$
- $a^2 + c^2 = q^2$
- $b^2 + c^2 = r^2$
- $a^2 + b^2 + c^2 = s^2$

*2) 4D Euler briks:*

- $a^2 + b^2 = p^2$
- $a^2 + c^2 = q^2$
- $b^2 + c^2 = r^2$
- $a^2 + d^2 = s^2$
- $b^2 + d^2 = t^2$
- $c^2 + d^2 = u^2$

*3) 4D perfect Euler bricks:*

- $a^2 + b^2 = p^2$
- $a^2 + c^2 = q^2$
- $b^2 + c^2 = r^2$
- $a^2 + d^2 = s^2$
- $b^2 + d^2 = t^2$
- $c^2 + d^2 = u^2$
- $a^2 + b^2 + c^2 + d^2 = v^2$

*4) Brocard's problem:* $n! + 1 = m^2, n > 7$, This problem is presented on 16, 32 bits format.

*5) H31 - The smallest (in H) open equation [2]:* $y(x^3 - y) = z^3 + 3$, This problem is presented on $80$ and $128$, $256$ bits format.

*6) Sum of three cubes:* $x^3 + y^3 + z^3 == k, k \in \{3, 33, 42, 165, 906, 114, 390, 579, 627, 633, 732, 921, 975\}$, This problem is for known solutions for $3, 33, 42$, to compare with large scale computation needed to solve, and search for unknown solutions over actual solutions. For un solved values is presented on $3 * 80$ and $3 * 128$ bits format.

## References

[1] Balyo , T , Froleyks , N , Heule , M , Iser , M , Järvisalo , M Suda , M (eds) 2021 , Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions . Department of Computer Science Report Series B , vol. B-2021-1 , Department of Computer Science, University of Helsinki , Helsinki .

[2] Grechuk, B. (2021). Diophantine equations: a systematic approach.

# Two Types of N-bits Inputs Multiplier Circuits Are Transformed to CNF

1st Shunyang Bi, 2nd Hailong You
*School of Microelectronics*
*XiDian University*
Xi'an China
shybi@stu.xidian.edu.cn,
hlyou@mail.xidian.edu.cn

***Abstract*—this description introduces our instances for the SAT Competition 2022. We generated instances that would select some circuits which are consisted of the n-bits inputs Carry-Save multiplier.**

## I. DATA

In the design of the n-bits multiplier, two multiplier types are widely applied in ordinary circuits: the Carry-Save multiplier and the Wallace-tree multiplier. For these two basic types of multipliers, the circuit first forms all the products $a_i * b_j$ where $a_i$ is a digit from the first number factor and $b_j$ is a digit from the second factor. The products are then added, but each multiplier circuit differs in the details of how the sum is done.

In the carry-save circuit, row i of the circuit adds the product from row i ($a_i * b_*$) with the sum and carry (shifted one column) to obtain a new sum and a new carry. A special adder is used to add the final sum and final carry. The products from the first row must go through a linear number of adders to get to the special adder.

In the Wallace-tree circuit, the rows (with appropriate shifts) are added in groups of three to produce sums and carries. The sums and carries (with appropriate shifts) are again added in groups of three, and this is repeated until there is just one sum and one carry. Then a special adder is used to add the final sum and the final carry. The products from any row need to go through only a logarithmic number of adders before they get to the special adder.

## II. SELECTION

The fast adder is an adder with log time complexity[1]. Whether the input of a designed multiplier circuit can be transformed into the CNF format based on the fast algorithm is very important for our circuit verification. We select some hard n-bits multipliers in the testing of the circuits and transform them into CNF format. TABLE Ⅰ shows the running time of 20 instances in Minisat.

TABLE I. RESULTS WITH MINISAT FOR 20 INSTANCES SUBMITTED FOR SAT COMPETITION-2022.

| *Instance name* | *Minisat Time(s)* | *Status* |
|---|---|---|
| Carry_Save_Fast_1.cnf | 5000 | UNKNOWN |
| Carry_Save_Fast_2.cnf | 5000 | UNKNOWN |
| Carry_Save_Fast_3.cnf | 5000 | UNKNOWN |
| Carry_Save_Bits_3.cnf | 101.888 | SAT |
| Carry_Save_Bits_4.cnf | 690.465 | SAT |
| Carry_Save_Bits_5.cnf | 5000 | UNKNOWN |
| Carry_Save_Bits_8.cnf | 5000 | UNKNOWN |
| Carry_Save_Bits_12.cnf | 367.49 | SAT |
| Carry_Save_Bits_16.cnf | 5000 | UNKNOWN |
| Carry_Save_Bits_18.cnf | 457.12 | SAT |
| Carry_Save_Bits_19.cnf | 1123.15 | SAT |
| Carry_Save_Bits_23.cnf | 4379.81 | SAT |
| Wallace_ Save_Bits_1.cnf | 5000 | UNKNOWN |
| Wallace_ Save_Bits_2.cnf | 247.39 | SAT |
| Wallace_ Save_Bits_3.cnf | 5000 | UNKNOWN |
| Wallace_ Save_Bits_4.cnf | 4320.78 | SAT |
| Wallace_ Save_Bits_5.cnf | 1783.81 | SAT |
| Wallace_ Save_Bits_6.cnf | 870.72 | SAT |
| Wallace_ Save_Bits_7.cnf | 5000 | UNKNOWN |
| Wallace_ Save_Bits_8.cnf | 5000 | UNKNOWN |

## III. TOOLS

We used CNF Generator for Factoring Problems to assist in transforming the multiplier of the circuits and generating the CNF formulas.

## REFERENCES

[1] Aho and Ullman, "Foundations of Computer Science" 1992.

# Time-indexed encoding of Multi-mode RCPSP

1st Jordi Coll
*Aix Marseille Univ, Université de Toulon,*
CNRS, LIS, Marseille, France
jordi.coll@lis-lab.fr

2nd Shuolin Li
*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
shuolin.li@etu.univ-amu.fr

3rd Chu-Min Li
*Université de Picardie Jules Verne*
Amiens, France
*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
chu-min.li@u-picardie.fr

4th Felip Manyà
*Artificial Intelligence Research Institute*
CSIC,Bellaterra, Spain
felip@iiia.csic.es

5th Djamal Habet
*Aix Marseille Univ, Université de Toulon*
CNRS, LIS, Marseille, France
Djamal.Habet@univ-amu.fr

***Abstract*—The proposed benchmarks are CNF encodings of the The Multi-mode Resource-Constrained Project Scheduling Problem (MRCPSP), obtained with Savile Row from a time-indexed model written in the Essence Prime constraint modelling language.**

## I. Problem Description

The Multi-mode Resource-Constrained Project Scheduling Problem (MRCPSP) [1] is an iconic scheduling problem with many industrial applications that has drawn a high research interest in the last decades, both for exact solving and approximate solving. The problem consists in finding a schedule with minimum makespan (total duration) for a project involving a finite set of activities $A$, and a finite set of renewable resources $R$ and non-renewable resources $N$ with limited capacity. Each activity has a set $M$ of execution modes available, and the duration and resource requirements of each activity depend on the selected mode. A solution must determine, for each activity $i \in A$, an integer start time $S_i$ and an execution mode $O_i$. Four main constraints appear in this problem:

- *Single mode selection*: exactly one execution mode must be selected for each activity.
- *Precedence relations*: there are predefined end-start precedence relations between pairs of activities $(i, j)$, meaning that activity $j$ cannot start until activity $i$ has ended.
- *Renewable resource constraints*: at any time instant, the sum of the demands on a renewable resource by activities that are running cannot surpass the capacity of that resource.
- *Non-renewable resource constraints*: the sum of demands on a non-renewable resource cannot surpass the capacity of that resource. The difference w.r.t. renewable resources is that the non-renewable are consumed, while the renewable are only occupied during the execution of the activities, and the resources recover the capacity used by an activity when this activity finishes.

The proposed benchmarks encode the decision version of MRCPSP, where the goal is to find a schedule whose makespan is not bigger than a given upper bound. The decision version of MRCPSP is NP-complete.

## II. Model Description

The CNF benchmarks are a subset of the CNF encodings of the decision version of the MRCPSP that were used in [2]. These formulas have been generated from the constraint model written in Essence Prime language from [3], which are encoded into CNF using the Savile Row reformulation tool [4]. The Essence Prime constraint model contains four main kinds of variables with finite domain: integer variable $jobStart_i$ specifies the start time of activity $i$; integer variable $mode_i$ specifies the selected execution mode for activity $i$; integer variable $duration_i$ specifies the duration of activity $i$ according to the selected execution mode; Boolean variable $jobActive_{i,o,t}$ is true iff activity $i$ is running at time $t$ (i.e. it has started but not finished), and in mode $o$. With these variables the single mode selection requirement is trivially satisfied, and the precedence relations between pairs of activities $(i, j)$ are modelled with expressions of the form $jobStart_i + duration_i \leq jobStart_j$. The dominating part of the model are the resource constraints, which are expressed as pseudo-Boolean (PB) constraints. In particular, renewable resource constraints are expressed as:

$$\sum_{i \in A, o \in M} usage_{i,o,r} \cdot jobActive_{i,o,t} \leq capacity_r$$

for all time instants $t$ up to a large enough horizon, and for every renewable resource $r \in R$. Similarly, non-renewable resource constraints are expressed as:

$$\sum_{i \in A, o \in M} usage_{i,o,r} \cdot (mode_i = o) \leq capacity_r$$

for every non-renewable resource $r \in N$. This kind of formulation is usually referred to as time-indexed formulation, since variables stating whether an activity is running are introduced for each time instant to express renewable resource constraints. Time-indexed formulations have shown to be state-of-the-art for MRCPSP exact solving [5].

## III. CNF Description

In the CNF formulas generated by Savile Row, the integer variables have been translated to Boolean variables by means of direct and order encodings, and a number of clauses is introduced to enforce the consistency between these encodings. The single mode selection constraint has been enforced by the direct encoding of $mode_i$. The precedence relations $jobStart_i + duration_i \leq jobStart_j$ have been translated into ternary clauses over the Boolean variables from the order encodings of $jobStart_i$,$duration_i$ and $jobStart_j$. Most of the clauses of the CNF formulas come from the CNF encodings of PB resource constraints, which also introduce a number of auxiliary variables. The provided benchmark set considers three different techniques from [2], [6] to encode PB constraints into SAT: GMTO, MDD and RGGT.

We select 5 of the hardest MRCPSP instances of the j30 benchmark set [7]. These instances contain 30 activities, 3 execution modes per activity, 2 renewable resources and 2 non-renewable resources. The upper bounds of the makespan are bigger than 40, hence the CNF formulase contain encodings of more than $2 * 40 = 80$ PB renewable resource constraints. Each PB constraint involves a maximum of $3 * 30 = 90$ $jobActive$ variables. Each MRCPSP instance is encoded a total of 6 times, thus having a total of 30 CNF formulas. The 6 encodings are the combinations of choosing one of the 3 PB encoding techniques, and 2 different upper bounds of the makespan: the best known value (always satisfiable), and the best minus 1 (unsatisfiable for the certified instances).

## References


[1] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch, "Resource-constrained project scheduling: Notation, classification, models, and methods," *European journal of operational research*, vol. 112, no. 1, pp. 3–41, 1999.

[2] M. Bofill, J. Coll, P. Nightingale, J. Suy, F. Ulrich-Oltean, and M. Villaret, "SAT encodings for pseudo-boolean constraints together with at-most-one constraints," *Artificial Intelligence*, vol. 302, p. 103604, 2022.

[3] C. Ansótegui, M. Bofill, J. Coll, N. Dang, J. L. Esteban, I. Miguel, P. Nightingale, A. Z. Salamon, J. Suy, and M. Villaret, "Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-boolean constraints," in *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11802. Springer, 2019, pp. 20–36.

[4] P. Nightingale, Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen, "Automatically improving constraint models in savile row," *Artificial Intelligence*, vol. 251, pp. 35–61, 2017.

[5] M. Bofill, J. Coll, J. Suy, and M. Villaret, "SMT encodings for resource-constrained project scheduling problems," *Computers & Industrial Engineering*, vol. 149, p. 106777, 2020.

[6] ——, "An mdd-based SAT encoding for pseudo-boolean constraints with at-most-one relations," *Artificial Intelligence Review*, vol. 53, no. 7, pp. 5157–5188, 2020.

[7] R. Kolisch and A. Sprecher, "Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program," *European journal of operational research*, vol. 96, no. 1, pp. 205–216, 1997.

# Circuit Model Checking with BMC

Xindi Zhang, Zhihan Chen, Shaowei Cai

[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{zhangxd,caisw,chenzh}@ios.ac.cn

## I. Introduction

With the increasing scale of integrated circuits, it becomes more and more difficult to verify the correctness of hardware in the application. We notice that there are gaps between real-world applications and SAT Competition instances in that the mathematical problems account for the vast majority of the latter one. Thus, we submit 5 instances that are generated by Bounded Model Checking (BMC) aiming at checking the safety properties of hardware.

## II. BMC Encoding for Circuits

Hardware Model Checking (HMC) is an automatic technique to decide whether a given circuit satisfies a given specification [3]. Current HMC is usually based on the Kripke structure which is a variation of the transition system and formalized as a finite-state transition system $M =< V, I, T, P >$, where $V, I, T, P$ stand for the states, initial states, transition relation between states, and the property each state holds. The state machine $M$ is usually encoded into the CNF, which can be handled by SAT solvers. Each state of $M$ can be formalized as some propositional variables. The inital state $I(V)$, the translation relationship $T(V_1, V_2)$ between two states $V_1$ and $V_2$ and the property $P(V)$ for a state $V$ can be encoded into some propositional clauses. The combinational circuit gates of the given hardware design can be encoded into SAT according to Tseitin's transformation [5]. For example an AND gate $O = AND(I_1, I_2)$ can be encoded into $(\neg I_1 \vee \neg I_2 \vee C) \wedge (I_1 \vee \neg O) \wedge (I_2 \vee \neg O)$, and a NOT gate $O = NOT(I)$ can be encoded into $(\neg O \vee \neg I) \wedge (I \vee O)$. The states in $M$ are variable assignments according to certain time stamps, and the latches or memory gates form the translation relations between adjacent time stamps.

BMC is a popular symbolic model checking algorithm for checking whether a property $P$ can be violated in $k$ steps [1]. A common practice for BMC solvers is to include an incremental SAT solver and iteratively use them as their core engine. The success of BMC lies in its ability to find counter-example. In each call, BMC tries to search for a counter-example within a given bounded limitation $k_{max}$. If a counter-example is found with step $k$, BMC returns the counter-example path. Else, the algorithm increases the step $k$ by 1 until reaches the given maximum bound $k_{max}$.

TABLE I
Selected instances submitted to SAT Competition 2022

| Index | Name |
|---|---|
| 1 | bmc_QICE_req_sfl_30.cnf |
| 2 | bmc_QICE_req_vld_30.cnf |
| 3 | bmc_QICE_rxrsp_vld_30.cnf |
| 4 | bmc_QICE_snp_vld_30.cnf |
| 5 | bmc_QICE_snp_vld_50.cnf |

The formula for checking for counter-examples of length $k$ can be built as follows.

$$F^k \equiv I(V^0) \wedge Trans(V^0, V^k) \wedge (\neg P(V^k)) \quad (1)$$

We use $V^t$ to stand for the state of timestamp $t$, and use

$$Trans(V^i, V^j) = T(V^i, V^{i+1}) \wedge ... \wedge T(V^{j-1}, V^j) \quad (2)$$

where $0 \leq i < j$, to represent the translation path with length $j - i + 1$.

### A. Benchmark Selection

We select 5 benchmarks from real hardware applications according to different bounds to check certain safety properties, which are shown in I. The instances are 'interesting', which are not too easy, and can be solved by our improved versions of the participants of SC21 [2], [4].

## References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In International conference on tools and algorithms for the construction and analysis of systems, pages 193–207. Springer, 1999.
[2] S. Cai and X. Zhang. Deep cooperation of cdcl and local search for sat. In International Conference on Theory and Applications of Satisfiability Testing, pages 64–81. Springer, 2021.
[3] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, et al. Handbook of model checking, volume 10. Springer, 2018.
[4] A. B. K. F. M. Fleury and M. Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. SAT COMPETITION 2020, page 50, 2020.
[5] G. S. Tseitin. On the complexity of derivation in propositional calculus. In Automation of reasoning, pages 466–483. Springer, 1983.

# Factory Worker Dispatching problem

Xindi Zhang, Zhihan Chen, Shaowei Cai

[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{zhangxd,caisw,chenzh}@ios.ac.cn

Abstract—In this document, we describe a real-world problem that many factories dispatch workers to cooperate on a joint task, and introduce a modeling scheme. We encode some randomly generated problems into Conjunctive Normal Form (CNF) and submit 15 instances to SAT Competition 2022 [1].

## I. Introduction

In reality, it usually happens that different companies or teams need to arrange for workers to cooperate on the same task, which is referred to as Worker Dispatching Problem (WDP) in this document. WDP is a decision problem that decides whether there is a feasible dispatching solution under given constraints, and the solution should be given when the answer is 'YES'. The WDP can be seen as a special case of the Timetable Scheduling Problem [1] or Job Scheduling Problem [2]. The WDP can be encoded into the SAT problem naturally, in this document, we introduce the modeling and encoding method for WDP.

## II. The Worker Dispatching Problem

Let us consider a scenario where $n$ factories $\{F_1, ..., F_n\}$ cooperate on a project. For factory $F_i$, there are $k_i$ workers that can be dispatched for this joint task, denoted by $W_i = \{W_i^1, ..., W_i^{k_i}\}$. We use $W = \bigcup_{1\leq i\leq n}\bigcup_{1\leq j\leq k_i} W_i^j$ to represent the set of all workers that can be dispatched in the $n$ factories. Due to the limitation of human resources, each factory $F_i$ can dispatch at most $M_i$ workers for this cooperation. In this document, we only consider a special case that $M_i = 1$. The joint project contains $m$ jobs, denoted as $J = \{j_1, ..., j_m\}$. According to the personal skills, each worker $w \in W$ is capable of a subset of jobs $J^w \subseteq J$, but can only be assigned to at most one job in $J^w$. Each job can be done if at least one worker is assigned to it.

The WDP problem asks whether there is a feasible dispatching solution.

## III. An Encoding Method

In this section, we assume readers are familiar with CNF. We use cardinality constraints at-most-one (AMO) in convenience for the expression, and the $AMO(x_1, x_2, ..., x_q)$ constraints are encoded pairwise as $\bigwedge_{1\leq i<j\leq q}(\neg x_i \vee \neg x_j)$. The variable $A(w, j)$ represents assigning worker $w$ to job $j$.

There are $|W| \cdot |m|$ variables for this encoding method.

- Every job needs to be handled by at least one worker.

$$\bigwedge_{1\leq i\leq m} \Big( \bigvee_{w\in W} A(w, J_i)\Big) \tag{1}$$

- Each factory can dispatch at most one worker, and each worker can be assigned at most one job.

$$\bigwedge_{1\leq i\leq n} AMO(\bigcup_{w\in W_i} AMO(\bigcup_{j\in J^w} A(w, j))) \tag{2}$$

Note that, the encoding method is simplified by neglecting variables $A(w, j)$ that represent impossible arrangements.

[1]The generator can be found in GitHub https://github.com/iHaN-o/Worker-instance-generator.

TABLE I
Selected instances submitted to SAT Competition 2022

| Name | #F | #W | #J | PR |
|---|---|---|---|---|
| worker_20_40_20_0.95.cnf | 20 | 40 | 20 | 0.95 |
| worker_20_60_20_0.9.cnf | 20 | 60 | 20 | 0.9 |
| worker_20_80_20_0.85.cnf | 20 | 80 | 20 | 0.85 |
| worker_30_60_25_0.9.cnf | 30 | 60 | 25 | 0.9 |
| worker_30_90_30_0.8.cnf | 30 | 90 | 30 | 0.8 |
| worker_30_120_30_0.8.cnf | 30 | 120 | 30 | 0.8 |
| worker_40_40_30_0.9.cnf | 40 | 40 | 30 | 0.9 |
| worker_40_80_35_0.85.cnf | 40 | 80 | 35 | 0.85 |
| worker_40_80_40_0.85.cnf | 40 | 80 | 40 | 0.85 |
| worker_50_100_40_0.95.cnf | 50 | 100 | 40 | 0.95 |
| worker_50_150_40_0.85.cnf | 50 | 150 | 40 | 0.85 |
| worker_50_50_30_0.8.cnf | 50 | 50 | 30 | 0.8 |
| worker_80_80_80_0.8.cnf | 80 | 80 | 80 | 0.8 |
| worker_550_550_550_0.3.cnf | 550 | 550 | 550 | 0.3 |
| worker_600_600_600_0.27.cnf | 600 | 600 | 600 | 0.27 |

## IV. Benchmark selection

We submit 15 'interesting' benchmarks for SAT Competition 2022, which cannot be solved by minisat within 1 minute, consisting of 13 unsatisfiable instances and 2 satisfiable instances. The 15 instances are described in Table I. The #F, #W and #J represent the number of factories, workers and jobs for a WDP instance, respectively. PR presents the probability of a worker can do a job.

## References


[1] H. Alghamdi, T. Alsubait, H. Alhakami, and A. Baz. A review of optimization algorithms for university timetable scheduling. Engineering, Technology & Applied Science Research, 10(6):6410–6417, 2020.

[2] A. Arisha, P. Young, and M. El Baradie. Job shop scheduling problem: an overview. In International Conference for Flexible Automation and Intelligent Manufacturing (FAIM 01), pages 682–693, 2001.

# Equivalence Checking of EPFL Benchmarks

Xinyan Chen[‡], Wenxuan Guo[‡], Wanqian Luo[†], Hui-Ling Zhen[†],
Xijun Li[†], Mingxuan Yuan[†] and Junchi Yan[‡]
[†]Huawei Noah's Ark Lab
[‡]Shanghai Jiao Tong University
{luowanqian1, zhenhuiling2, xijun.li, Yuan.Mingxuan}@huawei.com
{moss_chen, arya_g, yanjunchi}@sjtu.edu.cn

***Abstract*—To participate in SAT Competition 2022, we present an approach to generate CNFs via SAT encoding for circuit equivalence checking instances of EPFL Benchmarks.**

## I. INTRODUCTION

Circuit equivalence checking is to check whether two given circuits are equivalent in terms of function. Assume we have two circuits, i.e., golden circuit and implementation design circuit, needs to be checked. Note that the implementation design circuit is obtained by optimizing the golden circuit and the number of input gates of the two circuits is equal. A miter circuit is firstly constructed by connecting the output gates of above two circuits to a group of exclusive-OR (XOR) gates, as shown in Fig. 1. If the output of one of these XOR gates can be assigned to true, a certificate is found which shows that the two circuits are not equivalent. Otherwise, the circuits are equivalent, which means that the implementation circuit is valid optimization of golden circuit.

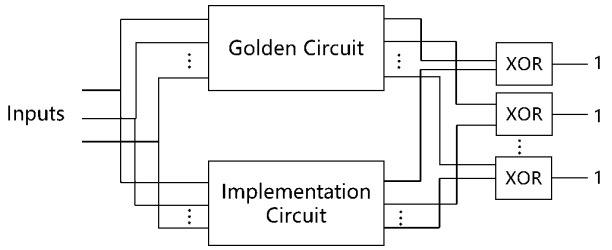



Fig. 1. Construction of miter circuits. A miter circuit is constructed by connecting the output gates of golden circuit and of implementation circuit to a group of exclusive-OR (XOR) gates.

The problem whether the output of miter circuit can be assigned to true can be equally converted into a Boolean Satisfiability (SAT) problem. Specifically, the Tseytin Encoding [1] is utilized to converted the miter circuit into corresponding CNF. And an unit clause of XOR is appended to the CNF. Similarly, the equivalence checking of multi-output circuits can be converted to multiple SAT problems.

## II. GENERATION PROCEDURE

The circuit data come from the public dataset of EPFL benchmarks [2] of which all circuits are combinatorial circuits. EPFL benchmark includes three sub benchmarks, i.e., Arithmetic benchmark, Random/Control benchmark and More than ten Miliong gates (MtM) benchmark. Here we adopt the Arithmetic benchmark in our generation procedure. In details, the golden circuits are provided by EPFL as baseline data and implementation circuits are selected from best LUT-6 results of historical submissions.

We firstly use *'miter'* command of ABC [3] to extract miter circuits. Then the golden circuits and implementation circuits involved in above miter circuits are extracted in the manner of depth first search (DFS). Finally, we use *'write_cnf'* command line to convert the above circuits to CNFs.

## III. BENCHMARKS

We submitted 20 benchmarks generated by the mentioned-above procedure. All benchmarks meet the requirement of benchmark submissions. In detail, we tested MiniSat Solver over the submitted benchmarks. The average solving time is around 578.55 seconds. All benchmarks are unsatisifiable. All tests were performed on Intel(R) Xeon(R) Platinum 8180M CPU@2.50GHz.

## REFERENCES

[1] https://en.wikipedia.org/wiki/Tseytin_transformation
[2] Amarù, Luca Gaetano et al. "The EPFL Combinational Benchmark Suite." (2015).
[3] R. K. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in CAV, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 24–40.

# The SAT Encoding for Graph Isomorphism

Yang Li[‡], Yuqi Jia[‡], Wanqian Luo[†], Hui-ling Zhen[†],
Xijun Li[†], Mingxuan Yuan[†] and Junchi Yan[‡]
[†]Huawei Noah's Ark Lab
[‡]Shanghai Jiao Tong University
{luowanqian1, zhenhuiling2, xijun.li, Yuan.Mingxuan}@huawei.com
{yanglily, jiayuqi001023, yanjunchi}@sjtu.edu.cn

***Abstract*—To participate in SAT Competition 2022, we present an approach to generate CNF via SAT encoding for graph isomorphism problem. The generated CNF consists of three classes, including SAT encoding based on basic graph isomorphism, SAT encoding based on transformed graph isomorphism and the ones obtained by random shuffling the first two classes.**

## I. INTRODUCTION

Graph Isomorphism (GI) problem is to check whether two given graphs are equivalent in terms of structure, which is one of the most studied topics in the field of complexity theory. There are a series of efficient algorithms to solve GI problem, such as nauty, traces [1], bliss [2], conauto [3], saucy [6], etc. Besides, the GI problem can be transformed into Boolean Satisfiability problem [6] and solved by calling SAT solvers. In this benchmark, we generate a group of hard SAT instances by transforming some of the public complex GI problems [6] into SAT encodings. The details of the generation procedure are presented in the following.

## II. GENERATION PROCEDURE

The raw data of graph isomorphism can be obtained from the public benchmark of [1]. The public GI benchmark mainly includes six groups of data which are of 'dimacs' format. The six groups of GI data fall into two categories, i.e., isomorphic and non-isomorphic. Among the six groups of data, cfi-rigid-r2 and cfi-rigid-t2 are isomorphic. The rest of them are non-isomorphic. All problems come mostly from research in Proof Complexity, such as pigeonhole principle, ordering principle and k-clique. Most problems are structured. Our CNFs are obtained by transforming from the above GI problems.

In order to raise the computing complexity of the generated CNFs, some transformations are applied over these generated CNFs. Firstly, we apply OR transformation over the CNFs obtained from CNFgen. In detail, we transform a CNF by substituting each variable with the original disjunction of three new variables. Considering the following CNF as an example.

$$(\neg X \vee Y) \wedge (\neg Z) \tag{1}$$

We substitute $X$ with $X_1, X_2$ and $X_3$. The similar substitution is also applied on $Y$ and $Z$. In this way, a new CNF can be obtained as follows:

$$\begin{aligned}&(\neg X_1 \vee Y_1 \vee Y_2 \vee Y_3)\wedge\\ &(\neg X_2 \vee Y_1 \vee Y_2 \vee Y_3)\wedge\\ &(\neg X_3 \vee Y_1 \vee Y_2 \vee Y_3)\wedge\\ &(\neg Z_1) \wedge (\neg Z_2) \wedge (\neg Z_3)\end{aligned} \tag{2}$$

Note that the number of variable substitutions in our transformation is set as two and three.

Then, on top of the above CNF, a random shuffling operation is applied to further increase the complexity of generated CNFs, which includes variable permutation, clause permutation and polarity flip. In detail, the variable permutation is to change the occurring order of original variables within each clause; and the clause permutation is to change the occurring order of clause within the CNF but the order of variables within each clause remains the same. The polarity flip is to randomly flip the polarity of literals of CNF.

## III. BENCHMARKS

We submitted 20 benchmarks generated by the mentioned-above procedure. All benchmarks meet the requirement of benchmark submissions. In detail, we tested MiniSat Solver over the submitted benchmarks. The average solving time is around 566 seconds. Among these benchmarks, eight of them are satisfiable and 12 of them are unsatisfiable. All tests were performed on Intel(R) Xeon(R) Platinum 8180M CPU@2.50GHz.

## REFERENCES

[1] B. D. McKay and A. Piperno. Practical graph isomorphism, II. J. Symb. Comput., 60:94–112, 2014.
[2] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, pages 135–149. SIAM, 2007.
[3] J. L. López-Presa, A. F. Anta, and L. N. Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. CoRR, abs/1108.1060, 2011.
[4] P. Codenotti, H. Katebi, K. A. Sakallah, and I. L. Markov. Conflict analysis and branching heuristics in the search for graph automorphisms. In 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013, pages 907–914.
[5] Neuen D, Schweitzer P. Benchmark graphs for practical graph isomorphism[J]. arXiv preprint arXiv:1705.03686, 2017.
[6] Torán J. On the resolution complexity of graph non-isomorphism[C]//International Conference on Theory and Applications of Satisfiability Testing. Springer, Berlin, Heidelberg, 2013: 52-66.

# Set Covering with Conflict Benchmarks

Jiongzhi Zheng[1] Kun He[1] Zhuo Chen[1] Jianrong Zhou[1] Chu-Min Li[2]
[1]School of Computer Science and Technology,
Huazhong University of Science and Technology, China
[2]MIS, Université de Picardie Jules Verne, France

## I. Introduction

We propose a new variant of the well-known set covering problem [1], called set covering problem with conflict (SCPC), and generate 20 SAT instances by transforming 20 SCPC instances.

## II. The Set Covering Problem with Conflict

Given a set of items $S = \{s_1, ..., s_m\}$ and a set of elements $E = \{e_1, ..., e_n\}$, where each item covers a subset of $E$ and each element $e_j$ ($j \in \{1, ..., n\}$ has a positive weight $w_j$, a conflict graph $G = (V, E')$ where the node set $V$ consists of all the items in $S$, an edge $(s_i, s_j)$ belonging to $E'$ indicates that items $s_i$ and $s_j$ are conflict with each other. The goal of SCPC is to find a subset $S' \subset S$ that any two items in $S'$ are not conflict, and the total weight of the covered elements is maximized.

## III. Transforming SCPC into SAT

Firstly, a Weighted Partial MaxSAT [2] instance can be generated by a SCPC instance as follows. We use each item $s_i$ to represent a variable $v_i$, each element $e_j$ that covered by $k$ items $\{s_{j1}, ..., s_{jk}\}$ to represent a soft clause with weight $w_j$ that consists of the positive literal of variables $\{v_{j1}, ..., v_{jk}\}$. We further use each pair of conflict items $s_a$ and $s_b$ to generate a hard clause, which consists of the negative literal of $v_a$ and $v_b$. Once a SCPC instance $I$ is generated, we first use a local search method to obtain a feasible solution $S'$ ($S' \subset S$), then we traverse each uncovered element three times to make it covered by a random item in $S'$ with a probability of 0.8 and obtain a new instance $I'$. Finally, we transform $I'$ into a MaxSAT instance $F$ by the above method, and regard all the soft clauses in $F$ as hard to obtain a SAT instance.

## IV. Benchmarks

To generate each SCPC instance, we set the density of the conflict graph to 0.1, the number of items to 500, and the number of elements to 3000, each element is covered by 20 items on average. We generate 20 random SCPC instances with the above settings, and transform them into 20 SAT instances by the aforementioned method.

All the instances are UNSAT. Each of the 20 instances can be solved by Kissat-MAB, the champion of the main track of SAT Competition 2021, within 600 seconds on a server using an Intel® Xeon® E5-2650 v3 2.30 GHz 10-core CPU, running Ubuntu 16.04 Linux operation system.

## References

[1] E. Balas, M. W. Padberg, "On the set-covering problem," Oper. Res., vol. 20(6), pp. 1152-1161, 1972.
[2] J. Zheng, K. He, J. Zhou, Y. Jin, C. M. Li, F. Manyà, "BandMaxSAT: A Local Search MaxSAT Solver with Multi-armed Bandit," IJCAI 2022.

# Sudoku Clue Generation Problem Instances

Zhenjiang Zhao*, Takahisa Toda*, Takashi Kitamura†
*Graduate School of Informatics and Engineering, University of Electro-Communications, Tokyo, Japan
{zhenjiang, toda}@disc.lab.uec.ac.jp
†National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan
t.kitamura@aist.go.jp

***Abstract*—We describe CNF formulas that are encoded from Sudoku clue generation problem instances.**

## I. Introduction

A Sudoku puzzle often has a regular pattern in the arrangement of initial digits, as depicted in the center grid of Fig. 1, and it is typically made solvable with known solving techniques called *strategies*. The *sudoku clue generation* [1] is the problem of determining digits (called *clues*) in given cells so that the grid can be completed by applying only prescribed strategies.

In Fig. 1, the Sudoku clue generation instance is specified by the gray cells, and suppose that only *naked single*, one of the most basic strategies, is allowed to be used. Here, naked single means that one can safely place $n$ in cell $(i, j)$ if no other candidate but $n$ remains at $(i, j)$. For instance, take a look at cell $(4, 2)$ in the center grid. We can observe that $8, 9, 2$ are present in the same row, $6, 7, 3$ is in the same column, and $1, 4$ is in the same block, all of which implies that all that remains is $5$.

In this setup, the output (solution) of the Sudoku clue generation is given in the center grid. Indeed, all empty cells in the grid can be filled by applying naked single only and the right grid will be finally obtained. Note that the solution of the current Sudoku clue generation is not the right grid but the center grid; the right grid is the solution when considering the center grid as "Sudoku instance".

## II. Benchmark Instances

The submitted benchmark instances consists of the 30 Sudoku clue generation instances listed in Table I. These instances are encoded by sgc_modeler [2]. As shown in the second column, grids with clue positions marked are represented as strings of zeros and asterisks; zeros denote empty cells and asterisks denote positions in which clues should be placed. In these instances, only naked singe is allowed to be used. These grids are obtained from some minimum Sudokus in the collection of Gordon F. Royle.

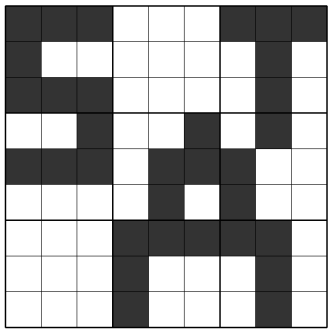
All the instances are confirmed to be unsatisfiable. The times required are shown in the third column of Table I. The computational environment is as follows. OS: Ubuntu 20.04.4 LTSCPU: Intel® Xeon® E5-2609 (1.70GHz), Main Memory: 32GB. SAT Solver: CaDiCaL 1.5.0 [3].

## References

[1] K. Nishikawa and T. Toda, "Exact method for generating strategy-solvable sudoku clues," *Algorithms*, vol. 13, no. 7, 2020. [Online]. Available: https://www.mdpi.com/1999-4893/13/7/171

[2] T. Toda, "Scg modeler," https://github.com/toda-lab/scg, accessed: 2012-04-29.

[3] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 5 | | | | 8 | 7 | 1 |
| 8 | | | | | | | 4 | |
| 4 | 7 | 9 | | | | | 3 | |
| | | 8 | | | 9 | | 2 | |
| 1 | 3 | 4 | | 7 | 2 | 5 | | |
| | | | | 6 | | 3 | | |
| | | | 6 | 3 | 7 | 1 | 9 | |
| | | | 4 | | | | 5 | |
| | | | 1 | | | | 8 | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 5 | 9 | 4 | 3 | 8 | 7 | 1 |
| 8 | 1 | 3 | 7 | 5 | 6 | 9 | 4 | 2 |
| 4 | 7 | 9 | 2 | 8 | 1 | 6 | 3 | 5 |
| 6 | 5 | 8 | 3 | 1 | 9 | 4 | 2 | 7 |
| 1 | 3 | 4 | 8 | 7 | 2 | 5 | 6 | 9 |
| 9 | 2 | 7 | 5 | 6 | 4 | 3 | 1 | 8 |
| 5 | 8 | 2 | 6 | 3 | 7 | 1 | 9 | 4 |
| 3 | 9 | 1 | 4 | 2 | 8 | 7 | 5 | 6 |
| 7 | 4 | 9 | 1 | 9 | 5 | 2 | 8 | 3 |

Fig. 1. Clue positions (left), initial Sudoku grid (center), and the solution (right)

TABLE I
SUDOKU GRIDS AND THE RUNNING TIMES (IN SECONDS): EACH GRID IS REPRESENTED BY A STRING OF ZEROS AND ASTERISKS DELIMITED BY THE PERIOD, WITH EACH DELIMITED SUBSTRING INDICATING A ROW OF THE GRID.

| Index | Grid | Time (s) |
|---|---|---|
| 1 | 0*0*00000.0000*0*00.0000000*0.0*0000*00.000*0*000.0000**000.000*00*0*.*0*000000.*00000000 | 2,834 |
| 2 | 0*0*0*000.*000000*0.000000000.000*00*0*.*00*00*00.*00000000.0**000000.0000*00*0.0000*0000 | 422 |
| 3 | 000*00**0.*0*000000.000*00000.0*00*00*0.00000**00.*000*0000.0*0*00000.0000*0*00.000000000 | 15,536 |
| 4 | 00*0000*0.000**0000.000000000.*00000*0*.00**00000.000000*00.000*0*0*0.0*00*0000.**0000000 | 5,104 |
| 5 | 0**000000.000*0000*.000000*00.*000**000.0000*00*0.*00000000.000*00*00.0*00000*0.*00*00000 | 423 |
| 6 | 0000000**.000*0*000.0*0000000.*0*000*00.0000*0000.000000*00.0*0**0000.*00000**0.000*00000 | 428 |
| 7 | *00*00*00.0000000*0.000000000.000000***.0*00**000.000000000.*0*000*00.000**0000.*00*00000 | 43,148 |
| 8 | 0000*0*00.0*00000*0.0000*0000.*00*000*0.00000*000.00*000000.**0*00000.000*00*0*.000000*00 | 5,073 |
| 9 | 000**0*00.*00*00000.*00000000.*000**000.000000**0.000000000.0*0*0*000.0000000**.0*0000000 | 502 |
| 10 | 0*00*00*0.*00*00*00.000000000.*00*0*000.*000000**.000*00000.00*000*00.0*00*0000.000000000 | 17,839 |
| 11 | 000*00**0.*0*000000.000*00000.000000*0*.0*0*00000.000000*00.*000**000.0*00*00*0.000000000 | 5,103 |
| 12 | *0*00*0*0.000*00*00.*00000000.0*00*0*00.00000*000.000*00000.0*0**0000.0000000**.000000000 | 428 |
| 13 | 0000*000*.00**00000.*00000000.**000*000.000*00**0.*00000000.00*000**0.0*00*0000.000000000 | 5,287 |
| 14 | *00*00000.000000*0*.00*000*00.000*00**0.**0000000.000*00000.*000000*0.0000**000.0000*0000 | 824 |
| 15 | 0*00**000.0000*0*00.000000000.0*000000*.000*000*0.*00*00000.*0*000*00.*00*00000.0000*0000 | 493 |
| 16 | *000000**.0000**000.000000000.000**000*.*00000*00.0**000000.**0*00000.000000**0.000000000 | 4,670 |
| 17 | 0*0**0000.0*000000*.0000000*0.0*0*00*00.*000*0000.00000*000.*0*000000.000*00*00.*00000000 | 744 |
| 18 | *000*0*0*.0*0*00000.*00000000.**00000*0.00000*000.0000*0000.000*000*0.000*00*00.00*000000 | 5,468 |
| 19 | *000*0000.0*00000*0.000000000.000*00***0.*0**00000.00*000000.000*00*00.0*000*000.*0000000* | 10,824 |
| 20 | 00*000*00.*000*0000.000*000*0.0000**000.0*00000*0.000*00000.**0*00000.000000*0*.000000*00 | 4,859 |
| 21 | 0*0000*00.0000**000.0000*0000.000*00*00.*0*000000.*0000000*.000**0000.*000000*0.000*000*0 | 741 |
| 22 | 0*0000*00.000*000*0.*000*0000.0**0000*0.000*000*0.0000*0000.*00000*0*.00000*000.000000*00 | 495 |
| 23 | *000*00**.*0000000*.000*000*0.0*0*00000.0000*0*00.0*0000000.*00000*00.000**0000.000000000 | 17,018 |
| 24 | *0*0*0*00.000*00000.*00000000.0*00000**.0000**000.000000000.**0*00000.000000**0.00000000* | 491 |
| 25 | 0*0000*00.0000**000.0000*0*00.*000000*0.0*0*00000.0000*0000.000*00*00.*0*000000.*00*00000 | 5,300 |
| 26 | *000*0000.000*00*00.000000*00.0000**0*0.00*000000.0*0000000.**00000*0.00000***0.000*00000 | 326 |
| 27 | 0000***00.*000000*0.000000000.*00*00000.000000*0*.000000**0.0**000000.000**0000.0*0*00000 | 297 |
| 28 | *00*00000.0000*0*00.0000000*0.0**000*00.0*0000*00.000*0000*.*00*0000*.0000*00*0.000000000 | 304 |
| 29 | *00*00000.0000000**.00000000*.000*0**00.*0*000000.*00*00000.0*0000*00.0000*00*0.0*0000000 | 4,825 |
| 30 | 00*0000*0.*00*00000.000*000*0.*00000*00.0000*0000.000*00000.00000**0*.**0000000.0*0000*00 | 21,215 |

# Solver Index

# Benchmark Index

# Author Index