

Proceedings of
SAT COMPETITION 2020
Solver and Benchmark Descriptions

Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Jarvisalo, and Martin Suda
(editors)

UNIVERSITY OF HELSINKI
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS B
REPORT B-2020-1

HELSINKI 2020

PREFACE

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

SAT Competition 2020 (SC 2020; <https://satcompetition.github.io/2020/>), a competitive event for SAT solvers, was organized as a satellite event of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020). SC 2020 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held 2002-2005, biannually during 2007-2013, and 2014, 2016-2018; the SAT Races held in 2006, 2008, 2010, 2015, and 2019; and SAT Challenge 2012.

SC 2020 consisted of a total of four tracks: Main Track (with Glucose Hack, Planning and No Limits sub-tracks), Incremental Library Track, Parallel Track, and Cloud Track. The planning sub-track represents a first instantiation of a more domain-specific track in the SAT competitions, complementing the otherwise general tracks. The Cloud Track is also a new inclusion into the SAT competition series for 2020, focusing on evaluating SAT solvers specifically built for running on cloud computing environments.

There were two ways of contributing to SC 2020: 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. Following the tradition put forth by SAT Challenge 2012, the rules of SC 2020 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 2020 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 2020. 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 thank CAS Software Karlsruhe for sponsoring the awards.

Finally, we would like to emphasize that a competition does not exist without participants: we thank all those who contributed to SC 2020 by submitting either solvers or benchmarks and the related description.

*Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, & Martin Suda
SAT Competition 2020 Organizers*

Contents

Preface	3
-------------------	---

Solver Descriptions

Cadical-trail, Cadical-alluip, Cadical-alluip-trail, and Maple-LCM-Dist-alluip-trail at SAT Competition 2020 <i>Randy Hickey, Nick Feng, and Fahiem Bacchus</i>	10
Undominated-LC-MapleLCMDiscChronoBT-DL <i>Jerry Lonlac and Engelbert Mephu Nguifo</i>	11
MLCMDChronoBT-DL-Scavel and its friends at the SAT Competition 2020 <i>Zhihui Li, Guanfeng Wu, Yang Xu, and Qingshan Chen</i>	13
Relaxed Backtracking with Rephasing <i>Xindi Zhang and Shaowei Cai</i>	15
ExMapleLCMDistChronoBT, upGlucose-3.0_PADC and PaInleSS_ExMapleLCMDistChronoBT in the SC20 <i>Rodrigue Konan Tchinda and Clémentin Tayou Djamegni</i>	17
MapleLCMDistChronoBT-DL-v3, the duplicate learnts heuristic-aided solver at the SAT Competition 2020 <i>Stepan Kochemazov, Oleg Zaikin, Victor Kondratiev, and Alexander Semenov</i>	20
F2TRC: deterministic modifications of SC2018-SR2019 winners <i>Stepan Kochemazov</i>	21
Four CDCL solvers based on expLRB, expVSIDS and Glue Bumping <i>Md Solimul Chowdhury, Martin Müller, and Jia-Huai You</i>	23
Optsat, Abcdsat and Solvers Based on Simplified Data Structure and Hybrid Solving Strategies <i>Jingchao Chen</i>	25
CryptoMiniSat with CCAnr at the SAT Competition 2020 <i>Mate Soos, Shaowei Cai, Jo Devriendt, Stephan Gocht, Arijit Shaw, and Kuldeep S. Meel</i>	27
CryptoMiniSat with WalkSAT at the SAT Competition 2020 <i>Mate Soos, Bart Selman, and Henry Kautz</i>	29
Glucose 3.0 — With Inprocessing <i>Norbert Manthey</i>	31
CTSat: C++ Template Abstractions for Dynamic Interchangeable SAT Solver Features <i>Marc Hartung</i>	32

PauSat	
<i>Benjamin Kaiser and Marc Hartung</i>	34
MapleCOMSPS_LRB_VSIDS for SAT Competition 2020	
<i>Jia Hui Liang, Chanseok Oh, Saeed Nejati, Pascal Poupart, and Vijay Ganesh</i>	36
MaplePainless-DC Parallel SAT Solver for SAT Competition 2020	
<i>Saeed Nejati and Vijay Ganesh</i>	38
MergeSAT	
<i>Norbert Manthey</i>	40
ParaFROST, ParaFROST_CBT, ParaFROST_HRE, ParaFROST_ALL at the SAT Race 2020	
<i>Muhammad Osama and Anton Wijs</i>	42
Riss 7	
<i>Norbert Manthey</i>	44
Engineering HordeSat Towards Malleability: mallob-mono in the SAT 2020 Cloud Track	
<i>Dominik Schreiber</i>	45
DurianSat at SAT Competition 2020	
<i>Arijit Shaw and Kuldeep S. Meel</i>	47
Maple_CM+dist, Maple_CM+dist+sattime, Maple_CM+dist+simp2- and Maple_CMused+dist in the SAT Competition 2020	
<i>Chu-Min Li, Mao Luo, Fan Xiao, Li Yu, Felip Manyà, and Zhipeng Lü</i>	48
CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020	
<i>Armin Biere, Katalin Fazekas, Mathias Fleury, and Max Heisinger</i>	50
GlucoseEsbpSel: accelerate the search while pruning	
<i>Hakan Metin, Vincent Vallade, Souheib Baarir, and Jo Devriendt</i>	54
P-MCOMSPS-STR: a Painless-based Portfolio of MapleCOMSPS with Clause Strengthening	
<i>Vincent Vallade, Ludovic Le Frious, Souheib Baarir, Julien Sopena, and Fabrice Kordon</i>	56
ManyGlucose 4.1-60	
<i>Hidetomo Nabeshima and Katsumi Inoue</i>	58
SLIME SAT Solver	
<i>Oscar Riveros</i>	59
TopoSAT2	
<i>Thorsten Ehlers, Mitja Kulczynski, Dirk Nowotka, and Philipp Sieweck</i>	60
 Benchmark Descriptions	
Main Track Benchmarks	
<i>Markus Iser</i>	62
Incremental Library Track Benchmarks	
<i>Markus Iser</i>	63
Planning Track Benchmarks	
<i>Nils Froleyks</i>	64
SAT Encodings for Discrete Logarithm Problem	
<i>Jingchao Chen</i>	65

Improving Directed Ramsey Numbers using SAT <i>David Neiman, Marijn Heule, and John Mackey</i>	66
Unit-Distance Strip Coloring Benchmarks <i>Peter Oostema, Ruben Martins, and Marijn Heule</i>	68
Edge-matching Puzzles as a Feature-Cardinality Problem <i>Dieter von Holten</i>	70
Population Safety: A SAT Benchmark based on Elementary Cellular Automaton <i>Md Solimul Chowdhury, Martin Müller, and Jia-Huai You</i>	75
Finding exact solutions to time-constrained variant of Influence Maximization Problem via SAT <i>Stepan Kochemazov</i>	77
SAT Benchmarks of AutoFault Attacking AES, LED and PRESENT <i>Tobias Paxian, Maël Gay, Devanshi Upadhyaya, Bernd Becker, and Iliia Polian</i>	79
Baseball Lineup Selection Benchmarks Description at SAT Competition 2020 <i>Randy Hickey, Nick Feng, and Fahiem Bacchus</i>	80
Duplex Encoding of Antibandwidth Feasibility Formulas Submitted to the SAT Compe- tition 2020 <i>Katalin Fazekas, Markus Sinnl, Armin Biere, and Sophie Parragh</i>	81
Combined SLS and CDCL instances at the SAT Competition 2020 <i>Mate Soos</i>	83
CNFMiter <i>Norbert Manthey</i>	85
Axiom Pinpointing Benchmarks for IPASIR Solvers <i>Norbert Manthey</i>	86
SAT Encodings for Flood-It Puzzle <i>Milan van Stiphout, Muhammad Osama, and Anton Wijs</i>	87
Crafted Benchmarks with Artificial Community Structure and Known Complexity <i>Marc Hartung and Benjamin Kaiser</i>	89
SAT Benchmarks based on Hypertree Decompositions <i>André Schindler and Stefan Szeider</i>	91
Lam’s Problem Benchmarks for the SAT Competition 2020 <i>Curtis Bright, Saeed Nejati, and Vijay Ganesh</i>	92
Polynomial Multiplication in SAT competition 2020 <i>Chu-Min Li, Mao Luo, Fan Xiao, Yu Li, Felip Manyà, and Zhipeng Lu</i>	94
PEQNP Python Library Benchmarks <i>Oscar Riveros</i>	96
A SAT Encoding of School Timetabling <i>Rodrigue Konan Tchinda and Clémentin Tayou Djamegni</i>	97
Station Repacking Problem <i>Xindi Zhang and Shaowei Cai</i>	100
Solver Index	101
Benchmark Index	102
Author Index	103

SOLVER DESCRIPTIONS

Cadical-trail, Cadical-alluip, Cadical-alluip-trail, and Maple-LCM-Dist-alluip-trail at SAT Competition 2020

Randy Hickey
University of Toronto

Nick Feng
University of Toronto

Fahiem Bacchus
University of Toronto

Abstract—This document describes Cadical-trail, Cadical-alluip, Cadical-alluip-trail and Maple-LCM-Dist-alluip-trail where two novel techniques "trail-saving" and "stable-alluip" are implemented.

I. INTRODUCTION

The base solvers we used to implement our techniques into are cadical v1.2.1, obtained from github as of January 1, 2020¹, and Maple-LCM-Dist, obtained from the 2017 SAT Competition [1]. Cadical-trail implements only trail saving and enhancements, cadical-alluip implements only all-UIP, cadical-alluip-trail implements both, and Maple-LCM-Dist-alluip-trail implements both (with fewer trail saving enhancements).

II. TRAIL SAVING

Trail saving is a technique that attempts to avoid doing repeated propagation steps after backtrack. For deeper analysis of the techniques and implementation details, please refer to the paper [2].

When using trail saving, upon each backtrack all literals that are unassigned are stored in order along with their reason clauses (implications) on an *oldtrail* vector. Upon redescend, whenever the top literal d on the *oldtrail* becomes true, all of the literals underneath it up until the next literal with no reason are also implied by unit propagation. If upon restoring a literal it has already been falsified, then a conflict is detected.

In addition, the following enhancements to trail saving were applied. First, each new *oldtrail* can be appended to the beginning of the existing *oldtrail* without discarding the *oldtrail* vector that is still there. Once in a while, this will have to be flushed to avoid growing indefinitely. Second, instead of only looking at the top of the *oldtrail*, one can examine several decision levels down the *oldtrail* to see if a literal has become falsified; if it has then following the same sequence of decisions as those that appear on the *oldtrail* is guaranteed to lead to a conflict. Lastly, restoring literals with their old reasons might keep a "bad" reason around. Whenever a literal about to be restored has a reason above a certain size and/or lbd, then we stop trail saving and start doing unit propagation.

III. STABLE ALLUIP

Stable-allUIP is a novel clause learning scheme to replace 1-UIP learning scheme [3]. Stable-allUIP scheme performs all-UIP [3] like ordered resolutions on top of 1-UIP learnt clause against the assertion trail, and it additionally enforces that the learnt stable-allUIP clause must have the same LBD [4] as the 1-UIP clause. If a literal cannot be resolved without increasing the clause's LBD, then the literal is kept in the learnt clause. The stable-allUIP clause is successfully added to the clause database if and only if it has smaller size. The solver also implemented stable-allUIP optimizations, t_{gap} .

a) t_{gap} : We say the gap value of a conflicting clause is the difference between the clause's LBD and size. We disable stable-allUIP for conflicting clauses with gap value smaller than a floating target, t_{gap} . Initially, $t_{gap} = 0$, and we count the number of times stable-allUIP is attempted and the number of times it successfully yields a shorter clause. On every restart if the success rate since the last restart is greater than 80%, we decrease t_{gap} by one (not allowing it to become negative), and if it is less than 80% we increase t_{gap} by one.

REFERENCES

- [1] F. Xiao, M. Luo, C.-M. Li, F. Manyá, and Z. Lü, "Maplelrb lcm, maple lcm, maple lcm dist, maplelrb lcmoccrestart and glucose-3.0+ width in sat competition 2017," *Proc. of SAT Competition*, pp. 22–23, 2017.
- [2] R. Hickey and F. Bacchus, "Trail saving on backtrack," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference [to appear]*, 2020.
- [3] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient conflict driven learning in boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, R. Ernst, Ed. IEEE Computer Society, 2001, pp. 279–285. [Online]. Available: <https://doi.org/10.1109/ICCAD.2001.968634>
- [4] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, C. Boutilier, Ed., 2009, pp. 399–404. [Online]. Available: <http://ijcai.org/Proceedings/09/Papers/074.pdf>

¹<https://github.com/arminbiere/cadical>

Undominated-LC-MapleLCMDiscChronoBT-DL

Jerry Lonlac
Research Center
IMT Lille Douai - University of Lille
Douai, France
jerry.lonlac@imt-lille-douai.fr

Engelbert Mephu Nguifo
LIMOS, CNRS, UMR 6158
University Clermont Auvergne
Clermont-Ferrand, France
engelbert.mephu_nguifo@uca.fr

Abstract—This paper describes our CDCL SAT solver *Undominated-LC-Maple* which we submit to the SAT competition 2020.

I. INTRODUCTION

Clause Learning [1], [2] is one of the most important components of a conflict driven clause learning (CDCL) SAT solver that is effective on industrial SAT instances. Since the number of learned clauses is proved to be exponential in the worst case, it is necessary to identify the most relevant clauses to maintain and delete the irrelevant ones. As reported in the literature, several learned clauses deletion strategies have been proposed. However the diversity in both the number of clauses to be removed at each step of reduction and the results obtained with each strategy increase the difficulty to determine which criterion is better. Thus, the problem to select which learned clauses are to be removed during the search step remains very challenging. Our SAT solvers *Undominated-LC-MapleLCMDiscChronoBT-DL* presented in this paper integrate an approach to identify the most relevant learned clauses without favoring or excluding any of the learned clause database cleaning strategies proposed, but by adopting the notion of dominance relationship among those measures. These solvers bypass the problem of results diversity and reach a compromise between the measures assessments. Furthermore, they also avoid another non-trivial problem which is the number of deleted clauses at each reduction of the learned clause database.

II. DOMINANCE RELATIONSHIP BETWEEN LEARNED CLAUSES IN MAPLELCMDISCHRONOBT-DL

Undominated-LC-MapleLCMDiscChronoBT-DL was implemented on top of the solver *MapleLCMDiscChronoBT-DL-v3* [3] the recent winning SAT solvers of the last SAT Race 2019 by integrating the learned clause database cleaning approach described in [4] cleaning the clauses in *LOCAL* set. Indeed, *MapleLCMDiscChronoBT-DL-v3* solver organises the learnt clauses in three sets: *CORE*, *TIER2* and *LOCAL*. The clauses of LBD [7] (Literal Block Distance) not greater than a threshold t_1 are stored in *CORE* and are never removed, the clauses of LBD greater than t_1 but not greater than another threshold t_2 are stored in *TIER2*, the remaining learnt clauses are stored in *LOCAL*. Clauses in *TIER2* are moved into *LOCAL* under some conditions and half clauses in *LOCAL*

are removed periodically according to their CVSIDS [8]. Our solver removes the clauses in *LOCAL* using the dominance approach.

More precisely, this approach is obtained by selecting at each cleaning step of the learned clauses database, a set of current undominated learned clauses [4]) according to a set of learned clauses relevant measures, and to delete all the learned clauses dominated by at least one of the current undominated learned clauses. *Undominated-LC-MapleLCMDiscChronoBT-DL* solver avoids another non-trivial problem which is the amount of learned clauses to be deleted at each reduction step of the learned clauses database by dynamically determining the number of learned clauses to delete at each cleaning step while the state of the art approaches removes half clauses at each cleaning step. Dominance relationship between learned clauses is described in more detail in [4].

We submit to the SAT competition 2020 an implementation of our *Undominated-LC-MapleLCMDiscChronoBT-DL* solver integrating three learned clauses relevant measures in the dominance relationship: *SIZE* [5], [6] that considers the shortest learned clauses as the most relevant, *LBD* that considers the clauses with the smallest LBD measure as the most relevant, and *CSIDS* that prefers the learned clauses most involved in recent conflict relationship.

III. ALGORITHM FOR FINDING UNDOMINATED LEARNED CLAUSES

During the search process, the CDCL SAT solvers learn a set of clauses which are stored in the learned clauses database $\Delta = \{c_1, c_2, \dots, c_n\}$. At each cleaning step, we evaluate these clauses with respect to a set $\mathcal{M} = \{m_1, m_2, \dots, m_k\}$ of relevant measures. We denote $m(c)$ the value of the measure m for the clause c , $c \in \Delta$, $m \in \mathcal{M}$. Since the evaluation of learned clauses varies from a measure to another one, using several measures could lead to different outputs (relevant clauses with respect to a measure). For example, consider the three learned clauses, c_1 , c_2 and c_3 with their values on the three relevant measures *LBD*, *SIZE* and *CVSIDS* [8]:

- $SIZE(c_1) = 8$, $LBD(c_1) = 3$, $CVSIDS(c_1) = 1e^{10}$;
- $SIZE(c_2) = 6$, $LBD(c_2) = 5$, $CVSIDS(c_2) = 1e^{20}$;
- $SIZE(c_3) = 5$, $LBD(c_3) = 4$, $CVSIDS(c_3) = 1e^{30}$.

It comes from the previous example that c_1 is the best clause with respect to the *LBD* measure whereas it is not the case according to the evaluation of *SIZE* measure which favors c_3 . This difference of evaluations is confusing for any process

of learned clauses selection. Hence, we can utilize the notion of dominance between learned clauses to address the selection of relevant ones.

Algorithm 1: reduceDB_Dominance_Relationship

Input: Δ : the learned clauses database; \mathcal{M} : a set of relevant measures
Output: Δ the new learned clauses database

```

1 sortLearntClauses(); /* by degree of compromise
  criterion */
2 ind = 1;
3 j = 1;
4 undoC = 1; /* the number of current undominated
  clauses */
5 while ind < |\Delta| do
6   c = \Delta[ind]; /* a learned clause */
7   if c.size() > 2 and c.lbd() > 2 then
8     cpt = 0;
9     while cpt < undoC and \neg dominates(\Delta[cpt], \Delta[ind],
10      \mathcal{M}) do
11       cpt++;
12       if cpt >= undoC then
13         saveClause();
14         j++;
15         undoC = j
16       else
17         removeClause();
18     else
19       saveClause();
20       j++;
21       undoC = j
22   ind++;
23 return \Delta;

23 Function dominates(cMin: a clause, c: a clause, \mathcal{M})
24 i = 0;
25 while i < |\mathcal{M}| do
26   m = \mathcal{M}[i]; /* a relevant measure */
27   if m(c) \succeq m(cMin) then
28     return FALSE;
29   i++;
30 return TRUE;

```

Algorithm 1 starts by sorting the set Δ of learned clauses according to their degree of compromise [4]. It is easy to see that the first clause of Δ is not dominated, it is the *top-1*. So, at the beginning of the algorithm, we have at least one undominated clause. In step *ind* ($ind > 1$) of the outermost while-loop, the clause in position *ind* is compared to at most $ind - 1$ undominated clauses. As soon as it is dominated, it is removed, otherwise, it is kept as undominated clauses.

Degree of compromise: Given a learned clause c , the degree of compromise of c with respect to the set of learned clauses relevant measures \mathcal{M} is defined by $DegComp(c) = \frac{\sum_{i=1}^{|\mathcal{M}|} \widehat{m}_i(c)}{|\mathcal{M}|}$, where $\widehat{m}_i(c)$ corresponds to the normalized value of the clause c on the measure m_i .

dominance value: Given a learned clause relevant measure m and two learned clauses c and c' , we say that $m(c)$ dominates $m(c')$, denoted by $m(c) \succ m(c')$, iff $m(c)$ is preferred to $m(c')$.

dominance clause: Given two learned clauses c, c' , the dominance relationship according to the set of learned clauses relevant measures \mathcal{M} is defined as follows: c dominates c' , denoted $c \succ c'$, iff $m(c) \succ m(c'), \forall m \in \mathcal{M}$.

IV. SUBMITTED VERSIONS

We submit two variants of our solver to the SAT competition 2020 and different scripts to start it with different parameters. One variant that maintains only the undominated learned clauses at each cleaning step and another variant *Topk-Undominated-LC-MapleLCMDiscChronoBT-DL* that deletes from the learned clauses database all the clauses dominated by the k first undominated learned clauses ranked in the increasing order of their degree of compromise at each cleaning step. To get the *Topk-Undominated-LC-MapleLCMDiscChronoBT-DL* version, we replace the lines 14 and 20 of algorithm 1 by $undoC = \min(k, j)$.

How to normalize the values of the learned clauses?

For the two variants of our solver submitted to the SAT competition 2020, we propose a way for normalizing the values of the learned clauses. Given a learned clause relevant measure m and a learned clause c , we normalize the value of the clause c on the measure m using the approach described in the following.

- If m higher values are better, then $\widehat{m}(c) = \frac{m(c)}{U}$, where U is the upper bound of the learned clause values on the measure m ;
- If m smaller values are better, then $\widehat{m}(c) = \frac{1}{m(c)}$.

For the *Topk-Undominated-LC-MapleLCMDiscChronoBT-DL* variant, we submit three versions with respectively the parameter $k = 16, 24, 36$ (*Top{16, 24, 36}-Undominated-LC-MapleLCMDiscChronoBT-DL*).

ACKNOWLEDGMENT

This research has been supported by Auvergne-Rhône-Alpes region and European Union through the European Regional Development Fund (ERDF) and University Clermont Auvergne. We also thank CRIL (Lens Computer Science Research Lab) Lab for providing them computing server and the authors of MapleLCMDiscChronoBT-DL SAT solver for putting the source code available.

REFERENCES

- [1] J. P. M. Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, 1999.
- [2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *38th Design Automation Conference (DAC)*, 2001, pp. 530–535.
- [3] S. Kochemazov, A. Nadel, O. Zaikin, V. Kondratiev, and A. Semenov, "Maplelcmdischronobt-dl, duplicate learnts heuristic-aided solvers," in *SAT Competition: Solver and Benchmark Descriptions*. vol. B-2019-1 of Dpt. of Computer Science Series, 2019, p. 29.
- [4] J. Lonlac and E. Mephu Nguifo, "Towards learned clauses database reduction strategies based on dominance relationship," *CoRR*, vol. abs/1705.10898, 2017.
- [5] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," *Discrete Applied Mathematics*, vol. 155, no. 12, pp. 1549 – 1561, 2007.
- [6] S. Jabbour, J. Lonlac, L. Saïs, and Y. Salhi, "Revisiting the learned clauses database reduction strategies," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 8, p. 1850033, 2018.
- [7] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, 2009, pp. 399–404.
- [8] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003, pp. 502–518.

MLCMDChronoBT-DL-Scavel and its friends at the SAT Competition 2020

Chengdu, China

lizhihui@swjtu.edu.cn, wgf1024@swjtu.edu.cn,
xuyang@swjtu.edu.cn, qschen@home.swjtu.edu.cn

1st Zhihui Li, 2th Guanfeng Wu, 3th Yang Xu, 4th Qingshan Chen
School of Mathematics

*National-Local Joint Engineering Laboratory of System Credibility
Automatic Verification, Southwest Jiaotong University*

Abstract— this document describes MLCMDChronoBT-DL-Scavel and its friends at the SAT Competition 2020.

I. INTRODUCTION

The community structure exists in the real problems. After formalizing the problems, the community structure characteristics can still be mined and utilized, although some community characteristics will be lost after the transformation[1][2][3]. Making full use of community characteristics in limited time is a research hotspot[4].

Based on the existing VSIDS and its variants, the selection strategy of decision variables continues to the selection of the first element of the restart. It make the solver unable to achieve a real restart. Our guiding idea is to adjust the selection method of starting decision variables so that the first variable of restart is generated from another selection mode. In this way, the first argument of the reboot may be different from the community structure of the part of the argument that has been trapped in the local space.

II. MLCMDCHRONOBT-DL-SCAVEL AND MLCMDCHRONOBT-DL-V2.2SCAVELRFV

Although there are many solvers that will carry out community solution in special time period, please make use of them, but all of them are aimed at the sample that can satisfy the solution time of small community, especially when the learning clause set is also involved, it is necessary to try to limit the community solution time[5].

So we settle for second best, and the competition of the solver version, the text will be more a part of the argument rough thought contained in one or more of the community, they are in the community may be related to solver last solving process finally enter the search area of different argument community, restart can be selected as the first argument optional argument set. Like the inIDGlucose solver, the characters of the original clause set text are first analyzed and applied to the selection of the first element in the solver restart stage. Secondly, one of the alternative arguments is taken as the first argument of restart. At the same time, in order to ensure the excellent characteristics of the existing solvers, the above selection mode of change decision variables is only used in a few restart stages, and is limited to the selection of the first variable to restart.

In addition, the ideological Improved learning clause management strategy we proposed in 2018 has also been considered[6]. This time we change the static threshold value to the dynamic threshold value, which is given dynamically

from the distance of the time the learning clause is used .For this purpose, we designed a special scoring function as (1).

$$\text{act_delta}[c_i]=1+\log(\text{conflicts}_i)*\log(\text{conflicts}_i) \quad (1)$$

Where conflicts_i is the i -th conflict since the beginning of the current search, and the learning clause c_i is used for conflict analysis and gets $\text{act_delta}[c_i]$ as the incremental value of the collision score.

The above changes are based on the 2019 champion solver (MapleLCMDistChronoBT-dl-v2.2). In addition to our own improvement ideas above, this revision also adds new methods such as PSIDS for polarity selection and COREFIRST for core clauses to be propagated first, which have already appeared in the 2019 competition.

III. ABOUT PARALLEL VERSION

We add the learnt clause used frequency strategy to other parallel solvers to see the effect of this strategy in other parallel solvers. We build our parallel Solvers based on Sryup[7] and abcd_SAT[8], So the name of parallel solvers are “syrup_Scavel” and “abcd_para18_Scavel”.

IV. ACKNOWLEDGMENTS

The author would like to thank the developers of all predecessors of Minisat, Syrup, MapleLCMDistChronoBT-DL, MapleCOMSPS CHB VSIDS, PSIDS MapleLCMDistChronoBT, MapleLCMDistCBTcoreFirst, and all the authors who contributed the modifications that have been integrated. Their solvers are the foundation of our learning and improvement. This work also supported by the Fundamental Research Funds for the Central Universities (No.2682020CX59) and the National Natural Science Foundation of P. R. China (Grant No. 61673320).

REFERENCES

- [1] Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, and L. Simon, “Impact of community structure on SAT solver performance,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8561 LNCS, doi: 10.1007/978-3-319-09284-3_20.
- [2] J. H. Liang and V. Ganesh, *Understanding VSIDS Branching Heuristic in CDCL SAT Solvers*. Springer International Publishing, 2015.
- [3] M. J. H. Heule, B. Kiesl, and A. Biere, “Encoding redundancy for satisfaction-driven clause learning,” in *International Conference*

on Tools and Algorithms for the Construction and Analysis of Systems, 2019, pp. 41–58.

- [4] C. Ansótegui, M. L. Bonet, J. Giráldez-Cru, and J. Levy, “Structure features for SAT instances classification,” *J. Appl. Log.*, vol. 23, pp. 27–39, 2017.
- [5] N. Manthey, “MergeSAT,” *Proc. SAT Compet.*, vol. 2019, 2019.
- [6] Y. Xu, G. Wu, Q. Chen, and S. Chen, “Maple LCM Scavel and Maple LCM Scavel 200,” *Proceeding SAT Compet.*, p. 27, 2018.
- [7] G. Audemard and L. Simon, “On the Glucose SAT Solver,” *Int. J. Artif. Intell. Tools*, vol. 27, no. 01, p. 1840001, 2018, doi: 10.1142/S0218213018400018.
- [8] J. Chen, “AbcdSAT and Glucose hack: Various Simplifications and Optimizations for CDCL SAT solvers,” *Proc. SAT Compet.*, pp. 10–12, 2018.

Relaxed Backtracking with Rephasing

Xindi Zhang^{1,2} and Shaowei Cai^{1,2*}

¹State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

²School of Computer Science and Technology, University of Chinese Academy of Sciences, China

dezhangxd@163.com, shaoweicai.cs@gmail.com

Abstract—A novel relaxing CDCL method and a new probability based phase saving technology are described. Based on this method, we develop three solvers called `Relaxed_LCMDCBDL`, `Relaxed_LCMDCBDL_noTimeParam` and `Relaxed_LCMDCBDL_newTech`, which are based on `MapleLCMDistChronoBT-DL`.

I. INTRODUCTION

We improve the relaxing CDCL method [3] using the information in CCAr [2] this year. By using some full assignments (also named phases) with certain probability before each inprocessing, the performance of solvers on satisfiable instances are improved.

II. METHODS

A. Relaxed CDCL Approach

The idea is to relax the backtracking process for protecting promising partial assignment, where a promising assignment is defined according to its consistency (no conflict) and length. When the CDCL process reaches a node with some conditions, the algorithm enters a non-backtracking phase until it gets a full assignment β . Then Local search process is then called to seek for a model near β . If the local search fails to find a model within certain limits, then the algorithm goes back to the normal CDCL search from the node where it was interrupted.

For a given conjunctive normal formula (CNF) with V variables, $|V|$ denotes the number of variables. And for a partial assignment α in CDCL process without conflicts, $|\alpha|$ is the number of assigned variables in α , then we name the max number of $|\alpha|$ in CDCL history as *max_trail*.

Here we control the entrance of local search process by p , q and c , where p , q presents $|\alpha|/|V|$ and $|\alpha|/max_trail$. And c presents the inprocessing times between two local search process.

B. Probability Based Phase Saving

Phase saving is a well-known technique which saves the assignment of variables when traceback and uses the assignment when variables are selected as decision variables. Like the rephase technique in CaDiCaL [1], we use vectors to save different phases, the difference is that we use probability to select which phase to use after each restart. The probability of each phase is shown in “Table. I”

* Corresponding author

TABLE I
PROBABILITY OF EACH PHASE

Index	Phase	Probability(%)
1	The best local search result with minimized unsatisfied clauses	10
2	The last local search result	30
3	Phase with <i>max_trail</i>	30
4	Reverse current phase	5
5	Reverse 1 phase	2.5
6	Reverse 3 phase	2.5
7	Rand phase	14
8	All True phase	0.5
9	All False phase	0.5
10	Keep current phase	5

III. IMPLEMENTATION AND MAIN PARAMETERS

A. Relaxed_LCMDCBDL

`Relaxed_LCMDCBDL` use both methods mentioned above and for the relaxing method, algorithms call local search process when $p \geq 0.4$ or $q \geq 0.9$ and $c \geq 400$. `MapleLCMDistChronoBT-DL` [4] is the base of the other solvers.

B. Relaxed_LCMDCBDL_noTimeParam

We find there is a switch in `MapleLCMDistChronoBT-DL` between VSIDS and CHB when the time reach 2500 seconds, this method will make the results unstable when tested on different clusters, or even on the same clusters but at different times. For the stableness, we replace the time based switch with a restart based switch, i.e., every n inprocessing, the algorithm will switch the branching algorithm once between VSIDS and LRB. For our solvers, $n = 500$.

C. Relaxed_LCMDCBDL_newTech

For better utilize the information in Local Search process, we use a vector *occ_num* to record the number of occurrences of variables in unsat clauses after each flip. And we see one percent of occurrences as one conflict in CDCL process.

For example, there is a CNF with two clauses $\{v_1, x_2\}, \{v_2, v_3\}$, which show up in the unsat clauses 10 and 20 times respectively. Assume the Local Search process conducts 100 flips, then $occ_num[v_1] = 10$, $occ_num[v_2] = 30$, $occ_num[v_3] = 20$, and v_1, v_2, v_3 are considered as encountering 10, 30 and 20 conflicts respectively. When the

Local Search process ends, the branching heuristic algorithm will utilize the information.

Relaxed_LCMDCBDL_newTech adjusts the local search entrance condition to $c \geq 300$ in order to adapt the local search information.

IV. ACKNOWLEDGEMENT

This work was supported by Beijing Academy of Artificial Intelligence (BAAI), and Youth Innovation Promotion Association, Chinese Academy of Sciences [No. 2017150].

REFERENCES

- [1] A. Biere. Cadical at the sat race 2019. *SAT RACE 2019*, page 8.
- [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. Four relaxed cdcl solvers. *SAT RACE 2019*, page 35.
- [4] S. Kochemazov, O. Zaikin, V. Kondratiev, and A. Semenov. Maplelcmdistchronobt-dl, duplicate learnts heuristic-aided solvers at the sat race 2019. *SAT RACE 2019*, page 24.

ExMapleLCMDistChronoBT, upGlucose-3.0_PADC and PaInleSS_ExMapleLCMDistChronoBT in the SC20

Rodrigue Konan Tchinda

*Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
rodriguetkonanktr@gmail.com*

Clémentin Tayou Djamegni

*Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
dtayou@gmail.com*

Abstract—This document describes the sequential solvers ExMapleLCMDistChronoBT and upGlucose-3.0_PADC and the parallel solver PaInleSS_ExMapleLCMDistChronoBT submitted to the 2020 SAT Competition.

I. INTRODUCTION

Today’s SAT solvers are becoming increasingly efficient. The class of instances that can now be tackled by state-of-the-art solvers is getting more and more greater. The explanation is the great advance of the research on SAT algorithms and heuristics. A lot has been done since the advent of CDCL (Conflict-Driven Clause Learning) [1]–[4] and today, various improvements have been integrated to solvers with success [5]–[13]. However, there is still much room for improvements.

It is common that huge performances gains are achieved through small modifications of solvers’ sources codes. The sequential solvers we present in this paper follow this idea by integrating into some well-performing SAT solvers a series of simple modifications. The latter include the *periodic aggressive learned clause database cleaning (PADC)* strategy [12], [14], the *polarity state independent decaying sum (PSIDS)* heuristic [14] and the *duplicate learned (DL)* heuristic [15]. Additionally, a heuristic for initializing and updating variable activities which we refer to as *occurrence-based variable activity update (OVAU in short)* is also considered. All these heuristics are implemented on top of MapleLCMDistChronoBT [17] with command line options to individually enable or disable any of them. The resulting solver is submitted to the main Track of the competition. The OVAU heuristic is used in combination with the PADC strategy within Glucose-3.0 to participate in the Glucose Hack Track. It is worth mentioning that all the above mentioned heuristics showed good performances during the past competitions. DL was implemented in MapleLCMDistChronoBT-DL [15] which won the Gold medal of the main Track of the SR19. PSIDS was implemented in PSIDS_MapleLCMDistChronoBT [14] which won the Bronze medal in the UNSAT Track of the SR19. PADC was integrated in glucose-3.0_PADC [12] that won the Bronze medal of the Random SAT Track of the SC18

and a special case of OVAU was implemented in inIDGlucose [16] which won the Silver medal of the Glucose Hack Track of the SC18. Hence, by integrating these heuristics into a single solver, we want to evaluate how performances can be affected by combining some of them. The latter solver is further used as a sequential engine with the PaInleSS [18] framework to build a parallel portfolio SAT solver that is submitted to the parallel Track.

II. OCCURRENCE-BASED VARIABLE ACTIVITY UPDATE HEURISTIC

Picking the right variables and assigning the right values to them is an important ingredient that could make a solver more efficient. This task is achieved by the branching heuristic based on information such as variable activity [19], learning rate [20], conflict history [21] etc. But at the beginning of the search the information is not accurate and therefore might not help in choosing the most suitable branching variables. As a consequence, solvers might make bad initial branchings that might direct them toward unfruitful subspaces and hence, greatly impacts the solving times. Even during the search, the branching heuristic could be strengthened by taking into account more aspects of the instance that is being solved. Initializing variable activities with good values at the beginning of the search has shown to be very promising considering the performance achieved by the solver inIDGlucose [16] at the 2018 SAT Competition. We follow a similar idea here by providing a slight generalization.

Given a formula \mathcal{F} and a set of learned clauses Δ , the score of a literal x is computed as follows :

$$score(x) = \sum_{c \in \mathcal{F} \cup \Delta, x \in c} \frac{1}{f(|c|)}$$

where f is an increasing positive function which we call the *penalizing function*.

Once the scores computed, the activity of a variable x , $act(x)$ is updated as follows:

$$act(x) += score(x) \times score(\neg x)$$

The idea here is similar to that of the MOMS heuristic which favors branching on variables that occur more frequently in short clauses. Activities are updated at the beginning of the search and optionally at specific moments during the search. This heuristic which we call *Occurrence-based Variable Activity Update (OVAU in short)* is implemented in all the solvers presented in this paper. It is worth noting that OVAU heuristic does not replace the branching heuristic of the solver, but just strengthens it. In our implementations, we considered the following penalizing functions : $f : x \mapsto x$, $f : x \mapsto x^2$, $f : x \mapsto x^3$ and $f : x \mapsto 2^x$.

Similarly to inIDGlucose [16], the initial polarities of a variable x is set to the truth value of $score(x) > score(\neg x)$. Note that we only update the polarities in this way at the beginning of the search.

III. EXMAPLELCMDISTCHRONOBT

We implemented the previously mentioned heuristics namely PADC, PSIDS, DL and OVAU on top of the 2018 SAT Competition winner MapleLCMDistChronoBT [17] with the possibility of individually enabling or disabling them. We called the resulting solver ExMapleLCMDistChronoBT. The solver ExMapleLCMDistChronoBT was submitted to the 2020 SAT Competition with four different configurations:

- The first configuration : PADC and DL enabled;
- The second Configuration : PSIDS and DL enabled;
- The third configuration: PADC, DL and OVAU enabled, where the penalizing function for OVAU is $f : x \mapsto 2^x$.
- The fourth configuration: PADC, DL and OVAU enabled with $f : x \mapsto x$ as penalizing function for OVAU.

In these versions, the OVAU heuristic is only used at the beginning of the search. Note that in MapleLCMDistChronoBT, there are three different arrays used to store variable activities: activity_CHB, activity_VSIDS and activity_distance. Our implementation allows to select which of these types of variable activities the OVAU heuristic is applied to.

IV. UPGLUCOSE-3.0_PADC

The solver upGlucose-3.0_PADC is a Glucose-3.0 Hack. It combines the PADC strategy [12], [14] and the OVAU heuristic where the penalizing function is $f : x \mapsto 2^x$. The parameters of the PADC strategy are the same as those of Glucose-3.0_PADC_10 [12] submitted to the SAT Competition 2018. The OVAU heuristic is used at the beginning of the search as well as during the search after deep cleaning steps.

V. PAInLeSS_EXMAPLELCMDISTCHRONOBT

PaInLeSS_ExMapleLCMDistChronoBT is a portfolio parallel solver built with the PaInLeSS framework [18] and that uses the previous solver ExMapleLCMDistChronoBT as sequential engine. Recall that PaInLeSS is a framework that greatly facilitates the implementation of parallel SAT solvers by letting developers to concentrate on functional aspects leaving common issues related to parallelization to the framework.

We submitted two versions of our parallel solver to the 2020 SAT Competition.

For the first version, workers are divided into two groups. The first group consists of solvers that collaborate by only exporting their learned clauses to the others. But they do not import any clause from them. By doing so, we want to prevent these solvers from being influenced by others which could impact their performances since shared learned clauses do not always positively impact efficiency. The second group consists of workers that export and import clauses to/from others. This second group leverages the benefit that can be gained through collaboration. Each worker shares with the others all its learned clauses having an LBD [22], [23] score under a certain threshold. Additionally, workers are configured to use specific combinations of the above mentioned heuristics. Table I gives the enabled heuristics for the first twelve workers. For the others, we use the configuration number $id\%12$ (where id is the identifier of the worker) and random initial activity scores and polarities for variables.

TABLE I
CONFIGURATIONS OF THE FIRST 12 WORKERS

Configuration	PADC	DL	PSIDS	OVAU
0	no	no	no	no
1	no	yes	no	no
2	yes	yes	no	yes
3	yes	yes	no	yes
4	yes	yes	no	no
5	yes	no	no	yes
6	yes	no	no	yes
7	yes	no	no	yes
8	no	yes	yes	no
9	yes	no	no	yes
10	no	yes	yes	yes
11	no	no	yes	yes

In the second version, all solvers are allowed to import and export clauses and only DL is enabled. Sharing is based on clause LBD scores as in the first version. Diversification is achieved through random initial activities and polarities for variables.

VI. ACKNOWLEDGMENTS

We would like to thanks the authors of MapleLCMDistChronoBT [17], MapleLCMDistChronoBT-DL [15], Glucose [22], [23], inIDGlucose [16] and PaInLeSS [18].

REFERENCES

- [1] J. Marques-Silva and K. Sakallah, “Grasp-a new search algorithm for satisfiability. iccad,” 1996.
- [2] H. Zhang, “Sato: An efficient propositional prover,” in *Automated DeductionCADE-14*. Springer, 1997, pp. 272–275.
- [3] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, “Efficient conflict driven learning in a boolean satisfiability solver,” in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2001, pp. 279–285.
- [4] C. P. Gomes, B. Selman, H. Kautz *et al.*, “Boosting combinatorial search through randomization,” *AAAI/IAAI*, vol. 98, pp. 431–437, 1998.
- [5] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, “Symmetry-breaking predicates for search problems,” *KR*, vol. 96, pp. 148–159, 1996.
- [6] F. A. Aloul, I. L. Markov, and K. A. Sakallah, “Shatter: efficient symmetry-breaking for boolean satisfiability,” in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 836–839.

- [7] B. Benhamou, T. Nabhani, R. Ostrowski, and M. R. Saidi, "Enhancing clause learning by symmetry in sat solvers," in *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, vol. 1. IEEE, 2010, pp. 329–335.
- [8] J. Huang, "Extended clause learning," *Artif. Intell.*, vol. 174, no. 15, pp. 1277–1284, Oct. 2010. [Online]. Available: <http://dx.doi.org/10.1016/j.artint.2010.07.008>
- [9] G. Audemard, G. Katsirelos, and L. Simon, "A restriction of extended resolution for clause learning sat solvers," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.
- [10] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker, "Improved static symmetry breaking for sat," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2016, pp. 104–122.
- [11] R. Konan Tchinda and C. Tayou Djamegni, "Enhancing static symmetry breaking with dynamic symmetry handling in CDCL SAT solvers," *International Journal on Artificial Intelligence Tools*, vol. 28, no. 03, p. 1950011, 2019. [Online]. Available: <https://doi.org/10.1142/S0218213019500118>
- [12] R. K. Tchinda and C. T. Djamegni, "Scalope, penelope mdlc and glucose-3.0 padc in sc18," *SAT COMPETITION 2018*, p. 42, 2018.
- [13] —, "Enhancing reasoning with the extension rule in cdcl sat solvers," in *Quatrimme Edition de la Confrence de Recherche en Informatique CRI2019*, 2019.
- [14] —, "Padc maplecmdistchronobt, padc maple lcm dist and psids maplecmdistchronobt in the sr19," *SAT RACE 2019*, p. 33.
- [15] S. Kochemazov, O. Zaikin, V. Kondratiev, and A. Semenov, "Maplecmdistchronobt-dl, duplicate learnts heuristic-aided solvers at the sat race 2019," *SAT RACE 2019*, p. 24.
- [16] D. Jo, "inidglucose," *SAT COMPETITION 2018*, p. 41, 2018.
- [17] V. Ryvchin and A. Nadel, "Maple_lcm_dist_chronobt: Featuring chronological backtracking," *SAT COMPETITION 2018*, p. 29, 2018.
- [18] L. Le Frioux, S. Baair, 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.
- [19] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.
- [20] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for sat solvers," in *Theory and Applications of Satisfiability Testing – SAT 2016*, N. Creignou and D. Le Berre, Eds. Cham: Springer International Publishing, 2016, pp. 123–140.
- [21] —, "Exponential recency weighted average branching heuristic for sat solvers," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
- [22] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, vol. 9, 2009, pp. 399–404.
- [23] —, "On the glucose sat solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 01, p. 1840001, 2018.

MapleLCMDistChronoBT-DL-v3, the duplicate learns heuristic-aided solver at the SAT Competition 2020

Stepan Kochemazov^{*}, Oleg Zaikin[†], Victor Kondratiev[‡] and Alexander Semenov[§]
Laboratory 6.2, ISDCT SB RAS
Irkutsk, Russia

Email: ^{*}veinamond@gmail.com, [†]zaikin.icc@gmail.com, [‡]vikseko@gmail.com, [§]biclop Rambler@yandex.ru

Abstract—This document describes the MapleLCMDistChronoBT-DL-v3 solver which is based on the SAT Competition 2018 winner, the MapleLCMDistChronoBT solver, augmented with duplicate learns 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 ≤ 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%.

The research was funded by Russian Science Foundation (project No. 16-11-10046).

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 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 (to appear),” in *ECAI*, 2020.
- [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.

F2TRC: deterministic modifications of SC2018-SR2019 winners

Stepan Kochemazov
Laboratory 6.2, ISDCT SB RAS
Irkutsk, Russia
Email: veinamond@gmail.com

Abstract—This document describes the three deterministic solvers which are the modifications of the SAT Competition 2018 winner, the `MapleLCMDistChronoBT` solver, and the SAT Race 2019 winner, the `MapleLCMDistChronoBT-DL-v3` solver.

I. INTRODUCTION

The SAT solvers participating in annual competitions in recent years are often heavily biased towards the winner(s) of the previous competition. For example, among 55 solvers that participated in SAT Race 2019 at least 16 were based on the SAT Competition 2018 winner, the `MapleLCMDistChronoBT` solver. The downside of this phenomenon is that sometimes undesirable traits of the solver are inherited and preserved in its offsprings just because the algorithm won in the competition. One of these traits is the nondeterministic switching between LRB [1] and VSIDS [2] phases at 2500 seconds, first introduced in `MapleCOMSPS` in 2016 [3], and then inherited by `Maple_LCM_Dist` in 2017 [4], `MapleLCMDistChronoBT` in 2018 [5] and many other solvers in SAT Competitions throughout 2017-2019.

The goal of this research is to present deterministic variants of the solvers, that attempt to fix some of the (in our opinion) unwanted features of the descendants of `MapleCOMSPS` without introducing completely new heuristics and techniques.

In more detail the motivation behind this work, along with vast experimental evaluation and detailed description of proposed modifications, can be found in [6]. Below is the technical description.

We work with two underlying solvers: the winner of SAT Competition 2018, `MapleLCMDistChronoBT` [5], and the winner of SAT Race 2019, `MapleLCMDistChronoBT-DL-v3` [7], [8]. Note, that the main distinction between the latter and the other solvers with duplicate learnts (DL) heuristic that participated in SAT Race 2019 is in the use of deterministic switching between LRB and VSIDS phases. Some of the improvements detailed below overwrite the switching variant of `MapleLCMDistChronoBT-DL-v3`, thus the resulting solver can be viewed as a modification of any of the three `MapleLCMDistChronoBT-DL` solvers from SAT Race 2019.

The research was funded by Russian Science Foundation (project No. 16-11-10046).

II. MAJOR MODIFICATIONS

The modifications of original solvers include: the deterministic LRB-VSIDS switching scheme, the change to the handling of *Tier2* learnts, and the introduction of `reduceDB`-like procedure for *Core* learnts.

A. Deterministic LRB-VSIDS switching strategy (*f2*)

After many experiments we chose the scheme that essentially copies the one first introduced in `MapleCOMSPS_LRB_VSIDS_2` [9]. It uses the `phase_allotment` variable to allocate the number of conflicts for the next VSIDS or LRB phase. The initial value of `phase_allotment` is 10000. The solver first allocates `phase_allotment` conflicts to the LRB phase, then the same amount of conflicts to the VSIDS phase, increases `phase_allotment` by 1.1 and repeats the cycle anew.

B. Changes to handling *Tier2* and *Core* learnts (*trc*)

The initial idea to separate learnt clauses into *Core*, *Tier2* and *local* is due to Chanseok Oh [10]. In our experiments it turned out that for $\approx 80\%$ instances from SAT Race 2019, `MapleLCMDistChronoBT-f2` accumulates more than 50 000 core learnts. It slows down the propagation considerably and that many *Core* learnts are not always useful. The average size of *Tier2* per instance varies from about 200 to 12000 learnts. Therefore, at times the learnt clause minimization (LCM) procedure [4], that improves the quality of *Tier2* learnts, has nothing to work with. In the `trc` version there are two modifications detailed below which are aimed at improving the solver behavior.

As soon as the *Core* size exceeds `core_size_limit` the `reduceDB_Core` procedure is invoked. It sorts *Core* learnts in the ascending order based on their `lbd` [11] and the size for equal `lbd`. Then all the clauses from the second half (with larger `lbd` and size) that did not participate in any of the most recent 100000 conflicts are moved to *Tier2*. The value of `core_size_limit` is initialized by 50000 and is multiplied by 1.1 each time the procedure is invoked.

The `reduceDB_Tier2` is also reorganized. In the baseline `MapleLCMDistChronoBT` the *Tier2* is reduced every 10000 conflicts and as a result all clauses that have not participated in the most recent 30000 conflicts are moved to

Local. In `trc` the *Tier2* learnts are accumulated until a pre-specified size limit (7000). During the purge only half of the clauses that participated in the recent conflicts is preserved. In particular, we order clauses in *Tier2* in the descending order in accordance with the number of the most recent conflict they participated in, and move the second half to *Local*.

C. Minor modifications

In order to accommodate the introduced major modifications, it was necessary to save the touched status for *core* learnts. Also, the touched is updated during LCM.

III. RESULTING SOLVERS

The proposed techniques were implemented in three solvers. Two of them are based on `MapleLCMDistChronoBT`. They are `MapleLCMDistChronoBT-f2trc` and `MapleLCMDistChronoBT-f2trc-s`. The third solver is based on the `MapleLCMDistChronoBT-DL-v3` and is called `MapleLCMDistChronoBT-DL-f2trc`. The solvers `MapleLCMDistChronoBT-f2trc` and `MapleLCMDistChronoBT-DL-f2trc` implement the techniques outlined earlier in the configuration described.

A. `MapleLCMDistChronoBT-f2trc-s`

This solver variant is an experimental implementation. It uses the same ideas as the other solvers, but in a slightly more sophisticated manner. In particular, there are additional parameters for invoking `reduceDB_Tier2` and `reduceDB_Core`, the procedures use slightly more complicated logic.

The main distinction between `f2trc-s` variant and `f2trc` is that the former uses a kind of *hot streak* heuristic. The idea behind it was inspired by the study of the *Cadical* solver [12], in particular of how it performs rephasing. However, the resulting implementation is different from that of *Cadical* and has a different purpose. `f2trc-s` tracks the maximum trail size that was achieved during the most recent restart and the overall maximum trail size. If the current `max_trail_size` is larger than the previous one and is at least $0.9 \times \text{overall_max_trail_size}$ then the solver doubles the next restart interval during LRB phase and increases the remaining number of conflicts for the current LRB/VSIDS phase to 10000 if it falls under this mark. The motivation here is to encourage the solver to find a satisfying assignment if it makes progress in this direction. In the experiments this version shows peculiar behavior, often resulting in lower runtimes on satisfiable instances. However, it is not currently clear whether the `max_trail_size` is really a good indication of the solver's progress and if the (sometimes) better behavior comes solely from the chains of increased restart intervals.

REFERENCES

- [1] 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.
- [2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01, 2001, pp. 530–535.
- [3] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maple-COMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *Proc. of SAT Competition 2016*, vol. B-2016-1, 2016, pp. 52–53.
- [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] A. Nadel and V. Ryvchin, "Chronological backtracking," in *SAT*, ser. LNCS, vol. 10929, 2018, pp. 111–121.
- [6] S. Kochemazov, "Improving implementation of sat competitions 2017-2019 winners (to appear)," in *SAT*, 2020.
- [7] S. Kochemazov, O. Zaikin, V. Kondratiev, and A. Semenov, "MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019," in *Proc. of SAT Race 2019*, vol. B-2019-1, 2019, p. 24.
- [8] S. Kochemazov, O. Zaikin, A. Semenov, and V. Kondratiev, "Speeding up CDCL inference with duplicate learnt clauses (to appear)," in *ECAI*, 2020.
- [9] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maple-COMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS," in *Proc. of SAT Competition 2017*, vol. B-2017-1, 2017, pp. 20–21.
- [10] C. Oh, "Between SAT and UNSAT: The fundamental difference in CDCL SAT," in *SAT*, ser. LNCS, vol. 9340, 2015, pp. 307–323.
- [11] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI*, 2009, pp. 399–404.
- [12] A. Biere, "CaDiCaL at the SAT Race 2019," in *Proc. of SAT Race 2019*, vol. B-2019-1, 2019, pp. 8–9.

Four CDCL solvers based on expLRB, expVSIDS and Glue Bumping

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

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

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

Abstract—This document describes four CDCL SAT solvers: `exp_V_LGB_MLD_CBT_DL`, `exp_V_L_MLDCBT_DL`, `exp_L_MLD_CBT_DL`, `exp_V_MLD_CBT_DL`, which are entering the SAT Competition-2020. These solvers are based on three new ideas: 1) Guidance of Learning Rate Based (LRB) and Variable State Independent Decaying Sum (VSIDS) branching heuristics via random exploration amid pathological phases of conflict depression and 2) Activity score bumping of variables which appear in the glue clauses.

I. GUIDANCE OF CDCL BRANCHING HEURISTICS VIA RANDOM EXPLORATION DURING CONFLICT DEPRESSION

This approach is based on our observation that CDCL SAT solving entails clear non-random patterns of bursts of conflicts followed by longer phases of *conflict depression* (CD) [1]. During a CD phase a CDCL SAT solver is unable to generate conflicts for a consecutive number of decisions. To correct the course of such a search, we propose to use exploration to combat conflict depression. We therefore design a new SAT solver, called *expSAT*, which uses random walks in the context of CDCL SAT solving. In a conflict depression phase, random walks help identify more promising variables for branching. As a contrast, while exploration explores *future* search states, LRB and VSIDS relies on conflicts generated from the *past* search states.

In [1], we proposed `expVSIDS`, the exploration based extension of VSIDS. In addition to `expVSIDS`, our solvers submitted to SAT Competition-2020, use `expLRB`, the exploration based extension of LRB.

A. *expSAT* algorithm

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

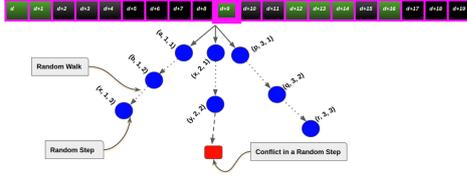
Given a CDCL SAT solver, *expSAT* modifies it as follows: (I) Before each branching decision, if a **substantially large CD phase** is detected then with probability p_{exp} , *expSAT* performs an *exploration episode*, consisting of a fixed number

nW of random walks. Each walk consists of a limited number of *random steps*. Each such step consists of (a) the uniform random selection of a currently unassigned *step variable* and assigning a boolean value to it using a standard CDCL *polarity* heuristic, and (b) a followed by Unit Propagation (UP). A walk terminates either when a conflict occurs during UP, or after a fixed number lW of random steps have been taken. Figure 1 illustrates an exploration episode amid a CD phase. (II) In an exploration episode of nW walks of maximum length lW , the *exploration score* $expScore$ of a decision variable v is the average of the *walk scores* $ws(v)$ of all those random walks within the same episode in which v was one of the randomly chosen decision variables. $ws(v)$ is computed as follows: (a) $ws(v) = 0$ if the walk ended without a conflict. (b) Otherwise, $ws(v) = \frac{\omega^d}{lbd(c)}$, with decay factor $0 < \omega \leq 1$, $lbd(c)$ the LBD score of the clause c learned for the current conflict, and $d \geq 0$ the *decision distance* between variable v and the conflict which ended the current walk: If v was assigned at some step j during the current walk, and the conflict occurred after step $j' \geq j$, then $d = j' - j$. We assign credit to all the step variables in a walk that ends with a conflict and give higher credit to variables closer to the conflict. (III) The novel branching heuristics `expLRB` (resp. `expVSIDS`) adds LRB (resp. VSDIS) score and $expScore$ of the variables that participated in the most recent exploration episode. For both `expLRB` and `expVSIDS`, a variable v^* with maximum combined score is selected for branching. (IV) All other components remain the same as in the underlying CDCL SAT solver.

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

Fig. 1: The 20 adjacent cells denote 20 consecutive decisions starting from the d^{th} decision, with $d > 0$, where a green cell denotes a decision with conflicts and a black cell denotes a decision without conflicts. Say that amid a CD phase, just before taking the $(d + 9)^{th}$ decision, *expSAT* performs an exploration episode via 3 random walks each limited to 3 steps. The second walk ends after 2 steps, due to a conflict. A triplet (v, i, j) represents that the variable v is randomly chosen at the j^{th} step of the i^{th} walk.



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 four CDCL SAT solvers to SAT Competition-2020, which are based on four combinations of the two approaches described in the previous sections. Our solvers are implemented on top of the solver MapleLCMDistChronoBT-DL-v2.2 [3], the runner up of SAT Race-2019. In the following, we describe our solvers:

a) *exp_V_L_MLD_CBT_DL*: In the baseline solver MapleLCMDistChronoBT-DL-v2.2, LRB starts execution after 50,000 conflicts and continues until 2,500 seconds (phase 1). After 2,500 seconds the solver switches over to VSIDS until the rest of the run of this solver (phase 2). This extended solver replaces LRB by $expLRB$ in phase 1 and VSIDS by $expVSIDS$ at phase 2.

b) *exp_V_MLD_CBT_DL*: The system replaces VSIDS with $expVSIDS$ at phase 2 and does not change any other aspects of the baseline system.

c) *exp_L_MLD_CBT_DL*: This system replaces LRB with $expLRB$ at phase 1 and does not change any other aspects of the baseline system.

d) *exp_V_LGB_MLD_CBT_DL*: This system extends the baseline by implementing the GB method on top of LRB and Dist heuristics, and replaces VSIDS with $expVSIDS$ at phase 2.

REFERENCES

- [1] Md Solimul Chowdhury and Martin Müller and Jia You, Guiding CDCL SAT Search via Random Exploration amid Conflict Depression, To appear in Proceedings of AAAI-2020.
- [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] Stepan Kochemazov, Oleg Zaikin, Victor Kondratiev, and Alexander Semenov. MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019, In Proceedings of SAT Race 2019:24-24.

Optsat, Abcdsat and Solvers Based on Simplified Data Structure and Hybrid Solving Strategies

Jingchao Chen

School of Informatics, Donghua University

2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China

chen-jc@dhu.edu.cn

Abstract—Based on hybrid and simplified technique, we modify MapleLCMDistChronoBT-DL and our previous SAT solvers such as abcdsat and optsat.

I. INTRODUCTION

Various decision variable branching policies have been proposed, for example, VSIDS (Variable State Independent Decay Sum), LRB (learning rate based branching heuristic) and distance branching policy etc. So far, no decision variable branching policy has absolute advantages. Here we mix various variable branching policies to modify the existing CDCL SAT solver. Different variable branching policies adopt different data structures. In order to facilitate unified data management, we need simplify the exiting data structure management technique.

II. SIMPLIFIED DATA STRUCTURE

The CoMiniSatPS [3] solver manages separately three different learnt clauses, while we do not adopt the separate management scheme. We use only one unified data list to store all learnt clauses, and set a mark flag for each learnt clause to distinguish which category a learnt clause belongs. Thus, our data structure is simpler than that of CoMiniSatPS. Except that each clause has an additional mark, our data structure storing learnt clauses is the same as that of Glucose. We can handle dynamics update of learnt clauses more easily than CoMiniSatPS. This modification seems to have a little bit impact on the performance of solvers, even if we use the same database reduction strategies as CoMiniSatPS. However, if we adopt different solving policies or database reduction strategies, it seems to gain the performance of solvers. CoMiniSatPS calls three different learnt clauses as *core*, *tier2* and *local*, respectively, we do them as *SMALL*, *MIDSZ* and *LONG*.

Our definition of different learnt clauses is a little different from that of CoMiniSatPS, but the same as that of MapleLCMDistChronoBT-DL [4]. LBD (literal block distance) is defined as the number of decision variables in a clause. According to LBD values, we define different type learnt clauses. In details, a learnt clause whose LBD value is less than 3 is defined as *SMALL*. A learnt clause whose LBD value is greater than 6 is defined as *LONG*. The other learnt clauses are called as *MIDSZ*.

III. MAPLE_MIX

Maple_mix is built on the top of MapleLCMDistChronoBT-DL [4]. It is for the main track. We found that the distance decision variable branching policy was not always efficient. Therefore, Maple_mix sets up two modes: distance and non-distance. In the distance mode, the running time is limited usually to 200 seconds. In addition, we make a slight modification on chronological backtracking [1]. The original chronological backtracking appears to be stuck in an infinite loop. our original chronological backtracking is limited to be at most 100000 conflicts one time.

IV. MAPLE_SIMP

Maple_simp simplifies the data structure of MapleLCMDistChronoBT-DL [4]. It participates the main track. All learnt clauses are stored in one unified data list. Three mark flags *SMALL*, *MIDSZ* and *LONG* are used to distinguish different types of learnt clauses. Furthermore, Maple_simp simplifies two re-learning subroutines to one re-learning subroutine. To ensure the performance, we modify the database reduction subroutine. Like Maple_mix, Maple_simp adopts also two two modes: distance and non-distance.

V. OPTSAT *m20*

This solver is submitted to the main track. The basic framework of Optsat *m20* is the same as Optsat *m19* [5]. The main difference between them is that Optsat_m20 is built on the top of Maple_simp, while Optsat *m19* is built on the top of the smallsat solver. Optsat *m20* contains the hyper binary resolution in-processing.

VI. ABCDSAT *i20*

This solver participates the incremental library track. Abcdsat *i20* simplify the original problem by lifting, un hiding, distilling, Tarjan's strongly connected components algorithm, tautology binary clause deletion and variable elimination etc. The simplified problem is solved by Maple_simp or Maple_mix.

VII. ABCDSAT *n20*

Abcdsat *n20* is submitted to the no limit track. It is similar to abcdsat *n18* [2], but removes the symmetry breaking preprocessing. Like Maple_simp, its scoring scheme alternates between Minisat-VSIDS (Variable State Independent Decay Sum) and LRB (learning rate based branching heuristic). Abcdsat *n20* replaces the tree-based search in abcdsat *n18* with the search in Maple_mix.

VIII. ABCDSAT *p20*

This solver is submitted to the parallel track. Abcdsat *p20* is the improved version of abcdsat *p18* [2]. It uses at most 25 threads. 8 out of 25 threads solve the subproblem $F \wedge p$, where p and F are a pivot and the original problem respectively. The other 17 threads solve either the original problem or the simplified problem. Once the thread of a subproblem ends, we re-use it to solve the simplified problem with learnt clauses generated so far. The main difference between threads is that they use different variable decay rates. Abcdsat *p20* is built on the top of Abcdsat *n20*.

REFERENCES

- [1] Nadel, A., Ryvchin, V.: Chronological Backtracking, SAT 2018, LNCS 10929, pp. 111–121 (2018)
- [2] Chen J.: AbcdSAT and Glucose hack: Various Simplifications and Optimizations for CDCL SAT solvers, Proceedings of SAT Competition 2018, pp.10-12
- [3] Oh C. : Between SAT and UNSAT: the fundamental difference in CDCL SAT, SAT 2015.
- [4] Kochemazov, S., Zaikin, O., Kondratiev, V., Semenov A.: MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019, Proceedings of SAT Race 2019, p.24
- [5] Chen J.: Smallsat, Optsat and MapleLCMdistCBTcoreFirst: containing core first unit propagation, Proceedings of SAT Race 2019, 31–32

CryptoMiniSat with CCAnr at the SAT Competition 2020

Mate Soos (National University of Singapore)

Shaowei Cai (State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences)

Jo Devriendt, Stephan Gocht (Lund University & University of Copenhagen)

Arijit Shaw, Kuldeep S. Meel (National University of Singapore)

I. INTRODUCTION

This paper presents the conflict-driven clause-learning (CLDL) SAT solver CryptoMiniSat (*CMS*) augmented with the Stochastic Local Search (SLS) [4] solver CCAnr as submitted to SAT Competition 2020.

CryptoMiniSat aims to be a modern, open source SAT solver using inprocessing techniques, optimized data structures and finely-tuned timeouts to have good control over both memory and time usage of inprocessing steps. CryptoMiniSat is authored by Mate Soos.

CCAnr [4] is a stochastic local search (SLS) solver for SAT, which is based on the configuration checking strategy and has good performance on non-random SAT instances. CCAnr switches between two modes: it flips a variable according to the CCA (configuration checking with aspiration) heuristic if any; otherwise, it flips a variable in a random unsatisfied clause (which we refer to as the focused local search mode). The main novelty of CCAnr lies on the greedy heuristic in the focused local search mode, which contributes significantly to its good performance on structured instances

II. COMPOSING THE TWO SOLVERS

The two solvers are composed together in a way that does *not* resemble portfolio solvers. The system runs the CDCL solver CryptoMiniSat, along with its periodic inprocessing, by default. However, at every 2nd inprocessing step, CryptoMiniSat's irredundant clauses are pushed into CCAnr (in case the predicted memory use is not too high). CCAnr is then allowed to run for a predefined number of steps. This in total leads to about 1% of all solving time dedicated to CCAnr. In case CCAnr finds a satisfying assignment, this is given back to the CDCL solver, which then performs all the necessary extension to the solution (e.g. for Bounded Variable Elimination, BVE [6]) and outputs the final solution.

In case CCAnr does not find a satisfying assignment, the following takes place. Firstly, the best variable setting found by CCAnr as measured by the number of satisfied clauses, is assigned as the polarity of the variables in the CDCL SAT solver. This idea has been taken from the solver CaDiCaL [3] as submitted to the 2019 SAT Race by Armin Biere. Secondly, after every successful execution of CCAnr, 100 variables' VSIDS are bumped in the following way.

CCAnr uses a clause weighting technique and clauses with greater weight can be considered more difficult to satisfy. Once CCAnr finishes, CCAnr's clauses are sorted according to their weights. Then, these clauses' variables' VSIDS are bumped, from hardest-to-easiest clause order, until 100 variables' VSIDS have been bumped. This shows clear improvement in the combined solver's performance. We believe these two integrations point to potential tighter, as-yet unexplored integration opportunities of the two solvers.

Note that the inclusion of the SLS solver is full in the sense that assumptions-based solving, library-based solver use, and all other uses of the SAT solver is fully supported with SLS solving enabled. Hence, this is not some form of portfolio where a simple shell script determines which solver to run and then runs that solver. Instead, the SLS solver is a full member of the solver, much like any other inprocessing system, and works in tandem with it. For example, in case an inprocessing step has reduced the number of variables through BVE or increased it through BVA [9], the SLS solver will then try to solve the problem thus modified. In case the SLS solver finds a solution, the main solver will then correctly manipulate it to fit the needs of the "outside world", i.e. the caller.

As the two solvers are well-coupled, the combination of the two solvers can solve problems that neither system can solve on its own. Hence, *the system is more than just a union of its parts* which is not the case for traditional portfolio solvers.

III. GAUSS-JORDAN ELIMINATION

As per the upcoming paper [12], the Gauss-Jordan elimination of CryptoMiniSat has been significantly improved. The average speed increase for moderately sized matrices is approx 3-6x, allowing the system to be ran at all times even when the matrix is not contributing as much to the overall solving. Hence, for the first time in CryptoMiniSat's 10 year history, Gauss-Jordan elimination is turned on by default for the NoLimits track.

IV. SYMMETRY BREAKING USING BREAKID AND BLISS

The BreakID [5] system is a cost-effective symmetry breaking preprocessor for SAT. Classic SAT symmetry preprocessing [1] detects symmetry by converting the input formula to a graph and computing generators for this graph's automorphism group, and adds symmetry breaking clauses

on a generator-by-generator basis. On top of this, BreakID heuristically searches for structure in the automorphism group, detecting *row interchangeability symmetry* (such as in the pigeonhole problem) and computing binary symmetry breaking clauses from orbits arising from the symmetry group. The resulting symmetry breaking clauses are more effective at reducing symmetrical assignments from the search space, both from a theory point of view as well as in practical experiments.

BreakID has been modified to work as a library. It can receive the clauses on-the-fly from the SAT solver, and produce the breaking clauses as a function return value. Various small bugs have also been fixed, such as memory leaks, which were not an issue when ran as a single executable, but created issues when ran as a library. Furthermore, the underlying highly sophisticated graph automorphism detection system, Bliss [7], has been slightly improved to allow for time-outs and it, too, has been fixed not to leak memory. BreakID is fully integrated into CryptoMiniSat by calling it on every 5th inprocessing iteration, and asked to contribute breaking clauses. These are always added with an assumption literal, so they can be removed when the solving finishes. Hence, symmetry breaking also works when CryptoMiniSat is used as a library.

V. PHASE SELECTION USING LSIDS

LSIDS is a literal activity-based phase selection heuristic [10]. LSIDS activity is maintained for each literal, and the activity for a literal is updated in a manner similar to VSIDS. Phase selection is made based on LSIDS activity only if the last backtrack is chronological. The LSIDS based phase selection heuristic looks at the activity of both the literals of a given variable and selects the literal with higher activity.

VI. FURTHER IMPROVEMENTS RELATIVE TO SAT RACE 2019

Many of the inprocessing parameters have been tuned. A few bugs related to clause activities have been fixed. Clause distillation (or clause vivification) [8] is now used a lot more, similarly to the previous years' winning solvers. The VSIDS and Maple decay factors are now iteratively changed between 0.70 and 0.90 for Maple and 0.92 and 0.99 for VSIDS. Between each iteration there is an inprocessing step, as before. This seems to add heterogeneity and avoids having to tune these parameters to a "single best" value. Polarity caching is still used, but once in a while, so-called "stable" polarities are used, as per CaDiCaL [3] in the SAT Race of 2019. Ternary resolution is also used at every inprocessing step, thanks to the suggestion by Armin Biere.

VII. THANKS

Kuldeep S Meel, Arijit Shaw, and Mate Soos were supported in part by the National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13].

Shaowei Cai was supported by Beijing Academy of Artificial Intelligence (BAAI), and Youth Innovation Pro-motion Association, Chinese Academy of Sciences [No. 2017150].

Stephan Gocht was funded by the Swedish Research Council (VR) grant 2016-00782.

The computational work for this article was performed on resources of the National Supercomputing Center, Singapore[2]. Mate Soos would also like to thank all the users of CryptoMiniSat who have submitted over 600 issues and pull requests to the GitHub CMS repository[11].

REFERENCES

- [1] Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for Boolean satisfiability. *IEEE Trans. Comput.* 55(5), 549–558 (May 2006), <https://doi.org/10.1109/TC.2006.75>
- [2] ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), <https://www.nscg.sg/about-nscg/overview/>
- [3] Biere, A.: CaDiCaL SAT solver GitHub page (2020), <https://github.com/arminbiere/cadical>
- [4] Cai, S., Luo, C., Su, K.: Ccanr: A configuration checking based local search solver for non-random satisfiability. In: Heule, M., Weaver, S.A. (eds.) *SAT 2015*. LNCS, vol. 9340. Springer (2015)
- [5] Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for SAT. In: Creignou, N., Le Berre, D. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2016*. pp. 104–122. Springer International Publishing, Cham (2016), <https://bitbucket.org/krr/breakid>
- [6] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) *Theory and Applications of Satisfiability Testing*. pp. 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
- [7] Junttila, T.A., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: *ALENEX 2007*. SIAM (2007)
- [8] Li, C., Xiao, F., Luo, M., Manyà, F., Lü, Z., Li, Y.: Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.* 279 (2020)
- [9] Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Biere, A., Nahir, A., Vos, T. (eds.) *Hardware and Software: Verification and Testing*. pp. 102–117. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
- [10] Shaw, A., Meel, K.S.: Designing new phase selection heuristics. In: *SAT 2020* (2020)
- [11] Soos, M.: CryptoMiniSat SAT solver GitHub page (2018), <https://github.com/msoos/cryptominisat>
- [12] Soos, M., Gocht, S., Meel, K.S.: Accelerating approximate techniques for counting and sampling models through refined CNF-XOR solving. In: *CAV 2020* (2020)

CryptoMiniSat with WalkSAT at the SAT Competition 2020

Mate Soos (National University of Singapore)

Bart Selman (Cornell University), Henry Kautz (University of Rochester)

Jo Devriendt, Stephan Goetz (Lund University & University of Copenhagen)

I. INTRODUCTION

This paper presents the conflict-driven clause-learning (CLDL) SAT solver CryptoMiniSat (*CMS*) augmented with the Stochastic Local Search (SLS) [11] solver WalkSAT v56 as submitted to SAT Competition 2020.

CryptoMiniSat aims to be a modern, open source SAT solver using inprocessing techniques, optimized data structures and finely-tuned timeouts to have good control over both memory and time usage of inprocessing steps. CryptoMiniSat is authored by Mate Soos.

WalkSAT [8] is a standard system to solve satisfiability problems using Stochastic Local Search. The version inside CryptoMiniSat is functionally equivalent to the “rnovelity” heuristic of WalkSAT v56 using an adaptive noise heuristic [6]. It behaves exactly as WalkSAT with the minor modification of performing early-abort in case the “low-bad” statistic (i.e. the quality indicator of the current best solution) indicates the solution is far. In these cases, we early abort, let the CDCL solver work longer to simplify the problem, and come back to WalkSAT later. The only major modification to WalkSAT has been to allow it to import variables and clauses directly from the main solver taking into account assumptions given by the user.

II. COMPOSING THE TWO SOLVERS

The two solvers are composed together in a way that does *not* resemble portfolio solvers. The system runs the CDCL solver CryptoMiniSat, along with its periodic inprocessing, by default. However, at every 2nd inprocessing step, CryptoMiniSat’s irredundant clauses are pushed into the SLS solver (in case the predicted memory use is not too high). The SLS solver is then allowed to run for a predefined number of steps. In case the SLS solver finds a solution, this is given back to the CDCL solver, which then performs all the necessary extension to the solution (e.g. for Bounded Variable Elimination, BVE [5]) and then outputs the solution.

Note that the inclusion of the SLS solver is full in the sense that assumptions-based solving, library-based solver use, and all other uses of the SAT solver is fully supported with SLS solving enabled. Hence, this is not some form of portfolio where a simple shell script determines which solver to run and then runs that solver. Instead, the SLS solver is a full member of the CDCL solver, much like any other inprocessing system, and works in tandem with it. For example, in case an inprocessing step has reduced the number of variables through BVE or increased it through

BVA [10], the SLS solver will then try to solve the problem thus modified. In case the SLS solver finds a solution, the main solver will then correctly manipulate it to fit the needs of the “outside world”, i.e. the caller.

As the two solvers are well-coupled, the combination of the two solvers can solve problems that neither system can solve on its own. Hence, *the system is more than just a union of its parts* which is not the case for traditional portfolio solvers.

III. GAUSS-JORDAN ELIMINATION

As per the upcoming paper [13], the Gauss-Jordan elimination of CryptoMiniSat has been significantly improved. The average speed increase for moderately sized matrices is approx 3-6x, allowing the system to be ran at all times even when the matrix is not contributing as much to the overall solving. Hence, for the first time in CryptoMiniSat’s 10 year history, Gauss-Jordan elimination is turned on by default for the NoLimits track.

IV. SYMMETRY BREAKING USING BREAKID AND BLISS

The BreakID [4] system is a cost-effective symmetry breaking preprocessor for SAT. Classic SAT symmetry preprocessing [1] detects symmetry by converting the input formula to a graph and computing generators for this graph’s automorphism group, and adds symmetry breaking clauses on a generator-by-generator basis. On top of this, BreakID heuristically searches for structure in the automorphism group, detecting *row interchangeability symmetry* (such as in the pigeonhole problem) and computing binary symmetry breaking clauses from orbits arising from the symmetry group. The resulting symmetry breaking clauses are more effective at reducing symmetrical assignments from the search space, both from a theory point of view as well as in practical experiments.

BreakID has been modified to work as a library. It can receive the clauses on-the-fly from the SAT solver, and produce the breaking clauses as a function return value. Various small bugs have also been fixed, such as memory leaks, which were not an issue when ran as a single executable, but created issues when ran as a library. Furthermore, the underlying highly sophisticated graph automorphism detection system, Bliss [7], has been slightly improved to allow for time-outs and it, too, has been fixed not to leak memory. BreakID is fully integrated into CryptoMiniSat by calling it on every 5th inprocessing iteration, and asked to contribute breaking clauses. These are always added with an assumption literal, so they can be removed when

the solving finishes. Hence, symmetry breaking also works when CryptoMiniSat is used as a library.

V. FURTHER IMPROVEMENTS RELATIVE TO SAT RACE 2019

Many of the inprocessing parameters have been tuned. A few bugs related to clause activities have been fixed. Clause distillation (or clause vivification) [9] is now used a lot more, similarly to the previous years' winning solvers. The VSIDS and Maple decay factors are now iteratively changed between 0.70 and 0.90 for Maple and 0.92 and 0.99 for VSIDS. Between each iteration there is an inprocessing step, as before. This seems to add heterogeneity and avoids having to tune these parameters to a "single best" value. Polarity caching is still used, but once in a while, so-called "stable" polarities are used, as per CaDiCaL [3] in the SAT Race of 2019. Ternary resolution is also used at every inprocessing step, thanks to the suggestion by Armin Biere.

VI. THANKS

Mate Soos was supported in part by the National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13].

Stephan Gocht was funded by the Swedish Research Council (VR) grant 2016-00782.

The computational work for this article was performed on resources of the National Supercomputing Center, Singapore[2]. Mate Soos would also like to thank all the users of CryptoMiniSat who have submitted over 600 issues and pull requests to the GitHub CMS repository[12].

REFERENCES

- [1] Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for Boolean satisfiability. *IEEE Trans. Comput.* 55(5), 549–558 (May 2006), <https://doi.org/10.1109/TC.2006.75>
- [2] ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), <https://www.nscg.sg/about-nscg/overview/>
- [3] Biere, A.: CaDiCaL SAT solver GitHub page (2020), <https://github.com/arminbiere/cadical>
- [4] Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for SAT. In: Creignou, N., Le Berre, D. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2016*. pp. 104–122. Springer International Publishing, Cham (2016), <https://bitbucket.org/krr/breakid>
- [5] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) *Theory and Applications of Satisfiability Testing*. pp. 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
- [6] Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Dechter, R., Kearns, M.J., Sutton, R.S. (eds.) *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*, July 28 - August 1, 2002, Edmonton, Alberta, Canada. pp. 655–660. AAAI Press / The MIT Press (2002), <http://www.aaai.org/Library/AAAI/2002/aaai02-098.php>
- [7] Junttila, T.A., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: *ALENEX 2007*. SIAM (2007)
- [8] Kautz, H.A., Selman, B.: Pushing the envelope: Planning, propositional logic and stochastic search. In: Clancey, W.J., Weld, D.S. (eds.) *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference*, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2. pp. 1194–1201. AAAI Press / The MIT Press (1996), <http://www.aaai.org/Library/AAAI/1996/aaai96-177.php>
- [9] Li, C., Xiao, F., Luo, M., Manyà, F., Lü, Z., Li, Y.: Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.* 279 (2020)
- [10] Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Biere, A., Nahir, A., Vos, T. (eds.) *Hardware and Software: Verification and Testing*. pp. 102–117. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
- [11] Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. pp. 521–532 (1995)
- [12] Soos, M.: CryptoMiniSat SAT solver GitHub page (2018), <https://github.com/msoos/cryptominisat>
- [13] Soos, M., Gocht, S., Meel, K.S.: Accelerating approximate techniques for counting and sampling models through refined CNF-XOR solving. In: *CAV 2020* (2020)

Glucose 3.0 – With Inprocessing

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

Abstract—The presented GLUCOSE 3.0 hack adds formula simplification during search, namely subsumption and self-subsuming resolution. Instead of full integration, this hack initiates required data structures. As learned clauses cannot be super-sets of existing clauses, each clause is considered for simplification exactly once.

I. INTRODUCTION

Since the 2009 SAT competition, some SAT solvers [2] have run formula simplification as part of their search. More recently, learned clause minimization has been extended to consider binary clauses [1], as well as vivification for learned clauses has been driving success [6], [8]. With vivification, learned clauses are reduced during search, based on other clauses. While general conflict driven clause learning does not learn the same clause twice – except a learned clause has been removed – vivification allows to introduce duplicates. The winning solver [9] of 2019 countered this effect by blocking the introduction of duplicate clauses via a hash-map.

With this hack, we take this approach a step further: instead of checking for duplicates, we check for subsumption as well as self-subsuming resolution. The subsumption check is more powerful than the duplicate check. Self-subsuming resolution allows to shrink learned clauses even further.

II. INTEGRATED TECHNIQUES AND FIXES

Analysis on simplifications in [7] showed that plain subsumption and self-subsuming resolution are as powerful as bounded variable elimination [4]. Therefore, this implementation focusses on implementing these two techniques. A subsumption check is implemented in a linear fashion using a linear hash table for fast accesses. The self-subsuming resolution check is implemented in the same loop, essentially checking whether all except one literals of the simplifying clause match, and the remaining literal matches as the complement. Each learned clause is considered as a simplification candidate at most once.

Simplification is scheduled after the clause database has been cleaned significantly, i.e. after at least 30 % of the clauses have been dropped. Furthermore, simplifications are scheduled with an exponential back-off, i.e. after 1, 2, 4, 8, ... attempts. Additionally, the current limit is increased by 2, in case no simplification was possible. The schedule is implemented in this way, because it is simple, and because the operation itself comes with an upfront cost. There might be more effective ways to implement this simplification, but given the space limitations, the proposed one was found to perform better than the unmodified solver.

A. Preliminary Testing

The modification has been tested on the evaluation benchmark for tuning solvers [5] with a timeout of 900 seconds. 190 instead of 185 formulas could be solved within the timeout.

III. AVAILABILITY

The source of the solver is publicly available under the MIT license at <https://github.com/comp-solutions/glucose3.0-hack-track>. The version in the branch “reloc-subsume” has been submitted to the 2020 competition. While the diff to the original system is below 1000 characters, as required, there is a more readable version available as well, in the same branch. There is an explicit commit (“443ff88: glucose-hack-track: get diff below 1000”) that reduces the diff. Mechanics to do that are to reuse variable, define a macro for printing clauses to drup proofs, and defining constants for common parts of the code, like “size()” statements. This commit highlights that the requirement to stay below a predefined distance in characters as a metric has flaws, and other measures, i.e. number of different program statements might be better suited.¹

REFERENCES

- [1] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern sat solvers,” in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, ser. IJCAI’09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, p. 399–404.
- [2] A. Biere, “Precosat system description,” <http://fmv.jku.at/precosat/precosat-sc09.pdf>, 2009.
- [3] 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.
- [4] N. Eén and A. Biere, “Effective preprocessing in SAT through variable and clause elimination,” F. Bacchus and T. Walsh, Eds., vol. 3569. Heidelberg: Springer, 2005, pp. 61–75.
- [5] H. H. Hoos, B. Kaufmann, T. Schaub, and M. Schneider, “Robust benchmark set selection for boolean constraint solvers,” in *LION*, ser. Lecture Notes in Computer Science, vol. 7997. Springer, 2013, pp. 138–152.
- [6] L. Kahlert, F. Krüger, N. Manthey, and A. Stephan, “Riss solver framework v5.05,” 2015.
- [7] N. Manthey, “Towards next generation sequential and parallel SAT solvers,” Ph.D. dissertation, TU Dresden, 2014.
- [8] F. X. F. M. Z. L. Mao Luo, Chu-Min Li, “An effective learnt clause minimization approach for cdcl sat solvers,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 703–711. [Online]. Available: <https://doi.org/10.24963/ijcai.2017/98>
- [9] V. K. Stepan Kochemazov, Oleg Zaikin and A. Semenov, “Maplelcmdistchronobt-dl, duplicate learnts heuristic-aided solvers at the sat race 2019,” 2019.

¹This could be based for example on the ELOC count CPROVER tools [3] can generate.

CTSat: C++ Template Abstractions for Dynamic Interchangeable SAT Solver Features

Marc Hartung
Zuse Institute Berlin
hartung@zib.de

Abstract—To provide a broad variety of solver heuristics while maintaining clean code and an efficient implementation, CTSat uses C++ templates to make important components of a CDCL solver interchangeable. Different configurations are compile-time defined to ensure a low overhead during runtime, but can also be dynamically chosen during runtime. Besides our compile-time approach, we use a multi-clause learning scheme and an extended in-conflict minimisation to improve solver behaviour during the search.

Index Terms—sat solver, software design, parallel computing, distributed computing

I. INTRODUCTION

The number of different approaches to SAT solving increases steadily with every new insight gained about the problem. Thereby, basic concepts, e.g. CDCL or SLS, and more often the used heuristics to solve a SAT instance in a specific way are diversified. Since there is no single dominant approach to solve every SAT instances fast, portfolio approaches in sequential SAT solvers become the state of the art. Thus, modern implementations tend to have a lot of branches to implement different heuristics in different solving phases. This leads to dead code regions for most of the solving process and unnecessarily evaluated branches. But in the first place, this limits the number of provided heuristics by a specific implementation, due to the increasing complexity of the code. To remedy this limitation, we started to abstract basic CDCL solver components using C++ templates in our solver CTSat that allows us to provide a wide variety of features without the need to evaluate complex branching structures or to perform costly dynamic type checks during runtime.

Our goal is not to create a sequential solver with a high rate of switches between heuristics during runtime. Our main targets are to be able to use a high number of different heuristics in a highly parallel SAT solver and to change the memory layout dynamically to be more efficient on certain SAT instances without many redundant code parts. The solver is based on MapleLCMDistChronoBT [1] and provides, besides the existing features, a multi-conflict-clause-learning scheme, we call Level Aware Conflict Analyzes (LAA), an extended in-conflict clause minimisation and a conflict based clause import inspired by tier-based clause reduction [2] in the parallel versions.

II. ABSTRACTED COMPONENTS

We split the base solver into different standalone components. The main components are the clause database, the

clause exchange in parallel solvers, the propagation scheme, the conflict analysis, and also the branching, reduce and restart heuristics. The submitted version provides only the most common heuristics, since our current focus is to define requirements for each component, so that the interfaces are flexible enough to support most of the current approaches and as far as possible future ones.

III. LEVEL-AWARE CONFLICT ANALYSIS

Based on the idea used in GRASP [3] and others [4], CTSat learns multiple clauses during conflict analysis. More precisely, we create additionally to the first unique implication point (UIP) cut clauses from a subset of the resolvents of the first UIP clause (length minimising cut) and from the resolvents between UIPs (multi UIP learning). To reduce the added overhead, CTSat only adds multiple clauses when it is currently below the average decision level.

IV. EXTENDED IN-CONFLICT CLAUSE MINIMISE

Both, state-of-the-art minimisations, i.e. redundancy check based on self-subsumption [5], and binary clause minimisation, use additional effort on short clauses and on clauses with low LBD. CTSat executes redundancy checks on not yet seen decision levels, which was originally prevented heuristically. This makes it possible to reduce the clause further, but possibly increases the LBD value of the clause. The latter minimisation is by default only executed using the first UIP literal. There, CTSat performs a binary clause implication check on every literal. These are minor changes, but have a tremendous impact in certain cases, while creating a minimal overhead.

V. CONFLICT- AND TIER-BASED CLAUSE EXCHANGE

In the Parallel and Cloud Track, CTSat uses a conflict-based clause import. Thereby, each imported clause will be deleted when it did not participate in any of 15,000 conflicts. When a clause was seen in a conflict, it is added to the learned clause database and treated as a normal learned clause. Exports are based on the lazy export policy [6].

VI. SEQUENTIAL TRACKS

The sequential build of CTSat uses the heuristics of MapleLCMDistChronoBT with the difference that LAA is used, wherefore additional clauses are learned when the current conflict level is 20 levels below the average of the last 20 levels. These are barely tested parameters, but we are eager to see how they will impact the behaviour.

VII. PARALLEL TRACK

For the parallel track, we use combinations of well-known heuristics (LRB, VSIDS, DIST; Tier-based reduction, Glucose reduction; Luby restart, Glucose restart). Also, LAA is configured to always check for additional learned clauses that are also asserting and smaller than the first UIP clause. In this case, the clauses will be swapped, and only the shortest is learned.

VIII. CLOUD TRACK

Since it is unclear how many threads one should use on different benchmarks, CTSat uses three different thread configurations with a different frequency for the MPI nodes. In six of ten nodes, CTSat uses half of the cores, in three of ten nodes every core, and in one of ten cases one core per NUMA-node is used. Each node is configured similar to the Parallel Track configuration. Imported clauses are filtered using hashes to identify duplicates. When more clauses are learned than space is available in the communication buffer during a communication period, only the clauses with the lowest LBD and size are exported.

ACKNOWLEDGMENT

We want to express our gratitude towards the organisers of the SAT Competition 2020 for making such an event possible, helping us setting up our solvers and expanding this years competition tracks. Additionally we like to thank Florian Schintke for his support, the IT and Data Services members of the Zuse Institute Berlin for providing the infrastructure.

REFERENCES

- [1] Vadim Ryvchin and Alexander Nadel, “Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking”, Proceedings of SAT Competition 2018, 2018
- [2] Oh Chanseok, “Between SAT and UNSAT: The Fundamental Difference in CDCL SAT”, SAT’15, 2015
- [3] J. P. Marques-Silva and K.A. Sakallah, “GRASP: A New Search Algorithm for Satisfiability”, ICCAD, 1996
- [4] Alexander Nadel and Vadim Ryvchin, “Flipped Recording”, POS’19, 2019
- [5] Niklas Een and Niklas Srensson, “MiniSat v1.13 A SAT Solver with Conflict-ClauseMinimization”, SAT, 2005
- [6] Gilles Audemard and Laurent Simon, “Lazy clause exchange policy for parallel SAT solvers”, SAT’14, 2014

PauSat

Benjamin Kaiser and Marc Hartung

Zuse Institute Berlin

Berlin, Germany

{kaiser,hartung}@zib.de

Abstract—This document describes the experimental SAT Solver PauSat, which implements an approach combining CDCL solving with SLS solver characteristics by introducing an additional assignment procedure.

Index Terms—SAT, SLS, CDCL, Hybrid

I. INTRODUCTION

The search routine of a typical *Conflict-Driven-Clause-Learning* (CDCL) solver starts in a state in which the variables of a subset of the variables used in the problem are assigned in such a way that there exists no clause in which all occurring literals are assigned false, i.e. a CDCL solver is at this moment always in a state of a *non-conflicting partial assignment*. A CDCL Solver then assigns one further still unassigned variable a truth-value and makes use of *Boolean Constraint Propagation* (BCP) to find out whether this last decision leads to a conflicting partial assignment and to infer further assignments. In the first case *conflict analysis* is invoked in order to learn an additional clause from the conflict, which is added to the problem, and conflicting variables are unassigned such that the partial assignment of the variables are non-conflicting. This search routine is repeated until either a full non-conflicting assignment is reached or the empty clause is learned during conflict-analysis, i.e. either a model of the problem is found or the problem is proven to be unsatisfiable. The decision which variable to assign next is based on an *activity* value assigned to each variable, which is updated during conflict analysis.

A typical *Stochastic Local Search* Solver on the other hand is at any given moment in a state in which all variables are assigned. If this full assignment is conflicting, i.e. if there exists a clause in which all occurring variables are assigned false, a variable is chosen whose assignment is changed from truth to false or vice versa. We say the truth value of this assignment is *flipped*. In order to choose a variable, first a clause is chosen in which all occurring literals are assigned false under the current assignment. In the following we will call such a clause a *conflicting clause*¹. Afterwards, the variable whose assignment is to be flipped is chosen randomly among the variables occurring in this conflicting clause using some probability distribution. This process is repeated until a non-conflicting full assignment is reached, in which case a model of the problem is found. Unlike a CDCL Solver, a typical SLS Solver cannot prove unsatisfiability of the problem.

¹not to be confused with the so-called *conflict clauses* learned during conflict analysis.

Among other techniques, both solver types, CDCL and SLS, make use of regularly scheduled restarts and furthermore a CDCL solver is forced to regularly delete some of the learned clauses due to the fact that too many (possibly useless) clauses in the problem highly affect the speed of BCP (and due to memory issues).

PauSat is a hybrid solver that tries to combine both Solver types using a novel approach of differentiating between two kinds of variable assignments. PauSat is an implementation of this approach based on the CDCL-Solver *Maple_LCM_Dist* [1].

II. PAUSAT'S HYBRID SEARCH

A. Soft Assignments, Hard Assignments and Dependency Lists

In the following, assignments used by BCP and conflict analysis in a standard CDCL solver are called *hard* assignments. Assignments done by our novel approach are called *soft*. With respect to all hard assignments PauSat is a typical CDCL Solver. However, at any given moment any variable which is not hard assigned, is furthermore assigned an additional soft truth-value. A variable is never both, soft and hard assigned concurrently.

So at the beginning of the CDCL search routine of PauSat the state of the Solver is determined by a potentially conflicting full assignment of the variables, which consist of a non-conflicting partial hard assignment used by the CDCL search routine and a soft assignment of all the variables which are not assigned hard. Note that at the beginning of the CDCL search routine no clause is conflicting with respect to all hard assignments. Thus, at this moment every conflicting clause contains at least one soft assigned variable.

To each variable a list of references of clauses is assigned. In the following this list will be called *dependency list* of this variable. Whenever in the search routine of PauSat a clause is found that is satisfied with respect to the current full assignment (both, soft and hard), one of the true literals in the clause is chosen and a reference of this clause is stored in the dependency list of the corresponding variable. This way it is remembered that this clause remains non-conflicting until the assignment of the chosen variable is flipped.

B. Initialization of the search

After preprocessing the problem at the very beginning of the search, for every variable a soft assignment is chosen randomly. PauSat also makes use of restarts, and after every restart every variable is soft assigned according to the last hard

assigned truth-value. Hence, the initial state of the search of PauSat is determined by a full assignment consisting of soft assignments only.

With respect to this initial full assignment every clause in the problem is either satisfied or conflicting, i.e. in every clause in the problem there either exists a literal in the clause which is (soft) assigned true or all literals in the clause are (soft) assigned false. For every non-conflicting clause, a reference to this clause is stored in the dependency list of the variable corresponding to one true (soft) assigned literal occurring in the clause.

All remaining clauses are sorted with respect to the maximum of the activities of the variables in the clause and saved in a stack with the clauses containing the variable with highest activity on top. At any given moment all clauses in the original problem that are conflicting are contained in this stack. However, this stack may contain satisfied clauses. In the following this stack is called *conflicting clause stack* (CCS).

C. Alteration of the CDCL Branching Heuristic

The top element of the CCS is removed. If this clause happens to be satisfied, it is added to the list of depending clauses of the variable corresponding to some true literal in the clause. Otherwise, among all soft assigned literals in the clause the one which has highest activity is chosen to be the next decision variable. All clauses in the dependency list of the corresponding variable are removed from this list and copied onto the top of the CSS. The assignment of this literal is changed to be hard and flipped. Now, the clause which was on top of the stack of possibly unsatisfied clauses is satisfied, and thus a reference to this clause is added to the dependency of the new decision variable.

D. BCP and Conflict Analysis

After choosing a new decision variable BCP is invoked in order to check whether the new hard assignment leads to a conflicting partial assignment with respect to all hard assignments and in order to infer further hard assignments. In this process soft assignments are possibly overwritten by hard assignments. Whenever a soft assignment of a variable is overwritten by a hard assignment and simultaneously the truth-value of this variable changes, all clauses in the dependency list of this variable are removed from this list and copied onto the top of the CCS.

If BCP finds a conflicting clause with respect to all hard assignments, the conflict is analyzed as in any CDCL solver. However, in order to restore a non-conflicting partial assignment with respect to all hard assignments, variables are not unassigned as in a typical CDCL solver, but rather the hard assignment is turned into a soft assignment and preserved.

ACKNOWLEDGMENT

We want to express our gratitude towards the organizers of the SAT Competition 2020 for making such an event possible. Additionally we like to thank Florian Schintke for his support and the IT and Data Services members of the Zuse Institute Berlin for providing the infrastructure and their fast help. Also we like to thank the authors of Maple_LCM_Dist and everyone else contributing to this solver.

REFERENCES

- [1] F. Xiao, M. Luo, C.-M. Li, F. Manyá, and Z. Lü, “MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width in SAT Competition 2017,” in *Proceedings of SAT Competition, 2017* p.22

MapleCOMSPS_LRB_VSIDS for SAT Competition 2020

Jia Hui Liang^{*†}, Chanseok Oh^{†‡}, Saeed Nejadi^{*†}, Pascal Poupart^{*}, Vijay Ganesh^{*}

^{*} University of Waterloo, Waterloo, Canada

[†] Google, New York, United States

[‡] Joint first authors

Abstract—This document describes the SAT solvers MapleCOMSPS_LRB_VSIDS that implements a machine learning branching heuristics called the *learning rate branching heuristic* (LRB) and a novel machine learning based initialization method called Bayesian Moment Matching (BMM) based initialization of variable activity and polarity.

I. INTRODUCTION

Over the last several years, the *learning rate branching heuristic* (LRB), which we introduced at SAT 2016 [1], has confirmed its place alongside VSIDS as one of the leading branching heuristics for industrial-strength SAT solvers. The LRB heuristic is designed to branch on variables that maximize the number of (high-quality) learnt clauses in a given unit of time that a solver learns during its run. To be more precise, branching heuristics can be viewed as methods that solve a bi-objective problem of selecting those branching variables that simultaneously maximize both the quantity and quality of the learnt clauses generated by a solver during its run. To simplify the optimization problem, we assume that the first-UIP clause learning scheme generates high quality learnt clauses, and thus we reduced the two objectives down to just one, that is, we attempt to maximize the quantity of learnt clauses.

Motivation for Machine Learning-based Solver Heuristics:

While a Boolean SAT solver is a decision procedure that decides whether an input formula is satisfiable, internally it can be seen as an optimization procedure whose goal is to minimize its runtime while correctly deciding the satisfiability of the input formula. Every sub-routine in a SAT solver can be viewed either as a logical reasoning engine (i.e., a proof system such as resolution in the case of conflict clause learning scheme or unit resolution in the case of BCP), or as a heuristic aimed at optimizing the sequencing, selection, and initialization of proof rules (e.g., variable selection, polarity selection, restarts, etc.). These optimization heuristic can in turn be implemented effectively using machine learning algorithms, since solvers are a data-rich environment. This philosophy, that we first articulated in our SAT 2016 paper [1] on the LRB branching heuristic, has since been widely adopted and underpins many solver heuristics for branching, restarts, and initialization developed in recent years.

II. LEARNING RATE BRANCHING

Before we can describe the LRB branching heuristic, we start by defining a concept called *learning rate* of a variable

that measures the quantity of learnt clauses generated by a variable of an input formula I during the run of the solver on I . The learning rate is defined as the following conditional probability (see our SAT 2016 paper for a detailed description [1]):

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

Ideally, if the learning rate of every variable was known a priori, then we claim that the a very effective branching policy is to branch on the variable with the highest learning rate in order to maximize the number of learnt clauses generated by the solver per unit time. Unfortunately, the learning rate, as defined above, is too difficult and too expensive to compute at each *branching point* (the point in time at which the solver selects a new variable to branch on). Hence, we cheaply estimate the learning rate using multi-armed bandits (MAB), a class of state-less reinforcement learning algorithms.

Briefly, the MAB-based abstraction of the branching problem can be described as follows: Conceptually, we first record the number of learnt clauses each variable participates in generating, under the condition that the variable is assigned and the solver is in conflict, since the beginning of the solver run. These observations are averaged using an exponential moving average (EMA) to estimate the current learning rate of each variable. The effect of using an EMA is that observations made in the “distant past” (with respect to a branching point) contribute very little to the average, while those in the “near past” contribute much more to the average.

This EMA-based method is implemented using the well-known *exponential recency weighted average algorithm* (ERWA) for multi-armed bandits [2] with learning rate as the reward. The variables are then ranked according to their ERWA score and the highest unassigned variable in this ranking is branched upon when the solver reaches a *branching point*.

Lastly, we extended the algorithm with two new ideas. The first extension is to encourage branching on variables that occur frequently on the reason side of the conflict analysis and adjacent to the learnt clause during conflict analysis. The second extension is to encourage locality of the branching heuristic [3] by decaying unplayed arms, similar to the decay reinforcement model [4], [5]. We call the final branching

heuristic with these two extensions the *learning rate branching* or LRB heuristic.

III. INITIALIZATION PROBLEM

Many modern branching heuristics in CDCL SAT solvers assume that all variables have the same *initial* activity score (typically 0) at the beginning of the run of a solver. However, it is well known that a solver’s runtime can be greatly improved if the *initial* order and value assignment of variables is not fixed a priori but chosen via appropriate static analysis of the formula. By the term initial variable order (resp., initial value assignment), we refer to the order (resp. value assignment) at the start of the run of a solver. This problem of determining the optimal initial variable order and value assignment is often referred to as the initialization problem.

In this paper, we propose a solution to the initialization problem based on a Bayesian moment matching (BMM) formulation of solving SAT instances and a concomitant method we refer to as BMM-based initialization. Our method is used as a pre-processing step before the solver starts its search (i.e., before it makes its first decision).

A. Bayesian Moment Matching (BMM)

The SAT problem, simply stated, is to determine whether a given Boolean formula is satisfiable. In order to reformulate the SAT problem in a Bayesian setting, we start by defining a random variable for each variable of the input formula, where $P(x = T)$ shows the probability of setting x to True in a satisfying assignment, assuming the formula is satisfiable. We assume that each of these variables has a Beta distribution, and collectively they form our prior distribution. We have the constraint that all of the clauses must be satisfied (i.e., it is assumed that the formula is satisfiable), therefore the clauses can be seen as evidence as to how the probability distribution should look like such that they are all satisfied. We then apply Bayesian inference using each clause as evidence to arrive at a posterior distribution. Applying Bayesian inference, gives us a mixture model, and this makes the learning intractable as the number of components grows exponentially with the number of clauses. To avoid this blow up, we use the method of moments to approximate the mixture Beta distribution with a single Beta distribution.

B. BMM as a Component

We implement an approximate version of the BMM method described above, since the complete method does not scale as the size of the input formulas increase. Fortunately, this approximate method is efficient and arrives at a promising point, as it attempts to satisfy as many clauses as possible. We take this starting point and initialize the preferred polarity and activity scores of each variable of an input formula, and then let the solver complete its search. The derived posterior distribution collectively represents a probabilistic assignment to the variables that satisfies most of the clauses. For polarity initialization, we used: $Polarity[x] = False$ if $P(x = T) < 0.5$ and True otherwise. For activity initialization, we gave

higher priority to variables based on the confidence that BMM has about their values, i.e., $Activity[x] = \max(P(x = T), 1 - P(x = T))$. We initialized both VSIDS and LRB scores with the aforementioned methods. (As the reader may have already guessed, the proposed BMM method works much better for satisfiable formulas than unsatisfiable ones.)

IV. SOLVERS

All solvers in this submission are modifications of Maple-COMSPS_LRB_VSIDS [6] that participated in SAT competition 2018 (which was a modification of COMiniSatPS [7] itself). The main two modifications in these solvers are: 1) activity/polarity initialization, described in Section III and 2) implementation of learnt clause minimization (LCM) [8].

V. AVAILABILITY AND LICENSE

The source code of all versions of our solver have been made freely available under the MIT license. All the solvers use the same license as COMiniSatPS. Note that the license of the M4RI library (which COMiniSatPS uses to implement Gaussian elimination) is GPLv2+.

ACKNOWLEDGMENT

We thank the authors of Glucose, GlueMiniSat, Lingeling, CryptoMiniSat, and MiniSAT for making their solvers available to us and answering many of our questions over the years.

REFERENCES

- [1] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Learning Rate Based Branching Heuristic for SAT Solvers,” in *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT’16, 2016.
- [2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.
- [3] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, “Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers,” in *Hardware and Software: Verification and Testing*. Springer, 2015, pp. 225–241.
- [4] I. Erev and A. E. Roth, “Predicting How People Play Games: Reinforcement Learning in Experimental Games with Unique, Mixed Strategy Equilibria,” *American Economic Review*, vol. 88, no. 4, pp. 848–881, 1998.
- [5] E. Yechiam and J. R. Busemeyer, “Comparison of basic assumptions embedded in learning models for experience-based decision making,” *Psychonomic Bulletin & Review*, vol. 12, no. 3, pp. 387–402.
- [6] J. H. Liang, C. Oh, K. Czarnecki, P. Poupart, and V. Ganesh, “Maple-comsps_lrb_vsids and maplecomsps_chb_vsids in the 2018 competition,” *Proc. of SAT Competition*, pp. 31–32, 2018.
- [7] C. Oh, “Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL,” Ph.D. dissertation, New York University, 2016.
- [8] M. Luo, C.-M. Li, F. Xiao, F. Manyá, and Z. Lü, “An effective learnt clause minimization approach for cdcl sat solvers,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 2017, pp. 703–711.

MaplePainless-DC Parallel SAT Solver for SAT Competition 2020

Saeed Nejati
University of Waterloo
Waterloo, Canada
snejati@uwaterloo.ca

Vijay Ganesh
University of Waterloo
Waterloo, Canada
vijay.ganesh@uwaterloo.ca

Abstract—This document describes the divide-and-conquer parallel SAT solver, MaplePainless-DC, that uses Painless parallel SAT framework and implements a machine learning base splitting heuristic.

I. INTRODUCTION

MaplePainless-DC is a divide-and-conquer parallel SAT solver, which is built on top of Painless parallel SAT framework [1]. The primary change we made to Painless is a new machine learning (ML) based splitting heuristic. More precisely, we modified the backend worker solvers to implement an ML-based splitting heuristic. Briefly, the backend worker solvers compute search statistical and structural features of variables of an input formula, query an offline-trained ML ranking model M , and return the highest ranked variable according to this ML model. The model M is trained such that splitting on the top variable predicted by it, ideally minimizes the runtime of solving the sub-formulas thus generated compared against splitting on any other variable in the input formula.

II. DESCRIPTION OF PAINLESS-DC

Painless-DC [2] uses a master-slave architecture. The master is responsible for maintaining the splitting tree, and the slaves are sequential backend worker solvers (potentially they can be parallel solvers as well, but in this setting, they are all sequential workers) that solve the sub-formulas generated by splitting. The backend worker solvers are also responsible for computing statistical and structural features of variables of the input formula that are then used by our ML-based splitting heuristic.

A. Master Solver and Load Balancing

The master node maintains a queue of idle workers to assign jobs to. Initially, the master node chooses a variable to split on and assigns the resultant sub-formulas to two workers. If the queue of idle workers is non-empty, the master node chooses a sub-formula from one of the busy workers and splits it into two sub-formulas, one of which is assigned to the busy worker and the other to one of the idle ones (work stealing model). This process is repeated until the queue of idle workers is empty. If during the solver's run a core becomes idle and is added to the idle queue (e.g., if it has established UNSAT for its input sub-formula), the above-mentioned process is invoked

until the idle queue becomes empty again. This form of load-balancing ensures that worker nodes are not allowed to idle for too long.

B. Backend Sequential Solvers and Splitting Heuristic

Each sequential worker receives the formula constrained with a set of assumptions that represents the sub-formula assigned to that worker. Workers start solving the sub-formula, until a threshold is reached. If the sub-formula is solved, they report back the SAT/UNSAT result to the master. In case they reach the search limit, and no other worker is idle, they continue the search, otherwise the problem is deemed too hard and the sub-formula is split further. The splitting procedure works as follows: the slave solver queries a ranking method that returns a total order over the variables in the input formula. These ranking methods are heuristics that analyze the formula in order to determine an order over the variables such that the higher a variable's rank, the better it is for the solver to split on it if the goal is to minimize overall solver runtime. The top-ranked variable v is then returned by the slave solver to the master, which splits the formula using the variable v into two sub-formulas. (It goes without saying that there are no guarantees of optimality here, since determining the optimal variable to split on is in general an NP-hard problem.)

III. DESCRIPTION OF MAPLEPAINLESS-DC

MaplePainless-DC is an instance of Painless-DC, whose splitting heuristic component has been replaced with a machine learning based splitting heuristic.

A. The Splitting Problem and an ML-based Splitting Heuristic

In order to rank splitting variables, we defined a performance metric of splitting a formula ϕ on a variable v as: $pm(\phi, v) :=$ The total runtime of the two sub-formulas of setting v to False and True ($\phi[v = F]$ and $\phi[v = T]$) in parallel. The splitting problem can be then simply stated as: $SplittingHeuristic(\phi) := argmin_v \{pm(\phi, v)\}$. The runtimes of solving a formula is not known a priori. One can build a machine learning model that given features of a formula, predicts the runtime of solving it using a specific SAT solver (Empirical hardness model). However, building such a model is well known as a challenging task for a variety of reasons. Our main observation in this setting is that, instead

of building an Empirical Hardness Model (i.e., predicting the runtimes exactly), we only need to provide a ranking of the runtimes. Therefore we built a machine learning model that approximates this function: $PW(\phi, v_i, v_j) = 1$ if $pm(\phi, v_i) < pm(\phi, v_j)$, and 0 otherwise. Because the output of PW is in $\{0, 1\}$, we built a binary classifier using random forest. This model can then be used as a comparator to find the minimum from the candidate list of splitting variables, that supposedly minimizes our performance target.

B. Sequential Solvers and Feature Computation

We used MapleCOMSPS [3] as the backend sequential worker solvers. We instrumented the worker solver to compute formula and variable features (e.g., number of times a variable is assigned, either decided or propagated) on the sub-formula to be split. Majority of the variable features are dynamic and their counters are updated whenever there is a related action performed during the search by the solver, thus their complexity is amortized over the run of the solver. The description of the variable features is listed in Table I. When the features are ready, the model trained for PW is queried as a comparator operator on the variables to find the minimum in a linear scan. The minimum variable is returned to the master node.

C. Other Settings

a) *Node switching strategy*: Whenever a worker W becomes idle and eventually another formula is split, this worker W is switched to solving one of the generated sub-formulas. The worker chooses between two solver states to continue: its own solver state, or the state of another worker solver that was solving a sub-formula before splitting it. In our version, we use the *clone* strategy, which is adopting the state of original worker solver.

b) *Sharing*: We use an *all-to-all* sharing strategy, in which all workers send and receive conflict clauses with an LBD of 4 or less.

TABLE I
VARIABLE FEATURES ($var_features(v)$).

Feature name	Description
activity	VSIDS activity [4]
LRBProduct	product of LRB [5] activities of v and $\neg v$ literals
numFlip	#times the implied value of v is different than its cached value [6]
propRate	average #propagation over #decision [7]
numDecided	#times v has been picked in branching
numAssigned	#times v got a value through branching/propagation
numLearnt	#times v appeared in a conflict clause
decisionLevel	average of decision levels of v at the end of the limited search
numInBinary	#times v appears in a clause of size 2
numInTernary	#times v appears in a clause of size 3

ACKNOWLEDGMENT

We would like to thank Ludovic Le Frioux, for answering our many questions regarding the Painless-DC framework.

REFERENCES

- [1] L. Le Frioux, S. Baair, J. Sopena, and F. Kordon, "Painless: a framework for parallel sat solving," in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2017, pp. 233–250.
- [2] —, "Modular and efficient divide-and-conquer sat solver on top of the painless framework," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 135–151.
- [3] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maplecomps lrb vsids and maplecomps chb vsids," *Proc. of SAT Competition*, pp. 20–21, 2017.
- [4] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.
- [5] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for sat solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer International Publishing, 2016, pp. 123–140.
- [6] G. Audemard, B. Hoessen, S. Jabbour, and C. Piette, "An effective distributed D&C approach for the satisfiability problem," in *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE, 2014, pp. 183–187.
- [7] S. Nejati, Z. Newsham, J. Scott, J. H. Liang, C. Gebotys, P. Poupart, and V. Ganesh, "A propagation rate based splitting heuristic for divide-and-conquer solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 251–260.

MergeSAT

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

Abstract—The sequential SAT solver MERGESAT is a fork of the 2018 competition winner, and adds known as well as novel implementation and search improvements. MERGESAT is setup to simplify merging solver contributions into one solver, to motivate more collaboration among solver developers.

I. INTRODUCTION

When looking at recent SAT competitions, the winner of the current year was typically last years winner plus a small modification. However, each year there are several novel ideas that the next winner does not pick up. Hence, lots of potential with respect to maximal performance is likely lost, and bug fixes of previous versions do not make it into novel versions.

The CDCL solver MERGESAT is based on the competition winner of 2018, MAPLE_LCM_DIST_CHRONOBT [14], and adds several known techniques, fixes, as well as adds some novel ideas. To make continuing the long list of work that influenced MERGESAT simpler, MERGESAT uses git to combine changes, and comes with continuous integration to simplify extending the solver further. Furthermore, starting in 2020, code style is enforced during CI as well, allowing to understand modifications better.

II. INTEGRATED TECHNIQUES AND FIXES

Most recently, backtracking has been improved by [10]. Backtracking improvements during restarts have already been proposed in [13]. MERGESAT uses the *partial trail* based backtracking during restarts.

Learned clause minimization (LCM) [7] is also kept. It is still open research in which order literals should be considered during vivification [11]. MERGESAT uses the improvement from [8], which repeats vivification in reverse order, in case a clause could be simplified with the first order. The original implementation of LCM adds a bit to the clause header to indicate that this clause has been considered already. However, no other bit has been dropped from the header, resulting in a 65 bit header structure. Along [5], this can result in a major slow down of the solver. Consequently, MERGESAT drops a bit in the size representation of the clause.

Large formulas require a long simplification time, even though simplification algorithms are polynomial. While for a 5000 second timeout, large simplification times are acceptable for effective simplifications, usually an incomplete simplification helps the solver already. Therefore, we introduce a step counter, that is increased whenever simplification touches a clause. Next, we interrupt simplification as soon as this counter reaches a predefined limit, similarly to [2]. To speed up simplification further, the linear subsumption implementation

and related optimizations from [3] have been integrated into MERGESAT.

Since the solver MAPLESAT [6], the decision heuristic is switched back from the currently selected one to VSIDS – after 2500 seconds. As solver execution does not correlate with run time, this decision results in solver runs not being reproducible. To fix this property, the switch to VSIDS is now dependent on the number of performed propagations as well as conflicts. Once, one of the two hits a predefined limit, the heuristic is switched back to VSIDS. This change enables reproducibility and deterministic behavior again. Based on [15], we added toggling VSIDS and LRB heuristic continuously.

MERGESAT implements an experimental – and hence disabled by default – heuristic to decide when to disable *phase saving* [12] during backtracking, which has been used in RISS [8] before: When the formula is parsed, for each non-unit clause it is tracked whether before applying unit propagation there is a positive literal. In case there is no positive literal, a *break* count is incremented. For the whole formula, this count approximates how close the formula is to being able to be solved by the *pure literal rule*. In case this break count is zero, or below a user defined threshold, no phase saving is used. The same rule is applied for negative literals. There exists benchmarks, where this heuristic with a threshold zero results in a much better performance. However, for a mixed benchmark, this feature has not been tested enough, and hence, remains disabled.

III. INPROCESSING

The simplification in MERGESAT has been limited via the number of allowed steps to perform. Hence, the potential to simplify further clauses is still there. Following the ideas in [2], starting 2020 MERGESAT runs subsumption and self-subsuming resolution using promising learnt clauses exactly once, as also implemented in an independent glucose hack [9]. This extension is also motivated by the fact that [15] checks LCM-simplified learnt clauses for duplicates, and drops those. With subsumption and simplification, we can drop even more redundant clauses.

IV. INCREMENTAL SAT

In previous variants of MAPLESAT, incremental solving via assumptions was disabled. To be able to use MERGESAT as backend in model checkers and other tools that require incremental solving capabilities, this feature is brought back. Furthermore, an extended version ¹ of the IPASIR interface [1]

¹<https://github.com/comp-solutions/ipasir>

is provided, which besides the usual functionality offers an additional function `ipasir_solve_final` to tell the SAT solver that this call is the final (or only) call. This function allows the solver to use formula simplification more extensively, because usually simplification cannot be applied during incremental solving.

MERGESAT implements *assumption prefetching*, which fast-forwards assumed literals, and triggers propagation only after all (from the previous calls used) assumptions have been assigned a value. In case of a conflict, the whole assumption stack is currently rolled back. Furthermore, the final conflict is simplified with LCM (in case the previous conflict was simplified successfully, otherwise, we skip once).

V. PARALLEL SOLVERS AND DOCKERFILE

We submit the sequential solver as a parallel solver to the competition, in 2 configurations. The solver configuration between the main track and the parallel track is the same.

The default (“parallel”) solver MERGESAT, which is compiled using the provided Dockerfile in the repository. This dockerfile links against a modified `glibc`, which enables *transparent huge pages* (THP) by default for the solver. On systems, where this feature is not enabled as “always” (like i.e. on the StarExec cluster), using THP can boost the solver runtime by 10% in average, with peaks of up to 20% improvement [4]. To demonstrate this behavior, the second submitted “parallel” solver, MERGESAT-NOTHP uses the exact same configuration, except not using THP.

The README of the MERGESAT repository contains the descriptions how to compile a statically linked binary of the solver to use it outside of the container as well. These instructions should work for any MINISAT 2.2 based solver. For other solvers and tools, additional dependencies would have to be added to the Dockerfile.

VI. AVAILABILITY

The source of the solver is publicly available under the MIT license at <https://github.com/conp-solutions/mergesat>. The version with the git tag “sat-comp-2020” is used for the submission. The submitted starexec package can be reproduced by running `./scripts/make-starexec.sh` on this commit.

VII. CONTINUOUS TESTING

The submitted version of MERGESAT compiles on Linux and Mac OS. GitHub allows to use continuous testing, which essentially build MERGESAT, and tests basic functionality: i) producing unsatisfiability proofs, ii) building the starexec package and producing proofs, iii) being used as an incremental SAT backend in Open-WBO as well as iv) solving via the IPASIR interface. All these steps are executed by executing the script `tools/ci.sh` from the repository, and the script can be used as a template to derive similar functionality. Independently, static code analysis with Coverity is used as part of continuous testing.

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.

REFERENCES

- [1] T. Balyo, A. Biere, M. Iser, and C. Sinz, “SAT race 2015,” *Artif. Intell.*, vol. 241, pp. 45–65, 2016.
- [2] A. Biere, “Precosat system description,” <http://fmv.jku.at/precosat/precosat-sc09.pdf>, 2009.
- [3] T. Ehlers and D. Nowotka, “Tuning parallel sat solvers,” in *Proceedings of Pragmatics of SAT 2015 and 2018*, ser. EPiC Series in Computing, D. L. Berre and M. J. Arvisalo, Eds., vol. 59. EasyChair, 2019, pp. 127–143. [Online]. Available: <https://easychair.org/publications/paper/NkG7>
- [4] 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>
- [5] 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.
- [6] V. G. K. C. Jia Hui Liang, Chanseok Oh and P. Poupart, “MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB,” in *Proceedings of SAT Competition 2016*, 2016. [Online]. Available: <http://hdl.handle.net/10138/164630>
- [7] 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>
- [8] N. Manthey, “Riss 7.1,” in *Proceedings of SAT Competition 2018*, 2018. [Online]. Available: <http://hdl.handle.net/10138/237063>
- [9] —, “Glucose 3.0 – With Inprocessing,” 2020, submitted.
- [10] 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.
- [11] C. Piette, Y. Hamadi, and L. Sais, “Vivifying propositional clausal formulae,” in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, vol. 178, 2008, pp. 525–529.
- [12] K. Pipatsrisawat and A. Darwiche, “A lightweight component caching scheme for satisfiability solvers,” in *SAT*, ser. LNCS, vol. 4501, 2007, pp. 294–299.
- [13] A. Ramos, P. van der Tak, and M. J. H. Heule, “Between restarts and backjumps,” in *Theory and Applications of Satisfiability Testing - SAT 2011*, K. A. Sakallah and L. Simon, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 216–229.
- [14] 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>
- [15] V. K. Stepan Kochemazov, Oleg Zaikin and A. Semenov, “Maplelcmdistchronobt-dl, duplicate learnts heuristic-aided solvers at the sat race 2019,” 2019.

ParaFROST, ParaFROST_CBT, ParaFROST_HRE, ParaFROST_ALL at the SAT Race 2020

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

I. INTRODUCTION

This paper presents a brief description to our solver ParaFROST which stands for *Parallel Formal Reasoning Of Satisfiability* in 4 different configurations. Our solver is based on state-of-the-art CDCL search [1]–[3], integrated with preprocessing as presented in our tool called SIGMA (SAT sImplification on GPU Architectures) [4], [5], and a new technique called Multiple Decision Making (MDM) [6]. Nevertheless, all submitted versions only permits a single-threaded CPU execution.

ParaFROST provides easy-to-use infrastructure for SAT solving and/or preprocessing with optimized data structures for both CPU/GPU architectures, and fine-tuned heuristic parameters. The *Parallel* keyword in ParaFROST intuitively means that SAT simplifications can be fully executed on variables in parallel as described in [4] using the Least Constrained Variable Elections (LCVE) algorithm. Moreover, via the MDM procedure [6], the solver is capable of making multiple decisions that can be assigned and propagated at once. In principle, choosing variables to preprocess or decisions relies heavily on *freezing* (that is where FROST is surfaced) mutually independent variables according to some logical properties.

II. PREPROCESSING

In previous work, we have shown how Bounded Variable Elimination (BVE) [7], [8] and Hybrid Subsumption Elimination (HSE) can be performed in parallel on Graphics Processing Units (GPU). The acceleration is proven to be effective in increasing the amount of reductions within a fraction of second, e.g. $66\times$ speedup compared to SatElite [8] when combined together in `ve+` mode [4]. This mode iterates over BVE and HSE in several rounds until no literals can be removed. Furthermore, we have added new implementations for Blocked Clause Elimination (BCE) and a new simplification technique, we call Hidden Redundancy Elimination (HRE) [5]. HRE 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 $C \equiv C_1 \otimes_x C_2$ and C is not a

This work is part of the GEARS project with project number TOP2.16.044, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

tautology, then let $\mathcal{S} := \mathcal{S} \setminus \{C\}$. The clause C is called a *hidden redundancy* and can be removed without altering the original satisfiability. For example, consider the formula $\mathcal{S} = \{\{a, \bar{c}\}, \{c, b\}, \{\bar{d}, \bar{c}\}, \{b, a\}, \{a, d\}\}$. Resolving the first two clauses gives the resolvent $\{a, b\}$ which is equivalent to the fourth clause in \mathcal{S} . Also, resolving the third clause with the last clause yields $\{a, \bar{c}\}$ which is equivalent to the first clause in \mathcal{S} . HRE can remove either $\{a, \bar{c}\}$ or $\{a, b\}$ but not both.

In this submission, a sequential implementation of all simplifications described above is provided as part of ParaFROST. By default, in ParaFROST, all simplifications are disabled. In ParaFROST_HRE, the `ve+` is enabled with number of `phases` set to 2. The `phases=<n>` option applies `ve+` for a configured number of iterations, with increasingly large values of the threshold μ (maximum occurrences of a variable) [4], [5]. After all phases are done, the `hre` method is executed once. On the other hand, the ParaFROST_ALL submission enables all simplifications along with `bce`.

both ParaFROST_HRE and ParaFROST_ALL delay preprocessing by a user-defined number of restarts. This gives the solver enough time to solve trivial problems (solved in few seconds) before simplifications are executed. The number of restarts needed to activate preprocessing is set to 50 through the option `pre-delay=<n>`. The solver supports geometric [9], Luby [2], and dynamic restarts [10]. However, in all submissions, we only enable dynamic restarts.

III. MULTIPLE DECISION MAKING

We proposed a new approach [6] to make multiple decisions in such a way, they can be assigned and propagated simultaneously or sequentially without causing any implications or conflicts. Originally, we did so to introduce a possible parallelisation strategy. This strategy is yet to pay off, but surprisingly, the MDM turned out to have a positive impact on standard, sequential CDCL, for many different formulas. In all configurations, the solver periodically calls MDM with a maximum of 3 rounds per search. Otherwise, a single decision is made as the standard CDCL procedure does. The number of MDM rounds is controlled via the option `PDM=<n>`.

IV. CHRONOLOGICAL BACKTRACKING

We adopted the *chronological backtracking* (CBT) introduced by the authors in [11], to help CDCL solvers avoid

jumping too far in certain situations. However, the procedure is computationally expensive in calculating the correct chronological level during a conflict. Therefore, we enabled this feature in a separate solver instance called ParaFROST_CBT (all simplifications are disabled). The CBT is triggered when the number of conflicts are multiple of 5000 (`cbt-conf=<n>`) and the jumping distance is 500 (`cbt-dist=<n>`). In ParaFROST_HRE, this option is disabled.

V. AUTOMATED TUNING

The GPU code tuner made by Ben van Werkhoven [12], [13] is used to optimize the parameter settings of all heuristics in ParaFROST_ALL. The tool is capable of tuning both CPU and GPU codes with support for many search optimization algorithms. In our case, we collected a sample of 48 different formulas, stemmed from different CNF families. The solving time per problem is expected to take 1000 seconds according to a solver experiment without tuning. Then, we ran a Python script to optimize the solver based on the accumulated running time of the selected benchmark suite. The tuned parameters are passed to the solver as command-line options. The basin hopping strategy is used to accelerate the tuning process.

Finally, the solver instance ParaFROST_ALL comprises all configurations described in the previous sections, in which HRE, CBT, and all simplifications are enabled.

REFERENCES

- [1] J. P. Marques-Silva and K. A. Sakallah, "Grasp: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.
- [2] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT*, ser. LNCS, vol. 2919. Springer, 2004, pp. 502–518.
- [3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, ser. IJCAI'09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, p. 399–404.
- [4] M. Osama and A. Wijs, "Parallel sat simplification on gpu architectures," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 21–40.
- [5] —, "Sigma: Gpu accelerated simplification of sat formulas," in *Integrated Formal Methods*, W. Ahrendt and S. L. Tapia Tarifa, Eds. Cham: Springer International Publishing, 2019, pp. 514–522.
- [6] —, "Multiple decision making in conflict-driven clause learning," Submitted.
- [7] S. Subbarayan and D. K. Pradhan, "NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances," in *SAT*, ser. LNCS, vol. 3542. Springer, 2004, pp. 276–291.
- [8] N. Eén and A. Biere, "Effective Preprocessing in SAT Through Variable and Clause Elimination," in *SAT*, ser. LNCS, vol. 3569. Springer, 2005, pp. 61–75.
- [9] T. Walsh *et al.*, "Search in a small world," in *Ijcai*, vol. 99, 1999, pp. 1172–1177.
- [10] G. Audemard and L. Simon, "Refining restarts strategies for sat and unsat," in *Principles and Practice of Constraint Programming*, M. Milano, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 118–126.
- [11] 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.
- [12] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347 – 358, 2019.
- [13] —, "Kernel tuner," 2020. [Online]. Available: https://github.com/benvanwerkhoven/kernel_tuner

Riss 7

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

Abstract—The sequential SAT solver RISS combines a heavily modified Minisat-style solving engine of GLUCOSE 2.2 with a state-of-the-art preprocessor COPROCESSOR and adds many modifications to the search process. RISS allows to use preprocessing based on COPROCESSOR. As unsatisfiability proofs are mandatory, but many simplification techniques cannot produce them, a special configuration is submitted, which first uses all relevant simplification techniques, and in case of unsatisfiability, falls back to the less powerful configuration that supports proofs.

I. INTRODUCTION

The CDCL solver RISS is a highly configurable SAT solver based on MINISAT [1] and GLUCOSE 2.2 [2], [3], implemented in C++. Many search algorithm extensions have been added, and RISS is equipped with the preprocessor COPROCESSOR [4]. Furthermore, RISS supports automated configuration selection based on CNF formulas features, emitting DRAT proofs for many techniques and comments why proof extensions are made, and incremental solving. The solver is continuously tested for being able to build, correctly solve CNFs with several configurations, and compile against the IPASIR interface. For automated configuration, RISS is also able to emit its parameter specification on a detail level specified by the user. The repository of the solver provides a basic tutorial on how it can be used, and the solver provides parameters that allow to emit detailed information about the executed algorithm in case it is compiled in debug mode (look for “debug” in the help output). While RISS also implements model enumeration, parallel solving, and parallel model enumeration, this document focusses only on the differences to RISS 7, which has been submitted to SAT Competition 2017. Compared to the version of 2018, only the NOUNSAT configuration has been added. For 2020, mainly defects reported by Coverity have been addressed.

II. SAT COMPETITION SPECIFICS – NOUNSAT CONFIGURATION

The default configuration uses only variable elimination [5] and bounded variable addition [6] as simplification, both of which can produce unsatisfiability proofs.

While recent SAT competitions come with a NOLIMITS track, this years event requires unsatisfiability proofs. To comply, simplification techniques that cannot produce proofs have been disabled in this situation. Differently, this years version comes with the NOUNSAT configuration, which basically cannot produce unsatisfiability answers. This means, that all simplification techniques are available for formulas that are satisfiable, or cannot be solved. In case the formula turns out to be unsatisfiable, the procedure is solved one more time, using the configuration that can produce unsatisfiability proofs.

III. AVAILABILITY

The source of the solver is publicly available under the LGPL v2 license at <https://github.com/comp-solutions/riss>. The version with the git tag “satcomp-2020” is used for the submission. The submitted starexec package can be reproduced by running “./scripts/make-starexec.sh” on this commit.

ACKNOWLEDGMENT

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

REFERENCES

- [1] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *SAT 2003*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Heidelberg: Springer, 2004, pp. 502–518.
- [2] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *IJCAI 2009*, C. Boutilier, Ed. Pasadena: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.
- [3] —, “Refining restarts strategies for sat and unsat,” in *CP’12*, 2012, pp. 118–126.
- [4] N. Manthey, “Coprocessor 2.0 – a flexible CNF simplifier,” in *SAT 2012*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Heidelberg: Springer, 2012, pp. 436–441.
- [5] N. 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.
- [6] N. Manthey, M. J. Heule, and A. Biere, “Automated reencoding of Boolean formulas,” in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science, A. Biere, A. Nahir, and T. Vos, Eds., vol. 7857. Springer Berlin Heidelberg, 2013, pp. 102–117. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39611-3_14

Engineering HordeSat Towards Malleability: mallob-mono in the SAT 2020 Cloud Track

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

Abstract—We briefly present the massively distributed SAT solver which we submit to the Cloud Track of the SAT Competition 2020, being the solver engine of a novel framework for massively parallel and distributed malleable job scheduling applied to SAT solving. Our solver is based on HordeSat; notable differences include completely asynchronous communication, a much more careful clause exchange, and some internal performance improvements.

Index Terms—Parallel SAT solving, distributed SAT solving

I. INTRODUCTION

In order to improve massively parallel problem solving “on demand” in a cloud context, we introduce malleability to parallel SAT solving as a part of a novel framework named “mallob” for massively parallel and distributed malleable job scheduling [1]. Malleability is the property of a computation to dynamically handle a varying amount of computational resources (i.e. cores or nodes) during its execution, opening up vast possibilities for performing highly dynamic load balancing on many jobs of varying demand and priority that run in parallel on some large-scale infrastructure.

However, the SAT competitions do not involve malleable computations nor solving multiple instances at the same time. As a consequence, we added a special configuration to our system for the sole purpose of solving a single instance with full computational power from the beginning and named it “mallob-mono” (mallob mono instance mode). In the following, we will describe the most relevant aspects of this solver engine and the surrounding architecture.

II. OVERVIEW

On each node we start one MPI process for each set of four available (virtual) CPUs such that each process can employ four solver threads. HordeSat [2] serves as a foundation for the solver engine residing on each process. Internally, we use Lingeling as a solver backend just like HordeSat’s default configuration. However, we updated the used Lingeling version from *ayv* (2014) [3] to *bcj* (2018) [4]. We also updated the native diversification routines of Lingeling according to the diversification of the 2018 version of Plingeling. We let one out of 14 solvers in our portfolio perform local search (using

YalSAT [4] as a backend) while the others are CDCL solvers with different set options.

We adjusted and replaced significant portions of the code-base of HordeSat in order to match the requirements of our malleable framework. As such, we enabled the suspension and resumption of particular solver instances, made all communication among the nodes completely asynchronous, and enabled descriptions of SAT formulae to be serialized and transferred directly over message passing instead of assuming that the formula resides on each node. Many of these changes are unimportant for the SAT competition. Some general performance improvements were integrated; for example, we reduce lots of unnecessary `getrusage` system calls by supplying a cheap and approximative time measuring callback over the Lingeling interface instead.

In the following we describe our clause exchange mechanism and the related clause filtering, which are the most prominent differences between HordeSat and our solver engine.

III. CLAUSE EXCHANGE

HordeSat initiates an All-to-all exchange of learnt clauses every second by a synchronous collective operation (`MPI_Allreduce`). The clause buffer size of each node is of fixed length 1500 and the entire buffer is sent around regardless of the degree to which it is filled. Duplicate clauses are detected by HordeSat’s clause filters only after the full operation succeeded. If an exported local clause buffer is filled to less than 80%, one of the local solver threads is asked to increase its clause production. Unit clauses are always shared and are exempt from being filtered. As a result, the first few clause exchanges are often flooded with large numbers of highly redundant unit clauses after first simplifications and preprocessing steps.

We have made the clause exchange entirely asynchronous while ensuring that one broadcast of a globally aggregated clause buffer takes place every second. We aggregate buffers of learnt clauses along a binary tree of all computing nodes. Clause buffers sent over this tree are always in compact shape, i.e., without any unused portions of memory. During the reduction, instead of just *concatenating* the buffers, inner nodes do a three-way *merge* of their local clauses and the clauses of their children, preferring short clauses and filtering out duplicates with an additional Bloom filter, a datastructure

Evaluation of this work was partially performed on the supercomputer ForHLR funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

that we took from original HordeSat [2]. Thereby, we limit the maximum length $b(u)$ of a merged clause aggregation containing clauses from u nodes:

$$b(u) = \lceil u \cdot \alpha^{\log_2(u)} \cdot 1500 \rceil$$

Note that $\alpha = 0.5$ makes the length of a clause aggregation converge to 1500 the more nodes are involved, and $\alpha = 1.0$ makes the limit grow linearly in the number of nodes just like in HordeSat. We set $\alpha = 0.75$ to find a middle ground between these extremes.

Additionally, no clauses of length greater than five are shared. With this strict limitation we expect to avoid a lot of communication volume and internal work in the SAT solvers while still sharing lots of potentially interesting information among the solvers.

After the reduction reaches the binary tree’s root node, the clause aggregation is broadcast through the tree to all other nodes and locally digested when appropriate.

IV. CLAUSE FILTERING

We also made some adjustments to HordeSat’s clause filtering mechanic used when clauses are exported or imported. We added duplicate checking for unit clauses both to each clause filter and to our duplicate checking during the reduction. This check does not rely on Bloom filters but functions with exact hash sets, using one of the commutative hash functions that are employed in the Bloom filters. This way we do not get any false positives for unit clauses and make sure that each such clause is being shared at least once.

Last but not least, we implemented a mechanic similar to restarts into the clause filters. The authors of original HordeSat already intended to periodically clear clause filters in order to be able to share clauses after some time, but it was not implemented. We introduce a quite careful “forgetting” of shared clauses: Every five minutes, in one iteration over all set bits in the filter each bit is unset with probability $\sqrt[4]{0.5} \approx 15.91\%$. As every clause inserted into the filter sets four bits from four hash functions, the probability that a clause is forgotten is close to $P(\text{forgotten}) = P(\geq 1 \text{ bit unset}) = 1 - P(0 \text{ bits unset}) = (\sqrt[4]{0.5})^4 = 0.5$. For the unit clauses, every element in the explicit set is forgotten with probability 0.5. Overall, approximately half of all clauses are effectively forgotten and can be shared again.

V. LICENSE

Our system mallob and, by extension, our submitted solver is licensed under the GNU Lesser General Public License (LGPLv3). As the licensing of Lingeling was changed to MIT with the 2018 version, our system consists of fully Free Software.

VI. CONCLUSION

We described the central aspects of our massively parallel SAT solver and are excited to see how it performs in the AWS environment of the competition.

While our competitor does include some computational overhead due to its malleable job scheduling aspects, we still expect that our solver will overall outperform original HordeSat due to various improvements of the internal workings of the portfolio solver and notably the improved clause exchange.

ACKNOWLEDGMENT

Many thanks to Tomáš Balyo and Markus Iser for fruitful discussions regarding HordeSat and its clause exchange.

REFERENCES

- [1] P. Sanders and D. Schreiber, “Massively parallel malleable job scheduling for SAT solving.” to be published.
- [2] 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.
- [3] 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.
- [4] A. Biere, “CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2018,” *Proc. of SAT Competition*, pp. 13–14, 2018.

DurianSat at SAT Competition 2020

Arijit Shaw
School of Computing
National University of Singapore
if.arijit@gmail.com

Kuldeep S. Meel
School of Computing
National University of Singapore
meel@comp.nus.edu.sg

Abstract—DurianSat is a patched Maple_LCM_Dist_ChronoBTv3 solver modifying the phase selection heuristic. DurianSat discriminates decisions based on whether the last backtrack was chronological or not. It applies a new literal score based phase selection heuristic called LSIDS if the last backtrack was chronological.

I. INTRODUCTION

Although phase-saving [5] is highly effective as a phase selection heuristic for SAT solving, that effectiveness is not observed if the solver is backtracking chronologically [4]. This observation is made in an upcoming paper [6]. DurianSat addresses this lack of integration between chronological backtrack (CB) and phase saving by implementing a literal activity based phase selection heuristic. This literal activity is called LSIDS.

II. LSIDS LITERAL ACTIVITY SCHEME

LSIDS, which is a VSIDS [7] like scoring scheme for literals, maintains activity for every literal. The activity for a literal is bumped in the following cases :

- If a literal l occurs in a learnt clause, bump activity for l .
- If assignment for a variable v gets canceled during backtrack; if the assignment was TRUE, then bump activity for v , otherwise the bump activity of $\neg v$.

Decay and rescore of activity is the same as VSIDS. Consult our SAT 2020 paper [6] or the github repository [1] for the details.

III. LSIDS PHASE SELECTION HEURISTIC

If the branching heuristic decides to branch on a variable v , an LSIDS based phase selection heuristic looks at the activity of both the literals of v and selects the literal with higher activity.

IV. DISCRIMINATING CB AND NCB

The solver Maple_LCM_Dist_ChronoBTv3 [3] uses a combination of chronological and non-chronological backtracks. DurianSat adds a patch on the above solver and opts for LSIDS based phase selection if the last backtrack is chronological.

ACKNOWLEDGMENT

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISGRP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore [2].

REFERENCES

- [1] DurianSat SAT solver GitHub page (2020), <https://github.com/meelgroup/duriansat/tree/satcomp20>
- [2] ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), <https://www.nscg.sg/about-nscg/overview/>
- [3] Heule, M.J., Jarvisalo, M., Suda, M.: Sat race 2019
- [4] Nadel, A., Ryvchin, V.: Chronological backtracking. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 111–121. Springer (2018)
- [5] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: International conference on theory and applications of satisfiability testing. pp. 294–299. Springer (2007)
- [6] Shaw, A., Meel, K.S.: Designing new phase selection heuristics. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT) (2020)
- [7] Silva, J.P.M., Sakallah, K.A.: Grasp—a new search algorithm for satisfiability. In: The Best of ICCAD, pp. 73–89. Springer (2003)

¹Durian, the “king of fruits”, is a commonplace fruit in Southeast Asia and part and parcel of Singapore’s culture. The label can be attributed to the fruit’s “formidable look and overpowering odour.”

Maple_CM+dist, Maple_CM+dist+sattime, Maple_CM+dist+simp2- and Maple_CMused+dist in the SAT Competition 2020

Chu-Min Li^{1,2}, Mao Luo¹, Fan Xiao¹, Yu Li², Felip Manyà³, Zhipeng Lü¹

¹School of Computer Science, Huazhong University of Science and Technology, Wuhan, China
{maoluo,zhipeng.lv}@hust.edu.cn, zhaoyangxiaofan@163.com

²MIS, University of Picardie Jules Verne, Amiens, France

{chu-min.li, yu.li}@u-picardie.fr

³Artificial Intelligence Research Institute (IIIA-CSIC), Barcelona, Spain. felip@iiia.csic.es

1. Introduction

The CDCL SAT solver Maple_LCM won the gold medal of the main track of the SAT Competition 2017. It was implemented on top of the solver MapleCOMSPS_DRUP [1], [2] by integrating the effective learnt clause minimization approach described in [3]. The CDCL SAT solver Maple_CM extends Maple_LCM by extending the learnt clause minimization approach to original clauses and by minimizing the clauses more than once in certain conditions [4]. Maple_CM won the bronze medal of the main track of the SAT Competition 2018. In the current competition, we propose Maple_CM+dist, which uses the distance heuristic described in [5] for the first 50,000 conflicts. In addition, we propose Maple_CM+dist+sattime, which integrates a local search in Maple_CM+dist and makes Maple_CM+dist work from the assignment obtained after applying local search, and Maple_CM+dist+simp2- and Maple_CMused+dist, two variants of Maple_CM+dist. The four solvers are described in the remaining sections.

2. Clause Minimization in Maple_CM+dist

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

- During preprocessing, each original clause is minimized. The minimization process stops when the total number of unit propagations is greater than 10^8 .
- During the search, Maple_CM organizes the learnt clauses in three sets as MapleCOMSPS_DRUP: CORE, TIER2 and LOCAL. The sets CORE and TIER2 roughly store the learnt clauses with $LBD \leq 6$, where LBD refers to the number of decision levels in a clause [6]. It also identifies a subset

of original clauses called *useful clauses* that are used to derive at least one learnt clause of $LBD \leq 20$ since the last clause minimization. Then, before a restart, Maple_CM minimizes each clause C such that $liveClause(C)$ (see below) returns true, provided that the number of clauses learnt since the last clause minimization is greater than or equal to $\alpha + 2 \times \beta \times \sigma$, where $\alpha = \beta = 1000$ and σ is the number of minimizations executed so far.

Function $liveClause(C)$ returns true if C is a learnt clause in CORE or TIER2 that has been never minimized or its LBD has been reduced twice since its last minimization, or if C is a useful original clause that has been never minimized or its LBD has been reduced three times since its last minimization.

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

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

A particular case is a clause with LBD 1. This clause is probably very powerful in unit propagation and its LBD value cannot be decreased anymore. So, a clause will be re-minimized if its LBD becomes 1 since its last minimization, no matter how many times the LBD value was decreased. See [4] for more details.

Maple_CM+Dist is Maple_CM in which the distance

heuristic [5] is used to select the decision variable during the first 50,000 conflicts.

3. Local search in Maple_CM+dist+sattime

Sattime is a local search algorithm presented in [7]. It is based on g2wsat [8]. Similar to adaptg2wsat, Sattime works with a (randomly generated) truth assignment. If the assignment satisfies all the clauses, the search stops. Otherwise, Sattime randomly selects a clause c falsified by the current assignment. If the best variable x in c (i.e., the variable whose flip allows to satisfy the greatest number of clauses) is not the most recent variable *satisfying* c in the past, it flips x . Otherwise, with an automatically adapted probability p , it flips the second best variable y in c and, with probability $1 - p$, it flips x . Note that adaptg2wsat flips the best variable x in c if x is not the most recent variable in c *falsifying* c in the past; otherwise, it flips the second best variable y in c with probability p , and it flips x with probability $1 - p$. This difference with adaptg2wsat in terms of satisfying versus falsifying makes Sattime efficient for both random and structured instances.

Maple_CM+Dist+Sattime solves an instance in three steps:

- 1) It minimizes the original clauses as in Maple_CM+Dist. Observe that this minimization can fix some variables in the instance. It repeats the minimization until no more variables can be fixed or the total number of unit propagations exceeds 2×10^9 .
- 2) If the total number of unit propagations does not exceed 2×10^9 , it executes the heuristic Sattime for 2×10^8 flips.
- 3) It executes the CDCL search from the assignment obtained in step 2.

It is well-known that local search is less effective than CDCL for structured instances. The use of local search in Maple_CM+Dist is based on the following two observations: (1) local search allows to obtain an assignment close to some solutions of a satisfiable instance. Working from this assignment would allow CDCL to find one of these solutions more easily; and (2) local search is not effective for the instances in which unit propagation can fix some variables. That is why local search in Maple_CM+Dist is applied to a simplified instance in which as many as possible variables are fixed in the clause minimization phase.

4. Further simplifying a simplified clause in Maple_CM+dist+simp2-

Maple_CM+dist+simp2- implements an idea of Riss 7.1 [9] on top of Maple_CM+dist: if a clause has been reduced by propagating its literals in their original order, then re-simplify it by propagating its remaining literals in the reverse order. It is not necessary to propagate the literals of a clause in the reverse order if it cannot be vivified in its original order.

5. Simplifying clauses that are used during the search in Maple_CMused+dist

A SAT instance usually contains a huge number of clauses. A CDCL solver also learns a large number of clauses during the search. However, a considerable number of these original or learnt clauses are rarely or not used during the search, and it is not necessary to vivify them. A useful conflict is a conflict such that the LBD of the clause learnt from the conflict is less than 20 [4]. Maple_CMused+dist is Maple_CM+dist but limits the learnt or original clauses to those used to derive at least 3 useful conflicts since the last clause vivification. Additionally, while Maple_CM+dist does not vivify the learnt clauses with LBD greater than 6 (i.e., learnt clauses stored in the LOCAL set), Maple_CMused+dist also vivifies the most active quart of learnt clauses in LOCAL, provided that these clauses were used to derive at least 3 useful conflicts since the last clause vivification.

Acknowledgments

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

References

- [1] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Proceedings of SAT*, 2016, pp. 123–140.
- [2] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maple-COMSPS, MapleCOMSPS LRB, MapleCOMSPS CHB," in *SAT Competition*, 2016, p. 52.
- [3] M. Luo, C.-M. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *Proceedings of IJCAI*, 2017, pp. 703–711.
- [4] C.-M. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li, "Clause vivification by unit propagation in cdcl sat solvers," *Artificial Intelligence*, vol. 279, 2020.
- [5] F. Xiao, C.-M. Li, M. Luo, F. Manyà, Z. Lü, and Y. Li, "A branching heuristic for sat solvers based on complete implication graphs," *SCIENCE CHINA Information Sciences*, vol. 62 (7), pp. 1–13, 2019.
- [6] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of IJCAI-2009*, 2009, pp. 399–404.
- [7] C. M. Li and Y. Li, "Satisfying versus falsifying in local search for satisfiability," in *Proceedings of SAT-2012*. Springer, 2012, pp. 477–478.
- [8] C. M. Li and W. Q. Huang, "Diversification and Determinism in Local Search for Satisfiability," in *Proceedings of SAT2005*, 2005, pp. 158–172.
- [9] N. Manthey, "Riss 7.1," in *Proceedings of SAT competition 2018*. Department of Computer Science Series of Publications B, University of Helsinki, 2018, pp. 36–37.

CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING Entering the SAT Competition 2020

Armin Biere Katalin Fazekas Mathias Fleury Maximilian Heisinger
Institute for Formal Models and Verification
Johannes Kepler University Linz

Abstract—This system description describes our new SAT solver KISSAT, how it differs from CADICAL, as well as changes made to CADICAL. We further present our new distributed cube-and-conquer solver PARACOOBA. Previous parallel SAT solvers PLINGELING and TREENGELING in essence remain unchanged.

I. CADICAL

Compared to the 2019 version of CADICAL [1], we have improved inprocessing by implementing conditioning [2]. However, this feature does not seem to improve performance and is not enabled by default (“--condition=true”). The major difference is the implementation of a three-tier system [3] to decide which clauses should be kept during clause reduction: tier-0 clauses ($LBD \leq 2$) are kept forever, tier-1 clauses ($2 < LBD \leq 6$) survive one round of reduction, whereas tier-2 clauses can be deleted immediately. In any case, clauses used since the last reduction are not deleted.

Moreover, even though CaDiCaL was designed to allow up to $INT_MAX = 2^{31} - 1 = 2\,147\,483\,647$ variables, represented with the `int` type of C++, it had a serious flaw because the idiom “for (int i = 1; i <= max_var; i++)” was used throughout the code. This led to undefined behaviour if INT_MAX variables are used even though it works fine for fewer. To avoid complicated iteration code and also to avoid such issues in the future we implemented variable and literal iterators, used as in “for (auto idx : vars)” or as in “for (auto lit : lits)”. We would like to thank Håkan Hjort for bringing this issue to our attention.

II. KISSAT

Experiments with large formulas, such as the DIMACS formula “p cnf 2147483647 0” resulted in the following observations. Even though CaDiCaL can handle formulas with INT_MAX variables, it needs a substantial amount of main memory (more than 512 GB) as well as long time for initialization. One reason is using the C++ container “`std::vector`” for most data structures (e.g., to hold flags, values, decision levels, reasons, scores). They are also mostly zero initialized. Instead, we now use the C memory allocator “`calloc`”. It

Supported by Austrian Science Fund (FWF) projects W1255-N23 and S11408-N23, by the LIT AI and LIT Secure and Correct Systems Labs and the LIT project LOGTECHEDU all three funded by the State of Upper Austria.

provides zero initialization on-demand by the virtual memory system and reduces resident set size accordingly.

This design decision also raised the question, whether we can reuse some other features of LINGELING [4] to further reduce memory. In KISSAT we therefore completely inline binary clauses in watcher stacks to reduce the size of watches from 16 bytes in CADICAL to 4 bytes for binary and 8 bytes for large clauses (due to the blocking literal). This in turn requires to use 4-byte offsets instead of pointers to reference large (non-binary) clauses. Note that binary clauses were still allocated in CADICAL in the memory arena holding clauses in the same way as larger clauses. In KISSAT they now really only exist in watcher lists. LINGELING even inlined ternary clauses which we consider less useful now.

We also revisited the data structure for holding watches (watched lists). In KISSAT we use a dedicated implementation of stacks of watchers, requiring only two offsets (of together 8 bytes in the compact competition configuration) instead of 3 pointers (requiring 24 bytes on a 64-bit architecture). This became possible by assuming that the all-bits-one word is not a legal watch and free memory in the watcher stack arena is marked with all-bits-one words. Pushing a watch on a watcher stack requires checking whether the word after the top element is illegal (all-bits-one). If so, it is overwritten. Otherwise the whole stack is moved to the end of the allocated part in the watcher arena. This produces the overhead that once in a while the watcher arena requires defragmentation and is usually performed after collecting redundant clauses in “reduce”.

In order to distinguish binary and large watches in watcher stacks, we use bit-stuffing as in LINGELING. This leaves effectively 31 bits to reference large clauses. Since these large clauses are allocated 8-byte aligned in the clause arena, the maximum size of this arena is $8 \cdot 2^{31}$ bytes (16 GB). Note that in practice many large CNFs consist mostly of binary clauses, which due to inlining do not require any space in this arena. Further, beside data structures for variables, watch lists occupy a large fraction of the overall memory. Actually the largest CNFs we ever encountered in applications easily stay below this limit while in total KISSAT reaches 100 GB memory usage. On top of that, other solvers including CADICAL often need more than 4 times more main memory than KISSAT.

Due to inlining binary clauses redundant and irredundant binary clauses have to be distinguished [5], which requires an-

other watcher bit (“redundant”). Finally, as in CADICAL, hyper binary resolvents are generated in vast amounts [6] during failed literal probing and vivification [7] and have to be recycled quite aggressively. To mark these hyper binary resolvents we need a third watcher bit (“hyper”) and the effective number of bits for literals is reduced to 29. Thus, the solver can only handle $268\,435\,455 = 2^{28} - 1$ variables.

In “dense mode” (during for instance variable elimination) the solver maintains full occurrence lists for all irredundant clauses. In the default “sparse mode” (during search) only two literals per large clause are watched and large clause watches have an additional blocking literal. Thus, as in LINGELING, watch sizes vary between one and two words, which lead to very cumbersome and verbose watch list traversal code in LINGELING repeated all over the source code. For KISSAT we were able to almost completely encapsulate this complexity using macros. The resulting code resembles ranged-based `for` loops in C++11 as introduced in CADICAL last year.

These improved data structures described above obviously require too many changes and we decided to start over with a new solver. In order to keep full control of memory layout, it was written in C. Otherwise we ported all the important algorithms from CADICAL, and were also able to reconfirm their effectiveness in a fresh implementation. In this regard using “target phases” as introduced last year in CADICAL [1] should be emphasized, which after careful porting, gave a large improvement on satisfiable instances.

We want to highlight the following algorithmic differences. The first version of CADICAL had a sophisticated implementation of forward subsumption, building on the one in SPLATZ inspired by [8], which was efficient enough to be applied to learned clauses too. Only later we added vivification [9], which is now used in most state-of-the-art solvers, and is particularly effective on learned clauses [7]. Thus subsumption on learned clauses becomes less important and we only apply it on irredundant clauses before and during bounded variable elimination. We have both a fast forward subsumption pass for all clauses as well incremental backward but now also forward subsumption during variable elimination, carefully monitoring variables occurring in added or removed (irredundant) clauses, which allows us to focus the inprocessing effort.

The clause arena keeps irredundant clauses before redundant clauses, which allows during reduction of learned clauses in “reduce” to traverse only the redundant part of the arena. Since watches contain offsets to large clauses in the arena we can completely avoid visiting irredundant (original) clauses during this procedure. This substantially reduces the hot-spot of flushing and reconnecting watchers in watch lists during clause reduction. Note, that “reduce” beside “restart” is the most frequently called procedure in a CDCL solver (after the core procedures “propagate”, “decide”, and “analyze”).

In comparison to CADICAL inprocessing procedures are scheduled slightly differently. First there is no forward subsumption of clauses outside of the “eliminate” procedure. In KISSAT compacting the variable range is part of “reduce” and actually always performed if new variables became inac-

tive (eliminated, substituted or unit). Otherwise “probe” and “eliminate” call the same algorithms as in CADICAL, except for vivification which became part of “probe” and duplicated binary clause removal (aka hyper unary resolution), which has moved from “subsume” (thus in CADICAL triggered during search and during variable elimination) to “eliminate”.

More importantly we have a more sophisticated scaling procedure for the number of conflicts between calls to “probe” and “eliminate”, which as in CADICAL takes the size of the formula into account, but now applies an additional scaling function instead of just linearly increasing the base interval in terms of n denoting how often the procedure was executed.

For variable elimination (“elim”) the scaling function of the base conflict interval is $n \cdot \log^2 n$. For “probe” it is $n \cdot \log n$. Similarly we scale the base conflict interval for “reduce” by $n / \log n$, while for “rephase” it remains linear. More precisely as logarithm we use $\log_{10}(n+10)$. Thus “reduce” occurs most often, followed by “rephase”, then “probe” and least often “elim”, all in the long run, independently of the base conflict interval, and the initial conflict interval.

Since boolean constraint propagation is considered the hot-spot for SAT solvers, CADICAL uses separate specialized propagation procedures during search, failed literal probing and vivification. In KISSAT we have factored out propagation code in a header file which can be instantiated slightly differently by these procedures, so taking advantage of dedicated propagation code while keeping the code in one place.

The concept of quiet “stable phases” without many restarts and “non-stable phases” with aggressive restarting was renamed. We call it now “stable mode” and “focused mode” to avoid the name clash with “phases” in “phase saving” (and “target phases”). We further realized that mode switching should not entirely be based on conflicts, since the conflict rate per second varies substantially with and without frequent restarts (as well as using target phases during stable mode).

Since the solver starts in focused mode, these focused mode intervals can still be based on a (quadratically) increasing conflict interval. For the next stable interval we then attempt to use the same time. Of course, in order to keep the solver deterministic, this requires to use another metric than run time. In CADICAL we simply doubled the conflict interval after each mode switch which did not perform as well in our experiments as this new scheme.

Our first attempt to limit the time spend in stable mode was to use the number of propagations as metric. But this was not precise enough, since propagations per second still vary substantially with and without many restarts. Instead we now count “ticks”, which approximate the number of cache lines accessed during propagations. This refines what Donald Knuth calls “mems” but lifted to cache lines and restricted to only count watcher stack access and large clause dereferences, ignoring for instance accessing the value of a literal.

Cache line counting is necessary because in certain large instances with almost exclusively binary clauses most time is spend in accessing the watches with inlined binary clauses in watcher stacks and not in dereferencing large clauses, while in

general, and for other instances with a more balanced fraction of large and binary clauses, a single clause dereference is still considerably more costly than accessing an inlined binary clause. Computing these “ticks” was useful limit the time spent in other procedures, e.g., vivification, in terms of time spent during search (more precisely the time spend in propagation).

While porting the idea of target phases [1], we realized that erasing the current saved phases by for instance setting them to random phases, might destroy the benefit of saved phases to remember satisfying assignments of disconnected components of the CNF [10]. Instead of decomposing the CNF explicitly into disconnected components, as suggested in [10], we simply compute the largest autarky of the full assignment represented by saved phases, following an algorithm originally proposed by Oliver Kullman (also described in [2]).

This unique autarky contains all the satisfying assignments for disconnected components (as well as for instance pure literals). If the autarky is non-empty, its variables are considered to be eliminated and all clauses touched by it are pushed on the reconstruction stack. We determine this autarky each time before we erase saved phases in “rephase” and once again if new saved phases have been determined through local search.

Finally, combining chronological backtracking [11] with CDCL turns out to break almost the same invariants [12] as on-the-fly self-subsuming resolution [13], [14] and thus we added both, while CADICAL is missing the latter. Both techniques produce additional conflicts without learning a clause and thus initially we based all scheduling on the number of learned clauses instead on the number of conflicts, but our experiments revealed that using the number of conflicts provides similar performance and we now rely on that for scheduling.

As last year for CADICAL we submit three configurations of KISSAT, one targeting satisfiable instances (“sat”) always using target phases (also in focused mode), one for unsatisfiable instances (“unsat”), which stays in focused mode, and the default configuration (“default”), which alternates between stable and focused mode as described above, but only uses target phases in stable mode.

III. PARACOOBA

Our new solver PARACOOBA [15] has been submitted to the cloud track. It is a distributed cube-and-conquer solver. The input DIMACS is split on the master node into various subproblems (cubes) that can be solved independently. The work is distributed over the network first from the master node to other nodes and then across nodes depending on the workload of nodes.

The “quality” of the cubes is important for the efficiency of the solver. We have submitted two versions to the competition: one relies on the state-of-the-art lookahead solver MARCH [16] for splitting; another uses our own implementation of tree-based lookahead [6]. Our implementation is part of CADICAL and is much less tuned than MARCH. It is run with a timeout and, whenever splitting takes too long (more than 30 s), we fall back on the number of occurrences.

During solving, whenever a subproblem takes too long, i.e., based on a moving average of solving times, then we split the problem again into two or more subproblems. If many nodes are unused, we generate more (and hopefully simpler) subproblems in order to increase the amount of work that can be distributed onto further nodes.

Generated subproblems are solved using the incremental version of CADICAL described below in Sect. V and we aim at solving similar cubes on the same CADICAL instance to reuse the results of previous inprocessing.

IV. PLINGELING AND TREENGELING

We submitted PLINGELING and TREENGELING to the parallel track. Compared to the version submitted to the 2018 SAT Competition [17] we have made essentially no changes to PLINGELING and TREENGELING nor to the SAT solver LINGELING that is used internally.

V. INCREMENTAL TRACK

CADICAL also enters the incremental track of the competition. It relies on our method [18] to identify and restore the necessary clauses when new clauses are added and can thereby make use of most of all the implemented inprocessing techniques. A sequence of incremental problems is considered as a stand-alone run from the perspective of inprocessing scheduling, i.e. none of the relevant inprocessing counters are reset in between iterations. The assumptions of each iteration are internally frozen (i.e. excluded from inprocessing), but beyond that there is no special treatment regarding them.

VI. LICENSE

Our solvers are all available under MIT license at <http://fmv.jku.at/cadical> for CADICAL, <http://fmv.jku.at/kissat> for KISSAT, <https://github.com/maximaximal/Paracooba> for PARACOOBA, and <https://github.com/arminbiere/lingeling> for PLINGELING and TREENGELING.

REFERENCES

- [1] A. Biere, “CaDiCaL at the SAT Race 2019,” in *SAT Race 2019*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.
- [2] B. Kiesl, M. J. H. Heule, and A. Biere, “Truth assignments as conditional autarkies,” in *ATVA 2019*, ser. LNCS, Y. Chen, C. Cheng, and J. Esparza, Eds., vol. 11781. Springer, 2019, pp. 48–64. [Online]. Available: https://doi.org/10.1007/978-3-030-31784-3_3
- [3] C. Oh, “Between SAT and UNSAT: the fundamental difference in CDCL SAT,” in *SAT 2015*, ser. LNCS, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 307–323. [Online]. Available: https://doi.org/10.1007/978-3-319-24318-4_23
- [4] A. Biere, “Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016,” in *SAT Competition 2016*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2016-1. University of Helsinki, 2016, pp. 44–45.
- [5] M. Järvisalo, M. Heule, and A. Biere, “Inprocessing rules,” in *IJCAR 2012*, ser. LNCS, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370. [Online]. Available: https://doi.org/10.1007/978-3-642-31365-3_28
- [6] M. Heule, M. Järvisalo, and A. Biere, “Revisiting hyper binary resolution,” in *CPAIOR 2013*, ser. LNCS, C. P. Gomes and M. Sellmann, Eds., vol. 7874. Springer, 2013, pp. 77–93. [Online]. Available: https://doi.org/10.1007/978-3-642-38171-3_6

- [7] C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li, “Clause vivification by unit propagation in CDCL SAT solvers,” *Artif. Intell.*, vol. 279, 2020. [Online]. Available: <https://doi.org/10.1016/j.artint.2019.103197>
- [8] R. J. Bayardo and B. Panda, “Fast algorithms for finding extremal sets,” in *SDM 2011*. SIAM / Omnipress, 2011, pp. 25–34. [Online]. Available: <https://doi.org/10.1137/1.9781611972818.3>
- [9] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, “An effective learnt clause minimization approach for CDCL SAT solvers,” in *IJCAI 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 703–711. [Online]. Available: <https://doi.org/10.24963/ijcai.2017/98>
- [10] A. Biere and C. Sinz, “Decomposing SAT problems into connected components,” *J. Satisf. Boolean Model. Comput.*, vol. 2, no. 1-4, pp. 201–208, 2006. [Online]. Available: <https://satassociation.org/jsat/index.php/jsat/article/view/26>
- [11] A. Nadel and V. Ryvchin, “Chronological backtracking,” in *SAT 2018e*, O. Beyersdorff and C. M. Wintersteiger, Eds., vol. 10929. Springer, 2018, pp. 111–121. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_7
- [12] S. Möhle and A. Biere, “Backing backtracking,” in *SAT 2019*, ser. LNCS, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 250–266. [Online]. Available: https://doi.org/10.1007/978-3-030-24258-9_18
- [13] H. Han and F. Somenzi, “On-the-fly clause improvement,” in *SAT 2009*, ser. LNCS, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 209–222. [Online]. Available: https://doi.org/10.1007/978-3-642-02777-2_21
- [14] Y. Hamadi, S. Jabbour, and L. Sais, “Learning for dynamic subsumption,” in *ICTAI 2009*. IEEE Computer Society, 2009, pp. 328–335. [Online]. Available: <https://doi.org/10.1109/ICTAI.2009.22>
- [15] M. Heisinger, M. Fleury, and A. Biere, “Distributed cube and conquer with Paracooba,” in *SAT 2020*, L. Pulina and M. Seidl, Eds. Springer, 2020, (Accepted).
- [16] M. J. H. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, “March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver,” in *SAT 2004*, ser. LNCS, vol. 3542. Springer, 2004, pp. 345–359. [Online]. Available: https://doi.org/10.1007/11527695_26
- [17] A. Biere, “CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018,” in *SAT Competition 2018*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2018-1. University of Helsinki, 2018, pp. 13–14.
- [18] K. Fazekas, A. Biere, and C. Scholl, “Incremental inprocessing in SAT solving,” in *SAT 2019*, ser. LNCS, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 136–154. [Online]. Available: https://doi.org/10.1007/978-3-030-24258-9_9

GlucoseEsbpSel: accelerate the search while pruning

Hakan Metin
Sorbonne Université
LIP6, UMR 7606, Paris, France
hakan.metin@lip6.fr

Souheib Baair
Université Paris Lumières
LIP6, UMR 7606, Paris, France
souheib.baair@lip6.fr

Vincent Vallade
Sorbonne Université
LIP6, UMR 7606, Paris, France
vincent.vallade@lip6.fr

Jo Devriendt
Lund University & University of Copenhagen
Lund, Sweden
jo.devriendt@cs.lth.se

Abstract—GlucoseEsbpSel combine the Glucose SAT solver with two techniques that handle symmetries dynamically. The first called effective symmetry breaking predicate (ESBP), prunes the search tree, while the second, called symmetric explanation learning (SEL), accelerates the traversal of this latter.

Index Terms—SAT, symmetry, breaking, learning, competition

I. INTRODUCTION

Many real-world problems exhibit symmetry, but the SAT competition and SAT race seldomly feature solvers that are able to exploit symmetry properties. Here, we propose a solver that exploits these properties in a complete dynamic way.

II. MAIN TECHNIQUES

The integrate to Glucose 4.0 [1] two complementary symmetry-based techniques: ESBP and SEL.

The idea of the ESBP approach is to break symmetries *on-the-fly*: when the current partial assignment can not be a prefix of a *lex-leader* of an equivalence class of assignments, a symmetry breaking predicate (sbp) that prunes this forbidden assignment and all its extensions is generated. It is then injected as a new (special) conflicting clause. The classical conflict analysis is then activated and a back-jumping is operated. The details of this approach can be found in [2].

The SEL approach is based on learning symmetric images of explanation clauses for unit propagations performed during search. A key idea is that these symmetric clauses are only learned when they would restrict the current search state, i.e., when they are unit or conflicting. So, this technique allows to accelerate the traversal of the search tree. The details of this approach can be found in [3].

III. SPECIAL ALGORITHMS, DATA STRUCTURES AND OTHER FEATURES

The symmetries of the treated instance are computed using the Bliss tool [4].

To be able to combine the two approach, an extra tagging system for the clauses is added to Glucose. It allows to distinguish the classical clauses from the sbp clauses: when

a clause is tagged to be an sbp or generated using an sbp, then the SEL approach has to be blocked from operating.

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

IV. AVAILABILITY

Source code and documentation for the combined approach is available under Glucose's license, and is available at https://github.com/lip6/ESBP_SEL/tree/experimental/core.

Besides, the standalone ESBP approach is implemented as a library, called cosy, that can be integrated with any CDCL-like solver. Cosy is released under GPL-v3 licence and is available at <https://github.com/lip6/cosy>.

ACKNOWLEDGMENT

We would like to thank the authors of Bliss, MiniSat [5] and Glucose.

REFERENCES

- [1] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, Jul. 2009, pp. 399–404.
- [2] H. Metin, S. Baair, M. Colange, and F. Kordon, "Cdclsym: Introducing effective symmetry breaking in sat solving," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 99–114.
- [3] J. Devriendt, B. Bogaerts, and M. Bruynooghe, "Symmetric explanation learning: Effective dynamic symmetry handling for sat," in *Theory and Applications of Satisfiability Testing – SAT 2017*, S. Gaspers and T. Walsh, Eds. Cham: Springer International Publishing, 2017, pp. 83–100. [Online]. Available: <https://bitbucket.org/krrt/glucose-sel>
- [4] 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*, D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, Eds. SIAM, 2007, pp. 135–149.

- [5] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Selected Revised Papers*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2004, pp. 502–518.

P-MCOMSPS-STR: a Painless-based Portfolio of MapleCOMSPS with Clause Strengthening.

Vincent Vallade*, Ludovic Le Frioux†, Souheib Baair*‡, Julien Sopena*§, Fabrice Kordon*

*Sorbonne Université, LIP6, CNRS, UMR 7606, Paris, France

†LRDE, EPITA, Le Kremlin-Bicêtre, France

‡Université Paris Nanterre, France

§INRIA, Delys Team, Paris, France

Abstract—This paper describes the solver P-MCOMSPS-STR submitted to the parallel track of the 2020's SAT Competition. It is a concurrent portfolio solver instantiated with the Painless (PARallel INSTantiabLe Sat Solver) framework and using MapleCOMSPS as core sequential solver.

I. INTRODUCTION

P-MCOMSPS-STR is a parallel SAT solvers built by instantiating components of the Painless parallel framework [1]. It is a portfolio-based [2] solver implementing a diversification strategy [3], fine control of learnt clause exchanges [4], using MapleCOMSPS [5] as a core sequential solver, and where learnt clause strengthening [6] has been integrated.

Section II gives an overview on Painless framework. Section III details the implementation of P-MCOMSPS-STR using Painless and MapleCOMSPS.

II. DESCRIPTION OF PAINLESS

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

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

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

A. Sequential Engines

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

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

B. Parallelization

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

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

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

C. Sharing

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

In Painless, solvers can export (import) clauses to (from) the others during the resolution process. Technically, this is done by using lock-free queues [7]. The sharing of these learnt clauses is dedicated to particular components called Sharers. Each Sharer is in charge of sets of producers and consumers and its behaviour reduces to a loop of sleeping and exchange phases.

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

III. P-MCOMSPS-STR

This section describes the overall behaviour of our competing instantiation named P-MCOMSPS-STR. Its architecture is highlighted in Fig. 1. It implements the Painless strengthening described in [8]. In the following, we highlight the outline.

A. MapleCOMSPS

MapleCOMSPS [5] is based on MiniSat [9], and relies on the classical VSIDS [10], and the more recently defined LRB [11] for its decision heuristics. These two are used in one-shot phases: first LRB, then VSIDS. Moreover, it uses Gaussian Elimination (GE) at preprocessing time.

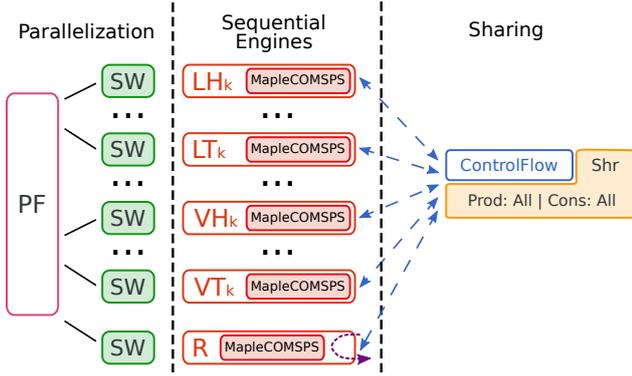


Fig. 1. Architecture of P-MCOMSPS-STR.

We adapt this solver for the parallel context as follows: (1) we parametrized the solver to select either LRB, or VSIDS for all solving process (noted respectively, L and V); (2) we added callbacks to export and import clauses; (3) we added an option to use or not the GE preprocessing; (4) we parametrized the solver to use as variable score comparator either $<$ or $<=$ (noted respectively head: H and tail: T).

B. Strengthenener

A *reducer* engine (R in Fig. 1) implements the algorithm introduced in [6].

We implemented the strengthening operation as a decorator of *SolverInterface*. This decorator is a *SolverInterface* itself that uses, by delegation, another *SolverInterface* to apply the strengthening, in the present case a MapleCOMSPS solver.

C. Portfolio and Diversification

P-MCOMSPS-STR is a solver implementing a basic portfolio strategy (PF), where one solver is used as a *reducer*, and the other underlying core engines are either LH , LT , VH or VT instances (i.e., combination of VSIDS or LRB, and head or tail).

For each type of instances, we apply a sparse random diversification similar to the one introduced in [3]. That is for each group of k solvers, the initial phase of a solver is randomly set according the following settings: every variable gets a probability $1/2k$ to be set to false, $1/2k$ to true, and $1 - 1/k$ not to be set.

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

D. Controlling the Flow of Shared Clauses

In P-MCOMSPS-STR, the sharing strategy `ControlFlow` is inspired from the one used by [3], [4]. We instantiate one *Sharer* for which all solvers are producers. It gets clauses from this producer and exports some of them to all others (the consumers).

The exchange strategy is defined as follows: each solver exports clauses having a LBD value under a given threshold (2 at the beginning). Every 1.5 seconds, 1500 literals (the sum of the size of the shared clauses) are selected from each

producers by the *Sharer* 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: respectively, less than 75% (1125 literals) and more than 98% (1470 literals).

E. Online Strengthening

The *reducer* engine is both a consumer and a producer of the sharer (*Shr*). It receives clauses from the different cores, strengthened them, in case of success it then exports them back. The sharing mechanism will then share this strengthened clauses to all the other solvers.

Since, a strengthened clause subsumes the original one, it is likely that cores will forget the original clause over time.

ACKNOWLEDGMENT

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

REFERENCES

- [1] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Painless: a framework for parallel sat solving," in *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] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maplecomsps lrb vsids, and maplecomsps chb vsids," pp. 20–21, 2017.
- [6] 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.
- [7] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 267–275, ACM, 1996.
- [8] 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.
- [9] N. Eén and N. Sörensson, "An extensible sat-solver," in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 502–518, Springer, 2003.
- [10] 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.
- [11] 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.

ManyGlucose 4.1-60

Hidetomo Nabeshima
University of Yamanashi, JAPAN

Katsumi Inoue
National Institute of Informatics, JAPAN

Abstract—ManyGlucose is a deterministic parallel SAT solver based on Glucose-syrup 4.1. In order to achieve reproducible behavior, ManyGlucose has a special mechanism called *delayed clause exchange* and accurate estimation of execution time of clause exchange interval between solvers.

I. INTRODUCTION

ManyGlucose 4.1-60 is a deterministic portfolio parallel SAT solver for shared memory multi-core systems. Given an instance, a deterministic solver has reproducible results in terms of solution (satisfying assignment or proof of unsatisfiability) and running time. ManyGlucose supports such reproducible behavior. The base solver is Glucose-syrup 4.1 [1] which is a non-deterministic parallel SAT solver. To achieve reproducible behavior, ManyGlucose has a special mechanism called *delayed clause exchange* and accurate estimation of execution time of clause exchange interval between solvers [2].

II. MAIN TECHNIQUES

ManySAT 2.0 [3] is the first parallel SAT solver that supports reproducibility. To achieve deterministic behavior, it periodically synchronizes all threads, each of which executes MiniSat 2.2 [4], before and after the clause exchange. The exchange interval is called a *period*. In ManySAT, all threads need to be synchronized periodically. Hence, waiting threads frequently occur in a many-core environment.

In order to reduce the idle time of threads, ManyGlucose uses the following two techniques [2]:

- 1) Delayed clause exchange: each thread receives learnt clauses acquired in m periods ago of the other threads. This eliminates the need to wait if the gap of the period of each thread is less than or equal to m , where m is an admissible delay, called *margin*.
- 2) Accurate estimation of execution time of period: In ManySAT, the length of a period is defined as the number of conflicts. The generation speed of conflicts fluctuates frequently since it is affected by the number and length of clauses. In ManyGlucose, two new definitions of a period are available. The first one is based on the number of literal accesses and the second one is based on the number of executions of blocks (statements enclosed in curly braces in C++).

From version 4.1-2 (SAT Competition 2018), the management of the clause database for exchange has been completely changed. In 4.1-2, there is one global clause database and mutual exclusion control is required to access

the database. In 4.1-60, each thread and each period has a clause database. As a result, the solver does not have to do mutually exclusive control to access the database.

III. MAIN PARAMETERS

We set the margin to 20 and use the block execution based period. The portfolio strategy of ManyGlucose is same as Glucose-syrup except that each thread uses different random seeds to hold the diversity of solvers. We submit ManyGlucose 4.1-60 with 32 threads and with 64 threads to Parallel track.

IV. AVAILABILITY

ManyGlucose 4.1-60 is developed based on Glucose-syrup 4.1. Permissions and copyrights of ManyGlucose follows that of Glucose-syrup. ManyGlucose can be downloaded at <https://github.com/nabesima/manyglucose-satcomp2020>.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP17K00300 and JP20K11934. In this research work we used the supercomputer of ACCMS, Kyoto University.

REFERENCES

- [1] G. Audemard and L. Simon, “Lazy clause exchange policy for parallel SAT solvers,” in *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014)*, LNCS 8561, 2014, pp. 197–205.
- [2] H. Nabeshima and K. Inoue, “Reproducible efficient parallel sat solving,” in *Proceedings of the 23th International Conference on Theory and Applications of Satisfiability Testing (SAT 2020)*, 2020, to appear.
- [3] 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.
- [4] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, LNCS 2919, 2003, pp. 502–518.

SLIME SAT Solver

1st Oscar Riveros

Research Architect at www.PEQNP.science

PEQNP

Santiago, Chile

oscar.riveros@peqnp.science

Abstract—SLIME 4 it's the evolution of the SAT Solver send to SAT Race 2019 (Version 1) [1] of the same name, integrate several techniques and improvements, for SAT Competition 2020 exist Cloud, Incremental, and Standard versions.

Index Terms—CNF, SAT, Cloud, Incremental

I. SLIME SAT SOLVER

Its state of the art SAT Solver, with the following general capabilities.

A. Geometric Temporal Rule

The internal flux of the solver is affected by a temporal switch that changes on geometric time, and this gives to the solver a small Random Behaviour depends on the performance of the machine, in practice present several advantages respect to the static execution.

B. The BOOST Heuristic

Boost Heuristic Based on the HESS algorithm, which is an Oracle-based deterministic black-box optimization algorithm, BOOST located on two strategic zones of the execution flux, this alternate according to the Exponential Temporal Rule, and manage the polarities of literals.

C. Distribution of polarities

For the standard version, the polarities of the literals initially distributed according to their parity; for the Cloud version, these polarities are uniformly randomized.

D. Cloud

SLIME Cloud is an MPI implementation that works like a portfolio sat solver but with SLIME as base, according to Randomization of Polarities and the Geometric Temporal Rule.

REFERENCES

- [1] Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions. / Heule, Marijn J.H. (Editor); Järvisalo, Matti (Editor); Suda, Martin (Editor). Helsinki : Department of Computer Science, University of Helsinki, 2019. 61 p. (Department of Computer Science Report Series B; Vol. B-2019-1).

Thanks to www.foresta.io for all these years of support.

TOPOSAT2

Thorsten Ehlers*, Mitja Kulczynski†, Dirk Nowotka† and Philipp Sieweck†

*German Aerospace Center (DLR)
Helmholtz Association, Hamburg, Germany
Email: thorsten.ehlers@dlr.de

†Department of Computer Science
Kiel University, Kiel, Germany
Email: {mku,dn,psi}@informatik.uni-kiel.de

Abstract—In this document we describe our parallel SAT solver TOPOSAT2 submitted to the SAT competition 2020 in the AWS Cloud track. We briefly review the key insights and the setup of our solver.

I. INSIGHTS OF THE SOLVER

TOPOSAT2 was mainly designed to run in a massively-parallel environment (> 1000 solver threads). Thus, we are curious to see how it performs in an AWS cloud system. It is built on top of Glucose 3.0, but uses a bug-fixed version of the lockless clause sharing mechanism from ManySAT [1] for communication on one compute node rather than the lock-based implementation from Glucose Syrup. The communication between nodes uses MPI.

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

A. Diversification

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

- Branching: Some solver threads use VSIDS, whereas other use LRB [2], as this branching heuristic was quite successful in the past SAT competitions. As VSIDS still works better on a significant amount of benchmarks, we use both.
- Restarts: We use the inner/outer restart scheme [3], Luby restarts, and the adaptive restart strategy from GLUCOSE [4].
- Learnt Clause DB management: Some solver threads use a scheme similar to the one suggested in [5]: Clauses with very low LBD (≤ 3) are stored permanently. Clauses of intermediate LBD are stored at least for some time, and there is a small activity-based clause storage. The LBD of clauses imported from other solver threads are initialised with the size of the clause. Thus, the clause must be used in order to update its LBD, and allowing it to be stored for a longer time. Some other solver threads use the default clause management strategy from GLUCOSE [6].

B. Lifting exported clauses

Wieringa et. al suggested to use some threads of a parallel SAT solver to strengthen learnt clauses [7]. Similarly, in [8]

some of the learnt clauses are strengthened during search. We use this technique when exporting clauses. Whenever one solver threads learns a clause of sufficiently low LBD, it is stored in an extra buffer. After the next restart, the clauses from this buffer are strengthened, and the results are exported to the other solver threads.

C. Parameter setup

The submitted version uses a variation of the clause import strategy of MANYSAT. During search, the trail size is monitored. Clauses are imported when some time has passed and the solver is somewhat close to the root of the search tree. In this way, we try to prevent the solver from backtracking too often when imported clauses are unit under the current assignment.

REFERENCES

- [1] Y. Hamadi, S. Jabbour, and L. Sais, “Manysat: a parallel SAT solver,” *JSAT*, vol. 6, no. 4, pp. 245–262, 2009.
- [2] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Learning rate based branching heuristic for SAT solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140.
- [3] A. Biere, “Picosat essentials,” *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.
- [4] G. Audemard and L. Simon, “Refining restarts strategies for SAT and UNSAT,” in *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Milano, Ed., vol. 7514. Springer, 2012, pp. 118–126.
- [5] C. Oh, “Between SAT and UNSAT: the fundamental difference in CDCL SAT,” in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Heule and S. Weaver, Eds., vol. 9340. Springer, 2015, pp. 307–323.
- [6] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, C. Boutilier, Ed., 2009, pp. 399–404.
- [7] S. Wieringa and K. Heljanko, “Concurrent clause strengthening,” in *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. Järvisalo and A. V. Gelder, Eds., vol. 7962. Springer, 2013, pp. 116–132.
- [8] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, “An effective learnt clause minimization approach for CDCL SAT solvers,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 703–711. [Online]. Available: <http://www.ijcai.org/Proceedings/2017/>

BENCHMARK DESCRIPTIONS

Main Track Benchmarks

Markus Iser
 Institute of Theoretical Informatics
 Karlsruhe Institute of Technology
 markus.iser@kit.edu

Abstract—We selected 400 benchmark instances for the main track of SAT Competition 2020. The selected instances are composed of 300 new submitted instances and 100 instances which have already been used in previous competitions.

I. INTRODUCTION

As in previous SAT competitions, the *BYOB rule* (bring your own benchmarks) required the participating teams to submit at least 20 benchmark instances.

Due to the immense feedback to our call for benchmarks we obtained a total of 1260 new benchmark instances, of which a total of 1012 instances satisfies our criterion of being interesting, i.e. they could not be solved by Minisat in less than 10 minutes.

Of the 1012 interesting new instances, we selected 300 instances for the competition by the procedure described below. We augmented this set by 100 instances of previous competitions, thus balancing the number of instances with *satisfiable*, *unsatisfiable* and *unknown* result.

II. BENCHMARK INSTANCE SUBMISSIONS

Per instance author, we included a maximum of 14 benchmark instances by using the following procedure. Let $I(a)$ be the set of instances submitted by author a . We created the partition $I(a) = I(a, \text{sat}) \cup I(a, \text{unsat}) \cup I(a, \text{unknown})$ of satisfiable, unsatisfiable or unknown subsets, respectively.

Per author a , we randomly selected 7 instances of $I(a, \text{sat})$ and 7 instances of $I(a, \text{unsat})$. In case of $|I(a, \text{sat})| < 7$ or $|I(a, \text{unsat})| < 7$, we added randomly selected instances of $I(a, \text{unknown})$.

By this procedure we obtained an initial selection of 308 benchmark instances, of which we knew 122 to be satisfiable, and 78 to be unsatisfiable, such that we randomly removed 8 satisfiable instances to get the final selection of 300 new benchmark instances.

Table I displays the numbers of submitted, interesting and finally selected instances split by problem family.

III. FINAL SET OF BENCHMARKS

As the number of satisfiable instances in the new set of benchmark instances is much larger than the number of unsatisfiable instances, we used the 100 old instances to balance the amounts of sat, unsat and unknown instances. Table II display the final numbers of instances selected.

In addition to this balancing criterion, we made sure that none of the the 100 randomly selected old instances belongs to an instance family which is already represented in the set of

submitted	interesting	selected	family
6	2	2	01-integer-programming
187	139	14	antibandwidth
40	20	13	baseball-lineup
393	333	14	bitvector
20	15	12	cellular-automata
38	18	7	cnf-miter
14	14	14	coloring
20	20	14	core-based-generator
18	13	13	cover
106	93	34	cryptography
20	19	7	discrete-logarithm
58	57	7	edge-matching
8	5	5	fermat
40	3	0	flood-it-puzzle
20	19	13	hgen
56	48	14	hypertree-decomposition
20	16	14	influence-maximization
20	20	9	lam-discrete-geometry
20	20	8	polynomial-multiplication
4	2	2	schur-coloring
20	20	12	station-repacking
23	19	7	stedman-triples
5	2	2	sum-subset
20	20	14	tensors
12	7	7	termination
20	16	14	timetable
16	16	14	tournament
36	36	14	vlsat
1260	1012	300	Σ

TABLE I
 FAMILIES AND AMOUNTS OF NEWLY SUBMITTED INSTANCES

new instances (see Table I), as far as the respective data was available and accessible. We also excluded randomly generated instances, planning instances (due to this years planning subtrack) and agile instances, and used “GBD Benchmark Database” (GBD)¹ to query for instances with the desired properties [1].

	SAT	UNSAT	UNKNOWN	Σ
new instances	114	78	108	300
old instances	21	57	22	100
Σ	135	135	130	400

TABLE II
 AMOUNT OF OLD AND NEW INSTANCES BY RESULT

REFERENCES

- [1] M. Iser and C. Sinz, “A Problem Meta-Data Library for Research in SAT”, Proceedings of Pragmatics of SAT 2018, Oxford, UK, July 7, 2018., pp. 144–152, 2018

¹<https://pypi.org/project/gbd-tools/>

Incremental Library Track Benchmarks

Markus Iser
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
markus.iser@kit.edu

Abstract—We used 6 applications with 50 benchmark instances per application to run the incremental library track.

I. INTRODUCTION

With the introduction of IPASIR (Re-entrant Incremental Solver API), the incremental library track took place for the first time in SAT Race 2015 [1], and then again in SAT Competitions 2016 and 2017. In SAT Competitions 2020, we run the track again and selected a total 300 benchmark instances for the 6 applications described below.

II. BENCHMARKS

A. Backbone Detection

The application *genipabones* reads a formula from a given file and transforms it using the dual rail encoding, i.e., it replaces each x by p_x and each \bar{x} by n_x and adds clauses of the form $(\overline{p_x} \vee \overline{n_x})$. Incrementally, each variable is then checked if it is a backbone variable or not. For this application, we selected 50 of the smallest and easiest satisfiable problems of previous SAT competitions.

B. Essential Variables Calculation

The application *genipaessentials* incrementally finds all the variables essential for the satisfiability of a given formula by testing each variable using the dual-rail encoded formula. For this application, we used the same easy satisfiable problems as in Section II-A.

C. Longest Simple Path Computation

The application *genipalsp* finds the longest simple path in a graph. For this application, we selected 50 instances of the smallest graphs provided by Balyo et al. [2].

D. MaxSAT

The application *genipamax* is a trivial partial MaxSAT solver based on adding activation literals to soft clauses and subsequent incremental optimization using a cardinality constraint [3] and assumptions. For this application, we selected 50 instances from MaxSAT Evaluation 2019¹.

E. QBF

ljtihad is a solver for Quantified Boolean Formulas (QBFs). The solver tackles the a formula iteratively, using counterexample-guided expansion [4]. For this application, we selected 50 instances from QBF Evaluation 2019².

F. Planning

Pasar is a planner which is based on the principles of counterexample guided abstraction refinement (CEGAR) [5]. For this application, we selected 50 *sas* planning instances.

REFERENCES

- [1] Tomáš Balyo, Armin Biere, Markus Iser, and Carsten Sinz, “SAT Race 2015”, *J. of Artif. Intell.*, vol. 241, pp. 45–65, 2016
- [2] Tomáš Balyo, Kai Fieger, Dominik Schreiber, and Christian Schulz, “Finding Optimal Longest Paths by Dynamic Programming in Parallel”, *Proceedings of the 12th Symposium on Combinatorial Search (SoCS 2019)*
- [3] Tobias Philipp and Peter Steinke, “PBLib A Library for Encoding Pseudo-Boolean Constraints into CNF” *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT 2015)*
- [4] Roderick Bloem, Nicolas Braud-Santoni, Vedad Hadzic, Uwe Egly, Florian Lonsing, and Martina Seidl, “Expansion-Based QBF Solving Without Recursion”, *Formal Methods in Computer Aided Design (FMCAD 2018)*
- [5] Nils Froleyks, Tomáš Balyo, and Dominik Schreiber, “PASAR Planning as Satisfiability with Abstraction Refinement”, *Proceedings of the 12th International Symposium on Combinatorial Search (SOCS 2019)*

¹<https://maxsat-evaluations.github.io/2019/>

²<http://www.qbflib.org/qbfeval19.php>

Planning Track Benchmarks

Nils Froleyks
Institute for Formal Models and Verification
Johannes Kepler University Linz
nils.froleyks@jku.at

Abstract—The benchmarks for the new planning track consist of classical planning problems encoded to SAT by *Madagascar* [1] and *PASAR* [2] as well as HTN planning problems encoded to SAT using *Tree-REX*. The original planning problems were used in the *IPC 2014* and *IPC 2018*. The generation script, including the used planners is available ¹.

I. INTRODUCTION

Classical planning is the problem of finding a sequence of actions – a plan – that transforms the world from some initial state to a goal state. In SAT-based planning the problem is encoded up to a certain number of *steps* (the *makespan*) as a Boolean formula F_i in such a way that F_i is satisfiable if and only if there is a plan with i steps or less. Depending on the encoding multiple actions can be executed in the same step. Therefore the minimum makespan i for which F_i is satisfiable depends on the problem and the used encoding.

In HTN planning the planner is provided with additional domain knowledge besides the problem description.

II. PLANNING BENCHMARKS

The classical planning benchmarks are selected from the *satisficing* and *optimal* tracks of the *International Planning Competitions 2014* ² and *2018* ³.

The HTN benchmarks were provided by the author of *Tree-REX*.

III. SAT-BASED PLANNERS

Madagascar provides multiple encodings to choose from. We used the default E-step encoding and the sequential encoding. PASAR uses the grounding procedure deployed by the well known planner *Fast Downward* [4] and allows the execution of multiple actions per step. *Tree-REX* was used to encode the HTN problems.

IV. INSTANCE NAMING

The benchmarks of planning track follow the following naming convention. For more details see the generation script.
`<SAT/UNSAT>_(encoding)_(pathToInstance)_(makespan).cnf`

¹https://satcompetition.github.io/2020/downloads/SAT2020_planning_generator.zip

²<https://helios.hud.ac.uk/scommv/IPC-14/repository/benchmarksV1.1.zip>

³<https://bitbucket.org/ipc2018-classical/domains>

TABLE I
NUMBER OF BENCHMARKS GENERATED BY EACH ENCODING.

Encoding	SAT	UNSAT
H Tree-REX	15	11
P PASAR	14	14
ME Madagascar E-step	5	10
MS Madagascar sequential	66	65
	100	100

REFERENCES

- [1] J. Rintanen, “Madagascar: Scalable planning with SAT” Proceedings of the 8th International Planning Competition (IPC-2014), pp. 1–5, 2014
- [2] N. Froleyks, T. Balyo and D. Schreiber, “PASAR—Planning as Satisfiability with Abstraction Refinement”, Twelfth Annual Symposium on Combinatorial Search, pp. 70–78 2019
- [3] D. Schreiber, D. Pellier, H. Fiorino and others, “Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning”, Proceedings of the International Conference on Automated Planning and Scheduling, pp. 382–390, 2019
- [4] M. Helmert, “The Fast Downward Planning System”, Journal of Artificial Intelligence Research (JAIR) 26, 2006

SAT Encodings for Discrete Logarithm Problem

Jingchao Chen

School of Informatics, Donghua University

2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China

chen-jc@dhu.edu.cn

Abstract—The discrete logarithm problem is to ask whether there exists an integer x such that $a^x = y \pmod n$, where a , y and n are integers given. In general, it is considered to belong to the intersection of the complexity classes NP, coNP, and BQP. Several algorithms in public-key cryptography assume that the discrete logarithm problem over chosen groups has no efficient solution.

I. INTRODUCTION

The discrete logarithm problem is formalized as follows:

Given three positive a , y and n , find an integer x such that $a^x = y \pmod n$.

This is considered to be computationally intractable. So far, no efficient classical algorithm solve it in polynomial time. The known efficient algorithms are usually inspired by similar algorithms for integer factorization. for example, baby-step giant-step, function field sieve, index calculus algorithm, number field sieve, PohligHellman algorithm, Pollard’s rho algorithm for logarithms, Pollard’s kangaroo algorithm. In 1997, Shor presented an efficient quantum algorithm [1]. Here we translate it into a SAT problem by an encoding. By our tests, the resulting SAT problem seems to be more difficult than the original problem.

II. ENCODING DISCRETE LOGARITHM PROBLEM BY FAST EXPONENTIATION

We use a basic fast exponentiation to translate the discrete logarithm problem into a SAT problem. Exponentiating by squaring is a basic method for fast computation of large positive integer powers of a number. This method is based on the fact that, for a positive integer x , we have

$$a^x = \begin{cases} a(a^2)^{\frac{x-1}{2}} & \text{if } x \text{ is odd} \\ (a^2)^{\frac{x}{2}} & \text{if } x \text{ is even} \end{cases}$$

Exponentiating by squaring uses the bits of the exponent to determine which powers are computed. If the bits of the exponent x is given, This method can be implemented in the pseudo-code shown in Algorithm 1.

It is easy to translate the multiplication of two numbers into a SAT problem. We do not encode modular arithmetic. Instead, we transform modular arithmetic into multiplication operation. Let $r = a \pmod n$. We have

$$a = qn + r$$

where q is a quotient of a divided by n . This expression can be converted to

$$a \pmod n = a - qn$$

Algorithm 1 Calculate the value of a^x after expanding the exponent in base 2

x has binary expansion $(x_m \dots x_2 x_1)_2$

$y = 1$

for $k = 1$ to m **do**

if $x_k = 1$ **then**

$y = a * y$

end if

$a = a * a$

end for

return y

Notice, $a \geq qn$. It is easy to see discrete logarithm can be done essentially by multiplication, addition and subtraction.

We produce 20 benchmarks by translating $a^x \pmod n$ in the different number of bits into the SAT problem. The bits range of a and x is from 9 to 32. The bits range of n is from 20 to 64. If using Pollard rho method, such discrete logarithm problems are easily resolved. However, such SAT problems seems to be hard.

REFERENCES

- [1] Shor Peter: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, *emphSIAM Journal on Computing*, 26 (5), pp.1484–1509, 1997.

Improving Directed Ramsey Numbers using SAT

David Neiman, Marijn Heule, and John Mackey
Carnegie Mellon University, Pittsburgh, United States

INTRO

A tournament is an orientation of a complete graph, or equivalently a directed graph D with no self-loops such that, for all pairs of distinct vertices u and v , exactly one of the edges uv or vu is in D . Intuitively, a tournament of order n represents the results of a round-robin tournament between n teams, where the existence of edge uv means team u beat team v in their head-to-head match. The exclusivity of uv and vu means that if team u beats team v , team v doesn't beat team u , and the inclusion of one of those edges reflects the fact that in a round-robin tournament, each team plays each other team. The non-existence of self-loops translates to the fact that no team plays itself.

A tournament is *transitive* if, for all vertices a , b , and c , the existence of edges ab and bc implies the existence of edge ac . If a tournament is not transitive, then it contains a directed cycle of length 3.

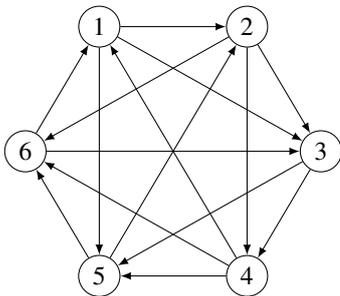


Fig. 1. The unique tournament of order 6 without a transitive sub-tournament of order 4.

DIRECTED RAMSEY NUMBERS

The directed Ramsey number $R(k)$ is the smallest integer n such that all tournaments on n vertices contain a transitive tournament of order k .

Directed Ramsey numbers were first introduced by Erdős and Moser [1]. In particular, they showed that $k \leq 2 \log_2(R(k)) + 1$ and also noted that $k \geq \log_2(R(k)) + 1$. In particular, this means $R(k)$ grows roughly as an exponential of k , with multiplier somewhere between $\sqrt{2}$ and 2. The precise limit of that multiplier is not known, but the literature has the following results on small directed Ramsey numbers:

- $R(2) = 2$
- $R(3) = 4$
- $R(4) = 8$
- $R(5) = 14$

- $R(6) = 28$ ([2])
- $32 \leq R(7) \leq 54$ ([3])

The tournaments of order 25, 26, and 27 that do not contain a transitive sub-tournament of order 6 are unique. We call them ST_{25} , ST_{26} , and ST_{27} , respectively. There are five tournaments of order 24 without a transitive sub-tournament of order 6.

SAT ENCODING

Lower bounds of directed Ramsey number $R(k)$ can be improved by constructing a complete directed graph without a transitive tournament of order k . The direct encoding into SAT would only use boolean variables for each edge with the truth value of a variable denoting the direction of the edge. Such an encoding uses many clauses and the resulting formulas are too large to improve the lower bound of $R(7)$. We therefore constructed a more compact encoding, which uses the fact that a transitive tournament lacks a directed cycle of length 3. For each triple of vertices in the graph, we introduce a new variable that is true if and only if the edges between them form a directed cycle. We use these auxiliary variables to encode the absence of transitive tournaments of order 7 more compactly.

BENCHMARKS

We submitted 16 benchmarks to the 2020 SAT Competition. The first 8 formulas encode whether ST_{25} , ST_{26} , ST_{27} , and the five tournaments of order 24 without a transitive sub-tournament of order 6 can be extended to a tournament of order 33 without creating a transitive sub-tournament of order 7. All these instances are satisfiable and thus establish an improved lower bound $R(7) > 33$. Figure 2 shows an example. The second 8 formulas are similar, but encode the existence of a tournament of order 34 without creating a transitive sub-tournament of order 7. It is not known whether these instances are satisfiable.

REFERENCES

- [1] P. Erdos and L. Moser, "On the representation of directed graphs as unions of orderings," *Math. Inst. Hung. Acad. Sci.*, vol. 9, pp. 125–132, 1964.
- [2] A. Sanchez-Flores, "On tournaments and their largest transitive subtournaments," *Graphs and Combinatorics*, vol. 10, no. 2-4, pp. 367–376, 1994.
- [3] —, "On tournaments free of large transitive subtournaments," *Graphs and Combinatorics*, vol. 14, no. 2, pp. 181–200, 1998.

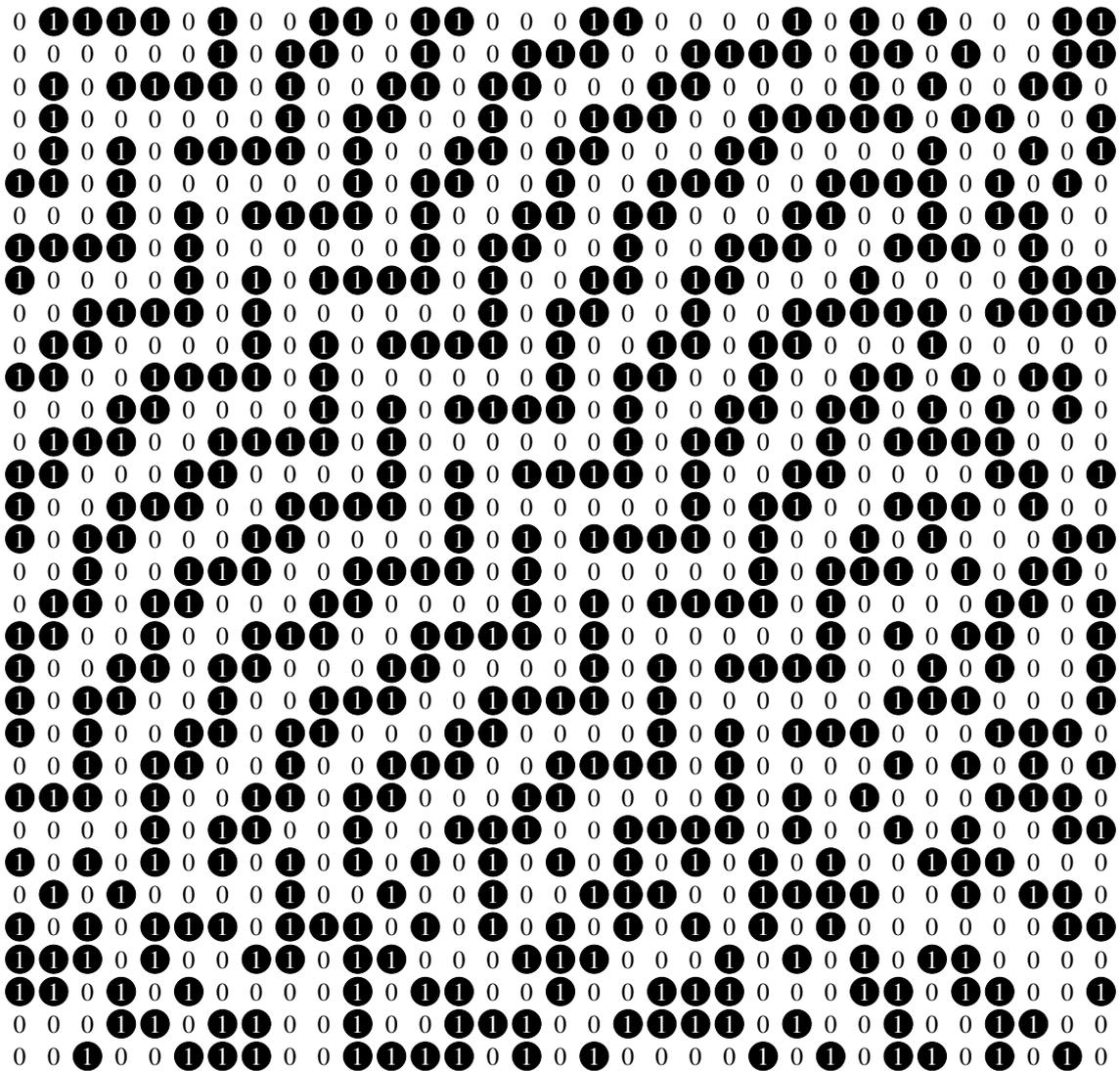


Fig. 2. Adjacency matrix of a 33-vertex tournament without a transitive sub-tournament of order 7.

Unit-Distance Strip Coloring Benchmarks

Peter Oostema, Ruben Martins, and Marijn Heule
Carnegie Mellon University, Pittsburgh, United States

INTRO

Coloring unit-distance strips is related to the Chromatic Number of the Plane (CNP), a problem first proposed by Nelson in 1950 [1]. The CNP asks how many colors are required to color the entire plane using the unit-distance constraint. Early results showed that at least four and at most seven colors are required.

Our benchmarks focus on coloring infinite strips with a given height instead of the entire plane. Table I summarizes the research in this direction. Early results for 3 and 4 colors are based on coloring the strips with rectangles of the same shape and with a height that equals the strip height [2]. A recent result improves the height for 4 colors using a much more involved pattern [3].

TABLE I: Colorings of infinite strips with different heights

# Colors	Height
3	$\sqrt{3}/2 \approx 0.866$ [4]
4	$2\sqrt{2}/3 \approx 0.94$ [4]
	$\sqrt{32/35} \approx 0.956$ [5] 0.959 [3]
5	$\sqrt{15}/4 \approx 0.968$ [2]
	$13/8 \approx 1.625$ [5]
	$9/2\sqrt{7} \approx 1.70084$ [6]]
6	$\sqrt{15}/2 + \sqrt{3} \approx 3.668$ [2]

SAT ENCODING

To translate our problem into SAT we first need to create a tessellation of the strip and make a graph from the shapes that contain points one distance away. Several polygons can be used to tessellate the plane. In this work, we tessellated the plane with squares and hexagons. Squares have the advantage of being able to draw straight lines, while hexagons are one of the roundest shapes which minimizes the number of conflicts between intersecting shapes.

Each shape was given two indices (one for each dimension) to identify its position as depicted in Figure 1a for hexagons. We build a conflict graph where each shape is a node and there is an edge between two shapes if there is a conflict between them, i.e. they cannot be colored with the same color. The graph is constructed by finding the conflicts from one shape to all others and repeating that pattern for each shape. For instance, squares that touch the red outer border area in Figure 1b are conflicting with the central (black) square. These conflicts can be found efficiently by searching downwards for the first shape that is one distance away. From here the adjacent shapes are traversed to find the complete set of shapes that are

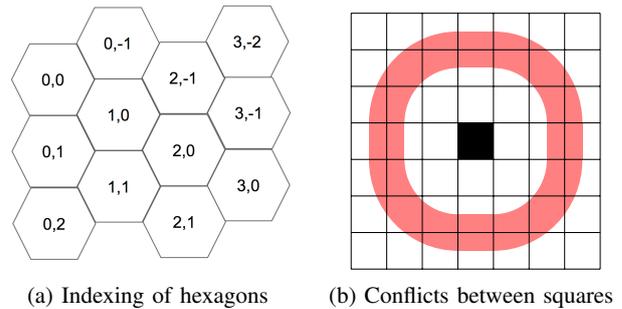


Fig. 1: Indexing and conflicts in a tessellation

one distance away. When a shape is found not in this range the traversal returns from that shape without adding a connection. The connections are stored as pairs of index offsets and easily translate to other shapes. For example, if two hexagons three rows apart share points one distance away then all hexagons will similarly share points with hexagons three rows above or below.

After constructing the graph, we follow a traditional encoding for graph coloring problems using SAT [7].

BENCHMARKS

We submitted various benchmarks to the 2020 SAT Competition based on strip coloring. The benchmarks encode whether strips of length 8 with different heights can be colored with 4, 5, or 6 colors using square or hexagon tiles. An example of a coloring that we found for a strip of height 1.64 using 5 colors and a hexagon tiling is shown in Figure 2. For more details regarding the benchmarks, we refer to the full paper [6].

REFERENCES

- [1] A. Soifer, *The Mathematical Coloring Book*. Springer New York, 2009.
- [2] M. Axenovich, J. Choi, M. A. Lastrina, T. McKay, J. Smith, and B. Stanton, “On the chromatic number of subsets of the euclidean plane,” *Graphs and Combinatorics*, vol. 30, no. 1, pp. 71–81, 2014. [Online]. Available: <https://doi.org/10.1007/s00373-012-1249-9>
- [3] F. Bock, “Epsilon-colorings of strips,” *Acta Mathematica Universitatis Comenianae*, vol. 88, no. 3, 2019.
- [4] B. L. Bauslaugh, “Tearing a strip off the plane,” *Journal of Graph Theory*, vol. 29, no. 1, pp. 17–33, 1998.
- [5] J. Parts, “Blog post on polymath16,” <https://dustingmixon.wordpress.com/2018/08/28/polymath16-tenth-thread-open-sat-instances/#comment-5488>, 2018, [Online]; accessed 21-February-2020.
- [6] P. Oostema, R. Martins, and M. Heule, “Coloring unit-distance strips using sat,” in *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, E. Albert and L. Kovacs, Eds., vol. 73. EasyChair, 2020, pp. 373–389. [Online]. Available: <https://easychair.org/publications/paper/69T4>

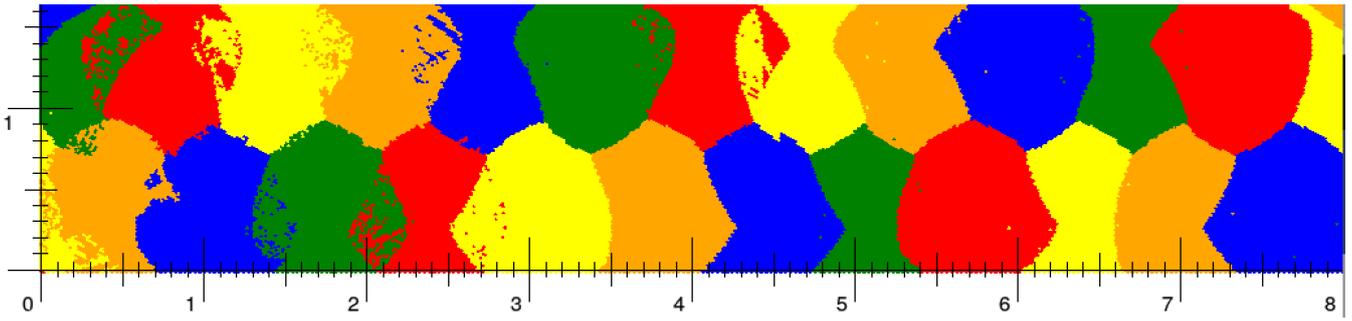


Fig. 2: A coloring of strip of height 1.64 using hexagon tiles and five colors.

- [7] A. V. Gelder, "Another look at graph coloring via propositional satisfiability," *Discrete Applied Mathematic*, vol. 156, no. 2, pp. 230–243, 2008.

Edge-matching Puzzles as a Feature-Cardinality Problem

Dieter von Holten

Abstract—This paper describes an encoding of edge-matching puzzles suitable for solving them with a SAT-solver. After introducing a minimalistic encoding, we add layers of redundant constraints, which vastly increase the size of the CNF, but reduce the time to solve.

I. INTRODUCTION

This approach first simplifies the problem by ignoring the border-tiles. Instead it just goes for a solution of the inner square. After a solution is found, the frame-tiles are placed manually to obtain a complete solution. The concept has been developed in the last year, although we find some overlaps with the paper of Marijn Heule from 2008 [MH08].

A. Grid Coloring

In the quest for a way to tackle the problem without falling back to backtracking, we came up with the idea of grid-coloring: we color the grid-lines. This immediately gives us a valid solution *by construction* - unfortunately most often for some other puzzle. When the obtained tile-set is equal to the puzzle-tile-set, we have the desired solution. Unfortunately, there is a huge number of possible colorings. A $nSize \times nSize$ grid has

$$nGridLines = 2 \times nSize \times (nSize + 1)$$

with c colors per gridline this results in

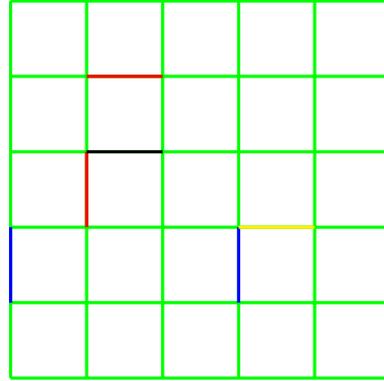
$$nColorings = c^{nGridLines}$$

For an Eternity 2 grade puzzle with $14 \times 14 = 196$ inner gridcells, that is 420 gridlines, and 17 colors this results in 6.145^{516} colorings - far too much to evaluate. This includes *all* colorings, even the not interesting ones like one green, one red, one blue, all others are yellow. We need something more restrictive - let's enforce a known edge-count per color.

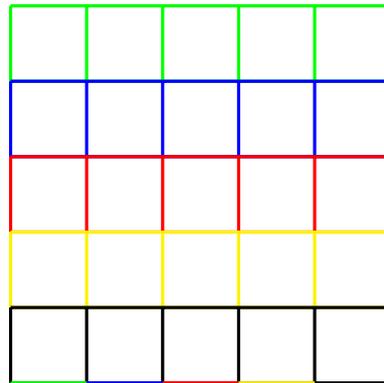
Let's assume we have 15 colors with 25 gridlines each, one color with 24 gridlines and one color with 23 gridlines. That would result in X tbd X colorings, much less than the first try.

But this still includes many uninteresting colorings, like: the first rows are all red, the next rows are all green, the next rows are blue and so on until the last few rows, which are all yellow. We need something more restrictive.

Now let's study the puzzle-tiles and see what we have there: we see (for example) that there are 3



wrong color-mix:
53 green, 2 red, 2 blue, 2 yellow,
1 black lines



good color mix,
but wrong distribution:
12 green, 12 blue, 12 red,
12 yellow, 12 black lines

Fig. 1: some grid-colorings

red/green corners, 7 green/blue and 4 blue/green corners, no black/white corners and no yellow/blue corners. We furthermore see 3 red-opposite-green edges, 2 black-opposite yellow-black edges, but no yellow-opposite-yellow edges. And we see around $nTiles$ 3-edge-patterns, most of them with just one occurrence. And we see $nTiles$ 4-edge-patterns.

II. THE FEATURES

The total number of features depends on the number of available colors. The key-insight here is, that the set of all possible edge-patterns is given by the Polya-Burnside necklace-enumeration of 4 beads and c colors. These form the *domain* of all tile-patterns. A concrete puzzle is a small *sample* of this domain.

TABLE I: Domain-size by color-count

c colors	patterns
10	2350
11	3696
12	5226
13	7189
14	9660
15	12720
16	16456
17	20961

Note, that these are the tile-patterns. The edge-patterns of a grid-cell are these patterns with 4 rotations.

Now let us dissect a tile-pattern into features:

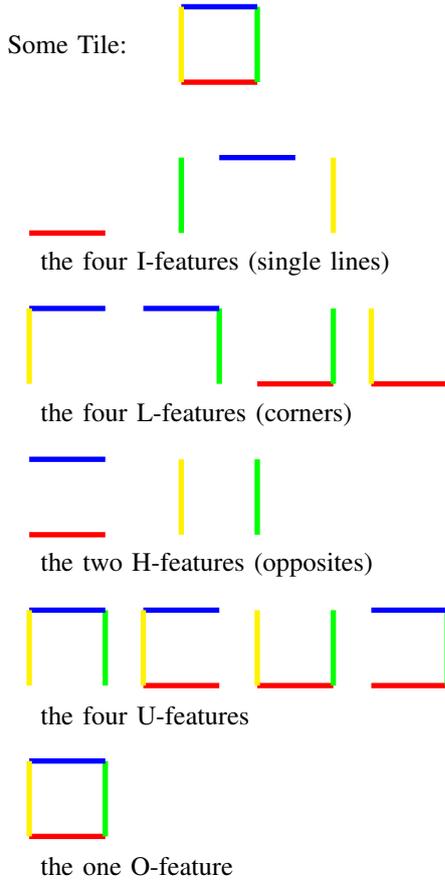


Fig. 2: Features

Table ?? shows a parts-list of features for various puzzles.

III. THE CONSTRAINTS

A. Cardinality = 0

This simply means 'the feature does not occur'. We don't have to deal with gridlines of non-existent colors. For the possible, but not used L-, H-, U- and O-features we create blocking clauses, with 4 rotations for most of them (a red-red H-feature has just two rotations, a blue-blue-blue-blue O-feature has just one rotation). We check for inclusion of the smaller within the larger features: a blocked red-red-corner prevents the existence of any full edge-pattern having a red corner. When a larger feature is 'covered/hidden' by a smaller one, the larger feature is ignored - no blocking clauses are generated. The result is the smallest possible set of the smallest possible blocking clauses.

These blocking clauses, written as one big conjunction, give a formula, which has $4 \times n_{Tiles}$ solutions - the valid tile-patterns in 4 rotations. These blocking clauses prevent the creation of invalid edge-patterns at the earliest possible point.

A grid-coloring with just these blocking-clauses in place would have only valid puzzle-tiles, albeit most likely with some duplicates and some missing.

B. Cardinality = 1

The most basic feature is a gridline of some color. We enforce, that a gridline has only one color with a 1ofN-constraint. For this, we use naive encoding.

From the tile-analysis we know the 'tile-defining feature' for each tile. We use the feature-existence-clauses to imply the tile-existence variables. We create a 1ofN-cardinality-constraint for each tile-existence-variable over all gridcells.

C. Cardinality > 1

From the puzzle-tile analysis we know the frequency of each color. The frequency of each color on the whole grid is implied by the solution: when we have one occurrence of each tile, we also have the right color-count. Therefore, it is redundant to enforce a count of 25 within the 420 gridline-color-variables. We make this constraint more precise by separating the gridlines around the edge of the inner square and within. It is an expensive redundant constraint, but it reduces solution-time.

We can impose cardinality constraints on each of the (not yet) used features. This comes with a considerable cost: per feature and per grid-cell we need 4 clauses (rotation!) of sizes between 3 and 5 to detect and hold the feature and we need a bitCounter of the size of the grid to enforce the proper count per feature.

IV. THE DATA-MODEL

We will now briefly describe the variables used to model the problem. To get some impression of the sizes, we assume a puzzle with $14 \times 14 = 196$ inner grid-cells, 420 grid-lines and 17 colors. We use these building-blocks to create the formula:

- a vector of variables, one variable per color, on each grid-line. A variable valued *true* designates the color of that gridline. A grid-line can have only one color, so we have a 1ofN-constraint over this vector, where n is the number of colors.
The example-puzzle costs $420 \text{ grid-lines} \times 17 \text{ colors} = 7140$ variables.
- a vector of variables, one variable per color, on each grid-cell *NoColorExistsVar*: a variable valued *true* says, that there is no edge of this color on this grid-cell. These variables are used in binary clauses to clear feature-exists and tile-exists-variables. The variables are the result of a tseitin-style AND of the 4 negated color-variables of a color on the 4 grid-lines around a grid-cell. In the example-puzzle, this costs $196 \text{ grid-cells} \times 17 \text{ colors} = 3332$ variables.
- a vector of variables, one variable per tile, on each grid-cell *TileExistenceVar*. A variable valued *true* defines the puzzle-tile sitting on this grid-cell. We do not need a local constraint here, because the blocking-clauses ensure, that we have only one valid tile per grid-cell. We need the variable just to see, which tile this actually is.

All variables for a certain tile, over all grid-cells, are collected in a vector. There is a 1ofN-constraint over this vector, which ensures, that we have only one instance of each tile. This costs $196 \text{ grid-cells} \times 196 \text{ tiles} = 3332$ variables.

A *TileExistenceVar* is set, when the defining feature exists on this grid-cell. The defining feature has a cardinality of 1, so when there is just one tile with a red-green corner, the grid-cell with a red-green corner holds that tile. We can define about 50 % of the tiles with just one binary L- or H-feature, most of the rest need a ternary U-feature, and just a few tiles need their O-feature for definition.

A *TileExistenceVar* is cleared, when a required color is not available on the grid-cell, that is, when the *NoColorExistsVar* becomes *true*. We furthermore use more precise clauses, which employ the (lack of the) edge-pattern: when the color of the north-edge is red, and the east-edge is not yellow, we clear the *TileExistenceVar* T77 on this grid-cell (see Fig. 3). This technique was used in [MH08].

- a vector of variables, one variable per feature, on each grid-cell *FeatureExistenceVar*. A variable valued *true* says, that this feature exists in this grid-cell. All we know is, that there are *some* features on each grid-cell, but we cannot

use a fixed constant. This *FeatureExistenceVar*-variables are set by clauses 'when this grid-line is red, and that one is green, the feature 'red-green-corner' exists. The *FeatureExistenceVar*-variables are cleared, when a required color is not available, that is, when the *NoColorExistsVar* becomes *true*. Note, that this is independent of rotation. A 4-colored grid-cell results in setting of 4 L-features, 2 H-features, 4 U-features and one O-feature.

These are about $13804 +$ the feature-existence variables. In addition, we have a considerable number of helper-variables within the large bit-counter constructs and the commander-encoded 1ofN-constraints.

V. OTHER REDUNDANT CLAUSES

We can easily derive obvious, but redundant constraints between tiles on neighboring grid-cells: when two tiles do not share a common color, they cannot sit on two neighboring grid-cells. This leads to a large number of binary blocking-clauses with two tile-existence-variables. This concept was already used in [MH08].

We extend this further to indirect neighbors: when tiles T77 and T88 have one common color (yellow), and sit beside each other, their rotation is fixed. This, in turn, fixes the colors of all their other edges. This in turn prevents the existence of certain tiles on adjacent grid-cells: the cell 6 on the left cannot contain a tile without a green edge, and the cell 3 on the right demands a tile with at least one orange edge. This leads to ternary blocking clauses, having two tile-existence variables describing the situation and the forbidden tile-existence variable.

	1	2	
6	T77	T88	3
	5	4	

tiles without required color are blocked, the tile on this cell must have at least one edge of
cell 1 : must have red
cell 2 : must have cyan
cell 3 : must have orange
cell 4 : must have black
cell 5 : must have blue
cell 6 : must have green

Fig. 3: blocked indirect neighbors

For size reasons, we cannot use all of these - we pick just random 25 % of these clauses. The full set of clauses for a 12×12 puzzle with a 10×10 inner grid

would have about 1×10^6 of these clauses, a 12×12 inner grid would have 4.4×10^6 of these and a 14×14 inner grid would have about 5.7×10^6 of them.

VI. THE BIT-COUNTER

The bit-counter is a major component of this problem-modelling. We collect the interesting variables in a vector and create some logic to count the set bits into some result-vector (integer). This result is then enforced to be equal to a constant. We use two kinds of bit-counters, composed of two kinds of adders:

- a bit-counter, composed of layers of adders, which reduce the intermediate results into the final value. This layout is described in [1] and [Wikipedia].
- a bit-counter, in which the bit-vector is broken into 7-bit wide chunks. The bit-count in a chunk is determined by some logic, without adders. The 3-bit results of the chunks could be added together for the final result, but here we build 3 new vectors from the result-bits: all bit0 -bits go into one vector, the bit1 - bits go into the second vector and bit2- bits go into a third vector. Then these 3 vectors are again 'recursively' bit-counted. The benefit is a reduced count of adders. We call this 'folded bit-counter'.
- one kind of full-adder is composed of gates and helper-variables, as documented in [Wikipedia].
- the other kind of full-adder is composed of just logic, without variables, so to say a truthable implementation.

The size of the input-vector is usually the number of grid-cells, that is from 100 for a 10×10 inner grid to 196 for a 14×14 grid. The number of required bits is usually small.

VII. ALTERNATIVE IMPLEMENTATIONS

As a benchmark for a SAT-competition it is obvious, that we provide CNF-formulations. The next implementation coming to mind could be SMT-formulas. Here, we have boolean as well as integer capabilities at hand. The blocking clauses and the feature-detection would be done with booleans, while the counting would be done with integers. Earlier experiments have found unsatisfactory performance with this mixed approach. Probably the SMT-world needs some decent benchmark-problems.

Another option would be the formulation with some 'higher-level' language like Picat. We have no experience with that.

Still another approach could be the implementation of a dedicated solver. That would turn the solution into some rule-based machinery, in which a fired rule increments a feature-count. When the allowed limit is surpassed, it backtracks and picks another color-assignment. The bit-counting mess would completely

disappear. The rule-handling can be implemented quite efficiently, however stuff like 'conflict based rule-learning' and restarts is currently out of reach.

VIII. SOME PARTS-LIST

The problem-generator places some statistics in the header of the of the CNF-file, let's have a look. Here we have the output for some 12×12 puzzle. It has an inner grid of $10 \times 10 = 100$ gridcells (and tiles). It has a total of 220 gridlines, 180 inner and 40 on the border. The n_of_180 and n_of_40 constraints are the color-count constraints for the inner grid and the border. The n_of_100 constraints are cardinality-constraints of some features, for example, we have 4 different features showing up 9 times. The 220 1_of_10 constraints are the one-color-per-gridline constraints. The weird 1_of_93 constraint is a tile-related constraint, ignoring the 7 hint-tiles.

```
var: 17_of_180-constraint : 2
var: 6_of_40-constraint : 1
var: 1_of_40-constraint : 2
var: 20_of_180-constraint : 1
var: 9_of_100-constraint : 4
var: 16_of_180-constraint : 2
var: 18_of_180-constraint : 3
var: 3_of_40-constraint : 1
var: 5_of_40-constraint : 1
var: 10_of_100-constraint : 1
var: 7_of_40-constraint : 1
var: 1_of_93-constraint : 93
var: 8_of_100-constraint : 2
var: 1_of_100-constraint : 100
var: 2_of_40-constraint : 1
var: 19_of_180-constraint : 1
var: 11_of_100-constraint : 1
var: 4_of_40-constraint : 4
var: 1_of_10-constraint : 220
var: FullAdderCount : 14023
var: 21_of_180-constraint : 1
var: 2_of_100-constraint : 84
```

IX. SUMMARY

The end is near - this is the way to go. The grid-coloring approach intuitively makes sense, and the feature cardinality enforcing is straight forward. The blocking-clauses for invalid edge-patterns and the indirect neighbor-constraints have a strong local influence on the available choices around a grid-cell. The cardinality-enforcement of color-lines and features provides global influence of any decision made. All of this is redundant, and would have been learnt anyway during solving.

However, it becomes visible, that the problem-formulation gets *really* big, the full E2 model can have in the order of 500 mio clauses and more than 500000 variables. Some hurdles remain:

- the most suitable bit-counter-implementation must be found. This may not be the smallest one, it needs to be a good fit to the solver.

- we can obtain huge numbers of constraints from the feature-cardinality and the invalid-neighbors. Is it really beneficial to use *all* of them? A search for a 'sweet spot' should be run, using various percentages of the available constraints. This must be carefully planned, otherwise the search for a 'good mix of ingredients' consumes more resources than solving the real thing.

X. THE FILES

The files were generated in 'series'. The puzzles are reused, that is a p1-14x14 is the same puzzle in different bit-counter and constraint selections.

- series 1, series 2 - omitted
- series 3: uses the standard bit-counter with standard adders, 20 % of the indirect-neighbor clauses and 50 % of the feature-constraints are used.
- series 4: uses the standard bit-counter with the no-variables adders, 20 % of the indirect-neighbor clauses and 50 % of the feature-constraints are used.
- series 5: uses the 'folded bit-counter' with the no-variables adders, 20 % of the indirect-neighbor clauses and 50 % of the feature-constraints are used.
- series 6: uses the 'folded bit-counter' with the standard adders, 20 % of the indirect-neighbor clauses and 50 % of the feature-constraints are used.
- series 7: uses the 'folded bit-counter' with the standard adders, 100 % of the indirect-neighbor clauses and 50 % of the feature-constraints are used.
- series 8: uses the 'folded bit-counter' with the standard adders, 50 % of the indirect-neighbor clauses and 100 % of the feature-constraints are used.

The filename is build as:

```
b <year><month><problemId>-<size> x
<size> c <colors> h <hintTiles>.cnf
```

REFERENCES

- [1] Henry S. Warren Jr, Hacker's delight 2013
[MH08] Marijn J.H. Heule, Solving edge-matching problems with satisfiability solvers. 2008
[Wikipedia] Wikipedia, tbd. year

Population Safety: A SAT Benchmark based on Elementary Cellular Automaton

Md Solimul Chowdhury, Martin Müller, Jia-Huai You
 Department of Computing Science, University of Alberta
 Edmonton, Alberta, Canada.
 {mdsolimu, mmueller, jyou}@ualberta.ca

Abstract—A *population* is an one-dimensional grid of $n \geq 1$ organisms, where each organism evolves between being alive (1) and dead (0) across chronological time steps by following a fixed rule of evolution. At any time step $t \geq 1$, the combined states of n organisms represent the state of the population at t . At t , a population is under the *threat of extinction*, if the number of alive organisms falls below $n * (P/100)$, where $0 < P \leq 100$ and *safe*, otherwise. We refer the above constraint as *safety constraint*.

In our proposed SAT benchmark *Population Safety (PS)*, given a population of n organisms, a maximum time step T , a safety threshold P , we verify if a population evolves safely at each time step upto T by following a fixed rule of evolution. For the SAT competition 2020, we have submitted 20 instances of the PS benchmark¹.

I. POPULATION SAFETY AS A CELLULAR AUTOMATON

State evolution in the Population Safety (PS) problem represents the state evolution of cells in finite elementary cellular automaton (CA) [3], with respect to the additional *safety constraint* at each time step.

In an elementary CA, at time step $t + 1$, the state of a cell c , which has cell l (resp. r) as its left (resp. right) neighbour, is computed based on a boolean combination the states of c , l , and r at time t . There are $2^3 = 8$ combinations of boolean values for l , c , and r at t , for each of which, there are 2 ways to set the value of the state of c at $t + 1$. Hence, there are $2^8 = 256$ ways to set the new state of the c at $t + 1$. Each of these 256 ways are called *rules* [3] for a given elementary CA.

We consider *Rule 30* [4] for the PS problem, which is known to produce chaotic patterns over time. At time $t + 1$, for a given center cell (*center*), its left (*left*) and right (*right*) neighbours, Rule 30 computes the state $center^{t+1}$ of the *center* cell as follows:

$$center^{t+1} \leftarrow left^t \text{ XOR } (center^t \text{ OR } right^t)$$

Figure 1 (taken from [4]) shows the evolution scheme for Rule 30. which is known to exhibit chaotic behavior for some initial states. Figure 2 shows such a chaotic evolution of a CA that follows Rule 30 (also taken from [4]).

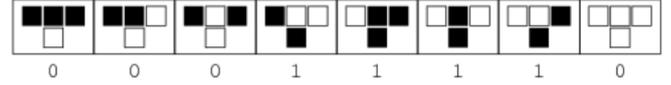


Fig. 1: State evolution for the center cell for Rule 30; black cells represents alive (1) cells, white cells represents dead (0) state.

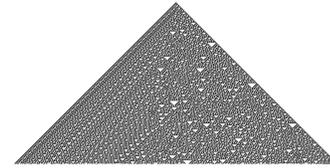


Fig. 2: Emergence of chaotic behavior with Rule 30

II. SAT ENCODING OF THE PS PROBLEM

A. PS as SAT Problem

Given a population of $n \geq 1$ organisms, a maximum time step $T \geq 2$, and a safety threshold $0 < P \leq 100$, the task of the PS problem is to determine if the population can evolve upto T by following Rule 30, with respect to the following safety constraint, *safety*:

safety: Total number of alive organisms in every time step $1 \leq t \leq T$ is at least $n * (P/100)$.

We can encode the population safety problem as a SAT formula. Let s_i^t be the state of the current cell i , where $1 \leq t \leq T$ and $1 \leq i \leq n$. Then we have the SAT encoding F_{PS} of the PS problem as follows

$$F_{PS} = F_{evolution} \cup F_{safety} \cup F_{boundary}$$

, where, $F_{evolution}$, F_{safety} , and $F_{boundary}$ are defined as follows:

$$F_{evolution} : \bigwedge_{t=1}^T \bigwedge_{i=1}^n (s_i^{t+1} = (s_{i-1}^t \oplus (s_i^t \vee s_{i+1}^t)))$$

$$F_{safety} : \bigwedge_{t=1}^T \sum_{i=1}^n s_i^t \geq n * (P/100)$$

$$F_{boundary} : \bigwedge_{t=1}^T \neg s_0^t \wedge \neg s_{n+1}^t$$

¹The code for PS instance generator is available at [1].

Over T time steps,

- $F_{evolution}$ encodes the evolution of the population of n organisms that follows Rule 30.
- F_{safety} encodes the population safety constraint.
- $F_{boundary}$ encodes the assertion that left neighbor (resp. right neighbor) of the leftmost (resp. rightmost) organism (resides outside of the *boundary* of a given population) of the population is always dead (0).

F_{PS} is SATISFIABLE, if the population can safely evolve upto time step T , otherwise, it is UNSATISFIABLE.

III. PROBLEM MODELING AND INSTANCE GENERATION FOR THE PS BENCHMARKS

A. Problem Modeling

picat [2] 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 PS benchmark, we use this CNF generation feature of **picat**. First, we modelled the PS problem in a **picat** program $picat_{PS}$. Then, for a given set of parameter values for (T, n, P) , we use this $picat_{PS}$ model to generate CNF F_{PS} by exploiting the CNF generation functionality of **picat**.

B. Instance Generation

We have performed a grid search with the $picat_{PS}$ over the parameter space of the PS model to generate 20 instances, 10 of which are interesting. Table I shows the experimental evaluation of these instances with MiniSAT, with 5,000 seconds timeout.

TABLE I: Results with MiniSAT for the 20 PS instances submitted for SAT Competition-2020.

Instance	Parameters			MiniSAT Time	Type
	T	n	P		
ps_200_300_70	200	300	70	550.85	UNSAT
ps_300_311_20	300	311	20	2969.23	SAT
ps_200_321_70	200	321	70	916.73	UNSAT
ps_300_312_20	300	312	20	3079.58	SAT
ps_200_310_70	200	310	70	1124.06	UNSAT
ps_300_322_20	300	322	20	1664.32	SAT
ps_200_305_70	200	305	70	2275.13	UNSAT
ps_300_314_20	300	314	20	2989.21	SAT
ps_200_306_70	200	306	70	2881.29	UNSAT
ps_200_317_70	200	317	70	Timeout	UNSAT
ps_300_300_70	300	300	70	Timeout	UNSAT
ps_200_319_70	200	319	70	Timeout	UNSAT
ps_200_323_70	200	323	70	Timeout	UNSAT
ps_300_301_30	300	301	30	Timeout	SAT
ps_200_318_70	200	318	70	Timeout	UNSAT
ps_200_301_70	200	301	70	Timeout	UNSAT
ps_200_307_70	200	307	70	Timeout	UNSAT
ps_200_309_70	200	309	70	Timeout	UNSAT
ps_200_312_70	200	312	70	Timeout	UNSAT
ps_200_316_70	200	316	70	Timeout	UNSAT

REFERENCES

- [1] Population Safety (PS) SAT Benchmark Instance Generator, https://figshare.com/articles/Population_Safety_PS_SAT_Benchmark_Instance_Generator/12218492, Accessed: 2020-04-29
- [2] Picat, <http://picat-lang.org/resources.html>, Accessed: 2020-04-09
- [3] Stephen Wolfram, A new kind of science. Wolfram-Media 2002, ISBN 978-1-57955-008-0, pp. I-XIV, 1-1197.
- [4] Rule 30 , <https://mathworld.wolfram.com/Rule30.html>, Accessed: 2020-04-09

Finding exact solutions to time-constrained variant of Influence Maximization Problem via SAT

Stepan Kochemazov
 Laboratory 6.2, ISDCT SB RAS
 Irkutsk, Russia
 Email: veinamond@gmail.com

Abstract—This document describes the benchmarks submitted to SAT Competition 2020 that encode problems of finding solutions to time-constrained variant of the well-known Influence Maximization Problem under deterministic Linear Threshold model with equivalent weights.

I. INTRODUCTION

The Influence Maximization Problem (IMP) [1] is one of the most well known problems in modern network sciences. Informally, it deals with the information dissemination in complex networks. It assumes that there is some network, the vertices of which are weighted. In the most simple case the weights are binary. The key component is a model of how information spreads, i.e. the rule according to which the weights of the vertices are recalculated. These rules usually involve several parameters assigned to network nodes and arcs. IMP is to find the so-called *seed set* of vertices such that their *influence spread* (the amount of network vertices to which they spread their information according to the used model) is maximized. There are many possible interpretations of IMP depending on the used parameters and information spread model. One of them is the modeling of the conforming behavior [2]. Other applications include viral marketing, voting, etc.

II. FORMAL DEFINITIONS

Denote a network as $G = (V, E)$, where V is a set of vertices, $|V| = n$ and E is the set of edges. Let us denote the set of vertices adjacent to v_i as its *neighborhood*:

$$N_i = \{v_k \in V : (v_k, v_i) \in E\}$$

The Deterministic linear threshold model [1], [2] with equivalent weights specifies that each network vertex $v_i \in V$ is assigned the so-called threshold θ_i , $1 \leq \theta_i \leq |N_i|$. Denote the weight of vertex v_i at time moment t as $w_i(t)$. The weights of vertices in the considered model are recalculated synchronously using the following formula:

$$w_i(t+1) = \begin{cases} 1 & \text{if } \sum_{v_k \in N_i} w_k(t) \geq \theta_i \\ 0 & \text{otherwise.} \end{cases}$$

Let us refer to the Boolean vector containing network vertices' weights at some time moment as to *network state* at this moment:

$$W(t) = (w_1(t), \dots, w_n(t)).$$

The research was funded by Russian Science Foundation (project No. 16-11-10046).

The specified model is a variant of a Synchronous Boolean Network [3].

We consider the following constrained variant of the Influence Maximization Problem: to find an initial configuration of weights $W(0)$ with minimal Hamming weight, such that after H time moments the Hamming weight of the network state will be at least L . The values of the parameters used in benchmarks were set as follows: $H = 10$, $L = 0.8 \times |V|$.

III. ENCODINGS

It is clear, that the corresponding problem can be naturally encoded to SAT using one of the plethora of encodings for cardinality constraints. The constructed benchmarks are built using the *Totalizer* encoding [4]. The tests were produced in the following manner: first for a given network a greedy algorithm from [5] is launched. Its output is a configuration W^{greedy} . We encode to SAT the process of information dissemination that starts from $W(0)$ and ends at $W(10)$, and add constraint on the final state:

$$Hamming_weight(W(10)) \geq 0.8 \times |V|.$$

Then we construct two SAT instances that differ in the value of the *RHS* constant at the right hand side of the constraint:

$$Hamming_weight(W(0)) \leq RHS$$

For one instance $RHS = |W^{greedy}| - 1$, for another $RHS = |W^{greedy}| - 2$.

IV. NETWORKS

The constructed SAT instances are relatively hard for state-of-the-art SAT solvers even when the dimension is small. For our benchmarks we took several fragments of the Twitter social network with number of vertices from 60 to 90. The Twitter fragments were taken from the Stanford network repository [6], [7]. The vertices' thresholds were picked as

$$\theta_i = \frac{|N_i|}{2} + rand() \bmod \left(\frac{|N_i|}{2} \right)$$

V. INSTANCE NAMING SCHEME

The benchmarks all follow the same naming convention:

$$DLTM_twitterI_N_RHS.cnf,$$

where DLTM is the abbreviation of Deterministic Linear Threshold Model, *twitterI* is the identifier of a network

fragment, N is the number of vertices in a corresponding network and RHS is the value in the constraint.

COMMENTS

It is clear that for the same network, the lower the value of the RHS constant, the harder is the problem. The attempt was made to pick such tests that the instance with $RHS = |W^{greedy}| - 1$ is satisfiable and the one with $RHS = |W^{greedy}| - 2$ is unsatisfiable, so that the solution found for $RHS = |W^{greedy}| - 1$ is the exact one, but it was not always possible.

REFERENCES

- [1] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03, 2003, pp. 137–146.
- [2] M. Granovetter, "Threshold models of collective behavior," *American Journal of Sociology*, vol. 83, no. 6, pp. 1420–1443, 1978.
- [3] S. Kochemazov and A. Semenov, "Using synchronous boolean networks to model several phenomena of collective behavior," *PLOS ONE*, vol. 9, no. 12, pp. 1–28, 12 2014.
- [4] O. Bailleux and Y. Boufkhad, "Efficient cnf encoding of boolean cardinality constraints," in *CP'03*, 2003, p. 108–122.
- [5] S. Kochemazov and A. A. Semenov, "Computational study of time constrained influence maximization problem under deterministic linear threshold model for networks with nonuniform thresholds," in *MIPRO 2019 proceedings*. IEEE, 2019, pp. 1121–1125.
- [6] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," <http://snap.stanford.edu/data>, Jun. 2014.
- [7] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *NIPS'12*, vol. 1, 2012, p. 539–547.

Baseball Lineup Selection Benchmarks Description at SAT Competition 2020

Randy Hickey
University of Toronto

Nick Feng
University of Toronto

Fahiem Bacchus
University of Toronto

Abstract—This document describes the SAT benchmarks we are submitting to SAT 2020. We generated benchmarks that would select a baseball lineup from a large pool of players based on some constraints.

I. DATA

We compiled a list of all baseball hitters from the 2019 MLB season and assigned each a number of boolean attributes based on their statistics. We eliminated all players that did not play in more than 80 games or appear in enough at bats to qualify for a batting average. This left 323 players. We then gave each player a 1 for a category if they were within 20% of the league leader and a 0 otherwise. We used a total of 16 statistical categories. Additionally, we gave each player a 1 for each position they played during the season and a 0 otherwise.

The data was parsed in this way to be a rough method of selecting a good lineup of players from a large pool that is balanced in every category, but there are many other ways one could select the data.

II. SELECTION

We then added constraints to the solver to determine whether or not a lineup of n players exists such that at least m of the players are within 20% of the league leader in every single category. Boolean variables x_i are constructed representing whether player i is selected or not, and Boolean variables y_{ij} are constructed where y_{ij} will equal 1 when both x_i is 1 and player i covers the j th category.

The graph of attributes is encoded into CNF in the standard way, at-least-k constraints are added on the y variables to ensure that each category is covered at least m times, and exactly-k constraints are added on the x variables to ensure that exactly x players are selected for the lineup.

For some of the benchmarks, we also enforce that, in addition to the above constraints, the selected lineup has at least m players who play every position. Such benchmarks are given the suffix "andpositions.cnf". At-least-k constraints are used to enforce this in the same way as for the statistical categories above.

III. TOOLS

To assist in generating the CNF formulas and adding cardinality constraints, we used Pysat [1]. The sequential cardinality encoding was used for both the exactly-k and at-least-k constraints.

REFERENCES

- [1] A. Ignatiev, A. Morgado, and J. Marques-Silva, "Pysat: A python toolkit for prototyping with SAT oracles," in *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, ser. Lecture Notes in Computer Science, O. Beyersdorff and C. M. Wintersteiger, Eds., vol. 10929. Springer, 2018, pp. 428–437. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26

Duplex Encoding of Antibandwidth Feasibility Formulas Submitted to the SAT Competition 2020

Katalin Fazekas¹ Markus Sinnl² Armin Biere¹ Sophie Parragh²

¹ Institute for Formal Models and Verification ² Institute of Production and Logistics Management
Johannes Kepler University Linz, Austria

I. THE ANTIBANDWIDTH PROBLEM

This benchmark set is originating in our recent work on the antibandwidth problem (in short ABP) presented in [1]. The ABP is a max-min optimization problem where, for a given graph $G = (V, E)$, the goal is to assign a unique label from the range $[1, \dots, |V|]$ to each vertex $v \in V$, such that the smallest difference between labels of neighbouring nodes is maximal. Applications of the ABP include for example scheduling, obnoxious facility location, radio frequency assignment.

To solve the ABP, in [2] an iterative solution method was proposed, where each iteration asked whether there exists a labelling to the graph, s.t. the smallest difference between labels of neighbours is greater than k . Finding the highest k where the answer is still affirmative determines the optimal solution of the ABP. In [1] we slightly refined the proposed formalization of [2] s.t. the question of each iteration can be stated as combinations of at-most-one constraints sliding over k -long sequences of binary variables. All in all, a feasibility query for a given k consists of the following constraints:

$$\forall \ell \in \{1, \dots, |V|\} : \sum_{i \in V} x_i^\ell = 1 \quad (\text{LABELS})$$

$$\forall i \in V : \sum_{\ell \in \{1, \dots, |V|\}} x_i^\ell = 1 \quad (\text{VERTICES})$$

$$\forall \{i, i'\} \in E, 1 \leq \lambda \leq |V| - k : \quad (\text{OBJ}_k)$$

$$\sum_{\ell=\lambda}^{(\lambda+k)} x_i^\ell \leq 1 \wedge \sum_{\ell=\lambda}^{(\lambda+k)} x_{i'}^\ell \leq 1 \wedge \left(\sum_{\ell=\lambda}^{(\lambda+k)} x_i^\ell \leq 0 \vee \sum_{\ell=\lambda}^{(\lambda+k)} x_{i'}^\ell \leq 0 \right),$$

where binary variables $x_i^\ell = 1$ ($i \in V, \ell \in \{1, \dots, |V|\}$) if and only if vertex i is assigned label ℓ . Constraints (LABELS) make sure that each label is used only once and constraints (VERTICES) ensure that each node $i \in V$ gets assigned one label. Constraints (OBJ_k) forbid for each neighbouring node to assign two labels from any k -wide range of labels.

II. SAT ENCODING

In [1] we defined a so-called staircase at-most-one constraint set (SCAMO) over a sequence of Boolean variables

Supported by the Austrian Science Fund (FWF) under projects W1255-N23, S11408-N23 and by the LIT AI Lab funded by the State of Upper Austria.

$X = \langle x_1 x_2 \dots x_n \rangle$ for a given width k (where $1 < k \leq n$) as

$$\text{SCAMO}(X, k) = \bigwedge_{i=0}^{(n-k)} \left(\sum_{j=i+1}^{(i+k)} x_j \leq 1 \right).$$

Then we proposed a linear size SAT encoding of this constraint set. The main idea of the encoding is to slice up the n -long sequence of Boolean variables into M k -long sequences and build up the complete SCAMO constraint set as a combination of smaller at-most-one and at-most-zero constraints, such that these smaller constraints can be efficiently shared and reused. Each smaller at-most-one and at-most-zero constraint is translated to SAT with standard BDD-based methods (see [3], [4]). However, each of these constraints is encoded twice, first considering a variable order $x_{i+1} < x_{i+2} < \dots < x_{i+k}$ in the BDD construction, then the reverse of that order. The result is an arc-consistent encoding of a SCAMO(X, k) constraint set with approximately $11 \cdot M \cdot k$ clauses, where $M = \lceil \frac{n}{k} \rceil$.

Consider a feasibility query with value k of the ABP over a graph G , as it was formalized in the previous section. We first encode for each vertex $v \in V$ a SCAMO(X, k) set, where X is a $|V|$ -long sequence of Boolean variables $\langle x_v^{\ell_1} x_v^{\ell_2} \dots x_v^{\ell_{|V|}} \rangle$ representing all possible labels of v . Then, we simply add the disjunction of the corresponding at-most-zero constraints belonging to neighbouring nodes to encode all the constraints of (OBJ_k). The exactly-one constraints of (VERTICES) are encoded as conjunctions of the constructed smaller at-most-one constraints and negations of at-most-zero constraints. The other set of exactly-one constraints in (LABELS) is encoded as conjunction of at-least-one constraints and the product encoding of at-most-one constraints introduced in [5].

The resulting formula consists mostly of unit, binary and ternary clauses from the SCAMO constraints. The larger clauses are either $|V|$ -long clauses from the at-least-one part of the exactly-one constraints in (LABELS), or M -long clauses from the at-least-one part of constraints (VERTICES).

III. GENERATED INSTANCES

To evaluate in [1] our proposed SAT-based solution method for the ABP, we implemented a C++ tool called DuplexEncoder. It takes as input a graph and a lower (LB) and an upper bound (UB) of the antibandwidth. For each value k starting from LB, it encodes the ABP as we presented in the previous section and invokes a SAT solver on it to decide

feasibility. If the formula is SAT, it moves to the next k , if it is UNSAT, the previous value was optimal and stops.

We experimented in [1] on the 24 graphs of the Harwell-Boeing Sparse Matrix Collection [6]. Our benchmark set was generated with `DuplexEncoder` from 12 graphs of [6], mostly where we could not solve the ABP in 1800 seconds in [1]. For each of these graphs first we considered every consecutive k values in a very wide range around the value of $(LB + UB)/2$ (see [1] for each value of LB and UB) and generated the corresponding SAT formula of the ABP for each.

From the resulting formulas we identified the “interesting” problems based on the description of the expected benchmarks on the homepage of the SAT competition. First we dropped all those formulas that Minisat [7] with default settings could solve in less than a minute. The remaining 539 formulas were tried to be solved in less than an hour with CaDiCaL [8], [9] on our cluster with Intel Xeon E5-2620 v4 @ 2.10GHz CPUs. CaDiCaL solved all in all 121 problems (91 SAT and 30 UNSAT) successfully and the required solving times of these instances ranged between few seconds and one hour with a very balanced distribution.

Due to our source AB problem, we know that whenever a formula with a specific k -value is satisfiable, all the other formulas with smaller k from the same graph must be SAT as well. Further, an unsatisfiable k means that every formula with larger k of the same graph is UNSAT as well. Based on these observations, we identified another 44 problem instances that must be satisfiable, but CaDiCaL could not solve. Since most of the unsatisfiable formulas were immediately solved with Minisat, we could not find further ones with this approach.

Further, for most of the graphs we collected two more yet unsolved instances that are hopefully not too far in difficulty from the solved ones. More precisely, for each graph where it was possible, we considered the highest k -value where the answer was SAT and picked the next formula (i.e. $k + 1$). Similarly, we considered the lowest formula where the answer was UNSAT and picked the next one (i.e. $k - 1$). For the resulting 22 formulas we do not know whether they are satisfiable or not, but hope that sooner or later they will be solved.

All in all, we included in our submission the 121 “interesting” problems together with the 44 unsolved satisfiable formulas (having at the end 135 SAT and 30 UNSAT problems) and the 22 unsolved, completely unknown problems. The result is a benchmark set of 187 problem instances, where approximately 12% is unknown whether satisfiable or not, 72% is SAT, the remaining 16% is UNSAT and CaDiCaL can solve approximately 65% of the problems in one hour. The file name of each submitted problem follows the `abw-[source-graph].w[k of query].cnf` pattern. The source code of the `DuplexEncoder` tool from [1] and the script that was used to generate the benchmarks are both available at <http://fmv.jku.at/duplex/>.

REFERENCES

- [1] K. Fazekas, M. Sinnl, A. Biere, and S. Parragh, “Duplex encoding of staircase at-most-one constraints for the antibandwidth problem,” in *CPAIOR*, ser. LNCS. Springer, 2020, to appear.
- [2] M. Sinnl, “A note on computational approaches for the antibandwidth problem,” *CoRR*, vol. abs/1910.03367, 2019. [Online]. Available: <http://arxiv.org/abs/1910.03367>
- [3] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger, “A new look at BDDs for pseudo-boolean constraints,” *J. Artif. Intell. Res.*, vol. 45, pp. 443–480, 2012. [Online]. Available: <https://doi.org/10.1613/jair.3653>
- [4] N. Eén and N. Sörensson, “Translating pseudo-boolean constraints into SAT,” *JSAT*, vol. 2, no. 1-4, pp. 1–26, 2006. [Online]. Available: <https://satassociation.org/jsat/index.php/jsat/article/view/18>
- [5] J. Chen, “A new sat encoding of the at-most-one constraint,” *Proc. Constraint Modelling and Reformulation*, 2010.
- [6] E. Rodríguez-Tello, H. Romero-Monsivais, J. Ramirez-Torres, and F. Lardeux, “Harwell-boeing graphs for the CB problem,” https://www.researchgate.net/publication/272022702_Harwell-Boeing_graphs_for_the_CB_problem, 2015.
- [7] N. Eén and N. Sörensson, “An extensible sat-solver,” in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518. [Online]. Available: https://doi.org/10.1007/978-3-540-24605-3_37
- [8] A. Biere, “CaDiCaL at the SAT Race 2019,” in *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.
- [9] 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*, M. Heule, M. Järvisalo, M. Suda, M. Iser, and T. Balyo, Eds., 2020.

Combined SLS and CDCL instances at the SAT Competition 2020

Mate Soos (National University of Singapore)

I. OVERVIEW

This paper showcases a system to generate problem instances that are a combination of an easily satisfiable random problem and an easily satisfiable structured problem. It combines these two satisfiable problems in a way that their sets of variables overlap on at least one variable that have matching polarities in their respective solutions. This ensures that the problems cannot easily be cut into two separate CNF files, while keeping the final CNF satisfiable.

The system tries to mimic real-world problems that are combinations of two easy problems, easily solvable separately by two *different* types of solving systems. However, the combination of them, truly combined but not highly overlapping, is hard to solve for non-hybrid solvers.

II. COMPONENTS

A. Fixed Clause Length Model Random CNF Generator

As part of the well-known WalkSAT [7] package, generates random formulas using the fixed clause length model. See WalkSAT and its source code for details. Parameters were set to generate 4.25 more clauses than the number of variables, each of length 3: "makewff -seed SEED -cnf 3 NVARs (4.25*NVARs)". The number of variables, NVARs, was varied between 300 and 500, and competition-matching values were picked out — too easy and too hard were discarded.

B. Grain of Salt Structured Problem Generator

The system uses Grain of Salt [8] to generate random problem from the CRYPTO-1 stream cipher [6], with preset number of outputs, a random IV and random key. When the number of OUTPUTS is low, it always generates a satisfiable problem. Increasing the number of OUTPUTS creates progressively harder problems which eventually become unsatisfiable. Parameters used were: "grainofsalt --seed SEED --crypto crypto1 --outputs OUTPUTS --karnaugh 0 --init no --num 1". The OUTPUTS parameter was varied between 30 and 50, and competition-matching values were picked out — too easy and too hard were discarded.

C. Multipart Combiner with Overlap

The multipart combiner is a tool that takes two satisfiable problems and their solutions, plus the parameter OVERLAP. It outputs a solution that is a combination of the two CNF's clauses, but the variable numbers have been shifted to be separate for both CNFs except for OVERLAP number of variables. These overlap variables are picked as the first OVERLAP variables that match on the provided solutions'

polarities. This ensures that the output is a satisfiable CNF and that the two problems cannot easily be cut apart.

D. Putting it all together

The random problem is first generated with a particular seed, and a solution is found for it using CCArr[4]. Then a cryptographic problem is generated with the same seed, and a solution is found for it using CaDiCaL[3]. Finally, the two CNFs and the two solutions are given to the multipart combiner tool which produces the final, satisfiable combined CNF.

III. HISTORICAL BACKGROUND

SLS solvers have been shown to be useful for some structured instances, as demonstrated by ProbSAT [2] and yalsat [1]. Early attempts at hybrid solving, such as ReasonLS [5] combined SLS and CDCL solvers in shell-scripted, non-cohesive way, nevertheless winning the 2018 SAT Competition's NoLimits track. This ignited interesting developments, culminating in CaDiCaL [3] and CryptoMiniSat [9] both having a hybrid SLS-CDCL strategy in the SAT Race 2019. This combined hybrid strategy has been proven useful for industrial instances. The CNFs generated by the system can be best solved by solvers utilizing such a hybrid strategy.

IV. RATIONALE

These CNFs were mostly created to encourage hybrid solvers such as SLS+CDCL, Gauss-Jordan elimination+CDCL, Groebner Basis+CDCL, etc. Note that with linearization, Gauss-Jordan elimination is capable of solving non-linear problems as well, and is therefore not restricted to XOR constraints. These hybrid systems could potentially prove very useful for the SAT community in the long run.

V. PARAMETERS FOR THE PROBLEM INSTANCES GENERATED

The problem instances problems fall into two categories, both of which have OVERLAP set to 2.

The first bunch contain 500 variables (and correspondingly, $4.25 * 500 = 22125$ clauses, each of length 3) for [?] and have 31 outputs given for the CRYPTO-1 algorithm. This can be considered moderately complex on the SLS domain and moderately simple on the CDCL domain.

The second bunch contain 450 variables (and correspondingly, $4.25 * 500 = 19912$ clauses, each of length 3) for [?] and have 40 outputs given for the CRYPTO-1 algorithm. This can be considered moderately simple on the SLS domain and moderately complex on the CDCL domain.

VI. LIST OF PROBLEM INSTANCES GENERATED

crypto1-wff-seed-1-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-12-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-15-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-16-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-18-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-19-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-21-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-22-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-24-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-25-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-26-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-28-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-3-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-32-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-4-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-5-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-8-wffvars-450-cryptocplx-40-overlap-2.cnf
crypto1-wff-seed-101-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-102-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-104-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-105-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-106-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-107-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-108-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-109-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-110-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-115-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-116-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-121-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-127-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-129-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-132-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-133-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-134-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-136-wffvars-500-cryptocplx-31-overlap-2.cnf
crypto1-wff-seed-138-wffvars-500-cryptocplx-31-overlap-2.cnf

REFERENCES

- [1] Balint, A., Biere, A., Fröhlich, A., Schöning, U.: Improving implementation of SLS solvers for SAT and new heuristics for k-sat with long clauses. In: Sinz, C., Egly, U. (eds.) *Theory and Applications of Satisfiability Testing - SAT*. LNCS, vol. 8561, pp. 302–316. Springer (2014)
- [2] Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Cimatti, A., Sebastiani, R. (eds.) *Theory and Applications of Satisfiability Testing - SAT 2012*. Lecture Notes in Computer Science, vol. 7317, pp. 16–29. Springer (2012)
- [3] Biere, A.: Cadical, Lingeling, Plingeling, Treengeling and YalSAT entering the sat competition 2018. *Proc. of SAT Competition* pp. 13–14 (2018)
- [4] Cai, S., Luo, C., Su, K.: Ccanr: A configuration checking based local search solver for non-random satisfiability. In: Heule, M., Weaver, S.A. (eds.) *SAT 2015*. LNCS, vol. 9340. Springer (2015)
- [5] Cai, S., Zhang, X.: ReasonLS. *Proc. of SAT Competition* pp. 52–53 (2018)
- [6] Garcia, F.D., de Koning Gans, G., Muijers, R., van Rossum, P., Verdult, R., Schreur, R.W., Jacobs, B.: Dismantling MIFARE classic. In: Jajodia, S., López, J. (eds.) *Computer Security - ESORICS 2008*. LNCS, vol. 5283, pp. 97–114. Springer (2008)
- [7] Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. pp. 521–532 (1995)
- [8] Soos, M.: Grain of salt—an automated way to test stream ciphers through sat solvers. In: *Tools*. vol. 10, pp. 131–144 (2010)
- [9] Soos, M.: The CryptoMiniSat 5.5 set of solvers at the SAT Competition 2018 (2018)

CNFMiter

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

Abstract—This document describes the CNFMiter tool, as well as the benchmarks that have been submitted to the SAT competition 2020, namely miter formulas of CNFs used for model counting, combined with a simplified, equivalent, variant.

I. INTRODUCTION

model counting uses CNF. We have CNF simplifications. number of models stays the same in case we only use equivalence preserving modifications (and drop defined variables – gates – but that’s harder to encode). To check whether CNF simplifications actually preserve equivalence, create miter CNFs.

II. CREATING MITER CNFS

The input for the cnfmiter tool are two uncompressed CNF files with the formulas F and G . For both formulas, a new variable f_i is introduced for each clause C_i in the formula F that is encoded to be \top when the current interpretation satisfies the clause. Note, the number of clauses in the formulas F and G can be different.

$$F' := \bigvee_{C_i \in F} f_i \leftrightarrow C_i$$

$$G' := \bigvee_{D_j \in G} g_j \leftrightarrow D_j$$

Next, we introduce a variable s_F that represents the information whether all clauses in a formula F are satisfied, again, for both formulas.

$$S_F := s_F \leftrightarrow \bigwedge_{C_i \in F} f_i$$

$$S_G := s_G \leftrightarrow \bigwedge_{D_j \in G} g_j$$

Finally, we need to make sure that the two formulas cannot be satisfied at the same time, which gives us the final miter formula:

$$M := F' \wedge G' \wedge S_F \wedge S_G \wedge (s_F \not\leftrightarrow s_G)$$

In case the two formulas F and G are equivalent, the resulting miter formula M is unsatisfiable. In case the variables in F and G do not match, the resulting formula M is satisfiable.

III. THE SUBMITTED BENCHMARK

The presented approach to check formulas for equivalence allows to check whether CNF simplifier implementations are buggy, i.e. when being combined with a CNF fuzzer like [2].

The submitted miter formulas take formulas as input, that come from the model counting domain. CNFs without variable weights have been taken from <http://www.cs.cornell.edu/~sabhar/software/benchmarks/IJCAI07-suite.tgz> and http://cs.stanford.edu/~ermon/code/nips2011_benchmark.zip.

Each formula has been attempted to be simplified, such that the resulting formula is logically equivalent, and all variables that have been present in the original formula are present in the result again.

For CNF simplification, we chose the latest version of COPROCESSOR [5] default simplifications. The simplifications are subsumption, self-subsuming resolution, equivalent literal elimination [3], reasoning on the binray implication graph [4], as well as deducing units and equivalences from XORs, and reason about cardinality constraints [1]. All techniques are setup to preserve all variables, e.g. eliminated equivalences and the resulting formula rewriting are performed, but before emitting the formula, the equivalence relations are encoded to the CNF again.

IV. AVAILABILITY

The source of the tool, as well as scripts to build the dependency coprocessor and create simplification miter formulas is publicly available under the MIT license at <https://github.com/conp-solutions/cnfmiter>. The file *scripts/README* in the repository explains how the submitted benchmarks have been created.

REFERENCES

- [1] A. Biere, D. Le Berre, E. Lonca, and N. Manthey, “Detecting cardinality constraints in CNF,” in *Theory and Applications of Satisfiability Testing – SAT 2014*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer International Publishing, 2014, pp. 285–301. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09284-3_22
- [2] R. Brummayer and A. Biere, “Fuzzing and delta-debugging SMT solvers,” in *Workshop SMT 2010*. ACM, 2009, pp. 1–5.
- [3] A. V. Gelder, “Toward leaner binary-clause reasoning in a satisfiability solver,” *Ann. Math. Artif. Intell.*, vol. 43, no. 1, pp. 239–253, 2005.
- [4] M. Heule, M. Jarvisalo, and A. Biere, “Efficient CNF simplification based on binary implication graphs,” in *SAT*, ser. LNCS, vol. 6695, 2011, pp. 201–215.
- [5] N. Manthey, “Coprocessor 2.0 - a flexible CNF simplifier - (tool presentation),” in *SAT*, ser. LNCS, vol. 7317, 2012, pp. 436–441.

Axiom Pinpointing Benchmarks for IPASIR Solvers

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

Abstract—This document briefly describes the axiom pinpointing benchmarks that have been submitted to the incremental track. In CNF form, axiom pin pointing searches for all subsets of a set of assumptions that make the formula unsatisfiable.

I. INTRODUCTION

In the description logic EL+, axiom pinpointing is the computations of all minimal subsets of axioms that imply another axiom in an ontology [1]. The task can be solved with SMT or SAT solvers, by converting the input problem accordingly [4]. The submitted benchmarks are based on the tool SATPIN [2], [3], which allows to use SAT solvers via the IPASIR interface. SATPIN also uses a modified MINISAT solver, heavily modifying the way incremental solving is done to speedup solving iterations. Experiments show, that at in 2015 IPASIR solvers could not keep up with this modified solver. Modifications include to avoid fully restarting the SAT solver between iterations, i.e. keeping as many decision levels as possible and adding new clauses while the interpretation is non-empty; testing whether an out-of-order assumption is already falsified before consider the next assumption as a decision; or sorting assumption literals to have stable assumptions in the front of the stack. The last mentioned modification results in automatically computing subset-free answers for the axiom pinpointing problem. Without this modification, more SAT calls might be required. See [2], [3] for more details.

II. THE SUBMITTED BENCHMARK

The submitted benchmark is based on the ontology *Full-Galen*, that has been converted to the CNF form [4]. SATPIN is called as usual, except that the backend is an IPASIR solvers, i.e. in case not all subsets that result in an unsatisfiable formula have been found, the SAT solver will be called again. Hence, finding small answers is beneficial, as it potentially avoids redundant SAT solver calls.

III. AVAILABILITY

SATPIN is publicly available under the MIT license at <https://github.com/conp-solutions/satpin>. The file *README* in the repository explains how to link IPASIR solvers, as well as how to call the tool with the converted anthologies.

REFERENCES

- [1] F. Baader, S. Brandt, and C. Lutz, “Pushing the \mathcal{EL} envelope.” Morgan-Kaufmann, 2005.
- [2] N. Manthey, R. Peñaloza, and S. Rudolph, “Efficient axiom pinpointing in EL using SAT technology,” in *Proc. of DL 2016*, ser. CEUR Ws, vol. 1577, 2016.
- [3] —, “SATPin Axiom Pinpointing for Lightweight Description Logics Through Incremental SAT.” Springer, 2020, in press.
- [4] R. Sebastiani and M. Vescovi, “Axiom pinpointing in large $\mathcal{EL}+$ ontologies via SAT and SMT techniques,” University of Trento, Italy, Tech. Rep. DISI-15-010, 2015.

SAT Encodings for Flood-It Puzzle

Milan van Stiphout

Faculty of Science

Radboud University, Nijmegen, The Netherlands

milanstiphout@gmail.com

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—Flood-it is a puzzle in which, the player aims to fill the game board (flood) with a minimum number of flooding moves. Finding an optimal solution to this game is an NP-Complete problem. In this paper, we describe a possible encoding of the game to propositional Satisfiability problem.

I. INTRODUCTION

Flood-It is a board game in which the player is given an $n \times n$ board of squares (we call them fields), where each one is allocated one of c colors. The goal is to fill the entire board with the same color via the shortest possible sequence of flooding moves from the top left. If $c \geq 3$, Flood-It complexity becomes NP-Hard [1], [2].

Each move dictates the player to choose a color. In turn, all flooded fields change their color to the chosen color. This procedure is recursive as any field touching the flooded region with the same color is also flooded. As the flooded region grows, new neighbors will be eligible for flooding in the next move. The process continues until the whole board is filled with a single color with lowest possible moves.

II. SAT ENCODING

The intuition of solving the game in SAT, is to let a SAT solver find the shortest sequence of moves that completes the puzzle. In this section, we briefly describe a strategy which is proposed by the first author in his bachelor thesis [3] to formulate the game rules with a set of predicates in Conjunctive Normal Form (CNF).

Nevertheless, solving a single encoding gives only part of the answer to the question “how many minimal moves is needed to solve Flood-It puzzle?”. The minimal solution is rather logarithmically approached by choosing a value m that allows the corresponding SAT formula satisfiable.

A. Game Properties

The properties of the game are expressed in the following predicates:

- $\mathcal{F}(t, f)$ is true if a field f is flooded at turn t .
- $\mathcal{M}(t, c)$ is true if a color c is chosen at turn t .
- $\mathcal{T}(t, f)$ is true if f is next to a flooded field at t .

These predicates define the Boolean variables of the SAT formula that encodes Flood-It puzzle. The assignments of the *move* predicates \mathcal{M} , along with the proposed conditions, give a possible sequence of moves that we need to solve the game.

B. Flood Conditions

The following conditions are designed in such a way to strictly follow the rules of Flood-It. This grants a tangible solution of the puzzle via a satisfying assignment to the corresponding SAT problem.

- 1) A cluster \mathcal{C} of orthogonal fields is flooded at $t = 0$ starting from the top-left-corner field $f = 0$ (precondition).

$$\bigwedge_{f=0}^{n^2-1} \begin{cases} \mathcal{F}(0, f) & , \text{if } f \in \mathcal{C} \\ \neg \mathcal{F}(0, f) & , \text{otherwise} \end{cases} \quad (1)$$

- 2) One move can be made at a turn with exactly one color. The notations m and k denote the maximum number of moves allowed and the number of colors available respectively.

$$\bigwedge_{t=0}^{m-1} \bigvee_{c=0}^{k-1} \mathcal{M}(t, c) \quad (2)$$

$$\bigwedge_{t=0}^{m-1} \bigwedge_{c=0}^{k-2} \bigwedge_{d=c+1}^{k-1} \neg \mathcal{M}(t, c) \vee \neg \mathcal{M}(t, d) \quad (3)$$

- 3) A flooded field will remain flooded to the end of the game

$$\bigwedge_{t=0}^{m-1} \bigwedge_{f=0}^{n^2-1} \mathcal{F}(t, f) \Rightarrow \mathcal{F}(t+1, f) \quad (4)$$

- 4) If a field f is not already flooded and not touched by any other field g_i such that $1 \leq i \leq k$, then f will not be flooded.

$$\bigwedge_{t=0}^m \bigwedge_{f=0}^{n^2-1} \neg \mathcal{F}(t, f) \wedge \neg \mathcal{T}(t, g_1) \wedge \dots \wedge \neg \mathcal{T}(t, g_k) \Rightarrow \neg \mathcal{F}(t+1, f) \quad (5)$$

- 5) If a move is made with an f 's color (c_f), and that field touches the flooded region, it is flooded.

$$\bigwedge_{t=0}^{m-1} \bigwedge_{f=0}^{n^2-1} \mathcal{M}(t, c_f) \wedge \mathcal{T}(t, f) \Rightarrow \mathcal{F}(t+1, f) \quad (6)$$

- 6) If a move is made of not c_f and f is not flooded already, then f is not flooded.

$$\bigwedge_{t=0}^{m-1} \bigwedge_{f=0}^{n^2-1} \neg \mathcal{M}(t, c_f) \wedge \neg \mathcal{F}(t, f) \Rightarrow \neg \mathcal{F}(t+1, f) \quad (7)$$

- 7) Whenever a field is flooded, all neighbors $(g_1 \dots g_k)$ with the same color are flooded as well.

$$\bigwedge_{t=0}^m \bigwedge_{f=0}^{n^2-1} \mathcal{F}(t, f) \Leftrightarrow \mathcal{F}(t, g_1) \vee \dots \vee \mathcal{F}(t, g_k) \quad (8)$$

- 8) A field is flooded iff it touches a direct orthogonal neighbor $(n_1 \dots n_k)$.

$$\bigwedge_{t=0}^m \bigwedge_{f=0}^{n^2-1} \mathcal{T}(t, f) \Leftrightarrow \mathcal{F}(t, n_1) \vee \dots \vee \mathcal{F}(t, n_k) \quad (9)$$

- 9) All fields are flooded after a maximum number of moves m at $t = m + 1$ (postcondition).

$$\bigwedge_{f=0}^{n^2-1} \mathcal{F}(m + 1, f) \quad (10)$$

III. BENCHMARKS

In benchmarks generation, we chose $k = 10, n = 70$, and $m = \{225, 229, 230\}$. The maximum number of moves m can be used to control the hardness of the encoded problem. For these settings, 20 instances are generated randomly.

REFERENCES

- [1] D. Arthur, R. Clifford, M. Jalsenius, A. Montanaro, and B. Sach, "The complexity of flood filling games," in *Fun with Algorithms*, P. Boldi and L. Gargano, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 307–318.
- [2] U. dos Santos Souza, F. Rosamond, M. R. Fellows, F. Protti, and M. D. da Silva, "The flood-it game parameterized by the vertex cover number," *Electronic Notes in Discrete Mathematics*, vol. 50, pp. 35 – 40, 2015, IAGOS'15 – VIII Latin-American Algorithms, Graphs and Optimization Symposium.
- [3] M. van Stiphout, "Flood-it as a sat problem," bachelor thesis, Faculty of Science, Radboud University, Nijmegen, The Netherlands, 2020. [Online]. Available: <https://www.cs.ru.nl/bachelors-theses>

Crafted Benchmarks with Artificial Community Structure and Known Complexity

Marc Hartung and Benjamin Kaiser
Zuse Institute Berlin
 Berlin, Germany
 {hartung,kaiser}@zib.de

Abstract—Insights about connections between structural properties and how a SAT solver behaves on such problems are either difficult to gain or only evaluated statistically. We propose a generator that creates benchmarks with a specific community structure, clause lengths, and an easy to understand proof. Thus, the effects of different solver implementations, heuristics, and parameters can be easier analyzed and compared.

I. INTRODUCTION

Different structural attributes of SAT problems influence the efficiency and the behaviour of a SAT solver. For example, certain watcher schemes are preferable, when a benchmark has a particular clause to variable ratio or a dominant clause length. Further, the underlying graph structure can determine which heuristics are more likely to succeed. A prominent example is to determine the usefulness of a learned clause in CDCL solvers by its LBD value [1], which mainly boosts the performance on application benchmarks, while maintaining the capability of solving other benchmark types. The underlying assumption is that the LBD reflects the community structure of the underlying graph of the SAT problem. Such universal insights are rare and hard to find. To further study the behaviour of SAT solvers and especially try to understand why certain heuristics, parameters, and implementations lead faster to a solution on certain problems, we implemented a benchmark generator that creates instances with a known graph structure and complexity (also called hardness), i.e. a known number of necessary conflicts to find the solution and how they can be solved optimally. With the generated problems, we can test variations of implementations and parameters to observe on which structures they succeed and in which they do not. Thus, possible connections between runtime statistics, used parameters, and problem structure can be easier exposed, since full information about the problem is known.

As an example, our first goal is to identify a connection between used data structures and hardware effects, e.g. cache usage for different benchmarks. Especially in parallel SAT solvers, the solve time fluctuates, which makes it necessary to execute hundreds of benchmarks multiple times to statistically evaluate different implementations. By using predefined branching sequences on our generated problems, we can compare different clause storage implementations deterministically. This is not possible in general because different data structures lead to different orders of propagated literals. This leads to different conflicts and finally to different solution

times. To prevent this, one has to know exactly which conflicts will occur through which branches.

This description is early work in progress and only covers how the submitted benchmarks are generated. We strongly encourage interested readers to get in touch with us to discuss and work on other possible applications of such generated benchmarks.

II. BASIC METHOD

The generated benchmarks are based on connecting multiple smaller SAT problems to an harder to solve instance. First, we use unsatisfiable problems as cores, further called UNSAT cores. The main requirement for an UNSAT core is that the minimal proof is known. This trivially includes which clauses are used in the proof and how many conflicts are necessary for a solution by knowing the resolution DAG. Such UNSAT cores are then linked together so that the resulting benchmark is still unsatisfiable and the minimal proof is still known.

In the current state, we link UNSAT cores together by adding a variable to a one of the proof clauses of the one core and the negation of the variable to a proof clause of the other core. This approach has limitations. As soon as cycles are created through linking multiple cores, it has to be ensured that the cycles do not introduce a satisfiable variable setting.

III. USED UNSAT CORE

In the first stage, we generate clauses from all sign permutations of m variables x_i , i.e. generate for each of the 2^m possible models a clause, which excludes it (Eq. 1). The proof of this m -core is simple and every deterministic branching heuristic leads to the same proof when no learned clauses are deleted. Also, every preprocessor using a NIVER approach [2] solves the problem optimally.

$$\{(s_1x_1 \vee s_2x_2 \vee \dots \vee s_nx_n) \mid s_i \in \{+, \neg\}\} \quad (1)$$

In the second stage, the generator transforms the m -core into a N-SAT problem, by splitting every clause using additional variables similar to linking multiple cores described above and shown for one 4-core clause in Eq. (2). This approach changes the number of resolutions in the proof but has no influence on the number of conflicts or the branching strategy.

$$\neg x_1 \vee x_2 \vee x_3 \vee \neg x_4 \xrightarrow[\text{split}]{\text{add } a} \begin{array}{l} \neg x_1 \vee x_2 \vee a \\ \neg a \vee x_3 \vee \neg x_4 \end{array} \quad (2)$$

In the third and last stage, three or more of the N-SAT m -cores are combined to a single problem using for each core the same variables for the N-SAT transformation, which is equal to a variable renaming step. Eq. (3) gives an example for the first clause of each of three combined 4-cores.

$$\begin{array}{ll}
 x_1 \vee x_2 \vee \mathbf{a} & x_1 \vee x_2 \vee \mathbf{a} \\
 \neg \mathbf{a} \vee x_3 \vee x_4 & \neg \mathbf{a} \vee x_3 \vee x_4 \\
 y_1 \vee y_2 \vee \mathbf{b} & y_1 \vee y_2 \vee \mathbf{a} \\
 \neg \mathbf{b} \vee y_3 \vee y_4 & \neg \mathbf{a} \vee y_3 \vee y_4 \\
 z_1 \vee z_2 \vee \mathbf{c} & z_1 \vee z_2 \vee \mathbf{a} \\
 \neg \mathbf{c} \vee z_3 \vee z_4 & \neg \mathbf{a} \vee z_3 \vee z_4
 \end{array} \quad (3)$$

The last transformation does not increase the hardness. The combined m -cores have still the complexity of 2^m , and the original unsat proofs are still valid. Thus, to make such a combined core satisfiable—which is also crucial for linking multiple cores—one clause of each combined m -core has to be deleted respectively to the variables used for the combination. For example, in Eq. (3) all six clauses using the same connection variable have to be deleted to make the complete problem satisfiable.

The last stage mainly adjusts the variable to clause ratio and prevents a typical preprocessor to be capable of solving the problem, which takes in most cases more time than directly using a CDCL solver.

IV. OTHER POSSIBLE UNSAT CORES

Certain other crafted benchmarks come also into consideration, which we may consider in the future. Even taking any minimal proof of a benchmark would be suitable. Regarding our first goal, it is very time-consuming to generate a branching sequence to non-trivial problems, which ensures the same solution path with different implementations. With our current approach, we can scale the problem size, the hardness, and adjust the variable to clause ratio easily, while the branching sequence is already known due to the simple base of the UNSAT cores.

V. DETAILS ABOUT SUBMITTED BENCHMARKS

All of the submitted benchmarks are relatively large. The used disk space for the not compressed CNFs are between 80 MB and 700 MB. Half of the 20 benchmarks are satisfiable, and the others are unsatisfiable. Details about the runtime are displayed in Table I.

The benchmarks are created from 17 to 19-cores with N-SAT transformations between 3 and 10 or from a high number of 4 to 7-cores reduced to 3-SAT problems. Every benchmark is homogeneously created, i.e. all linked and combined cores of a benchmark are created from the same number of variables. The used graph structure in each benchmark is a randomly generated acyclic graph. Since each benchmark is create so that the optimal branching sequence is always (x_1, x_2, \dots, x_n) we randomly renamed the variables using <https://github.com/vegard/cnf-utils>.

TABLE I
RESULTS FROM EXECUTING MINISAT 2.2 ON A DUAL SOCKET INTEL
XEON E5-2630V3 2.4 GHZ WITH 64 GB DRAM

name	solution	time in s
ncc_none_2_18_8_3_1_0_435991723.cnf	sat	352
ncc_none_3001_7_3_3_0_31_435991723.cnf	sat	932
ncc_none_3001_7_3_3_1_31_435991723.cnf	sat	965
ncc_none_5047_6_3_3_3_0_435991723.cnf	sat	1165
ncc_none_3047_7_3_3_1_0_1.cnf	sat	1323
ncc_none_2_17_3_3_1_0_435991723.cnf	sat	1688
ncc_none_12477_5_3_3_1_0_435991723.cnf	sat	2611
ncc_none_2_19_5_3_1_0_435991723.cnf	sat	6242
ncc_none_12477_6_3_3_1_0_435991723.cnf	sat	7000
ncc_none_21015_5_3_3_1_0_11.cnf	sat	7000
ncc_none_3_16_3_3_0_0_435991723.cnf	unsat	892
ncc_none_2_17_4_3_0_0_435991723.cnf	unsat	976
ncc_none_2_18_9_3_0_0_435991723.cnf	unsat	1138
ncc_none_5047_6_3_3_0_0_41.cnf	unsat	1573
ncc_none_3047_7_3_3_0_0_1.cnf	unsat	1809
ncc_none_2_18_5_3_0_0_435991723.cnf	unsat	2534
ncc_none_12477_5_3_3_0_0_435991723.cnf	unsat	3000
ncc_none_7047_6_3_3_0_0_420.cnf	unsat	3130
ncc_none_21015_5_3_3_0_0_11.cnf	unsat	7000
ncc_none_12477_6_3_3_0_0_435991723.cnf	unsat	7000

ACKNOWLEDGMENT

We want to express our gratitude towards the organisers of the SAT Competition 2020 for making such an event possible. Additionally we like to thank Florian Schintke for his support, the IT and Data Services members of the Zuse Institute Berlin for providing the infrastructure and their fast help and also Vegard Nossium for providing the CNF transform tools.

REFERENCES

- [1] Gilles Audemard and Laurent Simon, “Predicting Learnt Clauses Quality in Modern SAT Solvers”, IJCAI 2009, 2009
- [2] Sathiamoorthy Subbarayan, Dhiraj K. Pradhan and David G. Hoos, “NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances”, Theory and Applications of Satisfiability Testing, 2005

SAT Benchmarks based on Hypertree Decompositions

André Schidler
Algorithms and Complexity Group
TU Wien
Vienna, Austria
aschidler@ac.tuwien.ac.at

Stefan Szeider
Algorithms and Complexity Group
TU Wien
Vienna, Austria
sz@ac.tuwien.ac.at

This benchmark set contains several SAT instances used for computing hypertree decompositions. Similar to tree decompositions and generalized hypertree decomposition, hypertree decompositions are used to solve problems, that can be represented as hypergraphs (e.g. conjunctive queries), in time depending on the width of the decomposition.

I. PROBLEM DESCRIPTION AND ENCODING

A *tree decomposition* of a hypergraph $H = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ where $T = (V(T), E(T))$ is a tree and χ is a mapping that assigns to each $t \in V(T)$ a set $\chi(t) \subseteq V$ such that the following properties hold:

- T1 for each $v \in V$ there is some $t \in V(T)$ with $v \in \chi(t)$,
- T2 for each $e \in E$ there is some $t \in V(T)$ with $e \subseteq \chi(t)$,
- T3 for any three $t, t', t'' \in V(T)$ where t' lies on the path between t and t'' , we have $\chi(t) \cap \chi(t'') \subseteq \chi(t')$.

A *generalized hypertree decomposition* of H is a triple $D = (T, \chi_D, \lambda_D)$ where (T, χ_D) is a tree decomposition of H and λ_D is a mapping that assigns each $t \in V(T)$ an *edge cover* $\lambda_D(t) \subseteq E(H)$ of $\chi_D(t)$.

A *hypertree decomposition* [3] of H is a generalized hypertree decomposition $D = (T, \chi_D, \lambda_D)$ of H where T is a rooted tree that satisfies in addition to T1–T3 also a certain Special Condition (T4). To formulate this condition, we call a vertex v to be *omitted* at a node $t \in V(T)$, if $v \notin \chi_D(t)$, but $\lambda_D(t)$ contains a hyperedge e with $v \in e$. The *Special Condition* now states the following:

- T4 If a vertex v is omitted at t , then it must not appear in the bag $\chi_D(t')$ of any descendant node t' of t .

The *width* of D is the size of a largest edge cover $\lambda(t)$ over all $t \in V(T)$. The *hypertree width* of H is the smallest width over all hypertree decompositions of H .

We use a SAT encoding to compute hypertree decompositions of a given width for a given hypergraph. The encoding is based on the treewidth encoding by Samer and Veith [4] and its extension to generalized hypertree width [1], [2]. We extend this encoding with clauses representing the hierarchical structure of the rooted tree, which allows us to encode the Special Condition [5]. The width is enforced by placing sequence counter cardinality constraints [4], [6] on the edge covers.

II. INSTANCE GENERATION AND CATEGORIZATION

The hypergraphs used for the SAT instances are from the Hyperbench¹ collection. These hypergraphs represent queries from different database systems as well as CSP instances. Due to the sampling described below, the provided SAT instances are mainly generated from CSP hypergraphs.

The SAT instances were generated using our decomposer HtdSMT². Given a hypergraph H , the decomposer uses an upper bound on the hypertree width to start the search. Whenever a decomposition is found, the bound is decremented, until no decomposition can be found. Each SAT instance represents a run with a separate upper bound k . The instances are named using the convention `<hyperbench instance name>_<k>.cnf`. If the instance is satisfiable, the hypertree width of H is smaller or equal to k , otherwise it is larger than k .

The instances are separated into the categories *easy* and *hard*. Easy instances were solved on the test system in between 600 and 1800 seconds and hard instances took over 1800 seconds or were not solved at all. The instances were then subdivided into satisfiable instances (*sat*), unsatisfiable instances (*unsat*) and instances where the satisfiability is *unknown*. The categorization is reflected in the folder structure. The test system ran *glucose 4.0* using 16 GB RAM and an Intel i5-9600KF CPU running at 3.7 GHz.

REFERENCES

- [1] Jeremias Berg, Neha Lodha, Matti Järvisalo, and Stefan Szeider. Maxsat benchmarks based on determining generalized hypertree-width, 2017.
- [2] Johannes K. Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT approach to fractional hypertree width. In *Proceedings of CP 2018, the 24rd International Conference on Principles and Practice of Constraint Programming*, volume 11008, pages 109–127, 2018.
- [3] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.
- [4] Marko Samer and Helmut Veith. Encoding treewidth into sat. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 45–50, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [5] André Schidler and Stefan Szeider. A compact sat-encoding for hypertree width. submitted.
- [6] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

¹<http://hyperbench.dbai.tuwien.ac.at/>

²<https://github.com/ASchidler/htdsmt>

Lam’s Problem Benchmarks for the SAT Competition 2020

Curtis Bright
Carleton University & University of Waterloo
Ottawa & Waterloo, Canada
cbright@uwaterloo.ca

Saeed Nejati
University of Waterloo
Waterloo, Canada
snejati@uwaterloo.ca

Vijay Ganesh
University of Waterloo
Waterloo, Canada
vijay.ganesh@uwaterloo.ca

Abstract—This document describes a collection of satisfiability instances that arise in Lam’s problem from discrete geometry.

I. INTRODUCTION

Instances in this benchmark encode subproblems that arise in Lam’s problem from finite projective geometry—the problem of determining whether or not a finite projective plane of order ten exists. Studied since the 1800s, Lam’s problem was resolved in the late 1980s by a computer search culminating in months of computational effort on a CRAY-1A supercomputer [1]. Recently, we used SAT solvers to verify a significant portion of this search [2]–[4].

II. BACKGROUND

A finite projective plane of order ten is defined to consist of a collection of 111 points, 111 lines, and an incidence relationship between points and lines such that any two points are incident with a unique line and any two lines are incident with a unique point. Furthermore, every line is incident with exactly 11 points and every point is incident with 11 lines.

From a computational perspective, a convenient way of representing a finite projective plane of order ten is by a square incidence matrix A of order 111 whose (i, j) th entry contains a 1 exactly when the i th line is incident to the j th point. The projective plane incidence relationship requires that any two distinct rows or columns of A have an inner product of exactly one. It follows that A satisfies the relationship

$$AA^T = A^T A = 10I + J$$

where I denotes the identity matrix and J denotes the matrix consisting of all 1s.

It is hopeless to determine if such an A exists using this simple definition alone—even though the search space is finite it is far too large to be effectively searched. In the 1970s, coding theory was used to derive conditions that A must satisfy if it exists. In particular, it can be shown mathematically that the row-space of $A \pmod{2}$ must contain vectors of Hamming weight 15 or 19 [5]. Furthermore, the existence of such vectors greatly restrict the structure of A .

In particular, a vector of Hamming weight 15 appearing in A ’s row-space implies that every entry appearing in either the first 21 rows or 15 columns of A can be assumed without loss of generality [6]. Similarly, the vectors of Hamming

weight 19 are of three possible types (called oval, 16-type, or primitive [5]) and each case places restrictions on the possible structure of A .

The twenty benchmarks in this collection each specify a different starting configuration for A —one benchmark resulting from the weight 15 starting configuration, three benchmarks resulting from 16-type starting configurations, and sixteen benchmarks resulting from primitive weight 19 starting configurations.

III. ENCODING

Let $a_{i,j}$ be a Boolean variable that is true exactly when $A[i, j] = 1$. We say that two columns or rows of A intersect if they share a 1 in the same location. The projective plane incidence relationship requires that any two rows and any two columns of A intersect exactly once. In our encoding we require that

- 1) any two rows or columns intersect *at most* once, and
- 2) any row or column entirely specified by the starting configuration intersects every other row or column *at least* once.

These two conditions are sufficient to show that A cannot exist (at least in the starting configurations that occur in this collection of benchmarks). Moreover, these conditions are naturally encoded in conjunctive normal form.

In the first condition, suppose that i and j are arbitrary row indices. Then

$$\bigwedge_{1 \leq k < l \leq 111} (\neg a_{i,k} \vee \neg a_{i,l} \vee \neg a_{j,k} \vee \neg a_{j,l})$$

specify that rows i and j do not intersect twice (i.e., they intersect at most once). Conditions of this form are required for all $1 \leq i < j \leq 111$.

In the second condition, suppose that i is the index of a row completely specified by the starting configuration and that j is the index of another row. Then

$$\bigvee_{k:A[i,k]=1} a_{j,k}$$

specifies that row i and j intersect at least once. This clause is well-defined since all entries $A[i, k]$ for $1 \leq k \leq 111$ are known in the starting configuration. We also include similar

clauses for each column completely specified by the starting configuration.

Additionally, we used two optimizations of this encoding which in our experiments made the instances easier to solve.

First, we do not include all 111^2 variables in the instances. Instead, we choose a submatrix of A and only encode the constraints arising in that submatrix. The submatrix is experimentally chosen to be small while still ensuring that there are enough constraints to make the instance unsatisfiable. As a rule of thumb, about one third of the entries of A are usually required before the instance becomes unsatisfiable. In our collection of benchmarks the weight 15 instance used 75 columns and 51 rows, the 16-type instances used 65 columns and 80 rows, and the primitive weight 19 instances used up to 54 columns and all 111 rows.

Second, we included symmetry breaking clauses that remove symmetries from the search space. In particular, we enforce a lexicographical order on certain rows and columns that are otherwise identical in the starting configuration. There are also additional symmetries in the 16-type instances broken using a lexicographic method—see [3] for details.

IV. SUMMARY

The benchmarks in this collection naturally arise in the process of solving Lam’s problem from finite geometry. They have been selected in order to provide the satisfiability community a collection of instances relevant to solving an interesting and celebrated mathematical problem. They were generated as a part of the MathCheck project and can all be solved in under an hour on a modern desktop using the cube-and-conquer paradigm [7].

REFERENCES

- [1] C. W. H. Lam, L. Thiel, and S. Swiercz, “The non-existence of finite projective planes of order 10,” *Canadian Journal of Mathematics*, vol. 41, no. 6, pp. 1117–1123, 1989.
- [2] C. Bright, K. Cheung, B. Stevens, D. Roy, I. Kotsireas, and V. Ganesh, “A nonexistence certificate for projective planes of order ten with weight 15 codewords,” *Applicable Algebra in Engineering, Communication and Computing*, 2020.
- [3] C. Bright, K. K. H. Cheung, B. Stevens, I. Kotsireas, and V. Ganesh, “Unsatisfiability proofs for weight 16 codewords in Lam’s problem,” *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, to appear.
- [4] —, “Nonexistence certificates for ovals in a projective plane of order ten,” *Proceedings of the 31st International Workshop on Combinatorial Algorithms*, to appear.
- [5] M. Hall Jr., “Configurations in a plane of order ten,” in *Annals of Discrete Mathematics*. Elsevier, 1980, vol. 6, pp. 157–174.
- [6] F. J. MacWilliams, N. J. A. Sloane, and J. G. Thompson, “On the existence of a projective plane of order 10,” *Journal of Combinatorial Theory, Series A*, vol. 14, no. 1, pp. 66–78, 1973.
- [7] M. J. H. Heule, O. Kullmann, and A. Biere, “Cube-and-conquer for satisfiability,” in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 31–59.

Polynomial Multiplication in SAT competition 2020

Chu-Min Li^{1,2}, Mao Luo¹, Fan Xiao¹, Yu Li², Felip Manyà³, Zhipeng Lü¹
¹School of Computer Science, Huazhong University of Science and Technology, Wuhan, China
 {maoluo,zhipeng.lv}@hust.edu.cn, zhaoyangxiaofan@163.com
²MIS, University of Picardie Jules Verne, Amiens, France
 {chu-min.li, yu.li}@u-picardie.fr
³Artificial Intelligence Research Institute (IIIA-CSIC), Barcelona, Spain. felip@iiia.csic.es

Abstract—Multiplying two polynomials of degree $n - 1$ can need n^2 coefficient products, because each polynomial of degree $n - 1$ has n coefficients. If the coefficients are real numbers, the Fourier transformation allows to reduce the number of necessary coefficient products to $O(n * \log(n))$. However, when the coefficients are not real numbers (e.g., when the coefficients are matrix), the Fourier transformation cannot be used. In this case, it is important to reduce the number of necessary coefficient products using other methods to speed up the multiplication of two polynomials. In this short paper, we reduce the problem of multiplying two polynomials of degree $n - 1$ with t ($t \leq n^2$) coefficient products to SAT, in such a way that the solution of a satisfiable SAT instance tells how to use exactly t coefficient products to multiply two polynomials of degree $n - 1$, and the unsatisfiability of a SAT instance implies the infeasibility of using t coefficient products to multiply two polynomials of degree $n - 1$. We then provide 20 new crafted SAT instances.

1. Introduction

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

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

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

$$(ax + b)(cx + d) = acx^2 + \left((a + b)(c + d) - ac - bd \right) x + bd \quad (2)$$

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

In the general case, we want to multiply two polynomials of degree $n - 1$ using less than n^2 coefficient products. If the coefficients are real numbers, the Fourier transformation

allows to reduce the number of necessary coefficient products to $O(n * \log(n))$. However, when the coefficients are not real numbers (e.g., when the coefficients are matrix), the Fourier transformation cannot be used.

In the sequel, we describe how to reduce the problem of multiplying two polynomials of degree $n - 1$ using t ($t \leq n^2$) products to SAT. When the obtained SAT instance is satisfiable, the SAT solution gives a way to multiply two polynomials of degree $n - 1$ using t products. When the obtained SAT instance is unsatisfiable, we know that more than t coefficient products are needed to multiply two polynomials of degree $n - 1$. We refer to [1], [2] for other efficient algorithms for polynomials. The reduction is different from [3], [4] in that the solution of a satisfiable instance gives a precise way to compose each product and a precise way to use the t products to compute each coefficient of the product of the two polynomials, while the reduction presented in [3], [4] does not give such a way.

2. A SAT Encoding of Polynomial Multiplication Using t Products

Consider two polynomials of degree $n - 1$:

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

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

Their product is

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

We want to compute $A(x) \times B(x)$ using t ($t \leq n^2$) products: P_1, P_2, \dots, P_t , where each P_l ($1 \leq l \leq t$) is of the form $(\pm a'_1 \pm a'_2 \pm \dots)(\pm b'_1 \pm b'_2 \pm \dots)$ with $a'_1, a'_2, \dots \in \{a_{n-1}, a_{n-2}, \dots, a_0\}$ and $b'_1, b'_2, \dots \in \{b_{n-1}, b_{n-2}, \dots, b_0\}$. Addition and/or subtraction of these products give the coefficients $c_k = \pm P'_1 \pm P'_2 \pm \dots$ of $A(x) \times B(x)$, where $0 \leq k \leq 2n - 2$ and $P'_1, P'_2, \dots \in \{P_1, P_2, \dots, P_t\}$. The problem becomes to determine a'_i and b'_j and their sign for each product, as well as P'_l and their sign in c_k . To solve that problem, we first define the following Boolean variables.

- $pa_{il} = 1$ iff a_i is involved positively in product P_l ;
- $na_{il} = 1$ iff a_i is involved negatively in product P_l ;
- $pb_{jl} = 1$ iff b_j is involved positively in product P_l ;
- $nb_{jl} = 1$ iff b_j is involved negatively in product P_l ;
- $pc_{kl} = 1$ iff product P_l is used positively to compute c_k ;
- $nc_{kl} = 1$ iff product P_l is used negatively to compute c_k ;
- $px_{ijkl} = 1$ iff a_i and b_j are involved in product P_l , product P_l is used to compute c_k , and the number of negative signs among a_i , b_j and P_l in c_k is even;
- $nx_{ijkl} = 1$ iff a_i and b_j are involved in product P_l , product P_l is used to compute c_k , and the number of negative signs among a_i , b_j and P_l in c_k is odd;

We then define the clauses of the encoding as follows:

- Clauses linking px_{ijkl} and nx_{ijkl} with pa_{il} , pb_{jl} , pc_{kl} , na_{il} , nb_{jl} and nc_{kl} .
- For each i and j ($0 \leq i, j \leq n-1$) and for each k ($0 \leq k \leq 2n-2$) such that $i+j \neq k$, if a_i and b_j are involved in product P_l and P_l is used to produce c_k , then the product of a_i and b_j should be eliminated by subtraction using another product $P_{l'}$ involving a_i and b_j . If $i+j = k$, one product of a_i and b_j should remain in c_k . So,

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

In order to compute the difference $\sum_{l=1}^t px_{ijkl} - \sum_{l=1}^t nx_{ijkl}$, we introduce t new Boolean variables u_1, u_2, \dots, u_t that represent the value of px_{ijkl} ($1 \leq l \leq t$) in the decreasing order and t Boolean variables v_1, v_2, \dots, v_t that represent the value of nx_{ijkl} ($1 \leq l \leq t$) in the decreasing order. It is clear that $\forall i$ $u_i \geq v_i$ and there is exactly one i such that $u_i = 1$ and $v_i = 0$ to satisfy $\sum_{l=1}^t px_{ijkl} - \sum_{l=1}^t nx_{ijkl} = 1$. For example, let $t = 2$. We introduce $u_1 = px_{ijk1} \vee px_{ijk2}$ and $u_2 = px_{ijk1} \wedge px_{ijk2}$, and $v_1 = nx_{ijk1} \vee nx_{ijk2}$ and $v_2 = nx_{ijk1} \wedge nx_{ijk2}$. Then we have $u_1 \geq u_2$, $v_1 \geq v_2$, $u_1 \geq v_1$ and $u_2 \geq v_2$. If $v_1 = 1$, we deduce quickly $u_1 = u_2 = 1$ and $v_2 = 0$ to satisfy $\sum_{l=1}^t px_{ijkl} - \sum_{l=1}^t nx_{ijkl} = 1$.

3. Set of Submitted Instances

We generated 20 SAT instances by varying n and t , using the encoding of the previous section. Each combination of n and t gives an instance newpoln-t.

Table 1 shows, for each generated instances, its number of variables and clauses, the status of the formula (satisfiable, unsatisfiable or unknown), and the time needed by MiniSat [5] to solve the instance on a computer with Intel Xeon E5-2680 v4 processors at 2.40 GHz and 10GB of memory under Linux. The cutoff time is 5000 seconds.

TABLE 1. INFORMATION ABOUT THE GENERATED INSTANCES.

Instance	#Variables	#Clauses	Satisfiability	Time
newpol3-6	10247	32483	UNSAT	4346.01
newpol4-6	20133	64129	UNSAT	4573.78
newpol6-6	55745	178325	UNSAT	582.97
newpol6-7	70134	223345	unknown	timeout
newpol6-8	81975	260721	unknown	timeout
newpol7-5	65730	211263	UNSAT	652.43
newpol7-6	83487	267355	unknown	timeout
newpol23-4	1206085	3943811	UNSAT	1175.82
newpol27-4	1917373	6270811	UNSAT	1202.76
newpol28-4	2130735	6968861	UNSAT	1087.42
newpol29-4	2359369	7716887	UNSAT	1409.96
newpol30-4	2603803	8516617	UNSAT	719.09
newpol31-4	2864565	9369779	UNSAT	1984.97
newpol32-4	3142183	10278101	UNSAT	2134.57
newpol34-4	3750099	12267137	UNSAT	1103.73
newpol36-4	4431775	14497549	UNSAT	865.87
newpol37-4	4801593	15707591	UNSAT	1941.61
newpol38-4	5191435	16983161	UNSAT	893.79
newpol39-4	5601829	18325987	UNSAT	2415.15
newpol40-4	6033303	19737797	UNSAT	1022.88

Acknowledgments

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

References

- [1] D. Bini and V. I. Pan, *Polynomial and Matrix Computations: Fundamental Algorithms*. Birkhauser Verlag Basel, Switzerland, 1994.
- [2] R. Zippel, *Effective Polynomial Computation*. Springer Science & Business Media, 2012.
- [3] F. Xiao, C.-M. Li, M. Luo, F. Manyà, and Z. Lü, “Polynomial multiplication,” in *Proceedings of SAT competition 2017 Solver and Benchmark Descriptions*, 2017, pp. 43–44.
- [4] F. Xiao, C.-M. Li, M. Luo, F. Manyà, Z. Lü, and Y. Li, “Polynomial multiplication,” in *Proceedings of SAT competition 2018 Solver and Benchmark Descriptions*, 2018, pp. 61–62.
- [5] N. Eén and N. Sörensson, “An extensible sat-solver,” in *6th International Conference on Theory and Applications of Satisfiability Testing*, SAT, 2003, pp. 502–518.

PEQNP Python Library Benchmarks

1st Oscar Riveros

Research Architect at www.PEQNP.science

PEQNP

Santiago, Chile

oscar.riveros@peqnp.science

Abstract—The formulas that are generated by PEQNP Library represent some particular instances of the following NP-Problems: 01 Integer Programming, Fermat Factorization, Schur Triplets, Sum Subset.

Index Terms—CNF, CSP, Encoding, Benchmarks

I. THE PEQNP LIBRARY

The PEQNP System its an automatic CNF encoder and SAT Solver for General Constrained Diophantine Equations and NP-Complete Problems, fully integrated with Python [1].

II. SAT COMPETITION 2020 BENCHMARKS

The collected formulas have generated with PEQNP Library for the following problems:

A. 01 Integer Programming

Given an integer matrix C and integer vector d , exist a 0-1 vector x such that $Cx = d$?

B. Fermat Factorization

Given an integer pq and exist two integers p and q such that $(p - q)(p + q) = pq$ with $1 < p - q < p + q$?

C. Schur Triplets

Given a list of $3n$ distinct positive integers is there a partition of the list into n triples (a_i, b_i, c_i) such that $a_i + b_i = c_i$ for each triple i ?

D. Sum Subset

. Given a list of positive integers U with a target t exist a S such that $\Sigma(S) = t$?

III. AVAILABILITY

The source of PEQNP Library can be found at www.peqnp.science internally integrated with SLIME SAT Solver [2].

REFERENCES

- [1] www.python.org Python is an interpreted, high-level, general-purpose programming language.
- [2] www.peqnp.science SLIME 4: A Free World Class High Performance SAT Solver.

Thanks to www.foresta.io for all these years of support.

A SAT Encoding of School Timetabling

Rodrigue Konan Tchinda

Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
rodriguekonanktr@gmail.com

Clémentin Tayou Djamegni

Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
dtayou@gmail.com

Abstract—This document describes a SAT encoding of the school timetabling problem where all constraints are hard. The instances derived from this encoding are submitted to the 2020 SAT Competition.

I. INTRODUCTION

The tremendous progress achieved in the solving of the Boolean Satisfiability problem (SAT) has favored the used of the SAT technology to tackle a variety of related practical problems. The latter include software and hardware verification [1], [2] combinatorial problems [3], [4], planning in artificial intelligence [5], [6], cryptanalysis [7], dependency management [8], graph coloring, timetable construction [9], [10], etc.

In this paper, we are interested in the *school timetabling problem* [9] where we give an encoding of a specific version which is used to generate the benchmarks submitted to the 2020 SAT Competition.

II. TIMETABLING PROBLEM CONSTRAINTS AND ENCODING

A. Problem Specifications

To be valid, the timetable must comply with a number of constraints. All these constraints are considered hard in the sense that they are required to be satisfied. The problem specifications are the followings: A course is defined by a class and a subject in that class. A class here refers to a group of students that will be taught a particular subject. A course must have a teacher, a classroom and a number of time slots in which it is taught. Two courses taught by the same teacher must not take place in the same time slot. No more than one course may be taught in a classroom at the same time. No two courses in the same classroom may be held at the same time slot. Teachers may be unavailable at certain time slots. Each course has a required number of weekly time slots and must be scheduled in a limited number of time slots per day. The time slots allocated to a course in a day must be contiguous. Some teachers may not be qualified to teach some courses. All the courses taught by a given teacher are spread over a limited number of days in a week.

The problem here, is to find for each course of each class, the teacher who teaches it, the room in which it takes place and the time slots during which it is taught in a week.

B. Definitions and Notations

In our encoding, we consider the following Boolean variables:

- $ch_{i,j}$ is true iff the course i is scheduled in time slot j ;
- $cd_{i,j}$ is true iff the course i takes place in day j ;
- $cs_{i,j}$ is true iff the course i takes place in room j ;
- $ce_{i,j}$ is true iff the course i is taught by teacher j .
- $ed_{i,j}$ is true iff the teacher i teaches some course in day j .

We also consider the following sets and constants:

- T : The set of all time slots in a week;
- Cl : The entire set of classes;
- C : The entire set of courses;
- C_i : The set of courses taught in class i ;
- nt_i : The required number of time slots for course i in a week;
- S : The set of all rooms;
- S_i : The set of rooms that can hold course i ;
- E : The set of teachers;
- E_i : The set of teachers that can teach course i ;
- J : The entire set of class days;
- TJ_j : The set of the time slots in day j .
- $day(j)$ is the day of the time slot j .

We use the following notations:

Given a set A of Boolean variables and an integer k , $AMO(A)$, $EXO(A)$, $AM(k,A)$, $AL(k,A)$, $EX(k,A)$ denote respectively the cardinality constraints “At Most One”, “Exactly One”, “At Most k ”, “At Least k ” and “Exactly k ” defined over A .

C. Encoding of the Constraints

We then have the following encoding:

- 1) If a course takes place in a given time slot, then it also takes place on the day corresponding to that time slot. This is enforced by the following constraint:

$$\neg ch_{i,j} \vee cd_{i,day(j)}$$

Hence, for all courses and for all time slots, we have :

$$\bigwedge_{i \in C} \bigwedge_{j \in T} (\neg ch_{i,j} \vee cd_{i,day(j)})$$

- 2) If a course takes place in day j , then it also takes place in some time slot of this day; so for each course i and

each day j , whose time slots are t_{j1}, \dots, t_{jk} we have $\neg cd_{i,j} \vee ch_{i,t_{j1}} \vee \dots \vee ch_{i,t_{jk}}$ which gives the following CNF encoding:

$$\bigwedge_{i \in C} \bigwedge_{j \in J} (\neg cd_{i,j} \vee ch_{i,t_{j1}} \vee \dots \vee ch_{i,t_{jk}})$$

- 3) Each course i has exactly nt_i time slots per week.

$$\bigwedge_{i \in C} (\text{EX}(nt_i, \{ch_{i,j}, j \in T\}))$$

- 4) Each course must have exactly one teacher

$$\bigwedge_{i \in C} (\text{EXO}(\{ce_{i,j}, j \in E_i\}))$$

- 5) Each course must have exactly one classroom

$$\bigwedge_{i \in C} (\text{EXO}(\{cs_{i,j}, j \in S_i\}))$$

- 6) Two courses with the same teacher must not take place in the same time slot. T_j represents all the time slots during which teacher j is available.

$$\bigwedge_{j \in E} \bigwedge_{k \in T_j} \text{AMO}(\{ce_{i,j} \wedge ch_{i,k}, i \in C_j\})$$

- 7) Two courses with the same room must not take place at the same time slot.

$$\bigwedge_{j \in S} \bigwedge_{k \in T} \text{AMO}(\{cs_{i,j} \wedge ch_{i,k}, i \in C\})$$

- 8) Each class must have a maximum of one course per time slot.

$$\bigwedge_{k \in C} \bigwedge_{t \in T} (\text{AMO}(\{c_{i,t}, i \in C_k\}))$$

- 9) Each course i must have no more than $nbMaxTSDay_i$ time slots per day.

$$\bigwedge_{i \in C} \bigwedge_{j \in J} (cd_{i,j} \Rightarrow \text{AM}(nbMaxTSDay_i, \{ch_{i,t}, t \in TJ_j\}))$$

- 10) A course i must be scheduled on $nbDaysMax_i$ days at most per week.

$$\bigwedge_{i \in C} \text{AM}(nbDaysMax_i, \{cd_{i,j}, j \in J\})$$

- 11) Each teacher must have at least EJ_{min} and no more than EJ_{max} class days per week.

- $$\bigwedge_{j \in E} \bigwedge_{i \in C_j} \bigwedge_{k \in J} (\neg ce_{i,j} \vee \neg cd_{i,k} \vee ed_{j,k})$$

- $$\bigwedge_{i \in E} \text{AM}(EJ_{max}, \{ed_{j,k}, k \in J\})$$

- $$\bigwedge_{i \in E} \text{AL}(EJ_{min}, \{ed_{j,k}, k \in J\})$$

- 12) The time slots of the same course in the same day must be contiguous.

$$\bigwedge_{i \in C} \bigwedge_{j \in T} \left(\bigwedge_{k \in \text{TSP}(j)} \neg ch_{i,j} \vee (ch_{i,j-1}, \text{day}(j) = \text{day}(j-1) \vee \neg ch_{i,k}) \right)$$

$$\bigwedge_{k \in \text{TSF}(j)} \neg ch_{i,j} \vee (ch_{i,j+1}, \text{day}(j) = \text{day}(j+1) \vee \neg ch_{i,k})$$

where $\text{TSP}(j)$ (resp. $\text{TSF}(j)$) is the set of time slots preceding (resp. following) j in the same day as j .

D. Encoding of AMO, AM, ALO, AL and EX Constraints

“At Most One” constraints are encoded using *two product encoding* [11], the *nested encoding* [12] or the *naive encoding* (considering the number of clauses produced and auxiliary variables introduced). “At Most k ” and “At Least k ” constraints are encoded using *sequential counter encoding* [13]. As far as “Exactly k ” is concerned, we have $\text{EX}(k, A) \equiv \text{AM}(k, A) \wedge \text{AL}(k, A)$.

III. GENERATED BENCHMARKS FOR THE 2020 SAT COMPETITION

We generated 20 instances by varying the number of courses, teachers, classes and rooms. Table I gives some information on these benchmarks. In this Table for instance, the benchmark “Timetable_C_272_E_52_Cl_18_S_32” was generated with 272 courses, 52 teachers, 18 classes and 32 rooms. The table also reports experiments conducted with these benchmarks and the solver MapleLCMDistChronoBT [14] on the StarExec cluster [16] using a timeout of 5000 seconds and 24 GB of memory.

Table I
GENERATED INSTANCES

Benchmark	Result	Wallclock Time
Timetable_C_151_E_34_Cl_10_S_4	UNKNOWN	>5000
Timetable_C_181_E_34_Cl_12_S_10	UNKNOWN	>5000
Timetable_C_181_E_34_Cl_12_S_6	UNKNOWN	>5000
Timetable_C_196_E_34_Cl_13_S_12	UNKNOWN	>5000
Timetable_C_226_E_34_Cl_15_S_12	UNKNOWN	>5000
Timetable_C_241_E_34_Cl_16_S_12	UNKNOWN	>5000
Timetable_C_241_E_34_Cl_16_S_14	UNKNOWN	>5000
Timetable_C_241_E_45_Cl_16_S_14	UNKNOWN	>5000
Timetable_C_241_E_45_Cl_16_S_16	SAT	41.99
Timetable_C_272_E_52_Cl_18_S_32	SAT	80.65
Timetable_C_331_E_45_Cl_22_S_26	UNKNOWN	>5000
Timetable_C_331_E_50_Cl_22_S_26	SAT	62.55
Timetable_C_362_E_50_Cl_24_S_26	SAT	93.13
Timetable_C_392_E_50_Cl_26_S_26	SAT	106.56
Timetable_C_392_E_50_Cl_26_S_28	SAT	113.398
Timetable_C_392_E_62_Cl_26_S_28	SAT	89.44
Timetable_C_437_E_62_Cl_29_S_28	SAT	135.50
Timetable_C_437_E_62_Cl_29_S_32	SAT	147.75
Timetable_C_466_E_50_Cl_31_S_28	SAT	192.49
Timetable_C_497_E_62_Cl_33_S_30	SAT	204.01

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*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176.
- [2] S. Swords and J. Davis, “Bit-blasting ac12 theorems,” *arXiv preprint arXiv:1110.4676*, 2011.
- [3] M. J. Heule, O. Kullmann, and V. W. Marek, “Solving and verifying the Boolean pythagorean triples problem via cube-and-conquer,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2016, pp. 228–245.
- [4] M. J. Heule, “Schur number five,” in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [5] H. A. Kautz, B. Selman *et al.*, “Planning as satisfiability,” in *ECAI*, vol. 92. Citeseer, 1992, pp. 359–363.
- [6] J. Rintanen, “Planning as satisfiability: Heuristics,” *Artificial intelligence*, vol. 193, pp. 45–86, 2012.
- [7] F. Lafitte, J. Nakahara Jr, and D. Van Heule, “Applications of sat solvers in cryptanalysis: finding weak keys and preimages,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, no. 1, pp. 1–25, 2014.
- [8] D. L. Berre and A. Parrain, “On SAT technologies for dependency management and beyond,” in *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, 2008, pp. 197–200.
- [9] N. Pillay, “An overview of school timetabling research,” in *Proceedings of the international conference on the theory and practice of automated timetabling*, 2010, p. 321.
- [10] R. A. Achá and R. Nieuwenhuis, “Curriculum-based course timetabling with sat and maxsat,” *Annals of Operations Research*, vol. 218, no. 1, pp. 71–91, 2014.
- [11] J. Chen, “A new sat encoding of the at-most-one constraint,” *Proc. Constraint Modelling and Reformulation*, 2010.
- [12] A. Biere, D. Le Berre, E. Lonca, and N. Manthey, “Detecting cardinality constraints in cnf,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2014, pp. 285–301.
- [13] C. Sinz, “Towards an optimal cnf encoding of boolean cardinality constraints,” in *International conference on principles and practice of constraint programming*. Springer, 2005, pp. 827–831.
- [14] V. Ryvchin and A. Nadel, “Maple_lcm_dist_chronob: Featuring chronological backtracking,” *SAT COMPETITION 2018*, p. 29, 2018.
- [15] N. Eén and N. Sörensson, “An extensible sat-solver,” in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.
- [16] A. Stump, G. Sutcliffe, and C. Tinelli, “Starexec: a cross-community infrastructure for logic solving,” in *International Joint Conference on Automated Reasoning*. Springer, 2014, pp. 367–373.

Station Repacking Problem

Xindi Zhang^{1,2} and Shaowei Cai^{1,2*}

¹State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

²School of Computer Science and Technology, University of Chinese Academy of Sciences, China

dezhangxd@163.com, shaoweicai.cs@gmail.com

Abstract—This document describes a set of instances for SAT solvers, which are generated by SATFC for station repacking problem. SATFC is based on a SAT encoding and adopted by the FCC for use in the incentive auction.

I. INTRODUCTION

The US government held an “incentive auction” for radio spectrum, broadcast rights are relinquished with the television broadcasters paid through a “reverse auction”. Further, the remaining broadcasters are repacked into a narrower band of spectrum, and the cleared spectrum is sold. SATFC [1] is an open-source solver which is adopted by FCC for use in the incentive auction, and SATFC is a sat encoding based solver. The solver and data can be found at <http://www.cs.ubc.ca/labs/beta/Projects/SATFC>.

II. STATION REPACKING PROBLEM

The station repacking problem is formally defined below. Each television station $s \in S$ is currently assigned a channel $c_s \in C \subseteq \mathbb{N}$, and the assignment should ensure that channels will not excessively interfere with each other. We use $I \subseteq (S \times C)^2$ denoting a set of *forbidden station-channel pairs* $\{(s, c), (s', c')\}$ to represent that stations s and s' may not be concurrently assigned to channels c and c' , respectively.

The auction will lead to removing a set of channels, which will be reassigned to stations from a reduced set. This reduced set is defined by *clearing target*: channel $\bar{c} \in C$ such that all stations are only eligible to be assigned channels from $\bar{C} = \{c \in C : c < \bar{c}\}$. And the sets of channels a priori available to each station are given by a domain function $D : S \rightarrow 2^{\bar{C}}$ that maps from stations to these reduced sets.

So the station repacking problem is to find a repacking $\gamma : S \rightarrow \bar{C}$ which assigns each station a channel from its domain and satisfies the interference rule.

III. ENCODING

The station repacking problem is suitable to be encoded to SAT problem. Given station repacking problem (S, C) with domains D and interference constraints I , for every station-channel pair $(s, c) \in S \times C$, we create a boolean variable $x_{s,c}$ to represent station s is assigned to channel c . Then three types of clauses are created.

- (1) Each station is assigned at least one channel.

$$\bigvee_{d \in D(s)} x_{s,d}, \forall s \in S \quad (1)$$

- (2) Each station is assigned at most one channel.

$$\neg x_{s,c} \vee \neg x_{s,c'}, \forall s \in S, \forall c \neq c', c, c' \in D(s) \quad (2)$$

- (3) Respecting inference constraints

$$\neg x_{s,c} \vee \neg x_{s',c'}, \forall \{(s, c), (s', c')\} \in I \quad (3)$$

REFERENCES

- [1] A. Fréchette, N. Newman, and K. Leyton-Brown. Solving the station repacking problem. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 702–709, 2016.

* Corresponding author

Solver Index

Abcdsat, 25

CaDiCal, 50

Cadical-alluip, 10

Cadical-alluip-trail, 10

Cadical-trail, 10

CryptoMiniSat with CCAnr, 27

CryptoMiniSat with WalkSAT, 29

CTSsat, 32

DurianSat, 47

ExMapleLCMDistChronoBT, 17

exp_L_MLD_CBT_DL, 23

exp_V_L_MLDCBT_DL, 23

exp_V_LGB_MLD_CBT_DL,
23

exp_V_MLD_CBT_DL, 23

F2TRC, 21

Glucose 3.0 with Inprocessing, 31

GlucoseEsbpSel, 54

Kissat, 50

mallob-mono, 45

ManyGlucose 4.1-60, 58

Maple-LCM-Dist-alluip-trail, 10

Maple_CM+dist, 48

Maple_CM+dist+sattime, 48

Maple_CM+dist+simp2-, 48

Maple_CMused+dist, 48

Maple_mix, 25

Maple_simp, 25

MapleCOMSPS_LRB_VSIDS, 36

MapleLCMDistChronoBT-DL-v3,
20

MaplePainless-DC, 38

MergeSAT, 40

MLCMDChronoBT-DL-Scavel, 13

MLCMDChronoBT-DL-v2.2ScavelRFV,
13

Optsat, 25

P-MCOMSPS-STR, 56

PaInleSS_ExMapleLCMDistChronoBT,
17

Paracooba, 50

ParaFROST, 42

ParaFROST_ALL, 42

ParaFROST_CBT, 42

ParaFROST_HRE, 42

PauSat, 34

Plingeling, 50

Relaxed_LCMDCBDL, 15

Relaxed_LCMDCBDL_newTech,
15

Relaxed_LCMDCBDL_noTimeParam,
15

Riss 7, 44

SLIME, 59

TopoSAT2, 60

Treengeling, 50

Undominated-LC-MapleLCMDiscChronoBT-
DL, 11

upGlucose-3.0_PADC, 17

Benchmark Index

Antibandwidth feasibility, 81
AutoFault attacks on AES, LED,
and PRESENT, 79
Axiom pinpointing, 86

Baseball lineup selection, 80

Combinations of Random and Crypto
CNFs, 83
Community structures, 89

Directed Ramsey numbers, 66
Discrete logarithm, 65

Edge matching, 70

Flood-It puzzle, 87

Hypertree decomposition, 91

Incremental track, 63
Influence maximization, 77

Lam's problem, 92

Main track, 62
MITER CNFs, 85

PEQNP Python library bench-
marks, 96
Planning track, 64
Polynomial multiplication, 94
Population safety, 75

School timetabling, 97
Station repacking, 100

Unit-distance strip coloring, 68

Author Index

- Baarir, Souheib, 54, 56
Bacchus, Fahiem, 10, 80
Becker, Bernd, 79
Biere, Armin, 50, 81
Bright, Curtis, 92
- Cai, Shaowei, 15, 27, 100
Chen, Jingchao, 25, 65
Chen, Qingshan, 13
Chowdhury, Md Solimul, 23, 75
- Devriendt, Jo, 27, 54
Djamegni, Clémentin Tayou, 17, 97
- Ehlers, Thorsten, 60
- Fazekas, Katalin, 50, 81
Feng, Nick, 10, 80
Fleury, Mathias, 50
Froleyks, Nils, 64
- Ganesh, Vijay, 36, 38, 92
Gay, Maël, 79
Gocht, Stephan, 27
- Hartung, Marc, 32, 34, 89
Heisinger, Max, 50
Heule, Marijn, 66, 68
Hickey, Randy, 10, 80
- Inoue, Katsumi, 58
Iser, Markus, 62, 63
- Kaiser, Benjamin, 34, 89
Kautz, Henry, 29
Kochemazov, Stepan, 20, 21, 77
Kondratiev, Victor, 20
Kordon, Fabrice, 56
Kulczynski, Mitja, 60
- Lü, Zhipeng, 48
Le Frioux, Ludovic, 56
Li, Chu-Min, 48, 94
Li, Yu, 48, 94
Li, Zhihui, 13
Liang, Jia Hui, 36
Lonlac, Jerry, 11
- Lu, Zhipeng, 94
Luo, Mao, 48, 94
- Müller, Martin, 23, 75
Mackey, John, 66
Manthey, Norbert, 31, 40, 44, 85, 86
Manyà, Felip, 48, 94
Martins, Ruben, 68
Meel, Kuldeep S., 27, 47
Metin, Hakan, 54
- Nabeshima, Hidetomo, 58
Neiman, David, 66
Nejati, Saeed, 36, 38, 92
Nguifo, Engelbert Mephu, 11
Nowotka, Dirk, 60
- Oh, Chanseok, 36
Oostema, Peter, 68
Osama, Muhammad, 42, 87
- Parragh, Sophie, 81
Paxian, Tobias, 79
Polian, Ilia, 79
Poupart, Pascal, 36
- Riveros, Oscar, 59, 96
- Schindler, André, 91
Schreiber, Dominik, 45
Selman, Bart, 29
Semenov, Alexander, 20
Shaw, Arijit, 27, 47
Sieweck, Philipp, 60
Sinnl, Markus, 81
Soos, Mate, 27, 29, 83
Sopena, Julien, 56
Szeider, Stefan, 91
- Tchinda, Rodrigue Konan, 17, 97
- Upadhyaya, Devanshi, 79
- Vallade, Vincent, 54, 56
van Stiphout, Milan, 87
von Holten, Dieter, 70

Wijs, Anton, 42, 87

Wu, Guanfeng, 13

Xiao, Fan, 48, 94

Xu, Yang, 13

You, Jia-Huai, 23, 75

Zaikin, Oleg, 20

Zhang, Xindi, 15, 100