Proceedings of

SAT COMPETITION 2016

Solver and Benchmark Descriptions

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

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2016-1

ISSN 1458-4786
ISBN 978-951-51-2345-9 (PDF)
Helsinki 2016

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 2016 (SC 2016), an open competitive event for SAT solvers, was organized as a satellite event of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT 2016), Bordeaux, France. SC 2016 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; the SAT-Races held in 2006, 2008, 2010, and 2015; and SAT Challenge 2012.

SC 2016 consisted of several tracks, including a Main Track with subcategories and special tracks for parallel solvers, incremental solvers, solvers specificially developed for Random SAT, a Glucose Hack track, as well as a "No-Limits" track relaxing requirements on open source solvers and allowing any type of solvers—including solver portfolios—to compete.

There were two ways of contributing to SC 2016: by submitting one or more solvers for competing in one or more of the competition tracks, and by submitting interesting benchmark instances on which the submitted solvers could be evaluated on in the competition. Following the tradition put forth by SAT Challenge 2012, the rules of SC 2016 invited all contributors to submit a short, around 2-page long description as part of their contribution. This book contains these non-peer-reviewed descriptions in a single volume, providing a way of consistently citing the individual descriptions.

We hope this compilation is of value to the research community at large both at present and in the future, providing the reader new insights into the details of state-of-the-art SAT solver implementations and the SC 2016 benchmarks, and also as a future historical reference providing a snapshot of the SAT solver technology actively developed in 2016.

Matti Järvisalo

Contents

Benchmark Descriptions

SOLVER DESCRIPTIONS

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

CSCCSat in SAT Competition 2016

Chuan Luo*†, Shaowei Cai‡, Wei Wu§, Kaile Su¶‖
*Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
†School of Electronics Engineering and Computer Science, Peking University, Beijing, China
‡Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
§Center for Quantum Computation and Intelligent Systems, University of Technology, Sydney, Sydney, Australia
¶Department of Computer Science, Jinan University, Guangzhou, China
‖Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
{chuanluosaber, shaoweicai.cs, william.third.wu}@gmail.com; k.su@griffith.edu.au

***Abstract*—This document describes local search SAT solver *CSCCSat*.**

I. Introduction

Recently, a diversification strategy called configuration checking (CC) [1] has been proposed for handling the cycling problem, which is a serious issue in local search algorithms. In the context of SAT, there are two CC heuristics, i.e., the neighboring variables based configuration checking (NVCC) heuristic [2], [3], [4] and the clause states based configuration checking (CSCC) heuristic [5], [6], [7]. The CSCC heuristic has resulted in several efficient local search algorithms for SAT, such as *FrwCB* [6], [7] and *DCCASat* [8].

The *CSCCSat* solver is a local search solver, which is on the basis of the clause states based configuration checking (CSCC) heuristic. The *CSCCSat* solver is a combination of *FrwCB* and *DCCASat*. The *CSCCSat* solver won the '3rd Place Award' in the random SAT track of SAT Competition 2014.

II. Special Algorithms, Data Structures, and Other Features

The notation r denotes the clause-to-variable ratio of an SAT instance. The procedures of *CSCCSat* can be found in the solver description submitted to SAT Competition 2014 [9], and are described as follows. For random 3-SAT with $r \leq 4.24$, *FrwCB* is called; for random 3-SAT with $r > 4.24$, *DCCASat* is called. For random 4-SAT with $r \leq 9.35$, *FrwCB* is called; for random 4-SAT with $r > 9.35$, *DCCASat* is called. For random 5-SAT with $r \leq 20.1$, *FrwCB* is called; for random 5-SAT with $r > 20.1$, *DCCASat* is called. For random 6-SAT with $r \leq 41.2$, *FrwCB* is called; for random 6-SAT with $r > 41.2$, *DCCASat* is called. For random 7-SAT with $r \leq 80$, *FrwCB* is called; for random 7-SAT with $r > 80$, *DCCASat* is called.

III. Implementation details

The *CSCCSat* solver is implemented in programming language C/C++, and is developed on the basis of *FrwCB* and *DCCASat*.

IV. SAT Competition 2016 Specifics

The *CSCCSat* solver is submitted to Random SAT track, SAT Competition 2016. The command line of *CSCCSat* is described as follows.

```
./CSCCSat <instance> <seed>
```

References

[1] S. Cai, K. Su, and A. Sattar, "Local search with edge weighting and configuration checking heuristics for minimum vertex cover," *Artificial Intelligence*, vol. 175, no. 9-10, pp. 1672–1696, 2011.
[2] S. Cai and K. Su, "Local search with configuration checking for SAT," in *Proc. of ICTAI 2011*, 2011, pp. 59–66.
[3] ——, "Configuration checking with aspiration in local search for SAT," in *Proc. of AAAI 2012*, 2012, pp. 434–440.
[4] ——, "Local search for boolean satisfiability with configuration checking and subscore," *Artificial Intelligence*, vol. 204, pp. 75–98, 2013.
[5] C. Luo, K. Su, and S. Cai, "Improving local search for random 3-SAT using quantitative configuration checking," in *Proc. of ECAI 2012*, 2012, pp. 570–575.
[6] C. Luo, S. Cai, W. Wu, and K. Su, "Focused random walk with configuration checking and break minimum for satisfiability," in *Proc. of CP 2013*, 2013, pp. 481–496.
[7] C. Luo, S. Cai, K. Su, and W. Wu, "Clause states based configuration checking in local search for satisfiability," *IEEE Transactions on Cybernetics*, vol. 45, no. 5, pp. 1014–1027, 2015.
[8] C. Luo, S. Cai, W. Wu, and K. Su, "Double configuration checking in stochastic local search for satisfiability," in *Proc. of AAAI 2014*, 2014, pp. 2703–2709.
[9] ——, "CSCCSat2014 in SAT competition 2014," in *Proc. of SAT Competition 2014: Solver and Benchmark Descriptions*, 2014, pp. 25–26.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

DCCAlm in SAT Competition 2016

Chuan Luo*†, Shaowei Cai‡, Kaile Su§¶
*Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
†School of Electronics Engineering and Computer Science, Peking University, Beijing, China
‡Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
§Department of Computer Science, Jinan University, Guangzhou, China
¶Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
{chuanluosaber, shaoweicai.cs}@gmail.com; k.su@griffith.edu.au

***Abstract*—This document describes local search SAT solver *DCCAlm*.**

I. Introduction

The *DCCASat* solver [1] and the *WalkSATlm* solver [2], [3] show efficiency in solving random k-SAT instances at phase transition and large-scale random k-SAT instances, respectively.

The *DCCASat* solver benefits from the DCCA heuristic, which hierarchically combines neighboring variables based configuration checking (NVCC) [4], [5], [6] and clause states based configuration checking (CSCC) [7], [8], [9]. The *WalkSATlm* solver improves the original *WalkSAT* algorithm [10] by incorporating the multi-level make property [2], [3].

The *DCCAlm* solver is a combination of *DCCASat* and *WalkSATlm*. The main procedures of *DCCAlm* can be described as follows. For solving an SAT instance, *DCCAlm* first decides the type of this instance. Then based on the properties of the instance, *DCCAlm* calls either *DCCASat* or *WalkSATlm* to solve the instance.

II. Special Algorithms, Data Structures, and Other Features

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

III. Implementation details

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

IV. SAT Competition 2016 Specifics

The *DCCAlm* solver is submitted to Random SAT track, SAT Competition 2016. The command line of *DCCAlm* is described as follows.

```
./DCCAlm <instance> <seed>
```

References

[1] C. Luo, S. Cai, W. Wu, and K. Su, "Double configuration checking in stochastic local search for satisfiability," in *Proc. of AAAI 2014*, 2014, pp. 2703–2709.
[2] S. Cai, K. Su, and C. Luo, "Improving WalkSAT for random k-satisfiability problem with k > 3," in *Proc. of AAAI 2013*, 2013, pp. 145–151.
[3] S. Cai, C. Luo, and K. Su, "Improving WalkSAT by effective tie-breaking and efficient implementation," *The Computer Journal*, vol. 58, no. 11, pp. 2864–2875, 2015.
[4] S. Cai and K. Su, "Local search with configuration checking for SAT," in *Proc. of ICTAI 2011*, 2011, pp. 59–66.
[5] ——, "Configuration checking with aspiration in local search for SAT," in *Proc. of AAAI 2012*, 2012, pp. 434–440.
[6] ——, "Local search for boolean satisfiability with configuration checking and subscore," *Artificial Intelligence*, vol. 204, pp. 75–98, 2013.
[7] C. Luo, K. Su, and S. Cai, "Improving local search for random 3-SAT using quantitative configuration checking," in *Proc. of ECAI 2012*, 2012, pp. 570–575.
[8] C. Luo, S. Cai, W. Wu, and K. Su, "Focused random walk with configuration checking and break minimum for satisfiability," in *Proc. of CP 2013*, 2013, pp. 481–496.
[9] C. Luo, S. Cai, K. Su, and W. Wu, "Clause states based configuration checking in local search for satisfiability," *IEEE Transactions on Cybernetics*, vol. 45, no. 5, pp. 1014–1027, 2015.
[10] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proc. of AAAI 1994*, 1994, pp. 337–343.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Glue_alt: Hacking Glucose by Applying At-Least-One Recently Used Rule to Learnt Clause Management

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

***Abstract*—Glue_alt is a hack version that is built on the top of Glucose 3.0. It improves Glucose in the following components: phase selection and learnt clause database reduction, which are important elements in CDCL (Conflict Driven, Clause Learning) solvers.**

I. Introduction

Glue_alt is a hack version of Glucose 3.0 [1] . Compared with Glucose, Glue_alt adds at-least-one recently used strategy, bit-encoding phase selection strategy, and dynamic core and local learnt clause management strategy. In the subsequent sections, these strategies will be introduced.

II. At-Least-One Recently Used Policy

In the search process, CDCL (Conflict Driven, Clause Learning) solvers need to maintain the learnt clause database. This database maintenance should be similar to cache replacement in CPU cache management or page replacement in a computer operating system. There are many cache (page) replacement algorithms. For example, Least Recently Used (LRU), Most Recently Used (MRU), Pseudo-LRU (PLRU), Least-Frequently Used (LFU), Second Chance FIFO, Random Replacement (RR), Not Recently Used (NRU) [2] etc. Our at-least-one recently used (ALORU) algorithm is similar to NRU page replacement algorithm. ALORU algorithm favours keeping learnt clauses in database that have been recently used at least one time. If a learnt clause has not so far involved in any conflict analysis since it was generated, it will be discarded first. Implementing ALORU algorithm is very simple. When a conflict clause (called also learnt clause) is generated, its LBD (literal block distance) is usually set to the number of different decision levels involved in it. However, ALORU algorithm sets the initial LBD of a conflict clause to $+\infty$, not actual LBD value. In details, in the search procedure of Glucose, ALORU algorithm replaces "**setLBD(nblevels)**" with "**setLBD(0x3fffffff)**". Since any LBD never exceeds 0x3fffffff, we denote $+\infty$ with 0x3fffffff. If a learnt clause involves in a conflict analysis, Procedure **analyze** in Glue_alt sets its LBD value to the actual value.

III. Bit-Encoding Phase Selection Strategy

The bit-encoding phase selection strategy was proposed in 2014 [3], which is suitable for SAT instances. Its basic idea is that the phase of the n-th decision variable is the $(n \bmod 4)$-th of m, where m is a counter which is increased one every time it is used. In general, the decision levels where this strategy is applied are limited to 12. Furthermore, this strategy requires that the number of conflicts is less than 2×10^6. Implementing this strategy is also simple. In the procedure **pickBranchLit** of Glucose, the following statements are added.

```
if( bN >= 0 ) {
   int L=decisionLevel();
   if( L < 12) polarity[next] = (bN >>(L % 4)) & 1;
}
```

In the procedure **solve_** of Glucose, before statement "while (status == l_Undef)", statement "bN=-1" is added. After this "while", the following statement is added.

```
bN = conflicts > 2e6 || conflicts > 3e5 &&
      nVars() > 1e6? -1 : bN+1;
```

IV. Dynamic Core and Local Learnt Clause Management

Like SWDiA5BY [4], glue_alt classifies also learnt clauses into two categories: core and local. However, the classification of SWDiA5BY is static, while our classification is dynamic. In SWDiA5BY, the maximum LBD of core learnt clauses is fixed to a constant 5. However, in glue_alt, the maximum LBD of core learnt clauses is not fixed. Glue_alt divides the whole search process two stages. When the number of conflicts is less than 2×10^6, it is considered as the first stage. Otherwise, it is considered as the second stage. In the first stage, the maximum LBD of core learnt clauses is limited to 2. At this stage, core learnt clauses are kept indefinitely, unless eliminated when they are satisfied. In the second stage, the maximum LBD of core learnt clauses is limited to 5. This stage does not ensure that core learnt clauses are kept indefinitely. When clause database is reduced, we move 5000 core learnt clauses with LBD larger than or equal to 3 to local learnt clause database.

Whether the first or second stage, the number of local learnt clauses is maintained roughly between 10000 and 20000. That is, once the number of local learnt clauses reaches 20000, glue_alt will halve the number of the clauses. And the clauses with the smallest activity scores are removed first.

The computation of clause activity scores is consistent with MiniSat.

V. Conclusions

Glue_alt modified Glucose in three different ways, and resulted in two new ideas: learnt clause dynamic classification and at-least-one recently used notion. These new notions should have vitality.

References

[1] G. Audemard, L. Simon:Predicting learnt clauses quality in modern sat solvers, in *proceedings of IJCAI*, 2009, pp. 399–404.

[2] Amit S. Chavan, Kartik R. Nayak, Keval D. Vora, Manish D. Purohit, Pramila M. Chawan: A Comparison of Page Replacement Algorithms, IACSIT, vol.3, no.2, April 2011.

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

[4] C., Oh: MiniSat HACK 999ED, MiniSat HACK 1430ED, and SWDiA5BY, in *Proceedings of the SAT Competition 2014*, pp. 46–47.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

ParaGlueminisat, tbParaGlueminsat

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

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

***Abstract*—We briefly introduce our solver, ParaGlueminisat, and tbParaGlueminisat submitted to SAT competition 2016. These are parallel version of Glueminisat with several deterministic policies.**

I. Introduction

Diversification of search space has contributed to the rapid progress in SAT solving, and appears to be one of the most important keys in modern SAT solvers. It also plays an important role in portfolio-based parallel SAT solving. However, in portfolio solvers, maintenance of diversification among solvers is not that simple, especially for massively parallel machines. In this description we implements parallel version of glueminisat with several policies to diversify or intensify search space.

Details of our algorithm will be pressed. [1]

II. Proposal of Polarities to Search Space Index (PSSI)

Many state-of-the-art solvers are using phase saving to reuse its previous phase for intensive search after restarts. This phase has a strong relationship with learned clauses found from the current worker. However, clauses imported from other workers may doesn't fit current phase. By changing only a small part of the phase, we expect to maintain intensive search and may have an opportunity to use exported clauses.

For this, we convert the current phase to PSSI. First, divide the variable set into *k*-blocks $(B_1, B_2, ..., B_k)$. Second, calculate the ratio $(r_1, r_2, ..., r_k)$ of variables currently allocated to TRUE, and divide the ratio into uniform *m* sections uniformly, and each section having a value between 0 on the far left to *m* - 1 on the far right. For each block B_i, ratio is converted to integer b_i. For B_i, $b_i = p$ if $p/m \leq r_i < (p+1)/m$ where $p \in \{0, 1, ..., m-1\}$. After calculating each b_i, calculate PSSI (Polarity Search Space Index).

$$PSSI = \sum_{i=1}^{k} b_i \times m^{i-1}$$

Since PSSI is now only an integer, we can then easily though roughly compare the areas in the search space among the workers. Let's explain this with a simple example. Consider a problem with n variables x_1, x_2, x_3 and x_n. Solve this problem using the parallel SAT solver with 2 workers w_1 and w_2. Let's call p_i the current phase of w_i. If we simply calculate the hamming distance between workers, it takes only $O(n)$ time. However, to compute the distance between workers, they have to be synchronized, and this method would be unwieldy when the number of workers is increased.

When p_1 = 0, 1, 1, ..., 0 has been already visited and we fail to find a model, then we will want w_1 and w_2 to avoid the same status in the future. However, memoization for this needs a lot of memory. We do not compare these directly because of the synchronization problem, and we want to take into account past PSSI results.

III. Walk towards sparsely visited areas using history map

Using PSSIs we diversify the areas of the search space. Each worker calculates PSSI periodically, and we accumulate these data as a history map of PSSIs. Our main idea is to avoid the areas frequently visited, and to walk towards the sparsely visited areas. The history map is an 1-dimension array consisting of the PSSI counts. Each element counts how many times this area is visited. We can walk from the current area to the sparsely visited area by sharing the history map among workers and changing a phase dynamically. In this situation, we can not anticipate whether we will reach the sparsely visited area. It depends on the block division policy and the structure of the problem. Therefore we call this sparsely visited area the target area.

Algorithm 1 and 2 describe the pseudo-code of the SaSS (Sparsely visited area walking on Search Space) heuristic. In Alg. 1, after every *c* conflicts (line 1), each worker calculates the current area as PSSI (line 2), updates the history map of the PSSIs (shared for all workers) and gets a target area as a PSSI (line 3). If the target area is different from the current area (line 4), it changes polarities to walk towards the target area (line 5). A block is picked by calculating the bitwise XOR of *p* and *p′*, and each variable's polarity is updated. If the selected block is B_i, then each variable's polarity is allocated to TRUE b_i in *m*. In Alg 2, we get a target area using a history map. It updates history map (line2), but doesn't change the area in the early stages (line 3, 4). When the early stages end, it searchs the target area based on the current area (line 5). It checks areas within the hamming distance *d* from the current area, and the one with the minimal count in the history map is picked as the target area.

IV. ParaGlueminisat

In CHB [2], each variable has Q score, and is updated using Equation as follows based on reinforcement learning.

$$Q[v] = (1 - \alpha)Q[v] + \alpha r_v$$

Algorithm 1 SaSS heuristic: *changeCurrentArea*()

```
1: if conflicts % interval == 0 then
2:     p := getCurrArea();
3:     p′ := updateHistoryMap(p);
4:     if p != p′ then
5:         changeBlockPolarities(p, p′);
```

Algorithm 2 SaSS heuristic: *updateHistoryMap*(p)

Input: PSSI p
Output: PSSI p'

```
1: p′ := p;
2: historyMap[p′]++;
3: if p′ < c-threshold × thread number then
4:     return p′;
5: p′ := checkNearestAreas(p, d);
6: return p′;
```

We've selected several parameters those would change running time a lot to tune CHB and tested. The initial value of α is set to 0.4 in original CHB, and we changed this to 0.7 based on our tests. We've noticed CHB works very well with small problems, but VSIDS performs better than CHB for big problems. So, we divided problems for 2 groups by the number of variables. As default decision heuristic, our program choose VSIDS. If the number of variables is under 15000, CHB is activated and used behalf of VSIDS.

V. TBPARAGLUEMINSAT

Ties happen frequently in VSIDS. To break these, we update VSIDS scores after we obtain learned clauses. After a clause is obtained, we add 1 / (LBD of a clause) for each variables in that clause.

REFERENCES

[1] Moon, S., Inaba, M. Dynamic strategy to diversify search using a history map in parallel solving. LION 2016. (in press).

[2] Hui Liang, J., Ganesh, V., Poupart, P., Czarnecki, K. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

PolyPower: Random-SAT track participant in SAT Competition 2016 (PolyPower v1.0 and v2.0)

Sixue Liu
Institute for Interdisciplinary Information Sciences
Tsinghua University
Beijing, China

Periklis A. Papakonstantinou
School of Management Science and Information Systems
Rutgers University
New Jersey, USA

***Abstract*—This document describes the SAT solver "PolyPower", which is based on a new proposed theory for local search, with novel implementation and clause selection scheme.**

I. Introduction

Recently, a new theory for heuristics in SAT solving was proposed for the choices of the valuation function in WalkSAT [1]. This relies on the analysis of the break values: given a CNF and a complete assignment, the number of clauses turn to unsatisfied after flipping a variable. As a result, choosing a polynomial as the break valuation function is always better for any large enough uniform random k-CNF near the phase transition points (threshold) [1]. We also adopt separated-non-caching technology in the implementation to speed it up [2]. Additionally, a new clause selection scheme called *tbfs* is proposed, leading to a improvement for some categories of instances.

II. Main Techniques

Solver PolyPower is under the WalkSAT framework: start by a random complete assignment, in each step choose an unsatisfied clause c randomly or according to some rule. Within this clause c, if there exists 0-break variables then flip it[2], otherwise choose one variable v with probability $p(v) = \frac{f(\mathsf{break}(v))}{\sum_{v \in c} f(\mathsf{break}(v))}$ and flip v. Here the break valuation function is defined as:

$$f(x) = (((x-1)^{\frac{\kappa}{2}} + 2)^2 + \beta)^{-1}$$

κ and β are two parameters for PolyPower and will be adapted for different k-CNF and ratios.

Solver PolyPower adopts separated-non-caching technology for 3-SAT and 4-SAT, this separates the non-caching process in the break value calculation, resulted in an earlier termination of finding 0-break variables, which gives us a roughly 20% speed up.

Instead of randomly select an unsatisfied clause in each step, a new clause selection scheme *tbfs* is introduced: all the unsatisfied clauses are stored in an array U, new unsatisfied clauses are added to the end of U, and new satisfied clause are removed from the U, but keep the original order. In clause selection, *tbfs* first moves the first t clauses in U to the end of U, then choose the $t+1$ one. In other word, *tbfs* is similar to a "breadth-first-search" except the moving t clauses to the end operation. Empirical study shows that 3-SAT with ratio lower than threshold and 4-SAT benefit from *tbfs* with $t = 4$ distinctively.

[1]Rigourously it should be at the *anticipated phase transition points*, however this statement is also supported by our empirical study.

[2]Variable with break value of 0.

III. Main Parameters

	ratio	κ	β	clause selection
3-SAT	$[4.2, \infty)$	2	-0.08	random
	$[4.1, 4.2)$	2	-0.08	tbfs
	$(0, 4.1)$	2.3	-0.1	tbfs
4-SAT	$[9.7, \infty)$	4.0	0.06	tbfs
	$[8.8, 9.7)$	4.0	-0.1	tbfs
	$(0, 8.8)$	4.2	-0.1	tbfs
5-SAT	$[20.5, \infty)$	5.0	-0.17	random
	$[18, 20.5)$	4.6	-0.2	random
	$(0, 18)$	5.0	-1.1	random
6-SAT	$[40, \infty)$	6.0	0.2	random
	$[34, 40)$	6.4	-0.2	random
	$(0, 34)$	6.4	-0.4	random
7-SAT	$[80, \infty)$	7.0	0.35	random
	$[66, 80)$	7.0	-1.5	random
	$(0, 66)$	7.0	-1.8	random

TABLE I
Parameters Setting of PolyPower

The parameters setting of PolyPower is reported in Table 1. Note that the thresholds are contained in every first row of each k-SAT, and the optimal parameters found for these intervals are as same as for the thresholds: $r_3 = 4.267$, $r_4 = 9.931$, $r_5 = 21.117$, $r_6 = 43.37$, $r_7 = 87.79$ for 3-SAT to 7-SAT respectively.

IV. Implementation Details and SAT Competition 2016 Specifics

The current version of PolyPower is an incomplete solver designed for exact-k-SAT only[3], and $3 \leq k \leq 7$. For 3-SAT and 4-SAT, we adopt separated-non-caching in the implementation, while for 5-SAT and 6-SAT XOR-caching is adopted, and caching without XOR for 7-SAT.

Regarding *tbfs* scheme, we implement this using an array with dynamic size. Since moving new satisfied clause out of U causes empty slots in U, a garbage collection mechanism is required: when the size of array exceed the pre-set threshold, carry out defragmentation. We set this threshold value and design defragmentation delicately to make sure the amortized time complexity of *tbfs* is affordable per step.

Solver PolyPower is implemented in programming language C++, and compiled by "g++ polypower.cpp -O3 -static -o polypower". PolyPower is submitted to random SAT Track of SAT Competition 2016. The command line to run it is described as follows:

```
./PolyPower <instance filename> <seed>
```

The second argument "seed" is optional, and if not specified, the current system time is chosen as the initial seed. PolyPower 2.0 differs from PolyPower 1.0 in only minor code differences that we expect to only slightly affect the performance – we expect these minor changes to benefit PolyPower 2.0.

V. Availability

The PolyPower solver is open source and publicly available for only research purposes.

Acknowledgments

Many thanks to Spiros Papadimitriou from Rutgers (MSIS), and Wei Xu from Tsinghua (at Andrew Yao's institute) for providing us with the necessary computational power.

References

[1] S. Liu and P. A. Papakonstantinou, "Local search for hard sat formulas: the strength of the polynomial law," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*. AAAI Press, 2016.

[2] S. Liu, "An efficient implementation for walksat," *CoRR*, vol. abs/1510.07217, 2015. [Online]. Available: http://arxiv.org/abs/1510.07217

[3]Each clause has exactly k literals.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Scavel_SAT

Yang Xu
Southwest Jiaotong University
Chengdu, China

***Abstract*—Scavel_SAT is submitted to SAT Competition 2016. It is based on logic deduction, and proposes a new branching heuristics.**

I. Introduction

In this short paper, we present a derived version of MiniSat called Scavel_SAT. Scavel_SAT is developed from the code source of MiniSat 2.2.0 [1] with the bit-encoding phase selection strategy [2], while there is a significant change in Scavel_SAT by applying a kind of logic deduction compared with MiniSat 2.2.0. Specifically, based on this logic deduction, we can (i) obtain a set of partial assignment argument sequences; and (ii) obtain a new learning clause that is then added to the original CNF formula. The principle of this logic deduction applied in Scavel_SAT is summarized in Section II.

II. Logic Deduction

Resolution principle [3] is one of the most important methods for validating the unsatisfiability of logical formulas. Due to its simplicity, soundness and completeness, it has been adopted by most popular modern automated deduction systems. For further improving the efficiency of resolution, many refined resolution methods have been proposed such as linear resolution, semantic resolution, and lock resolution, etc. In this section, some preliminary concepts about binary resolution and linear resolution are reviewed firstly. A new way of deriving a local satisfiable assignment for a given clause set is then proposed by using the linear logic resolution deduction.

Definition 2.1 [3] Let C_1 and C_2 be clauses and L_1 a propositional variable. The clause $R(C_1, C_2) = C_1^{'} \vee C_2^{'}$ is called *a resolvent of clauses* $C_1 = L_1 \vee C_1^{'}$ and $C_2 = \neg L_1 \vee C_2^{'}$.

Definition 2.2 [3] Let S be a clause set. $\omega = \{C_1, C_2, \ldots, C_k\}$ is called a *resolution deduction* from S to C_k, if $C_i\,(i = 1, \cdots, k)$ is either a clause in S, or the resolvent of C_j and $C_r (j < i, r < i)$.

Definition 2.3 [3] Let S be a clause set, C_0 a clause in S, $\omega = \{C_1, C_2, \ldots, C_k\}$ is called a *linear resolution deduction* from S to C_k with the top clause C_0 if it satisfies:

1) C_{i+1} is the resolvent of C_i (a center clause) and B_i (a side clause), where $i = 0, 1, \ldots, n-1$.
2) $B_i \in S$ or $B_i = C_j (j < i)$.

Remark 2.1 In fact, linear resolution deduction provides a special and simple resolution deduction structure now that we just need to choose the side clauses.

Using the logic deduction, i.e., a linear resolution deduction method, we can derive a local satisfiable assignment for a given clause set S following the algorithm below:

Algorithm 1:
Step 1: firstly we choose a clause C_0 in S as the top clause;
Step 2: get a linear resolution deduction of S with the top clause C_0 where any complementary pair should not be appeared in all side clauses.

Therefore, for the given clause set S, there are two possible cases as below:

1) if an empty clause is derived from S using the above linear resolution, then S is unsatisfiable.
2) otherwise, we can obtain a local satisfiable assignment for S that are the resolvent literals in all side clauses, that is, $(B_{01}, B_{11}, \ldots, B_{i1})$, where B_{i1} is the resolvent literals in B_i.

Based on the local satisfiable assignment of S, we can construct or extend it to a global satisfiable assignment for S by using **Algorithm 1**.

In MiniSat, a decision variable p can be derived through the function pickBranchLit(), therefore we can start the logic linear resolution deduction from p. Since p is not a clause, in order to better combine the linear resolution deduction with the MiniSat decision variable selection strategy, we assume that p is from the tautology $\{p, \neg p\}$, and the resolvent R of contains $\neg p$ as a consequence.

Following several linear resolutions, we can obtain a set of local/partial assignment argument sequences. Then a new learning clause can be obtained that is then added to the original CNF formula in MiniSat. Due to the soundness and completeness of linear resolution, the Scavel_SAT is also sound and complete.

III. SAT Competition 2016 Specifics

We submit Scavel_SAT to the main track of SAT Competition2016.

Acknowledgment

Our work is partially supported by the National Natural Science Foundation of China (Grant No. 11526171,

61305074, 61175055 and 61100046), and the Fundamental Research Funds for the Central Universities of China (GrantNo. A0920502051305-24, 2682015CX060).

REFERENCES

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

[2] J. Chen, "A bit-encoding phase selection strategy for satisfiability solvers," in *in Proceedings of Theory and Applications of Models of Computation(TAMC14)*, ser. LNCS, vol. 8402, Chennai, India, 2014, pp. 158–167.

[3] C. L. Chang and R. C. T. Lee, *Symbolic logic and mechanical theorem proving*. USA: Academic Press, 1997.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

AICR_PeneLope 2016

Hitoshi Togasaki
The University of Tokyo
Tokyo, Japan

***Abstract*—In this paper, we show that simple introduction of our solver AICR_PeneLope 2016 submitted to SAT Competition 2016. We implemented Activate Idle Clause Restart(AICR) in PeneLope 2014(submitted SAT Competition 2014).**

I. Introduction

Portfolio algorithm is mainstream of parallel SAT Solvers. In Portfolio, to maintain diversification, Each worker choose different search strategy and parameter. Portfolio SAT Solvers solve problems efficiently by sharing useful learnt clauses. Search diversification and intensification are one of the most important factor for SAT Solving[1]. For this problem, we extract the areas of search space relatively not searched by using imported clause.

II. Activate Idle Clause Restart

Learnt clause sharing among workers is a fundamental task in the parallel SAT solvers. Learnt clauses not only prevent reappearances of the same conflits but also accelerate pruning of search spaces. In order to reduce communication costs, workers share only useful learnt clauses[2], i.e., clauses with short length or a low LBD value However, the workers not always utilize the imported learnt clauses. We define a learnt clause imported from other worker and never used in the search of the worker as a "idle clause" in this paper. In this context, a worker "uses" a clause if this clause appears in the propagation phase. The idle clauses are related with search spaces where the worker does not conduct the search. By forcing the workers to assign values so that the idle clauses become unit, we can change the search spaces of the workers. Especially, it is important that the idle clauses are relevant to the search spaces where the worker does not conduct the search but other wokers do.

We show the pseudocodes of our proposal in Algorithm 1 and 2. The main function is in Algorithm 1. This function is called for every restart. The input of this function is a list of worker IDs. The worker activates the idle clauses from the workers with IDs in this list. In addition, we select the clauses with a low LBD value (i.e., lower than average LBD value of imported). For each selected idle clause, we increase the VSIDS score of each variable in the clause according to the "BUMP_RATIO" constant and add to "idle clause vars" sets. Note that the "BUMP_RATIO" is always 1 in the original VSIDS. In this function, the polarity (true or false) is also assigned so that the literals in the target clause become false.

The decision fucntion is in Algorithm 2. This function is called for every decision. We select a variable with the highest VSIDS score, and call this v. If v is included in the "idle clause vars" set, we allocate each variable to make idle clause false, and erase v from "idle clause vars" set.

Algorithm 1 ActivateIdleClauseRestart

Input: $tid \Leftarrow$ target worker IDs
1: $ICVars \Leftarrow \emptyset$
2: **for** $clause \in$ learnt clauses **do**
3: **if** $clause$ is imported clause from workers with ID $\in tid$ **and** is idle **and** its LBD value is less than the average LBD values of imported **then**
4: **for** $v \in clause$ **do**
5: **if** $v \notin ICVars$ **then**
6: $ICVars \Leftarrow ICVars \cup v$
7: $ICPolarity[v] \Leftarrow$ not $sign(v)$
8: $BumpVSIDS(v, BUMP\_RATIO)$
9: **end if**
10: **end for**
11: **end if**
12: **end for**

Algorithm 2 Idle Clause based Decision

1: $v \Leftarrow \mathrm{argmax}\, VSIDS$
2: **if** $v \in ICVars$ **then**
3: $ICVars \Leftarrow ICVars \setminus v$
4: $decision(v, ICPolarity[v])$
5: **else**
6: $decision(v, Polarity[v])$
7: **end if**

III. AICR_PeneLope

We implemented AICR in PeneLope[3](version 2014) and modified the parameter(related threads). We submitted two solvers AICR_PeneLope(threads 24 and 48) and set parameter for 'BUMP_RATIO=100'and tid(below).
Letting p as the number of workers, the ID of workers ranges from 0, to $p-1$. For each worker with ID i,

- Pairwise: $tid \Leftarrow \{i+1\}$ if i is even, otherwise $tid \Leftarrow \{i-1\}$

References

[1] L. Guo, Y. Hamadi, S. Jabbour, and L. Sais, "Diversification and intensification in parallel sat solving," in *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, ser. CP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 252–265.

[2] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs, *On Freezing and Reactivating Learnt Clauses*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 188–200.
[3] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette, *Revisiting Clause Exchange in Parallel SAT Solving*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 200–213.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

AmPharoS, An Adaptive Parallel Solver

Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, Sébastien Tabary
Univ. Lille-Nord de France
CRIL/CNRS UMR8188

***Abstract*—We present AMPHAROS, a new parallel SAT solver based on the divide and conquer paradigm. This solver, designed to work on a great number of cores, runs workers on sub-formulas restricted to cubes. In addition to classical clause sharing, it also exchange extra information associated to cubes.**

I. MODUS OPERANDI

AMPHAROS is a parallel distributed SAT solver that uses the divided and conquer approach. All details of AMPHAROS are available here [1].

Even if AMPHAROS is a divide and conquer based solver, it is important to stress that, contrary to [2], it does not use the work stealing strategy. In our case, the division is performed in a classical way as in [3], [4]. More precisely, our approach generates guiding paths, restricted to cubes, that cover all the search space. This way, the outcome of the division is a tree where nodes are variables and the left (resp. right) edge corresponds to the assignment of the variable to true (resp. false). Then, solvers operate on leaves (represented by the symbol NIL) and solve (under assumptions) the initial formula restricted to a cube which corresponds to the path from the root to the related leaf. Fig. 1 shows an example of a tree containing three open leaves (cubes $[x_1, \neg x_2, x_4]$, $[x_1, \neg x_2, \neg x_4]$ and $[\neg x_1, \neg x_3]$), two closed branches (already proven unsatisfiable) and four solvers ($S_1 \dots S_4$) working on these leaves.

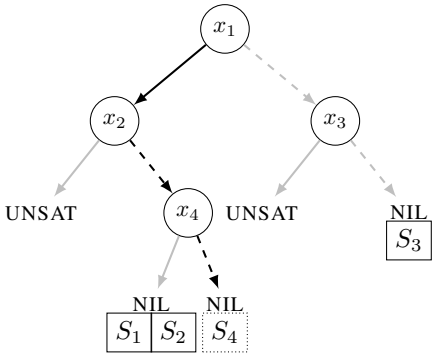


Fig. 1. AMPHAROS architecture.

In our architecture, solvers choose by themselves which cubes they try to solve. Then, solvers can work on the same cube (as solvers S_1 and S_2 in Fig. 1a) and can stop working before finding a solution or a contradiction. In AMPHAROS, each time a solver shares information or asks to solve a new cube, it communicates with a dedicated worker, called MANAGER. Its main mission is to manage the cubes and the communication between the solvers (here CDCL solvers). Thus, when a solver decides to stop solving a given cube (without having solved the instance), it can ask the MANAGER to enlarge this one. Indeed, cubes that seem difficult to solve are extended. This is done by adding to each leaf a counter increased each time a solver is not able to solve it in a given amount of conflicts.

Another situation where a solver stops, is once a branch is proved to be unsatisfiable. In this case, a message informs the MANAGER and the tree is updated in consequence. In both cases, when a solver stops it goes through the tree and starts solving a new cube (potentially the same). The end of the solving process finally occurs either when a cube is proved to be satisfiable or when the tree is proved to be unsatisfiable.

Different knowledge sharing are achieved during the solving process. First, classical learnt clauses that appear to be good with respect to LBD measure [5]. Note that solvers communicate learnt clauses to the MANAGER that takes over distibution to other solvers. AMPHAROS shares also assumtive unit literals, that is, literals that are unit with respect to the current cube. This allows to reduce communications and to be sure that learnt clauses that have a key role under such assumptions are effectivelly shared.

Finally, AMPHAROS contains a strategy that permit to intensify or diversify the search. This is done by measuring the number of subsumed clauses that MANAGER recovers. This can be summarized here:

Few subsumed Clauses	**Many subsumed clauses**
Reduce extension Increase the number of imported clauses	Favour extension Limit the number of imported clauses
Intensification	**Diversification**

Results (cactus plots, scatter plots, ...) and further explanations are available in http://www.cril.univ-artois.fr/ampharos/.

II. ALGORITHM AND IMPLEMENTATION DETAILS

AMPHAROS uses Open MPI library to ensure communication. It uses 3 different CDCL solvers: GLUCOSE [5], MINISAT [6] and MINISATPSM [7].

REFERENCES

[1] G. Audemard, J.-M. Lagniez, N. Szczepanski, and S. Tabary, "An adaptive parallel sat solver," in *Proc of International Conference on Principles and Practice of Constraint Programming*, 2016, p. To appear.

[2] T. Schubert, M. Lewis, and B. Becker, "Pamiraxt: Parallel SAT solving with threads and message passing," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, no. 4, pp. 203–222, 2009.
[3] G. Audemard, B. Hoessen, S. Jabbour, and C. Piette, "An effective distributed d&c approach for the satisfiability problem," in *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014, pp. 183–187.
[4] G. Chu, P. J. Stuckey, and A. Harwood, "Pminisat: a parallelization of minisat 2.0," SAT Race, Tech. Rep., 2008.
[5] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, 2009.
[6] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.
[7] G. Audemard, J. Lagniez, B. Mazure, and L. Sais, "On freezing and reactivating learnt clauses," in *International Conference on Theory and Applications of Satisfiability Testing*, 2011, pp. 188–200.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

"Beans and Eggs" Proteins for Glucose 3.0

Markus Iser
Karlsruhe Institute of Technology
Karlsruhe, Germany

***Abstract*—This document describes the Glucose 3.0 Hack "Beans and Eggs" that combines a gate recognition algorithm with the variable selection heuristic.**

I. Introduction

This system description describes the "Beans and Eggs" hack of Glucose 3.0 ([1], [2]). The hack includes application of the gate-recognition algorithm that has been presented in [3]. Based on the result of gate structure analysis, we use a variant of input branching ([4], [5]).

We use a slim version of gate-recognition [3] as a preprocessor. The recognition result is used to initialize the activity values of the input variables. In this context an input variable is any variable that our algorithm did not recognized as being the output of a gate. Note that if no gate is recognized, every variable is treated as input variable.

II. Implementation details

Given the input formula F, our slim version of the gate-recognition algorithm uses the unit clauses $U \subseteq F$ to start its recursion. If there are no unit clauses, no recognition takes place; i.e we got rid of the clause-selection loop and heuristic we described in [3].

The recursive part of the gate-recognition is displayed in Algorithm 1. For every $u \in U$ we enter the recursion by invocation of $\mathsf{extractGates}(u, F \setminus U)$.

Algorithm 1: $\mathsf{extractGates}(o, F)$

Data: F : CNF formula, o : output literal
Result: C : subset of F that is part of the gate structure

```
1  C ← ∅
2  if blockedSet(o, F_ō, F_o) then
3      if ¬inp[ō] ∨ fullEnc(o, F_ō ∪ F_o) then
4          C ← C ∪ {F_ō ∪ F_o}
5          output[o] ← true
6          for p ∈ literals(F_ō) \ {ō} do
7              inp[p] ← true
8              if inp[ō] then inp[p̄] ← true
9          for p ∈ literals(F_ō) \ {ō} do
10             C ← C ∪ extractGates(p, F \ C)
11 return C
```

For the non-monotonic case, i.e. when *both* implications in a Tseitin encoding are needed, we utilize a simple pattern-based recognition that is capable of n-ary `and`, `or` and binary `xor` detection.

A. Activity Initialization

Let $h(v)$ be the number of occurrences of the variable v and $m = \max_{v \in V}(h(v))$ be its maximum. We bump each input variable v by the scaled value $\frac{h(v)}{m} \leq 1$.

```
for v ∈ vars(F) do
    if ¬output[v] ∧ ¬output[v̄] then
        varBumpActivity(v, h(v)/m);
```

We hereby establish a pre-ordering of variables that favours frequent input variables. The ordering blurs rapidly as activity-values are adjusted during the solving process.

III. Availability

The hack is available at the website `http://baldur.iti.kit.edu/sat-competition-2016`.

Acknowledgment

The author would like to thank Felix Kutzner for many fruitful related discussions.

References

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

[2] 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*, 2003, pp. 502–518.

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

[4] M. Järvisalo and T. A. Junttila, "Limitations of restricted branching in clause learning," in *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, 2007, pp. 348–363.

[5] M. Iser, M. Taghdiri, and C. Sinz, "Optimizing minisat variable orderings for the relational model finder kodkod - (poster presentation)," in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, 2012, pp. 483–484.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

CBPeneLoPe2016, CCSPeneLoPe2016, Gulch at the SAT Competition 2016

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

***Abstract*—In this description, we provide a brief introduction of our solvers: PeneLoPe2016, CCSPeneLoPe2016, and Gulch. PeneLoPe2016 and CCSPeneLoPe2016 are based on the parallel SAT solver PeneLoPe. Gulch is based on Glucose version 3.**

I. PeneLoPe2016

PeneLoPe2016 is a parallel portfolio SAT solver based on PeneLoPe [3] and a new version of ones submitted in the SAT Competition 2014 and SAT Race 2015. PeneLoPe2016 implements *community branching* [9], a diversification [7] technique using community structure of SAT instances [1]. The community branching assigns a different set of variables (community) to each worker and forces them to select these variables as decision variables in early decision levels, aiming to avoid overlaps of search spaces between the workers more vigorously than the existing diversification methods.

In order to create communities, we construct a graph where a vertex corresponds to a variable and an edge corresponds to a relation between two variables in the same clause, proposed as Variable Incidence Graph (VIG) in [1]. After that, we apply Louvain method [5], one of the modularity-based community detection algorithms, to identify communities of a VIG. Variables in a community have strong relationships, and a distributed search for different communities can benefit the whole search. Recently, interesting works based on the community structure or from a point of view of graph theory are introduced [2], [6].

The differences between PeneLoPe2016 and previous versions are as follows.

- Community based learnt clause sharing
- Changes of some parameters
- Refactoring for some parts of the program

A. Community-Based Learnt Clause Sharing

In the previous version of PeneLoPe2016, we used the default function of original PeneLoPe for learnt clause sharing between workers. However, there are so many learnt clauses that are imported but deleted without being used ("used" stands for being used in BCP). Importing such clauses results in vain hence we should cease importing them to reduce various costs (e.g., memory consumption, lock waiting time). One of the reasons why such clauses can be deleted is that the worker searches in different regions (or simply already satisfies these clauses). Using the community branching, the workers can be forced to search specific areas (based on the assigned communities). Thus, the most likely used learnt clauses are ones that include the variables belonging to the assigned communities.

We propose a new method for clause sharing in parallel SAT solvers, *community-based learnt clause sharing (CLCS)*. The CLCS conducts the community detection algorithm on the VIG of target SAT instance. Then, this method restricts the sharing of each learnt clause to workers that conducts the search for the variables related with communities in the target learnt clause. By combining the community branching, the CLCS distributes the target clauses to the workers with related communities. For example, if a learnt clause $(a \vee b \vee c)$ is to be shared among the workers, and the variable a and b belong to a community C_1 and the variable c belongs to a community C_2, this clause is distributed only to the workers that are assigned the community C_1 or C_2 by the community branching.

II. CCSPeneLoPe2016

CCSPeneLoPe2016 is a parallel portfolio solver based on PeneLoPe. The features of CCSPeneLoPe2016 are as follows.

- Conflict history-based branching heuristic (CHB) [8] for some workers
- CLCS prioritizing high VSIDS or CHB scores

The CHB is good at cryptographic instances in [8]. In CCSPeneLoPe2016, some workers use this heuristic with different sets of its parameters. For the CLCS, each worker calculates an average activity score (VSIDS or CHB) of variables for each community and chooses the highest scored community as a "desired community". The CLCS distributes the target clause to the workers that desire to share that clause (i.e., including the variables that belong to the desired community).

III. Gulch

Gulch is a sequential solver based on Glucose version 3 [4]. The decision heuristic of Gulch is based on the CHB. In preliminary experiments, we confirmed that CHB was not good at some types of unsatisfiable instances. To mitigate this issue, we add a simple heuristic based on the number of variables and clauses. This heuristic switches the CHB and VSIDS alternately for every K restarts, and randomized the

activity scores. In Gulch, K is set to 1024, and 128 in the "agile" version of Gulch. In the "once" version of Gulch, the switching from CHB to VSIDS is conducted only once after 50 seconds passes.

IV. Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 15K16057.

References

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

[2] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Using community structure to detect relevant learnt clauses. In *Theory and Applications of Satisfiability Testing*, SAT'15, pages 238–254, 2015.

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

[4] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 399–404, 2009.

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

[6] Jesús Giráldez-Cru and Jordi Levy. A modularity-based random SAT instances generator. In *International Joint Conference on Artificial Intelligence*, IJCAI'15, 2015.

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

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

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

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

CHBR_glucose

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

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

***Abstract*—We briefly introduce our solver CHBR_glucose, CHBR_glucose_tuned, tb_glucose and tc_glucose submitted to SAT-Competition 2016. All solvers are based on glucose3.0, and CHB, introduced at AAAI 2016, is implemented in CHBR_glucose, CHBR_glucose_tuned, and tc_glucose. CHBR_glucose_tuned is for entering the Glocose Hack track in the SAT Competition 2016.**

I. Introduction

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

Recently new branching heuristic CHB[5] was provised and it showed significant improvements for some benchmarks.

In our program, we implemented CHB and select decision heuristic between VSIDS and CHB dynamically.

When a variable is selected by the score of VSIDS a lot of ties happened. We added some scores to VSIDS to reduce ties, and select more valuable variable from ties.

II. CHB tuned

In CHB, each variable has Q score, and is updated using Equation as follows based on reinforcement learning.

$$Q[v] = (1 - \alpha)Q[v] + \alpha r_v$$

We've selected several parameters those would change running time a lot to tune CHB and tested. The initial value of α is set to 0.4 in original CHB, and we changed this to 0.7 based on our tests.

III. CHBR_glucose

We've noticed CHB works very well with small problems, but VSIDS performs better than CHB for big problems. So, we divided problems for 2 groups by the number of variables. As default decision heuristic, our program choose VSIDS. If the number of variables is under 15000, CHB is activated and used behalf of VSIDS.

IV. CHBR_glucose_tuned

We've tuned CHB parameters based on 24 combination tests. Some instances work better than default parameter values. We've changed initial value of α, minimum of α, and $multiplier$ for small problems.

$if(2000 < numberofvariables < 7000)$
$\alpha = 0.4, \alpha_{min} = 0.03, multiplier = 0.5$

V. tb_glucose

Ties happen frequently in VSIDS. To break these, we update VSIDS scores after we obtain learned clauses. After a clause is obtained, we add 1 / (LBD of a clause) for each variables in that clause. We call this TBVSIDS.

VI. tc_glucose

This is a hybrid version of CHBR_glucose and tb_glucose. We use TBVSIDS as a default decision heuristic and use CHB when the number of variables is under 15000.

References

[1] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S. Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Design Automation Conference, pp 530–535, 2001.
[2] Dershowitz, Nachum and Hanna, Ziyad and Nadel, Alexander. A Clause-Based Heuristic for SAT Solvers. Theory and Applications of Satisfiability Testing, pp 46–60, 2005.
[3] Goldberg, Evgueni and Novikov, Yakov. BerkMin: A Fast and Robust Sat-Solver. Design, Automation, and Test in Europe, pp 465–478, 2008.
[4] L.Ryan. Efficient algorithms for clause-learning SAT solvers. Matser's thesis, Simon Fraser University, 2004.
[5] Hui Liang, J., Ganesh, V., Poupart, P., Czarnecki, K. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016.
[6] Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment. (2008).

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

The CryptoMiniSat 5 set of solvers at SAT Competition 2016

Mate Soos

I. Introduction

This paper presents the conflict-driven clause-learning SAT solver CryptoMiniSat v5 (*CMS5*) as submitted to SAT Competition 2016. CMS5 aims to be a modern, open-source SAT solver that allows for multi-threaded in-processing techniques while still retaining a strong CDCL component. In this description only the features relative to CMS4.4, the previous year's submission, are explained. Please refer to the previous years' description for details. In general, CMS5 is a in-processing SAT solver that usues optimized datastructures and finely-tuned timeouts to have good control over both memory and time usage of simplification steps.

A. Removal of uneeded code

Over the years, many lines of code has been added to CMS that in the end didn't help and often was detrimental to both maintinability and efficiency of the solver. Many such additions have now been removed. This simplifies understanding and developing the system. Further, it allows the system to be more lean especially in the tight loops such as propagation and conflict analysis where most of the time is spent.

B. Integration of ideas from COMiniSatPS

Some of the ideas from COMiniSatPS[1] have been included into CMS. In particular, the clause cleaning system employed and the switching restart have both made their way into CMS.

C. On-the-fly Gaussian Elimination

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

D. Clause usefulness guessing

Besides glues and clause activites, CMS5 also tries to guess clause usefulness based on the trail size, the backjump level and the activity of the variables in the ancestor of the learnt clause. Although this is at a very early stage of development, it has been found to be helpful.

E. Auto-tuning

The version 'autotune' reconfigures itself after about 160K conflicts. The configuration picked is one of 2 different setups that vary many different parameters of the solving such as learnt clause removal strategy, restart strategy, and in-processing strategies. CMS5 was run on all SAT Comp'09 + 11 + 13 + 14 + 15 problems with both configurations, extracting relevant information from the all problems after they have been solved and simplified for 160K conflicts. configurations were then given to a machine learning algorithm (C5.0[2]) which built a decision tree from this data. This decision tree was then translated into C++ and compiled into the CMS5 source code.

References

[1] Oh, C.: MiniSat HACK 999ED, MiniSat HACK 1430ED and SWDiA5BY. In: SAT Competition 2014 Booklet. (201)

[2] Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS

Chanseok Oh
Google
New York, NY, USA

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

I. Introduction

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

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

[1]Source is available at http://www.cs.nyu.edu/~chanseok/cominisatps/.

II. COMiniSatPS the Chandrasekhar Limit

Differences from the last year's COMiniSatPS Main Sequence [5] are as follows:

- Always performs pre-processing.
- Applies a small patch implementing what we call *Incrementally Relaxed Bounded Variable Elimination*[2] which was first proposed by GlueMiniSat last year [6].
- LBD [3] of new learned clauses is one less than what it used to be. Code that compares LBD values has been modified accordingly in a few locations.
- Performs on-the-fly failed literal detection through advanced stamping [7], however, very sparingly. It may be triggered only when learning unit clauses. We referenced the implementation of Lingeling [8]. This is the only feature that resulted in a complex implementation which unfortunately contradicts the minimalistic spirit of COMiniSatPS.

III. Incremental SAT Solving

Nothing has changed since last year except that solvers now use a sane strategy and a reasonable parameter value; solvers submitted to SAT Race 2015 last year intentionally used an unreasonable strategy for demonstration purposes. Specifically, in this year's solvers, mid-tier and local learned clauses are not purged but preserved after each incremental run.

A. 2Sun

Like the last year's 1Sun version, the 2Sun version does not employ the hybrid restart strategy of COMiniSatPS. Incremental variable elimination [9] is turned off for small problem instances.

B. 2Sun_nopre

Incremental variable elimination is always turned off. The threshold for purging core learned clauses after each incremental run has been slightly relaxed: when the size (not LBD) is greater than 8 (increased from 5).

[2]The authors of GlueMiniSat call this feature *Incremental Variable Elimination*. However, we avoid the original term because we have been using it to refer to variable elimination in the context of incremental SAT.

IV. GHackCOMSPS

This solver implements many of the core features of COMiniSatPS on top of Glucose 3.0: 1) the 3-tiered learned clause management; 2) the hybrid restart strategy; and 3) the alternating variable decay factors (but without the separate activity score sets or variable priority queues). There are several other minor changes too. GHackCOMSPS qualifies as a Glucose hack.

V. Availability and License

Source is available for download for all the versions in this paper. COMiniSatPS uses the same MIT license as MiniSat's.

Acknowledgment

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

References

[1] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003.

[2] C. Oh, "Improving SAT solvers by exploiting empirical characteristics of CDCL," Ph.D. dissertation, New York University, 2016.

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

[4] C. Oh, "Between SAT and UNSAT: The fundamental difference in CDCL SAT," in *SAT*, 2015.

[5] ——, "Patching MiniSat to deliver performance of modern SAT solvers," in *SAT-RACE*, 2015.

[6] H. Nabeshima, K. Iwanuma, and K. Inoue, "GlueMiniSat 2.2.10 & 2.2.10-5," in *SAT-RACE*, 2015.

[7] M. Heule, M. Järvisalo, and A. Biere, "Efficient CNF simplification based on binary implication graphs," in *SAT*, 2011.

[8] A. Biere, "Lingeling, Plingeling and Treengeling entering the SAT Competition 2013," in *SAT-COMP*, 2013.

[9] J. Ezick, J. Springer, T. Henretty, and C. Oh, "Extreme SAT-based constraint solving with R-Solve," in *HPEC*, 2014.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

BreakIDCOMiniSatPS

Jo Devriendt
University of Leuven
Leuven, Belgium

Bart Bogaerts
Aalto University
Espoo, Finland

***Abstract*—BreakIDCOMiniSatPS combines the COMiniSatPS SAT solver with the symmetry breaking preprocessor BreakID.**

I. Introduction

Many real-world problems exhibit symmetry, but the SAT competition and SAT race seldomly feature solvers who are able to exploit symmetry properties. This discrepancy can be explained by the assumption that for most of the problems in these competitions, symmetry exploitation is not worth the incurred overhead.

We tested this hypothesis in 2013's SAT competition and 2015's SAT race, and now participate again 2016's SAT competition. Symmetry is broken in the spirit of Shatter [1]. As symmetry breaking preprocessor we use BreakID [2] version 2.2, a slightly more efficient version of BreakID than the 2.0 version used in last year's SAT race. As SAT solver we use COMiniSatPS [3], with the same source as the COMiniSatPS competition submission without symmetry breaking. We expect COMiniSatPS to be more effective than the Glucose [4] solver used in last year's SAT race.

II. Main Techniques

The workflow of BreakIDCOMiniSatPS is straightforward:

1) BreakID uses Saucy [5] to enumerate symmetry generators for an input CNF theory.
2) BreakID analyzes these generators for certain properties, and constructs effective symmetry breaking clauses.
3) COMiniSatPS solves the resulting CNF theory, consisting of the original clauses and the symmetry breaking clauses.

III. Main Parameters

The main user-provided parameters control:

- How much time should be allocated to Saucy for symmetry detection. This does not limit further analysis of these generators to construct symmetry breaking clauses. Given a time limit of 5000 seconds to solve one CNF instance, Saucy gets 200 seconds to detect symmetry generators.
- How large the symmetry breaking sentences are allowed to grow, measured in the number of auxiliary variables introduced by a symmetry breaking formula. We limit this to 50 auxiliary variables.

IV. Special Algorithms, Data Structures, and Other Features

In comparison with Shatter, BreakID offers the following improvements:

- A more efficient clausal encoding of the Lex-Leader symmetry breaking formula [6].
- Detection of row interchangeability symmetry groups, which can be broken completely.
- Construction of binary symmetry breaking clauses, which potentially break much symmetry at little cost.
- Construction of a more suitable variable ordering with which to break symmetry.
- A limit on the size of a symmetry breaking formula.
- A time limit on Saucy's symmetry detection routine.

Lastly, Saucy requires a slightly cleaned CNF as input, so the BreakID preprocessor also employs a small preprocessing step:

- Removing duplicate and tautological clauses from the input CNF theory.

V. Implementation details

BreakID was written from scratch in C++. We refer to the webpages of Saucy and COMiniSatPS for their implementation details.

VI. SAT Competition 2016 Specifics

BreakIDCOMiniSatPS participates in the No-Limit track since BreakID does not support outputting DRAT proofs of unsatisfiability. The compiler used is the one provided by the competition organizers.

VII. Availability

Source code and documentation for BreakID is available under a non-commercial license [7].

Acknowledgment

We would like to thank the authors of Saucy for providing the source code to Saucy. Thanks also go to the authors of MiniSat [8], Glucose and COMiniSatPS for making their solvers publically available.

REFERENCES

[1] F. A. Aloul, K. A. Sakallah, and I. L. Markov, "Efficient symmetry breaking for Boolean satisfiability," *IEEE Transactions on Computers*, vol. 55, no. 5, pp. 549–558, 2006.

[2] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker, "Improved static symmetry breaking for sat," to appear in the proceedings of SAT'16, 2016.

[3] C. Oh, "Improving sat solvers by exploiting empirical characteristics of cdcl," PhD thesis, New York University, cs.nyu.edu/media/publications/oh_chanseok.pdf, 2016.

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

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

[6] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy, "Symmetry-Breaking Predicates for Search Problems," in *Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1996, pp. 148–159.

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

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

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Dissolve in the SAT Competition 2016

Julien Henry
University of Wisconsin-Madison

Aditya Thakur
Google, Inc.

Nick Kidd
Google, Inc.

Thomas Reps
University of Wisconsin-Madison
Grammatech, Inc.

***Abstract*—Dissolve is a distributed SAT solver that uses a search-space partitioning and merging mechanism to divide the work and to share information among a pool of sequential CDCL solvers. Learnt clauses are shared among solvers, and partitioning evolves over time as a function of the current states of the solvers.**

I. Introduction

Dissolve is a new distributed solver that is being entered into the SAT competition for the first time in 2016. It is designed to work with any underlying sequential CDCL SAT solver that implements a small API. The implementation uses Glucose 3.0 [1], which is itself based on Minisat [2]. Each sequential solver communicates with a master solver written in the Go programming language. In essence, the algorithm implemented in Dissolve explores the search space using an approach to search-space exploration that was inspired by Stålmarck's method [3]:

- split the problem into subproblems
- learn as much as possible about the subproblems that is relevant to the original problem instance within a limited period of time
- merge the knowledge from all instances, and restart with a new and different split.

Dissolve uses a pool of CDCL SAT solvers, each of which processes queries that it receives from the master solver. Because each query is the conjunction of the original formula with a partial assignment to a subset of k variables, the solvers can maintain their state across queries: the k input assignments are treated as the first k decisions and everything that is learnt holds for the original formula. Solver state includes learnt clauses, polarities, VSIDS information, LBD values, etc. In the competition, we use a pool of 48 sequential Glucose 3.0 solvers—one for each logical CPU of a single node of the competition server. These solvers never stay idle: even if one does not receive a query from the master solver (which can happen for short periods of time), it initiates a search (seeded with the next random value). This activity is useful because the solver will start the next query with a "better" initial state.

II. Main Parameters

Sequential solvers have their own performance-sensitive parameters. For the competition, we used the default parameters of Glucose 3.0. Below, we discuss the parameters of the master solver.

1) Splitting strategy: One of the most important parameters in Dissolve is the number of variables k that we use for the splits. We call a *round* the solving of the 2^k SAT queries obtained by selecting a vector of k variables from the original formula and assigning them all possible combinations of truth values. The selection of the variables used for splitting is done by a vote among all solvers. Every solver returns a sequence of the first 100 decision variables that would have been chosen in the sequential case (using the `pickBranchLit` method from Minisat). The first one is assigned a vote 100, the second 99, etc. The k variables with the highest total score are those selected for the next round. The first round does not split with k variables; instead, each sequential solver is run on the original formula with a different random seed. In the competition, we used a value of $k = 5$, each round consisting of 32 queries. This means that different rounds can be solved in parallel at the same time.

2) Merging strategy: SAT queries that are sent to the various sequential solvers do not usually run until they finish; instead, they return once a budget limit has been reached. The budget can be based either on the number of propagations, the number of conflicts, or a timeout limit. For the competition, we set a timeout of 5 seconds or 2.10^7 propagations, whatever comes first. When the budget limit is reached, the sequential solver returns the list of the N *most-useful* clauses it has learnt. We chose N to be equal to $min(1000, 100000/2^k)$.

A UBTree (*Unlimited Branching Tree*) [4] is a data structure for storing a set of clauses that allows subsumption checking to be performed relatively inexpensively. In Dissolve, the master solver inserts clauses into three different UBTrees, depending on their importance: Dissolve's sequential solvers report *small* clauses (size ≤ 2), *important* clauses (the 100 *best* clauses according to Glucose's heuristic based on LBD), and *other* clauses.

When a sequential solver returns SAT, the master solver interrupts all computations and returns SAT. When a solver returns from a query with an UNSAT answer, it also returns a conflict clause. If the conflict clause is empty, the master solver returns UNSAT and the problem is solved. If the conflict clause is not empty, the master solver cancels the queries that the given conflict clause implies are UNSAT.

3) Clause sharing: When a new query is sent to a sequential solver, the master solver also sends a set of at most 50,000 learnt clauses that the given sequential solver has not yet received. Up to 50,000 learnt clauses are sent using the priorities *small*, *important*, *other*. To avoid the accumulation of too many clauses, the master solver's clauses are incorporated

by each sequential solver after discarding as many clauses in its local database as the number of master-solver clauses received.

4) Random seeds: With each new query, a sequential solver also receives a new unique random seed to replace the previous one.

III. Implementation Details

Dissolve has been implemented to be run in a distributed setting (i.e., on a cloud-computing platform), and some of our design decisions have been made with that goal in mind. Consequently, Dissolve relies on heavier-weight communication mechanisms than the threads and shared memory of a shared-memory multicore machine. In particular, all information between the master solver and the sequential solvers is exchanged using the Google protocol-buffer binary format. While other parallel solvers exchange learnt clauses at very high rates, Dissolve exchanges information only every couple of seconds. Because we wish Dissolve to be able to scale to a large number of slave sequential solvers, we designed it to use low-cost approaches to (a) problem splitting, (b) obtaining learnt clauses that can be used unconditionally for all solvers, (c) merging learnt clauses plus heuristic-search information from the different slave sequential solvers, and (d) propagating information from the master to the slaves—while simultaneously attaining near-100% CPU utilization. (The tuning and evaluation of Dissolve on a cloud-computing platform is underway.)

IV. Related Work

The algorithm used in Dissolve has similarities with the *Cube-and-Conquer* approach of Heule et al. [5]. One of the main differences is that the algorithm in Dissolve performs merging and readjusts the splitting variables in subsequent rounds based on the sequential solvers' states. The algorithm in Dissolve is also related to previous work by Hyvärinen et al. [6], [7], [8], but adopts different approaches and heuristics for splitting, merging, and sharing learnt clauses. (See Section III.)

V. SAT Competition 2016 Specifics

To make sure that all sequential solvers have been started and are listening to the network for new queries, we allow a fixed time of 1.5s seconds before the solving of a benchmark actually starts, which means that solving even very simple benchmarks takes at least 1.5 seconds.

References

[1] G. Audemard and L. Simon, "Lazy clause exchange policy for parallel SAT solvers," in *SAT*, 2014.

[2] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003.

[3] M. Sheeran and G. Stålmarck, "A tutorial on Stålmarck's proof procedure for propositional logic," *FMSD*, vol. 16, no. 1, pp. 23–58, 2000.

[4] J. Hoffmann and J. Koehler, "A new method to index and query sets," in *IJCAI*, 1999, pp. 462–467.

[5] M. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *HVC*, 2011, pp. 50–65.

[6] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä, "Partitioning SAT instances for distributed solving," in *LPAR*, 2010, pp. 372–386.

[7] ——, "Grid-based SAT solving with iterative partitioning and clause learning," in *CP*, 2011, pp. 385–399.

[8] A. E. J. Hyvärinen and N. Manthey, "Designing scalable parallel SAT solvers," in *SAT*, 2012, pp. 214–227.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Glucose_nbSat

Chu Min LI*†, Fan Xiao* and Ruchu XU*
*Huazhong University of Science and Technology, China
†MIS,Universit de Picardie Jules Verne, France

***Abstract*—This document describes the SAT solver "GlucoseNBSAT", a solver based on glucose 3.0. We present a new measure called nbSAT based on the saved assignment to predict the usefulness of a learnt clause when reducing clause database.**

I. Introduction

In this paper, we present a versions based on Glucose called *Glucose_nbSat* . Glucose_nbSat is developed from the code source of Glucose-3.0[1], [2] by implementing a significant change in the learnt clause database management and a limited redundant clause simplification and removing.

II. Main features of Glucose

Glucose is a very efficient CDCL-based complete SAT solver. It is always one of the awarded winning SAT solvers in SAT competitions (challenge, race) since 2009. The main features of Glucose and its updated versions include a measurement of learnt clause usefulness called LBD and used in the cleaning of the learnt clause database. In the recent versions of Glucose such as Glucose-3.0, once the number of clauses learnt since the last database cleaning reaches 2000 + 300*x, where x is the number of database cleanings performed so far, the cleaning process is fired (i.e., the learnt clauses are sorted in the decreasing order of their LBD, and the first half of learnt clauses are removed except binary clauses, clauses whose LBD value is equal to 2, and the clauses that are reasons of the current partial assignment. In addition, Glucose-3.0 uses a very aggressive restart strategy [3], in such a way that the solver is very frequently restarted.

Our new learnt clause database management is based on the above features of Glucose-3.0.

III. Cleaning learnt clause database at the root of a search tree

A CDCL based SAT solver usually uses the restart mechanism [4], every restart constructing a binary search tree from scratch. In Glucose, as well as in most CDCL-based solvers, a learnt clause database cleaning process can be fired inside a binary search tree. Two observations can be made about this strategy: (1) there are locked clauses, i.e. clauses that are reasons of the current partial assignment, that cannot be removed, (2) the part of the tree before the cleaning and the part of the tree after the cleaning are constructed with very different learnt clause databases.

Glucose_nbSat differs from Glucose in that Glucose_nbSat cleans the learnt clause database always at the beginning of each restart, i.e., at the root of the search tree that is going to be constructed, when the number of learnt clauses becomes bigger or equal to 2000 + 300*x since the last database cleaning. In this way, clauses satisfied by variables fixed at the root are simply removed, as well as the literals falsified in the remaining clauses. Note that no clause is locked at the root of a search tree. Moreover, since the cleaning is not done inside the search tree, the search tree is constructed with the same incremental learnt clause database.

Compared with Glucose, the database cleaning is delayed in Glucose_nbSat, because it is not fired as soon as the number of the newly learnt clauses reaches a limit, but should wait for the next restart. However the delay is not important, since Glucose_nbSat performs fast restart as Glucose.

IV. Using a new measurement to predict the learnt clause usefulness

Modern CDCL-based SAT solvers usually save the last truth value of each variable when backtracking. When a free variable is picked as a decision variable, it is assigned the saved value. It is easy to see that at least one clause in which literals are all falsified by the saved assignment will become unit and change the saved value of a variable during unit propagation. More generally, a clause has more chance to become unit if the number of literals satisfied by the current saved truth value is smaller. On the contrary, those clauses with many literals satisfied by the saved truth value have little chance to become unit and should be removed.

Based on the above observation, we introduce a new measurement to predict the usefulness of a learnt clause, namely the number of literals satisfied by the saved assignment, denoted by *nbSat*, and implement the following learnt clause database cleaning strategy in Glucose_nbSat:

1) compute the number of literals satisfied by the saved assignment for each learnt clause, denoted by *nbSat*;
2) Sort all learnt clauses in the decreasing order of their nbSat value, breaking ties using the decreasing order of their LBD value. The remaining ties are broken using the clause activity value as in Glucose.
3) Remove the first half of learnt clauses (i.e. those with bigger nbSat values), by keeping binary clauses and clauses whose LBD is 2 as in Glucose.

Note that the saved assignment changes frequently during search. The measurement nbSat works only when the learnt clause database cleaning is fired frequently, because otherwise, it does not reflect the current search state after many conflicts. This is not a problem with Glucose_nbSat, because Glucose_nbSat cleans the database frequently as Glucose, making

it relevant to use the nbSat measurement in the database cleaning.

V. Other embedded techniques

When a learnt clause is in the first half after all learnt clauses are sorted in the decreasing order of their nbSat value, i.e., when it is going to be removed by the database cleaning process, we check if it subsumes an original clause or if it can be resolved with an original clause to produce a resolvent that subsumes the original clause. In the first case, the learnt clause replaces the original clause and will never be removed. In the second case, the produced resolvent replaces the original clause and will never be removed.

Example. Let $x_1 \vee x_2 \vee x_3 \vee x_4$ be an original clause, and $\bar{x}_2 \vee x_3 \vee x_4$ be a learnt clause, then the resolvent $x_1 \vee x_3 \vee x_4$ is added as an original clause that is never removed, and $x_1 \vee x_2 \vee x_3 \vee x_4$ is removed.

The above process is also applied to simplify the set of original clauses as a preprocessing in Glucose_nbSat. More concretely, the original clauses are sorted in the decreasing order of their size: c_1, c_2, ..., c_m. Each c_i ($1 \leq i \leq m$) is checked if there is a c_k ($k < i$) such that c_i subsumes c_k or if c_i and c_k can be resolved to produce a resolvent that subsumes c_k. In both cases, c_k is removed. The resolvent in the second case is inserted in the set of original clauses.

References

[1] G. Audemard and L. Simon, "Glucose: a solver that predicts learnt clauses quality," *IJCAI'09*, 2009.

[2] ——, "Glucose in the sat 2014 competition," *in Proceedings of the 2014 SAT competition*, 2014.

[3] ——, "Refining restarts strategies for sat and unsat formulae," *in Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP-12)*, 2012.

[4] C. P. Gomes, B. Selman, and K. Henry, "Boosting combinatorial search through randomization," *in Proc. AAAI-98*, Madison, WI, July 1998.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

dimetheus

Oliver Gableske
oliver@gableske.net

***Abstract*—This document describes the `dimetheus` SAT solver as submitted to the random SAT track of the SAT Competition 2016.**

I. Introduction

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

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

II. Main Techniques

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

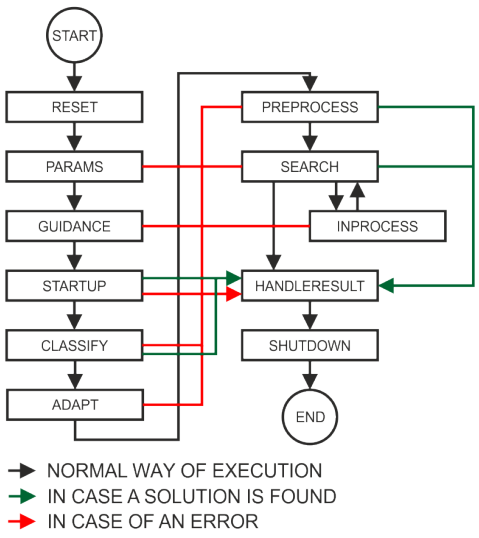


Fig. 1. A flow chart that visualizes the execution of `dimetheus`.

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

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

The classify phase will then determine what type of CNF formula the solver must solve. Since the solver is submitted to the random SAT track of the SAT Competition it will determine what type of random formula it has to solve (e.g., it will determine the size of the formula, the clause lengths, the ratio). The classifier then reports to the adapt phase for what type of formula the solver must adapt its internal parameters (e.g., a uniform random 3-CNF formula with 50000 variables and a ratio of 4.2). The adapter will then enforce a Specific parameter setting that is known to work well when solving this type of formula.

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

This is followed by the search phase in which the solver tries to find a satisfying assignment for the formula (inprocessing is turned off when the solver solves random formulas). The approach that the solver applies is best understood as bias-based decimation followed by stochastic local search. The bias-based decimation applies a Message Passing algorithm to calculate biases for individual variables. These biases indicate how likely it is to observe a variable assigned to one or zero when taking into account the models of the formula. For more information see [1]. Afterwards, a fraction of the variables with the largest bias are assigned and unit propagation (UP) is performed which then leads to a simplified remaining formula. The bias calculation and the UP-based assignment of variables with the largest bias is repeated until one of two cases occurs. First, a model is found. In this case the solver merely outputs the model and terminates. Second, UP runs into a conflict. In this case the solver will undo all assignments and initializes an SLS solver. The starting assignment for the SLS is comprised of all the assignments made until the confilct arose as well as random assignments to the remaining variables. From this point onwards the SLS takes place until either a time-out is hit or a model is found. The `dimetheus` solver, as it runs in the SAT Competition 2016, is therefore an incomplete solver that cannot detect unsatisfiability.

III. Main Parameters

The solver is started with the two following parameters.

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

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

As mentioned in the previous section, the solver will determine an optimal parameter setting based on the provided formula-type information. The parameter adapter will then internally tune a wide variety of parameters that are explained in [1]. Unfortunately, it is not possible to correctly explain the abundance of parameters here which is why the reader is addressed to the given reference for details.

IV. Implementation Details

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

V. SAT Competition 2016 Specifics

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

VI. Availability and License Information

The `dimetheus` solver is publicly available and can be downloaded from https://www.gableske.net/dimetheus. The solver is provided under the Creative-Commons Non-Commercial No-Derivs license version 4.0.

Acknowledgments

The author would like to thank Marijn Heule and Uwe Schöning for their continuous support.

References

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

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

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

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

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

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Sequential and Parallel Glucose Hacks

Thorsten Ehlers
Kiel University
Kiel, Germany

Dirk Nowotka
Kiel University
Kiel, Germany

***Abstract*—This document describes the SAT solvers we submitted to the SAT Competition 2016.**

I. Introduction

We submit a solver to the Glucose Hack Track. As this seems to performed quite well on easy benchmarks in our tests, we also submit it to the agile track. Furthermore, we submit a parallel version of Glucose to the parallel track.

II. Glucose Hack Track

For the glucose hack track, we applied two small, but useful changes.

What does LBD mean if the value is large?

The literal block distance (LBD) has become one of the most important measures for learnt clause quality [1]. However, there are different opinions about the reasons for this. We found experimentally that for values of LBD larger than 2, a trivial measure like clause size worked astonishingly well on the benchmarks of the SAT competition 2015. Therefore, we swith the clause deletion strategy, and sort clauses according to their size. We are curious to see how this performs on this year's benchmarks.

A. Delete Everything!

Glucose 3.0 never deletes clauses of LBD≤ 2. Although this appears to be a great decision in general, it may be somewhat misleading, e.g. if a clause is learnt which subsumes an LBD2-clause. Therefore, we seek to delete clauses if they have not been used for a long time. This is, we simply delete clauses if their activity drops to zero. With the standard value for cla-decay, 0.999, this is too aggressive, therefore we increased it to 0.9999.

B. Don't restart too early!

In [2], Audemard et. al suggest to block restarts if the solver seems to be close to finding a SAT-answer. This decision is based on the average trail size on which conflicts occur. On some benchmarks, this does not work well, if many unit clauses are found at decision level 0. Therefore, we adjust this measure, and consider the difference between trail size and trail size on decision level 0, i.e. "trail.size()-trail_lim[0]". This lead to a significant improvement on satisfiable formulas in our experiments.

III. Parallel Track

We submit a glucose-hack to the parallel track. In contrast to our massively-parallel solver TopoSAT [3], this is a way simpler solver. Similar to Plingeling, we organise our processes such that one of them organises the search, wheras all others peform a portfolio search. Learnt clauses are shared among all solver, if their LBD is at most 5, and the size at most 100 literals. As the "master"-process tends to be idle for small numbers of processes, we apply some simple inprocessing-techniques as unhiding and failed literal branching here. Furthermore, we implement an option to split the solvers in two groups, one which is tuned for UNSAT instances, and the other one for SAT instances. For SAT instances, we apply the above mentioned adjustment of the restart blocking technique of glucose. Furthermore, we only import clauses of LBD at most 2. In the UNSAT case, the solvers import all clauses they receive.

IV. Availability

The solver sources are submitted, and will become publicly available after the submission deadline.

References

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

[2] ——, "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.

[3] T. Ehlers, D. Nowotka, and P. Sieweck, "Communication in massively-parallel SAT solving," in *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*. IEEE Computer Society, 2014, pp. 709–716.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Glucose and Syrup in the SAT'16

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

Laurent Simon
Univ. Bordeaux
LABRI
lsimon@labri.fr

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

I. Introduction

Since 2009, Glucose enters SAT competitions/races [1], [2]... Glucose is based on minisat [3] and depends heavily on the concept of Literal Block Distance, a measure that is able to estimate the quality of learnt clauses [4]. Indeed, learnt clauses removal, restarts, small modifications of the VSIDS heuristic are based on the concept of LBD. The core engine of Glucose (and Syrup) is 7 years old.

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

II. Adaptative solver

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

A. Adaptation in Glucose

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

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

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

III. Playing with the phase

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

IV. Specificities of the parallel version

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

V. Incremental track

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

VI. Algorithm and Implementation details

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

References

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

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

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

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

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

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

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

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

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

GlucosePLE

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

***Abstract*—GlucosePLE is a CDCL solver heavily based on glucose 3.0, with a special focus on the strategy of pure literal elimination. Frequent execution of pure literal extraction would lead to an excessive computing cost. For solving this problem, we define a dynamic adjustment approach that can optimize the execution frequency of our algorithm.**

I. Introduction

Glucose [1] is an open-source CDCL-based SAT solver [2] [3] that has achieved numerous excellent performance in past SAT Competition. In the major solving procedure of glucose based solvers, variable assignment essentially happens in two different situations: the one consists of the identification of unit clauses and the creation of the associated implications which carries out *Boolean Constraint Propagation* (BCP); the other one is decision assignment which picks an unassigned literal by a decision strategy, called *Variable State Independent Decaying Sum* (VSIDS) heuristic. With defining $p()$ as the priority evaluation of variable assignment, we can obviously know that p(BCP) $>$ p(VSIDS). We consider that pure literal elimination (PLE) [4] as the third situation of variable assignment which has a higher priority than decision assignment, but lower than BCP. In general, we have p(BCP) $>$ p(PLE) $>$ p(VSIDS).

II. Main Techniques

With a standard structure of occurrence vector for each literal, which is recorded by the ID of clauses where the literal occurs, we introduce an algorithm that can be easily implemented to extract the pure literals within linear time in number of variables.

We perform pure literal extraction for variable decision. If we succeed in extracting pure literals, we manipulate them as decision variables and give each literal a value of 1. We introduce our algorithm into glucose 3.0. The extended solver carries out PLE by setting a new decision level for each of extracted pure literals, then skip the VSIDS heuristic and run the decision assignment. However, we are well aware of that pure literal extraction will keep a high execution frequency in solving process, which might reduce the efficiency of solving.

III. Dynamic Adjustment Approach

We define an approach that can effectively utilize the features of instance and current states to set an optimal frequency. Assume that in unit time ($\Delta time$) the *Decision Level* increases ($\Delta DecisionLvl$) by 100, without considering the circumstances under *backtracking* [5], we should make the extraction frequency to be less than or equal to 100. We let the *Difficulty Coefficient* be (*nVars* $*$ *nClauses*), and the *Computing Coefficient* be ($\Delta propagation$ / $\Delta time$). The dynamic adjustment approach sets an optimal frequency as follows:

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

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

Acknowledgment

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

References

[1] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers." in *IJCAI*, vol. 9, 2009, pp. 399–404.
[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*. ACM, 2001, pp. 530–535.
[3] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and applications of satisfiability testing*. Springer, 2004, pp. 502–518.
[4] J. Johannsen, "The complexity of pure literal elimination," in *SAT 2005*. Springer, 2006, pp. 89–95.
[5] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. ios press, 2009, vol. 185.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

GLUEMINISAT 2.2.10-81

Hidetomo Nabeshima
University of Yamanashi, JAPAN

Koji Iwanuma
University of Yamanashi, JAPAN

Katsumi Inoue
National Institute of Informatics, JAPAN

***Abstract*—GLUEMINISAT 2.2.10-81 is a SAT solver based on MINISAT 2.2 and the LBD evaluation criteria of learned clauses. New features of 2.2.10-81 are an inprocessing technique based on at-most-one constraints and the support of the UNSAT certification in binary DRAT format.**

I. INTRODUCTION

GLUEMINISAT is a SAT solver based on MINISAT 2.2 [1] and the LBD evaluation criteria of learned clauses [2]. One of the feature of GLUEMINISAT is the on-the-fly lazy simplification techniques based on binary resolvents [3], which are inprocessing techniques and are executed frequently during the search process of the satisfiability checking. These techniques try to identify the truth value of variables and to detect equivalent literals, and to simplify (learned) clauses by binary self-subsuming resolution. These were introduced from 2.2.7 and some of them are refined and extended in 2.2.10.

The version 2.2.10 was submitted to SAT Race 2015 [4]. New features of 2.2.10-81 are an inprocessing technique based on at-most-one constraints and the support of the UNSAT certification in binary DRAT format.

II. MAIN TECHNIQUES

A feature of 2.2.10-81 is a simplification technique based on at-most-one constraints which are automatically extracted by two ways. First is the extraction of pairwise encoding of at-most-one constraints. This extraction is executed at the end of preprocessing with the help of an efficient maximal clique enumerator called MACE [5]. Second is the semantic extraction based on binary resolvents and executed periodically during solving. For example, if we have $l_1 + l_2 + l_3 \leq 1$ and the binary implications $\forall i(\neg l_4 \rightarrow l_i)$ are detected in the process of unit propagations, then we can extend it to $l_1 + l_2 + l_3 + l_4 \leq 1$. The extracted at-most-one constraints are used for the identification of the truth value of variables. For example, if we have $l_1 + l_2 + l3 + l4 \leq 1$ and the binary resolvent $l_1 \vee l_2$ is detected, then $l_3 = l_4 =$ *false* holds.

III. SAT COMPETITION 2016 SPECIFICS

GLUEMINISAT 2.2.10-81 is submitted to Main, Agile and No-limits tracks.

- **Main track**: To reduce the generation and verification cost of UNSAT proof, every lazy simplification techniques [3] and the simplification technique based on at-most-one constraints are disabled.
- **Agile track**: The incremental variable elimination [4] and the simplification technique based on at-most-one constraints are disabled since sometimes these take time compared with other simplification techniques.
- **No-limits track**: Every techniques are enabled.

IV. AVAILABILITY

GLUEMINISAT is developed based on MINISAT 2.2. Permissions and copyrights of GLUEMINISAT are exactly the same as MINISAT. GLUEMINISAT can be downloaded at http://glueminisat.nabelab.org/. MACE is available for only academic use and refer [5] for details.

ACKNOWLEDGMENT

This research is supported in part by Grant-in-Aid for Scientific Research (No. 26330248 and 16H02803) from Japan Society for the Promotion of Science and by Kayamori Foundation of Informational Science Advancement.

REFERENCES

[1] 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.

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

[3] H. Nabeshima, K. Iwanuma, and K. Inoue, "On-the-fly lazy clause simplification based on binary resolvents," in *ICTAI*. IEEE, 2013, pp. 987–995.

[4] ——, "GLUEMINISAT 2.2.10 & 2.2.10-5," 2015, SAT Race 2015 Solver Description.

[5] T. Uno, "MACE: Maximal clique enumerator, ver 2.0," http://research.nii.ac.jp/~uno/code/mace.html.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016

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

***Abstract*—This paper serves as solver description for our new SAT solver Splatz and further documents the versions of our other solvers submitted to the SAT Competition 2016, which are Lingeling, its two parallel variants Treengeling and Plingeling, and our local search solver YalSAT.**

LINGELING

Our sequential solver Lingeling version *bbc* was submitted to all but the RandomSAT track. For the Main track we linked it with our Druplig library to print proof traces. For the other tracks the library was not included. This results in two different solvers according to the competition rules. In essence the same effect could have been achieved by changing command line options, but would result in a small overhead for those checks that skip trace generation.

This version *bbc* of the submission is in essence the same as version *bal* of Lingeling submitted to the SAT Race 2015 last year. It incorporates insights from [1], [2] and is described in the corresponding SAT Race 2015 solver description [3]. There is a small improvement in cardinality reasoning. Another new technique, which resets the reduce interval is disabled. The same applies to restart blocking and parameter selection through bucket classification, using a k-means classifier, which all do not seem to pay off, and are disabled.

As Lingeling has many preprocessing and inprocessing algorithms implemented we expect certain benchmarks to be uniquely solved by it. Because of this feature the amount of time Lingeling is allowed to spend in preprocessing and inprocessing is high. As a consequence this high effort parameter setting used in the submission may not work well for the Agile track even though it could be tuned to do so. It should be beneficial for long runs in the other tracks though.

Since proof trace generation through Druplig is still not completely implemented for all preprocessing and inprocessing techniques yet, only a subset of techniques is enabled in the Main track. For instance cardinality and XOR reasoning are disabled in the Main track.

PLINGELING, TREENGELING

The parallel solvers Plingeling and Treengeling are based on Lingeling and use exactly the same version *bbc* as the submitted sequential version. The front-ends have not changed.

Further, as before, the submitted Treengeling solver links to YalSAT version *03r*, which is run during inprocessing in a small fraction of parallel Lingeling instances. This is expected to be beneficial for crafted instances used in past competitions.

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

YALSAT

To the RandomSAT track we submitted our sequential local search solver YalSAT version *03r*. Even though we experimented with focusing on eagerly flipping break zero variables, the submitted version *03r* does not incorporate algorithmic nor heuristic changes and should behave as the previous version *03l* used in 2014 [4].

Since YalSAT solves some hard satisfiable crafted but also application instances used in past competitions, we also submitted it to the other tracks (Agile,Main,NoLimit). As YalSAT does not use any preprocessing nor any portfolio style combination with a CDCL solver, we do not expect top-class performance in the RandomSAT track, compared to other participating solvers.

The main purpose of submitting YalSAT is to see whether a local search solver can solve some interesting non-random benchmarks, for which other solvers have a hard time to solve them and on the other hand determine its base performance for the random track.

SPLATZ

First, it is pretty challenging to make changes to Lingeling and thus new ideas are hard to add and explore. Further, as in SAT solving, we argue that restarts are valuable and might trigger new ideas. Finally, it is important to consolidate already existing ideas in order to understand their effectiveness.

This lead to the development of our new SAT solver Splatz, with version *03v* submitted to the competition. This solver is developed from scratch in C++. It is a sequential stand-alone SAT solver, with static data-structures, non-reentrant and without API nor incremental usage. However, it is much better documented than say Lingeling and contains many cross-references and explanations.

More specifically, we wanted to implement a new solver, which first has a slightly less optimized, but easier to change clause storage and watching mechanism than Lingeling. This enabled us to implement an inprocessing version of blocked clause decomposition and SAT sweeping, which was left as future work in [5].

The decision heuristics uses stamping based VMTF instead of VSIDS as proposed in [1]. Restart scheduling follows [2].

As in the submitted version of Lingeling we further replaced Glucose style restart blocking by delaying restarts, which works as follows. Assume a restart is supposed to occur. This is called "forced" in Glucose terminology, and according to [2] is triggered if the fast moving glucose level average is above the slow moving average. If in this situation the current decision level is smaller than 50% of the (exponential moving) backjump level average, then restarting is delayed.

We also made the subsumption phase, usually interleaved with bounded variable elimination [6], more efficient by incorporating ideas from [7]. This allows to apply subsumption and shrinking to learned clauses as well. The major benefit is that this in turn allows to remove subsumed (and shrink) "sticky" clauses. As in Glucose [8] these sticky clauses, or low glucose level (LBD) clauses, are clauses which are never removed during learned clause cleaning (also called "reduction"), for instance clauses of glucose level 2. In MiniSAT [9] subsumed learned clauses become inactive and thus by activity based clause cleaning get removed automatically.

On the other side we realized that using a static limit on the glucose level to determine which clauses are sticky, can be replaced by a dynamic limit, by measuring the average glucose level and size of clauses resolved during conflict analysis. Clauses are considered important and become sticky if their glucose level and size is below these measured averages.

This new solver is lacking preprocessing and inprocessing techniques implemented in Lingeling, which we consider to be useful and eventually should be added. Thus the performance of Splatz is not expected to match the performance of Lingeling yet.

A (probably partial) list of implemented data-structures and algorithms is provided here:

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption
- stamping based VMTF instead of VSIDS
- subsumption for both irredundant and learned clauses
- blocked clause decomposition (BCD) enabling ...
- ... SAT sweeping for backbones and equivalences
- equivalent literal substitution (ELS)
- bounded variable elimination (BVE)
- blocked clause elimination (BCE)
- dynamic sticky clause reduction
- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

For details about these and other ideas implemented in the solver, which due to space constraints can not all be discussed nor referenced here, we recommend to consult the source code and its documentation with more references and explanations.

LICENSE

REFERENCES

[1] A. Biere and A. Fröhlich, "Evaluating CDCL variable scoring schemes," 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. 405–422.

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

[3] A. Biere, "Lingeling and friends entering the SAT Race 2015," Johannes Kepler University, Linz, Austria, FMV Report Series Technical Report 15/2, April 2015.

[4] ——, "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014," in *Proc. of SAT Competition 2014*, ser. Department of Computer Science Series of Publications B, A. Belov, M. J. H. Heule, and M. Järvisalo, Eds., vol. B-2014-2. University of Helsinki, 2014, pp. 39–40.

[5] M. Heule and A. Biere, "Blocked clause decomposition," in *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. L. McMillan, A. Middeldorp, and A. Voronkov, Eds., vol. 8312. Springer, 2013, pp. 423–438.

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

[7] R. J. Bayardo and B. Panda, "Fast algorithms for finding extremal sets," in *Proceedings of the Eleventh SIAM International Conference on Data Mining, SDM 2011, April 28-30, 2011, Mesa, Arizona, USA*. SIAM / Omnipress, 2011, pp. 25–34.

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

[9] 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.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

StocBCD: a Stochastic Local Search solver Based on Blocked Clause Decomposition

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

***Abstract*—StocBCD is a simple stochastic local search (SLS) solver. This solver sets an initial assignment of the input formula according to blocked sets obtained blocked clause decomposition (BCD). Its search is based on probability distributions similar to probSAT [2]. In addition, we speed up the search progress by block flipping heuristic.**

I. Introduction

The StocBCD solver consists mainly of three phases such as initialization, local search, block flipping. The initialization phase sets an initial assignment for the formula to be solved. The local search phase flips a variable randomly selected from unsatisfied clauses according to a probability distribution. The local search is trapped often in a deep local minima. To escape from such a local minima, we set up the block flipping phase.

II. Initialization techniques

Many stochastic local search (SLS) solvers start with a random assignment. However, StocBCD does not like this. It sets up an assignment in the order of blocked clauses. The notion about blocked clauses may be described as follows.

Given a CNF formula F, a clause C, a literal $l \in C$ is said to block C w.r.t. F if (i) C is a tautology w.r.t. l, or (ii) for each clause $C' \in F$ with $\bar{l} \in C'$, $C' \otimes_l C$ is a tautology. In this case, the clause C is called a blocked clause.

Any CNF formula can be decomposed into two blocked subsets such that both can be eliminated into a empty set by blocked clause elimination (BCE). On the other hand, any blocked set is satisfiable. Based on the two properties, we decide to use blocked clause decomposition (BCD) technique to do the initialization. At first, we decompose the formula into two blocked subsets: large subset L and small subset S, which is done by the LessInterfereBCD algorithm [1] recently proposed. Then we append S to L. In the reverse order of clause elimination obtained by BCE, we generate a number for each blocked clause. The number of the clause eliminated before is larger than the number of one eliminated after. The number of a clause in L precedes that of a clause in S. Under such assumption, let $L \wedge S = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where C_i $(i = 1, 2, \ldots, n)$ is a clause. According to this order of clauses, an initial assignment (which is assumed to be stored in array $value$) is set as follows.

Algorithm initialize_assignment
  for $i = 1$ to n **do**
    for each literal x in clause C_i **do**
      if $value[x]$ is undefined **then** $value[x] \leftarrow 1$

III. Local search techniques

Here we recall briefly the local search technique of the probSAT solver [3]. ProbSAT is a SLS solver that uses only the break values of a variable. It flips variable v according to probability $\frac{f(v,\mathbf{a})}{\sum_{x \in UC} f(x,\mathbf{a})}$, where $\mathbf{a}$ denotes the current assignment, UC is the set of unsatisfiable clauses, and $f(x, a)$ is a probability distribution function, which is defined as an exponential or a polynomial shape as shown below.

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

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

where $break(x, \mathbf{a})$ denotes the number of clauses which become false by flipping variable x under the current assignment $\mathbf{a}$. StocBCD is also based on probability $\frac{f(v,\mathbf{a})}{\sum_{x \in UC} f(x,\mathbf{a})}$ to flip variable v. However, StocBCD uses different $f(x, a)$. In details, its $f(x, a)$ is defined as

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

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

where

$$blocking(x) = \begin{cases} 1 & x \text{ is not a blocking literal} \\ 0.92 & x \in L \text{ and } x \in S \\ 1.2 & x \in L \text{ or } x \in S \end{cases}$$

Function $blocking(x)$ is actually a weight function, which is set to larger value if x is a blocking literal in only one blocked subset, and smaller value if x is a blocking literal in both blocked subsets. This may be based on the fact that the blocking literal that occurs in only one blocked subset should be a critical literal, and more important than the other literals.

IV. Parameter dynamic settings

Like probSAT, StocBCD sets ϵ to 0.9. However, parameter c_b setting is different. ProbSAT sets up statically parameter c_b, while stocBCD does dynamically it. The following table shows dynamic formulas for parameter c_b.

s	$c_b[0]$	$c_b[t]$	ϵ
3	2.30	2.3-0.085t	0.9
4	2.95	3.05-0.0625t	-
5	3.88	3.95-0.075t	-
6	4.831	4.831+0.092t	-
7	5.825	6.2-0.205t	-
≥ 8	5.4	5.8-0.15t	-

where s is the size of the longest clause in CNF formula F. Let $\#flip$ denote the total number of flipping. When $\#flip < 5 \times 10^8$, parameter c_b is set to $c_b[0]$. Otherwise, c_b is set to $c_b[t]$, where $t = \frac{\#flip}{2\times 10^8} \bmod 4$. The dynamic change period is 4.

V. Block flipping heuristic

Here we present a block flipping heuristic to escape from local minima. When the total number of unsatisfied clauses is one, we generate n bit patterns every 12 variables. In general, $n \leq 100$. Let $D(a, k, p)$ denote the difference between the number of unsatisfied clauses under assignment a and the number of unsatisfied clauses when the value of the $(k+i)$-th variable become the i-th bit value of p, where $1 \leq i \leq 12$, and the values of the remaining variables are consistent with assignment a. The algorithm for generating n closest bit patterns may be described as follows.

Algorithm build_bit_pattern
for $i = 0$ to $\frac{\#var}{12}$ **do**
  for $p = 0$ to 4096 **do**
    if $p < n$ **then** $pat[i][p] \leftarrow p$
    else if $D(a, 12i, p) < \max\{D(a, 12i, pat[i][j])\}$
      then Let $D(a, 12i, pat[i][k])$ is max
        $pat[i][k] \leftarrow p$

Using pattern pat, every 3×10^7 flips, we carry out block flipping according to the following algorithm.

Algorithm block_flip(k)
for $i = 0$ to $\frac{\#var}{12}$ **do**
  $p \leftarrow pat[i][k]$
  for $j = 0$ to 12 **do**
    if $value[v_{(12i+j)}] \neq j$-th bit of p
    then flip variable $v_{(12i+j)}$

VI. Conclusions

StocBCD adopted a few new heuristics, including BCD-based initialization, BCD-based probability distribution computation, block flipping, dynamic parameter setting etc. Based on our experimental observation, these new techniques were efficient. However, it is not clear whether they are suitable for the benchmarks of the SAT competition 2016.

References

[1] J.C. Chen: Fast Blocked Clause Decomposition with High Quality, 2015, http://arxiv.org/abs/1507.00459

[2] A. Balint, U. Schning: Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break Lecture Notes in Computer Science, 2012, Volume 7317, Theory and Applications of Satisfiability Testing - SAT 2012, pp. 16-29

[3] A. Balint, U. Schning: ProbSAT and pprobSAT, Proceedings of the SAT Competition 2014, pp. 37–38.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Improving abcdSAT by At-Least-One Recently Used Clause Management Strategy

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

***Abstract*—We improve further the 2015 version of abcdSAT by various heuristics such as at-least-one recently used strategy, learnt clause database approximation reduction etc. Based on the requirement of different tracks at the SAT Competition 2016, we develop three versions of abcdSAT: *drup*, *inc* and *lim*, which participate in the competition of main (agile), incremental library and no-limit track, respectively.**

I. Introduction

The abcdSAT solver submitted to the SAT Competition 2016 is the improved version of abcdSAT 2015 [1], which are built on the top of Glucose 2.3 [6], [7]. Here we provide three versions of abcdSAT: *drup*, *inc* and *lim*, which are submitted to main (agile) track, incremental library track and no-limit track, respectively. The main techniques use by the three versions include at-least-one recently used strategy, learnt clause database approximation reduction, recursive s-plitting solving, decision variable selection based on blocked clause decomposition [2], [3], bit-encoding phase selection [4], simplification such as lifting, probing, distillation, elimination, hyper binary resolution etc. Of course, all the simplification techniques used here are the existing techniques, so we will omit the description on them.

II. At-Least-One Recently Used Strategy

In the search process of CDCL (Conflict Driven, Clause Learning) solvers, the learnt clause database is required to be maintained. Based on our experimental observation, the clause database maintenance is actually similar to cache replacement in CPU cache management or page replacement in a computer operating system. There are many cache (page) replacement algorithms. For example, Least Recently Used (LRU), Most Recently Used (MRU), Pseudo-LRU (PLRU), Least-Frequently Used (LFU), Second Chance FIFO, Random Replacement (RR), Not Recently Used (NRU) [9] etc. Our At-Least-One Recently Used (ALORU) algorithm is similar to NRU page replacement algorithm, but different from the clause freezing mechanism proposed by Audemard et al [8]. ALORU favours keeping learnt clauses in database that have been recently used at least one time. If a learnt clause has not so far involved in any conflict analysis since it was generated, it will be discarded first. Implementing ALORU is very simple. When a conflict clause (called also learnt clause) is generated, its LBD (Literal Block Distance, for its definition, see [7]) is usually set to the number of different decision levels involved in it. However, ALORU sets the initial LBD of a conflict clause to $+\infty$, not actual current LBD. In details, in the **search** procedure, ALORU replaces "**setLBD(nblevels)**" with "**setLBD(0x3fffffff)**". Since any LBD in real instances that can be solved never exceeds 0x3fffffff, we denote $+\infty$ with 0x3fffffff. If a learnt clause involves in a conflict analysis, the procedure for conflict analysis sets its LBD to the actual value.

III. Learnt Clause Database Approximation Reduction

The target of learnt clause database reduction is two fold: remove useless clauses and avoid the expansion of database. Almost all the existing reduction algorithms in CDCL solvers are to sort learnt clauses according to the score (e.g. LBD) of clauses, then remove a given number of clauses in the sorted order. This can be viewed as exact reduction. Our approximation reduction is different from the exact reduction. It has no sorting, and replaces sorting with selection. In details, our approximation reduction finds firstly the clause with the k-th smallest (or largest) score, where k is the number of clauses to be removed. Secondly, it removes k clauses with the score less than or equal to the k-th smallest score. Notice, the clauses with the score equal to that of the k-th clause are not often unique. And the clauses with the score less than to the k-th smallest score are not necessarily removed. Therefore, the parameter k is an approximation value or estimate, not exact. Due to this reason, we call reduction implemented by finding the k-th item approximation reduction. Here we choose QUICKSELECT or Hoare's FIND algorithm [10] to find the k-th item.

If all database reductions are done in this approximation way, solving is not the most efficient. Therefore, we apply the approximation reduction when the number of conflicts is larger than 300000 for special CNF instances. In the other cases, we apply still the exact approximation.

IV. Dynamic Core and Local Learnt Clause Management

Like SWDiA5BY [11], glue_alt classifies also learnt clauses into two categories: core and local. However, the classification of SWDiA5BY is static, while our classification is dynamic. In SWDiA5BY, the maximum LBD of core learnt clauses is fixed to a constant 5. However, in abcdSAT, the maximum LBD of core learnt clauses is not fixed. AbcdSAT divides

the whole search process two stages. When the number of conflicts is less than 2×10^6, it is considered as the first stage. Otherwise, it is considered as the second stage. In the first stage, the maximum LBD of core learnt clauses is limited to 2. At this stage, core learnt clauses are kept indefinitely, unless eliminated when they are satisfied. In the second stage, the maximum LBD of core learnt clauses is limited to 5. This stage does not ensure that core learnt clauses are kept indefinitely. When local learnt clause database is reduced, we move 5000 core learnt clauses with LBD larger than or equal to 3 to local learnt clause database.

Whether the first or second stage, the number of local learnt clauses is maintained roughly between 9000 and 24000. That is, once the number of local learnt clauses reaches a upper bound, say 18000, abcdSAT will halve the number of the clauses. And the clauses with the smallest activity scores are removed first. The computation of clause activity scores here is consistent with MiniSat.

V. Recursive Splitting Solving

Any CNF formula $\mathcal{F}$ can be split into two subproblems $\mathcal{F} \cup \{x\}$ and $\mathcal{F} \cup \{\neg x\}$, where x is a variable in $\mathcal{F}$. We can obtain the solution the original problem by solving each subproblem. Solving subproblem in the same way results in a recursive solving algorithm. In general, we limit recursive depth to 10. Here is the pseudo-code of this recursive solving framework.

Algorithm SplitSolve($\mathcal{F}$, $level$)
  if $level \geq 10$ **then return** abcdSAT($\mathcal{F}$, 2×10^6)
  $\langle ret, \mathcal{F}' \rangle \leftarrow$ abcdSAT($\mathcal{F}$, 500)
  if $ret =$ SAT or UNSAT **then return** ret
  $x \leftarrow$ GetBranchVariable($\mathcal{F}'$)
  SplitSolve($\mathcal{F}' \cup \{x\}$, $level + 1$)
  SplitSolve($\mathcal{F}' \cup \{\neg x\}$, $level + 1$)

The 2nd parameter of abcdSAT in the above algorithm denotes the limit of the number of conflicts. abcdSAT($\mathcal{F}$, 500) means that it searches a solution of $\mathcal{F}$ until the number of conflicts reaches 500. Procedure GetBranchVariable selects a branch variable according to the rule given in [12].

This solving framework is suitable for small formulas.

VI. abcdSAT *drug*

Because each solver participating in the main track is required to provide a DRUP proof in UNSAT case, we add a DRUP patch in the original abcdSAT. In addition to this patch, abcdSAT *drug* adds learnt clause database approximation reduction, at-least-one recently used strategy, but excludes XOR and cardinality constraint simplification. In particular, XOR simplification is difficult to provide a DRUP proof. The splitting and merging technique used in the original abcdSAT cannot provide a DRUP proof. So abcdSAT *drug* simplifies it into recursive splitting solving technique given in previous section.

VII. abcdSAT *inc*

The solver submitted to the incremental library track is called abcdSAT *inc*. This version has no DRUP patch. The main difference between abcdSAT *inc* and abcdSAT *drug* is that abcdSAT *inc* adopts dynamic core and local learnt clause management policy, while abcdSAT *drug* adopts Glucose-style learnt clause management policy.

VIII. abcdSAT *lim*

This is the version submitted to the no-limit track. AbcdSAT *lim* not only includes various techniques given above, but also XOR and cardinality constraint simplification. With respect to learnt clause management, what abcdSAT *lim* adopts is Glucose-style learnt clause management policy. For a few special instances, abcdSAT *lim* switches to Lingeling 587f [13] to solve them. When the average LBD score of an instance to be solved is small, say less than 16, this version uses splitting and merging (reconstructing) strategy described in [5], rather than recursive splitting solving strategy mentioned above.

References

[1] J.C. Chen: MiniSAT_BCD and abcdSAT: solvers based on blocked clause decomposition, in *Proceedings of the SAT Competition 2015*

[2] J.C. Chen: Fast Blocked Clause Decomposition with High Quality, 2015, http://arxiv.org/abs/1507.00459

[3] Chen, J.C.: Improving SAT Solvers via Blocked Clause Decomposition, 2016, http://arxiv.org/abs/1604.00536

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

[5] J.C. Chen: Glue_lgl_split and GlueSplit_clasp with a Split and Merging Strategy, in *Proceedings of the SAT Competition 2014*, pp. 37–39.

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

[7] G. Audemard, L. Simon:Predicting learnt clauses quality in modern sat solvers, in *proceedings of IJCAI*, 2009, pp. 399–404.

[8] G. Audemard, J.M. Lagniez, B. Mazure, L. Saïs:On freezing and reactivating learnt clauses, in *Proceedings of SAT 2011*, ser. LNCS, vol. 6695, pp. 188–200.

[9] Amit S. Chavan, Kartik R. Nayak, Keval D. Vora, Manish D. Purohit, Pramila M. Chawan: A comparison of page replacement algorithms, IACSIT, vol.3, no.2, April 2011.

[10] C. Hoare, Algorithm 63 (PARTITION), Algorithm 64 (QUICKSORT) and Algorithm 65 (FIND). Communications of the ACM 1961,4(7), pp. 321–322.

[11] C., Oh: MiniSat HACK 999ED, MiniSat HACK 1430ED, and SWDi-A5BY, in *Proceedings of the SAT Competition 2014*, pp. 46–47.

[12] J.C. Chen: Building a hybrid sat solver via conflict-driven, look-ahead and xor reasoning techniques,in *Proceedings of SAT 2009*, ser. LNCS, vol. 5584, pp. 298–311.

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

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

MapleGlucose and MapleCMS

Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, Pascal Poupart
University of Waterloo
Waterloo, Canada

***Abstract*—This document describes the SAT solvers MapleGlucose and MapleCMS, two solvers implementing our machine learning branching heuristic called the *learning rate branching heuristic* (LRB).**

I. Introduction

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

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

II. Learning Rate Branching

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

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

If the learning rate of every variable was known, then the branching heuristic should branch on the variable with the highest learning rate. The learning rate is too difficult and too expensive to compute at each branching, so we cheaply estimate the learning rate using multi-armed bandits, a special class of reinforcement learning. Essentially, we observe the number of learnt clauses each variable participates in generating, under the condition that the variable is assigned and the solver is in conflict. These observations are averaged using an exponential moving average to estimate the current learning rate of each variable. This is implemented using the well-known *exponential recency weighted average algorithm* for multi-armed bandits [2] with learning rate as the reward.

Lastly, we extended the algorithm with two new ideas. The first extension is to encourage branching on variables that occur frequently on the reason side of the conflict analysis and adjacent to the learnt clause during conflict analysis. The second extension is to encourage locality of the branching heuristic [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 heuristic*.

III. MapleGlucose

MapleGlucose is a hack version of the Glucose 3.0 [6] solver. The solver simply runs LRB for the first 2500 seconds, then switches to VSIDS for the remaining time.

IV. MapleCMS

MapleCMS is a version of CryptoMiniSat where VSIDS is completely replaced with LRB.

V. SAT Competition 2016 Specifics

1) Both solvers are participating in the Main, Agile, and No-Limits tracks.
2) In addition, MapleGlucose is participating in the Glucose Hack Subtrack.
3) We used the same LRB parameters as presented in our paper [1].

VI. Availability

MapleGlucose and MapleCMS use the same licenses as Glucose and CryptoMiniSat respectively.

Acknowledgment

We thank Gilles Audemard and Laurent Simon, the authors of Glucose. Additionally, we thank Mate Soos, the author of CryptoMiniSat.

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] G. Audemard and L. Simon, “Glucose in the SAT 2014 Competition,” in *Proceedings of SAT Competition 2014*, 2014, p. 31.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB

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

***Abstract*—This document describes the SAT solvers MapleCOMSPS, MapleCOMSPS_LRB, and MapleCOMSPS_CHB, three solvers implementing our machine learning branching heuristics called the *learning rate branching heuristic* (LRB) and *conflict history-based branching heuristic* (CHB).**

I. Introduction

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

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

II. Learning Rate Branching

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

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

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

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

III. Conflict History-Based Branching

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

IV. Solvers

All the solvers are modifications of COMiniSatPS [7]. We used the same COMiniSatPS version that also participates in the competition [8]. However, we changed VSIDS slightly for our MapleCOMSPS solvers: during conflict analysis, if decision levels of variables are greater (resp., less) than a backtrack level, such variables get more (resp., less) bumps to activity scores. MapleCOMSPS and MapleCOMSPS_CHB also disable on-the-fly failed literal detection [9].

A. MapleCOMSPS

The difference from COMiniSatPS is that, basically, it runs LRB for the first 2500 seconds, then switches to VSIDS for the remaining time. (However, it still runs VSIDS first for the first 10000 conflicts for initialization purposes as in COMiniSatPS.) The solver employs Luby restarts and Glucose-style restarts for LRB and VSIDS, respectively.

B. MapleCOMSPS_LRB

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

C. MapleCOMSPS_CHB

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

V. SAT Competition 2016 Specifics

1) The three solvers are participating in the Main, Agile, and No-Limits tracks.
2) We used the same LRB and CHB parameters as presented in our papers [1], [2].

VI. Availability

MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB use the same licenses as COMiniSatPS (MIT license).

Acknowledgment

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

References

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

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

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

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

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

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

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

[8] ——, "COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS," in *SAT Competition*, 2016.

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

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

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

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

multi-SAT: An Adaptive SAT Solver

Sajjad Siddiqi
Jubail University College
Jubail Industrial City, Saudi Arabia
siddiqis@ucj.edu.sa

Jinbo Huang
Australian National University
Canberra, Australia
jinbo.huang@anu.edu.au

***Abstract*—Mainstream clause learning SAT solvers use decision heuristics that are based on incrementing the scores of variables involved in the learning of conflict clauses. Such a decision strategy emphasizes locality, and accounts for much of the success of modern SAT solvers. However, in some cases the resulting bias toward limited portions of the search space can be detrimental to efficient solutions. The present work has originated on the assumption that efficient solutions can be extended to a greater class of problems if the decision strategy is made more adaptive. In other words, bias in different directions may be beneficial for different types of problems. In this spirit, we propose a new decision framework for SAT that incorporates multiple decision heuristics (for both the initial ordering of variables and their dynamic reordering), periodic assessment of their effectiveness during search, and mechanism to switch on the fly between them based upon the outcomes of their assessments. Incorporating this framework we build a solver named** MULTI-SAT **based upon** GLUCOSE 2.0 **[1] and submit to the random satisfied benchmarks track of SAT Competition 2016 for evaluation.**

I. Detailed Description

It is well known that a decision heuristic may perform well on only a class of problems and no heuristic is expected to work well on all problems. The motivation behind this work is to somehow combine the strengths of different heuristics into a single solver to solve more problems. Since each heuristic is expected to perform well on a particular set of problems, maintaining a reasonably good set of heuristics may enable efficient solutions of a larger set than an ordinary SAT solver can achieve. The same idea has previously motivated researchers to develop portfolio-based SAT solvers [2], [3], [4], which maintain portfolio of several SAT solvers and select the best solver to run on an input problem using a heuristic that is driven by features of the input problem, where such a heuristic is empirically constructed. The work presented in this paper is similar in several ways to porfolio approach but is fundamentally different in terms of selection heuristic.

The new adaptive framework, builds upon the ideas presented in [5], which can maintain a set of decision heuristics each with particular schemes for initial and dynamic variable ordering. The framework allows switching amongst the decision heuristics on the fly based upon an estimate of how good the heuristic is expected to perform on a particular SAT instance at a particular time. For this purpose it periodically performs sample executions of each individual heuristic on the given instance and estimates which heuristic is likely to be more effective at a particular time in the varying search conditions. Several criteria can be used to measure the effectiveness of a particular heuristic. We use a simple criteria based upon known idea of *satisfaction power* described as the tendency of a heuristic toward satisfying the clauses in current clause database), and a measure of solver progress [6] (used in the restart strategy of GLUCOSE 2.0).

Given a set of heuristics, the adaptive mechanism works as follows. Since it is impossible to know in advance which heuristic would perform best on a particular SAT instance. Therefore sample executions of all heuristics are performed periodically and predictions about their effectiveness are made. For this purpose, at the beginning and at certain normal restarts the solver goes into an *assessment mode* in which a sample execution of every decision heuristic is performed. During assessment mode a decision heuristic is allowed to run for only c number of conflicts and then a restart is forced. At the restart the control is switched to the next heuristic. The assessment mode is finished once all heuristics have been executed, at which place the best heuristic is selected (according to criteria mentioned above) and executed for a number of conflicts determined by the heuristic's own restart policy.

Our implementation mechanism allows that the scores, phases, and statistics maintained by decision heuristics are isolated from each other. The execution of a particular heuristic does not directly change the scores, phases, and statistics maintained by other heuristics. Also, when a heuristic is executed another time during search it uses its old scores that were saved during its last execution. However, the clause database and the inference engine of the solver are shared by all heuristics. As a result, when a heuristic decides to perform clause database reduction it may delete clauses learnt by other heuristics.

We have incorporated 10 arbitrary decision heuristics. A decision heuristic for satisfiability is characterised by particular schemes for initial and dynamic variable ordering, phase saving / selection, score decaying, restarts, clause learning, and clause database reduction. Variations in these schemes lead to different decision heuristics. However, in the current implementation all heuristics, although somewhat independent from each other, work in the same way in all aspects as the default heuristic in GLUCOSE with just two modifications.

1) Every heuristic differs from others in only the initial variable ordering.
2) Every heuristic, in addition to its normal schedule for restarts, forces a solver restart after every time clause database is reduced. At this forced restart new samples

of the decision heuristics are also taken in view of the potentially significant change in the search conditions due to the clause database reduction (modern SAT solvers often employ an aggressive clause deletion policy). These samples are taken in addition to the normal sampling strategy of the solver (described above).

Initial variable orders are generated using simple topology-based methods by performing depth-first traversals of the CNF graph [7]. Such schemes tend to keep the variables that are topologically close to each other together in the variable order. By using different initial variable orders in different heuristics, we hope that even only the differences in the initial variable orders will lead each heuristic to different portions of the search space and will help break the bias generated by a single initial variable order in a traditional solver.

REFERENCES

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

[2] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, "Boosting as a metaphor for algorithm design," in *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP)*, 2003, pp. 899–903.

[3] ——, "A portfolio approach to algorithm selection," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, 2003, pp. 1542–1543.

[4] G. D. Liberto, S. Kadioglu, K. Leo, and Y. Malitsky, "DASH: dynamic approach for switching heuristics," *CoRR*, vol. abs/1307.4689, 2013.

[5] O. Shacham and K. Yorav, "Adaptive application of SAT solving techniques," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 1, pp. 35–50, 2006.

[6] G. Audemard and L. Simon, "Refining restarts strategies for SAT and UNSAT," in *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP)*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 118–126.

[7] M. Rice and S. Kulhari, "A survey of static variable ordering heuristics for efficient BDD/MDD construction," *University of California Technical Report 2008*, 2008.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Riss 6 Solver and Derivatives

Norbert Manthey, Aaron Stephan and Elias Werner
Knowledge Representation and Reasoning Group
TU Dresden, Germany

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

I. INTRODUCTION

The CDCL solver RISS is a highly configurable SAT solver based on MINISAT [1] and GLUCOSE 2.2 [2], [3]. Many search algorithm extensions have been added, and RISS is equipped with the preprocessor COPROCESSOR [4]. Finally, RISS supports automated configuration selection based on CNF formulas features, emitting DRAT proofs for many techniques, and incremental solving. The parallel solver PRISS is a portfolio solver that allows to use COPROCESSOR for formula simplification commonly for all solver incarnations, provides inprocessing for all solver incarnations, and can handle sharing of sets of equivalent literals.

This document mentions only the differences to RISS 5.05 that has been submitted to SAT Race 2015. Most differences come from bug fixes, where the bugs have been found with SPYBUG [5], and from disabling techniques that cannot print (short) DRAT proofs for their reasoning.

II. MODIFICATIONS OF THE SEARCH

RISS 5.05 used a vivification based learned clause minimization. As it turned out to be ineffective for formulas of more recent years, this minimization is disabled by default. Furthermore, the input formula is scanned for being easily solvable by assigning all variables $\top$ or assigning all variables $\bot$. Otherwise, both the activity and the polarity information to perform search decisions are pre-initialized: the activity per variable decreases linearly, and the initial polarity is set according to the Jeroslow-Wang score.

III. MODIFICATIONS OF COPROCESSOR

To be able to emit DRAT proofs many simplification techniques of Coprocessor had to be disabled, among them reasoning with XORs and cardinality constraints [6]. However, we added lazy hyper binary resolution, as used in ABCDSAT, to the portfolio of simplification techniques.

IV. CONFIGURATION SELECTION

The machine learning front end is based on the feature extraction routines implemented in RISS [7]. Compared to RISS 5.05 we included only configurations that allow to print DRAT proofs. The used configurations have been picked by hand, or have been produced by running SMAC [8] on sets of formulas that are known to be challenging for RISS.

The knowledge base for prediction is integrated into the solver. From the features and measured times that are available, we use the *information gain ratio* to select improtant features and remove redundancy. Next, with *principal component analysis* we further reduce the dimension of the feature space, where during training the information loss has to be small. Additionally more redundancy in form of correlation between the features is removed. Based on the features, we perform a k-nearest neighbor search.

The implementation of PCA is based on two external libraries: *Libpca*[1] and *armadillo*[2].

V. INCREMENTAL SAT SOLVING WITH RISS

The two configurations that have been submitted for the incremental track: RISS 5.21 uses formula simplification only during the first incremental call, while RISS 6 uses inprocessing with very long intermediate intervals, starting after the first 5000 conflicts.

VI. SAT COMPETITION SPECIFICS

RISS and COPROCESSOR are implemented in C++. The DS version of RISS implements watch lists based on an own allocator – as also done in LINGELING – so that garbage collection on watch lists is possible. PRISS implements the parallel code for multi-core architectures with the pthreads library.

The solvers are submitted to all tracks that are offered, except the *random SAT* track.

VII. AVAILABILITY

All tools in the solver collection are available for research. The source of the solver will be made publicly available under the LGPL v2 license after the competition at http://tools.computational-logic.org.

[1] http://sourceforge.net/projects/libpca/
[2] http://arma.sourceforge.net/

ACKNOWLEDGMENT

The author would like to thank the developers of GLUCOSE 2.2 and MINISAT 2.2. The computational resources to develop, evaluate and configure the SAT solver have been provided by the ZIH of TU Dresden. This project is supported by the DFG grant HO 1294/11-1.

REFERENCES

[1] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Heidelberg: Springer, 2004, pp. 502–518.

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

[3] ——, "Refining restarts strategies for sat and unsat," in *CP'12*, 2012, pp. 118–126.

[4] N. Manthey, "Coprocessor 2.0 – a flexible CNF simplifier," in *SAT 2012*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Heidelberg: Springer, 2012, pp. 436–441.

[5] N. Manthey and M. Lindauer, "Spybug: Automated bug detection in the configuration space of SAT solvers," in *SAT*, 2014, accepted.

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

[7] E. Alfonso and N. Manthey, "New CNF features and formula classification," in *POS-14*, ser. EPiC Series, D. L. Berre, Ed., vol. 27. EasyChair, 2014, pp. 57–71.

[8] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *LION*, ser. LNCS, 2011, vol. 6683, pp. 507–523.

BENCHMARK DESCRIPTIONS

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Generating the Uniform Random Benchmarks

Marijn J. H. Heule

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

***Abstract*—The uniform random k-SAT instances described here, together with the hard satisfiable random instances described on pages 60–62 of this compilation, constitute the benchmark set of the Random Track of SAT Competition 2016.**

INTRO

This description explains how the benchmarks were created of the uniform random categories of the SAT Competition 2016. These categories consists of uniform random k-SAT instances with $k \in 3, 4, 5, 6, 7$ – Boolean formulas for which all clauses have length k. For each k the same number of benchmarks have been generated.

GENERATING THE SATISFIABLE BENCHMARKS

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

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

TABLE I
PARAMETERS OF GENERATING THE SATISFIABLE BENCHMARKS

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

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

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Using Algorithm Configuration Tools to Generate Hard Random Satisfiable Benchmarks

Tomáš Balyo
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
tomas.balyo@kit.edu

***Abstract*—The benchmarks described here, together with the uniform random k-SAT instances described on page 59 of this compilation, constitute the benchmark set of the Random Track of SAT Competition 2016.**

Tools for optimizing algorithm parameters have been successfully used to speed up local search satisfiability (SAT) solvers and other search algorithms by orders of magnitude. In this paper we show that such tools are also very useful for generating hard SAT formulas with a planted solution. Our experiments with state-of-the-art local search SAT solvers show, that using this approach we can randomly generate satisfiable formulas significantly harder than uniform random formulas of the same size from the phase-transition region or formulas generated by previous approaches. Additionally we show how to generate small satisfiable formulas that are hard to solve by CDCL solvers. The generation of difficult compact formulas with a planted solution is useful for benchmarking SAT solving algorithms and also has cryptographic applications.

I. Introduction

One of the most popular class of randomly generated satisfiability (SAT) benchmarks is uniform random 3SAT formulas with a variable-clause ratio corresponding to the phase-transition threshold [1]. This ratio causes that half of the generated formulas is satisfiable. These formulas are considered very hard to solve relative to their size and are often used to evaluate the performance of SAT solvers, in particular local search SAT solvers [2] (the satisfiable instances).

However, there are methods that can generate even harder instances having the same or smaller size. These methods have additional advantages such as always generating a satisfiable instance and the possibility to hide a predefined solution in the formula. The main motivation for hard satisfiable benchmarks is to evaluate, compare and improve SAT solvers (local search solvers in particular) but generators of hard satisfiable formulas with a predefined solution can also be used in cryptography as one-way functions [3] (for example a password can be coded as the solution of randomly generated formula, which is easy to verify but hard to find).

In this paper we describe a new method that allows us to generate hard satisfiable formulas with a predefined solution. We will focus on 3SAT formulas, however the method can be easily generalized to generate formulas with arbitrary clause lengths. We provide experimental results to demonstrate the hardness of the generated formulas for both CDCL and local-search SAT solvers. We can generate instances especially difficult for stochastic local search algorithms [2]. In particular, we obtained satisfiable 3SAT formulas with as few as 60 variables that cannot be solved by state-of-the-art local search SAT solvers such as ProbSAT [4] and Dimetheus [5] in 10 minutes.

```
    cdc-generate (vars, φ)
CDC0  F := ∅
CDC1  while |F| < r * vars do
CDC2    C = generateRandom3Clause(vars)
CDC3    i = numberOfSatLiterals(C,φ)
CDC4    if i > 0 then
CDC5      with probability p_i do F = F ∪ {C}
CDC6  return F
```

Fig. 1. Pseudo-code of a CDC algorithm which has 2 inputs (the number of variables and a truth assignment) and parameters p_i and r. It produces a random 3SAT formula with the given number of variables that is satisfied by the given assignment.

II. Related Work

A commonly used method (at SAT competitions and in local-search solver papers) for generating hard random satisfiable formulas is to generate uniform random formulas from the SAT phase-transition [1] region and filter out the unsatisfiable instances [6][1] This method generates small and hard instances, but it has several disadvantages. We cannot generate a formula with a predefined solution and we need to be able to solve the formulas in order to filter out the unsatisfiable instances.

A simple approach to generate a formula with a given solution ϕ is to generate random clauses while filtering out those that are not satisfied by ϕ until we reach the desired amount of clauses [7]. This approach, called the 1-hidden algorithm, has the disadvantage that the generated formulas are easy to solve, especially by local search solvers [7]. The hardness can be increased by hiding 2 or more solutions [7].

Another typical approach is to use a clause distribution control (CDC) algorithm [8]. A CDC algorithm for generating k-SAT formulas has $k+1$ parameters $0 < p_1, \ldots, p_k < 1$ and $r \in \mathbb{R}$. It is outlined in Figure 1. The parameter r represents the clause/variable ratio and each p_i is the probability that

[1]The filtering is usually done by running a local search SAT solver with a large time limit and removing the instances it cannot solve. This has the obvious problem that only instances that local search can solve are selected as benchmarks (which are then mostly used to evaluate other local search solvers).

a clause which has exactly i satisfied literals under a given assignment ϕ gets into the formula. Implementations of the CDC algorithm define the values of p_i and r differently. For example the q-hidden algorithm defines $p_i = q^i$ for a parameter q [9] and the generator of Barthel et al. [8] uses the diluted spin-glass model to theoretically compute good values of p_i.

A more detailed overview of algorithms for generating random formulas with hidden solutions can be found in a recent paper by Liu et al.[10].

III. Our Approach

Similar to previous successful generators [8], [9], [10] our method is also based on the clause distribution control (CDC) approach. The difference is that to obtain the values of the parameters p_i and r we use an automatic software parameter optimization tool instead of analyzing theoretical models.

Parameter optimization tools have been used successfully to improve performance of SAT solvers, especially local search solvers [11]. In a sense, in this paper, we use these tools for the opposite purpose – to slow down SAT solvers (by generating hard benchmarks). For our experiments we use the 2.10.03 version of the parameter optimization tool SMAC [12].

IV. Obtaining The CDC Parameters

Using SMAC we optimized a shell script that evaluates a given configuration for a particular solver. The evaluation algorithm is outlined in Figure 2 and its goal, in essence, is to compute how many formulas can be solved until they become too hard to solve. The evaluation algorithm relies on the fact that formulas get harder with a larger size (more variables and clauses) for any given configuration.

For each formula size (number of variables) starting from 20 and increasing by 5 until 600 eight formulas are generated using the CDC algorithm (Figure 1)[2]. These 8 instances are then solved in parallel[3] by the SAT solver (with a time limit of 1 minute). If at least half (4) of the instances is solved then we continue to the next size otherwise we return the total number of solved formulas so far as the score of this configuration. Lower score means the configuration gives harder instances which means that the configuration is better for our purposes. The SMAC tool is then used to find the values of p_i and r with a minimal score.

V. SAT Competition 2016 Benchmarks

Using four representative state-of-the-art SAT solvers – two local search solvers: ProbSAT [4] (version SC13.2) and Dimetheus [5] (version 2.100.994) and two CDCL solvers: Lingeling [13] (version bal) and Glucose [14] (version 4.0) – we obtained the following configuration of the CDC parameters: $c_1 = 0.414$, $c_2 = 0.028$, $c_3 = 0.503$ and $c_r = 4.408571$. We generated 60 instances, 10 for each of the following number of variables: 350, 400, ..., 600

[2]The planted solution is generated randomly.

[3]Each core of the computer is running one solver-benchmark pair.

evaluate-configuration (c_1, c_2, c_3, c_r)
SC0 $score := 0$
SC1 **for** $vars \in \{20, 25, \ldots, 600\}$ **do**
SC2 $solved := 0$
SC3 **repeat** 8 times:
SC4 $F =$ cdc-generate$(vars, c_1, c_2, c_3, c_r)$
SC5 **if** F is solved in 1 minute **then** $solved$++
SC6 $score := score + solved$
SC7 **if** $solved < 4$ **then break**
SC8 **return** $score$

Fig. 2. Pseudo-code of a configuration evaluation algorithm which has 4 inputs – p_1, p_2, p_3 and r. Its goal is to estimate the size of the largest formula that can be solved under one minute for a given configuration using a particular SAT solver.

VI. Conclusion

A random SAT formula generator algorithm can be automatically configured to generate very hard instances in the same way as the SAT solvers are tuned. Our method can generate formulas with a planted solution that are small in size and hard to solve by several state-of-the-art SAT solvers. A generator of such formulas is useful not only for benchmarking SAT solving algorithms but also in cryptographic applications.

Acknowledgment

This research was partially supported by DFG project SA 933/11-1

References

[1] I. P. Gent and T. Walsh, "The sat phase transition," in *ECAI*, vol. 94. PITMAN, 1994, pp. 105–109.
[2] B. Selman, H. J. Levesque, D. G. Mitchell *et al.*, "A new method for solving hard satisfiability problems." in *AAAI*, vol. 92, 1992, pp. 440–446.
[3] C. H. Papadimitriou, *Computational complexity*. John Wiley and Sons Ltd., 2003.
[4] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *SAT*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 16–29.
[5] O. Gableske, "Solver description of dimetheus v. 1.700 for the sat competition 2013," *Proceedings of SAT Competition 2013*, p. 30, 2013.
[6] O. Kullmann, "The sat 2005 solver competition on random instances," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 61–102, 2006.
[7] D. Achlioptas, H. Jia, and C. Moore, "Hiding satisfying assignments: two are better than one," *Journal of Artificial Intelligence Research*, pp. 623–639, 2005.
[8] W. Barthel, A. K. Hartmann, M. Leone, F. Ricci-Tersenghi, M. Weigt, and R. Zecchina, "Hiding solutions in random satisfiability problems: A statistical mechanics approach," *Physical review letters*, vol. 88, no. 18, p. 188701, 2002.
[9] H. Jia, C. Moore, and D. Strain, "Generating hard satisfiable formulas by hiding solutions deceptively." in *Proceedings of the National Conference on Artificial Intelligence*, vol. 20, no. 1. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005, p. 384.
[10] R. Liu, W. Luo, and L. Yue, "Hiding multiple solutions in a hard 3-sat formula," *Data & Knowledge Engineering*, vol. 100, pp. 1–18, 2015.
[11] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stuetzle, "ParamILS: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, October 2009.
[12] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization*. Springer, 2011, pp. 507–523.

[13] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013." in *In Proceedings of SAT Competition 2013, A. Balint, A. Belov, M. J. H. Heule, M. Järvisalo (editors), vol. B-2013-1 of Department of Computer Science Series of Publications B pages 51-52, University of Helsinki, 2013.*, 2013.
[14] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers." in *IJCAI*, vol. 9, 2009, pp. 399–404.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Avoiding Monochromatic Solutions of $a + b = c$ and $a^2 + b^2 = c^2$

Marijn J. H. Heule

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

INTRO

No satisfiable crafted benchmarks were submitted to SAT Competition 2016. The organizers therefore generated some interesting hard satisfiable (and some similar unsatisfiable) benchmarks based on problems in Ramsey Theory.

GENERATING SCHUR BENCHMARKS

The well-known Schur theorem [1] states that it is impossible to color the natural numbers $\{1, 2, \dots\}$ with a finite number of colors such that there exist no monochromatic solutions of the equation $a + b = c$. The largest monochromatic-free colorings for $k < 5$ colors are known. The best known lower bound of monochromatic-free colorings using five colors is for $\{1, \dots, 160\}$. Yet finding a monochromatic-free coloring of $\{1, \dots, 160\}$ is hard for SAT solvers. The problem gets easier for most solvers if the problem is restricted as follows: the first n numbers cannot be colored with the fifth color. Up to $n = 43$, all these benchmarks are satisfiable. The competition suite has 24 satisfiable benchmarks of the problem that differ only in the restriction ($20 \leq n \leq 43$). Also included in the suite are 24 problems of coloring $\{1, \dots, 161\}$ with five colors and the same restrictions. All these benchmarks are expected to be unsatisfiable. For all 48 problems we added symmetry-breaking predicates to make them easier.

GENERATING PYTHAGOREAN TRIPLES BENCHMARKS

The Pythagorean Triples problem deals with avoiding monochromatic solutions of the equation $a^2 + b^2 = c^2$. It has recently been shown [2] that the numbers $\{1, \dots, 7824\}$ can be colored with red and blue such that for all Pythagorean triples (a, b, c) with $a^2 + b^2 = c^2$ and $c \leq 7824$ holds that a, b, or c is colored red and a, b, or c is colored blue. This is impossible for the numbers $\{1, \dots, 7825\}$. Let F_{7824} denote the formula encoding the existence of a valid 2-coloring of $\{1, \dots, 7824\}$, i.e, no monochromatic Pythagorean triple. This formula uses variables x_i. If x_i is assigned to true/false, then i is colored red/blue, respectively. Solving F_{7824} is hard, in particular for CDCL solvers.

The backbone of a formula is the set of literals that is assigned to true in all solutions. First, we break the symmetry in F_{7824} by adding the unit clause (x_{2520}) resulting in the formula F^*_{7824}. The backbone of F^*_{7824} consists of 2304 backbone literals. Adding backbone literals to F^*_{7824} makes it easier to solve. After the addition of about 20 backbone literals, the problem F^*_{7824} becomes reasonably easy.

The competition suite consists of 21 benchmarks of F^*_{7824} with a different number of backbone literals added. For each $i \in \{0, \dots, 20\}$ there is a benchmark of F^*_{7824} with i backbone literals. By construction, all these benchmarks are satisfiable.

REFERENCES

[1] I. Schur, "Über die Kongruenz $x^m + y^m = z^m \pmod p$," *Jahresbericht der Deutschen Mathematikervereinigung*, vol. 25, pp. 114–117, 1917.

[2] M. J. H. Heule, O. Kullmann, and V. W. Marek, *Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer*. Cham: Springer International Publishing, 2016, pp. 228–245. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-40970-2_15

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

CNF From Tools Driven By SAT Solvers

Norbert Manthey
Knowledge Representation and Reasoning Group
TU Dresden, Germany

***Abstract*—Many tools utilize SAT solvers to solve higher level problems. The set of submitted formulas has been created by using a hand picked set of tools.**

I. Hardware Model Checking with ShiftBMC

The formulas encode hardware model checking problems. The original circuit description in the AIGER format is given to the ShiftBMC (http://tools.computational-logic.org/) tool which applies circuit simplifications with the ABC toolkit. Next, the transition formula in CNF is furthermore simplified with Coprocessor. The encoded formula represents the model checking problem for a single step.

The submitted formulas are considered being difficult. In the hardware model checking competition in 2014 none of the submitted tools for the deep bound track have been able to return a result for this problems.

II. Software Model Checking with CBMC

The submitted formulas have been encoded by the software verification tool CBMC (www.cprover.org/cbmc). All model checking tasks have been taken from the concurrency track of the Competition of Software Verification (SV-COMP) 2015. The corresponding C benchmarks can be found at https://github.com/sosy-lab/sv-benchmarks/tree/master/c/.

The CNF files have been generated by calling CBMC with

```
./cbmc <benchmark> --unwind 2 --dimacs
```

The used version of CBMC is version 5.3. Glucose 3.0 was used to determine the satisfiability for the formulas, where the resource limits 6.5GB memory and a 1h timeout have been applied. From the set of generated CNF files we randomly selected 45 satisfiable formulas, 45 unsatisfiable benchmarks, and we added at most 10 formulas that could not be solved in the time limit. The formulas should be labeled as software verification.

III. SAT Modulo Theory

The submitted formulas have been encoded from problems specified in SMT. All benchmarks have been taken from the quantifier free bit vector track of the SMT library. The corresponding SMT benchmarks can be found at "http://www.cs.nyu.edu/ barrett/smtlib/QF_BV_rest.zip". Two SMT solvers have been used to generate CNF files: CVC4 (https://github.com/lianah/cvc4) and STP (https://github.com/stp/stp).

The CVC4 CNF files have been generated by calling CVC4 with

```
./cvc4 --bitblast=eager
     --bvminisat-dump-dimacs <benchmark>
```

The used version of CVC4 is "cvc4 1.5-prerelease (git branch bvminisat-dump-cnf)".

The STP CNF files have been generated by calling STP with

```
./stp-2.1.2 --output-CNF
         --exit-after-CNF $f <benchmark>
```

While STP usually created multiple CNF files, for the given benchmark and call only a single CNF was generated per SMT benchmark. The used version of STP is "stp-2.1.2".

Glucose 3.0 was used to determine the satisfiability for the formulas, where the resource limits 6.5GB memory and a 1h timeout have been applied. For each set of submitted formulas we randomly selected 45 satisfiable formulas, 45 unsatisfiable benchmarks, and we added at most 10 formulas that could not be solved in the time limit. The submission of the formulas contains hardware verification problems as well as software verification problems.

Acknowledgment

The computational resources to preselect the formulas have been provided by the ZIH of TU Dresden. This project is supported by the DFG grant HO 1294/11-1.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016

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

***Abstract*—In this short note we present a collection of benchmarks submitted to the SAT Competition 2016. Most of them stem from other sources, some crafted ones are new, but all present equivalence checking problems (miters) for arithmetic circuits, such as multipliers.**

INTRODUCTION

Two invited talks by Anna Slobodova and Aaron Tomb, as well as a tutorial by Priyank Kalla in Austin as part of SAT'16 and FMCAD'15 argued, that checking arithmetic miters is still a challenge, both in hardware and software verification, even after more than 20 years after the Pentium FDIV bug. As a consequence even today, verifying arithmetic circuits requires cumbersome manual case splitting or simply gives up on obtaining a formal proof and uses simulation instead.

The reason is that these circuits do not have internal equivalence points, i.e., in essence only the outputs are pair-wise equivalent. It is further conjectured that resolution is not strong enough to obtain polynomial proofs even for such simple tasks as checking commutativity of bit-vector multiplication after bit-blasting and CNF encoding.

In order to help trying to attack this challenge we collected existing arithmetic miters and also generated some new crafted benchmarks. All the submitted benchmarks are published at http://fmv.jku.at/datapath. The README files available there give more information on how exactly the benchmarks were derived. The benchmark archive also contains structural versions for some of the benchmarks in various formats beside CNF in DIMACS format.

CRAFTED MITERS

The CRAFTED benchmark set contains the old subset LINVRINV, which was suggested by Stephen Cook during his invited talk at SAT'04, for which we previously already submitted a C generator to the competition. Pre-generated CNFs up to square matrix size 7 are included, which are still considered to be really challenging. The structure of the propositional arithmetic in this benchmark subset has some flavor or multiplier miters, but might need even more powerful reasoning. The remaining benchmark subsets in the CRAFTED set, check simple properties of bit-vector multiplication for various bit-widths, more precisely, commutativity $x \cdot y = y \cdot x$, associativity $x \cdot (y \cdot z) = (x \cdot y) \cdot z$, and distributivity $x \cdot (y + z) = x \cdot y + x \cdot z$, as well as the property $x \cdot (x + 1) = x \cdot x + x$.

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

We consider these problems as crafted, since bit-vector rewriting can prove them trivially. However, disabling rewriting and bit-blasting them to AIGs with Boolector [1], then encoding them into CNF, produces pretty challenging benchmarks too. We included SMT, AIG and of course CNF versions of these benchmarks up to the bit-width, for which we do not know of any known technique which can solve the CNF versions in a reasonable amount of time (16 bits for commutativity and associativity, 12 bits for distributivity and 24 bits for the last property).

Note however, that these benchmarks, as well as probably most of the benchmarks in this submission, have nice linear parallel speed-ups using cube-and-conquer solving [2]. So we expect Treengeling [3] to be able to go a few bits further than other solvers, particularly sequential ones, depending on the number of processor cores.

EPFL MITERS

The benchmark set EPFL was generated by Mathias Soeken using ABC [4]. These 10 miters check correctness of the smallest optimized variant of circuits in the "The EPFL Combinational Benchmark Suite" [5]. Only arithmetic circuits were used for generating miters in this submission. A few benchmarks are considered trivial, most of them challenging. This original set of optimized circuits is still evolving and might be good a source for more miter benchmarks.

MULTIPLIER MITERS BY MATTI JÄRVISALO

The benchmark set JARVISALO was submitted to the SAT Competition 2007 before by Matti Järvisalo [6] and has been used in the competition for quite some time (file name prefix "eqatreebraun"). It consists of miters for checking equivalence of one particular optimized multiplier architecture against a reference multiplier. We only include these because they model the same problem as other benchmarks in this submission.

I. FMCAD 2015 EXAMPLE BY PRIYANK KALLA

Syntactically different polynomials modulo 2^n might still represent the same function. One of them might have less coefficient bits. This in turn might yield a more compact circuit implementation. Checking equivalence of the original circuit versus the optimized implementation again produces a miter. The single benchmark we have in this benchmark set KALLAFMCAD15 is from an example given by Priyank Kalla in his Tutorial at FMCAD'16 [7] on implementing

$F = 1/2\sqrt{a^2 + b^2}$ by the polynomial of its Taylor expansion on $x = a^2 + b^2$, where x is a bit-vector of size 16.

This setting might yield more interesting benchmarks and the same applies to similar problems in the context of verifying arithmetic circuits for signal processing, such as considering Galois field multipliers.

II. Multiplier Miters from Arist Kojevnikov

There exists a generator suite to produce an actually quite large set of multiplier miters, which was published already in 2005 by Arist Kojevnikov. This was used for developing and benchmarking a boolean algebraic solver [8]. These generator scripts produce ISCAS miters, which we translated to AIGs and then to CNF. We generated benchmarks for bit-widths 4,8,9-16,32,64,128, and generated 144 miters per bit-width. We also included buggy multipliers, which yield satisfiable miters all having "bg" in their file name. Some of the miters compare structurally very similar (or even identical) circuits. Those are then much simpler. The other correct miters with high bit-widths are a real challenge.

III. Miters from Kaiserslautern

The first benchmark set WEDLER from the group of Wolfgang Kunz in Kaiserslautern, is based on miters in SMT format as used in an ASPDAC'08 paper on bit-level arithmetic circuit verification [9]. We obtained the actual SMT files from Markus Wedler. These only include miters for their own generated multipliers and not the industrial IBM multipliers, which were used in addition in that paper. These generated benchmarks have further been used already in many papers on arithmetic circuit verification. The 108 SMT files have different combinations of operand size and output sizes, and also differ w.r.t. signedness and whether Booth encoding was used. The SMT files were bit-blasted again with Boolector [1] to obtain CNF files.

The other benchmark set WIELAND from Kaiserslautern is related to their CAV'08 paper [10], which uses algebraic word-level techniques. These benchmarks were submitted to the SMT-LIB [11], and we simply bit-blasted them with Boolector [1]. This set consists of three generic miters over the bit-widths 4,8,16,32,48,64, thus 18 benchmarks altogether.

IV. Conclusion

We consider the effort of collecting a meaningful set of arithmetic problems encoded in CNF as not finished yet. For instance, we tried to obtain some more multiplier miters used in a recent DATE'16 paper [12], but the original multiplier designs are not publicly available. Furthermore, some of the sources of benchmarks used above might yield more benchmarks. Then there are these challenges mentioned in his invited talk by Aaron Tomb last year, and already discussed above, in the context of verifying correctness of the implementation of cryptographic functions. Some of them are already available as SMT-LIB [11] benchmarks. Finally, benchmarks in the context of verifying floating point operations might be interesting too.

References

[1] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *JSAT*, vol. 9, pp. 53–58, 2015.

[2] M. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, K. Eder, J. Lourenço, and O. Shehory, Eds., vol. 7261. Springer, 2011, pp. 50–65.

[3] A. Biere, "Lingeling and friends entering the SAT Race 2015," Johannes Kepler University, Linz, Austria, FMV Report Series Technical Report 15/2, April 2015.

[4] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 24–40.

[5] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, no. EPFL-CONF-207551, 2015.

[6] M. Järvisalo, "Equivalence checking hardware multiplier designs," 2007, sAT Competition 2007 benchmark description. Available at http://www.satcompetition.org/2007/contestants.html.

[7] P. Kalla, "Formal verification of arithmetic datapaths using algebraic geometry and symbolic computation," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, p. 2.

[8] E. Hirsch, D. Itsykson, A. Kojevnikov, A. Kulikov, and S. Nikolenko, "Report on the mixed boolean-algebraic solver," Citeseer, Tech. Rep., 2005.

[9] U. Krautz, M. Wedler, W. Kunz, K. Weber, C. Jacobi, and M. Pflanz, "Verifying full-custom multipliers by boolean equivalence checking and an arithmetic bit level proof," in *Proceedings of the 13th Asia South Pacific Design Automation Conference, ASP-DAC 2008, Seoul, Korea, January 21-24, 2008*, C. Kyung, K. Choi, and S. Ha, Eds. IEEE, 2008, pp. 398–403.

[10] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 473–486.

[11] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.5," Department of Computer Science, The University of Iowa, Tech. Rep., 2015, available at `www.SMT-LIB.org`.

[12] A. A. R. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining gröbner basis with logic reduction," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, L. Fanucci and J. Teich, Eds. IEEE, 2016, pp. 1048–1053.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Documentation of some combinatorial benchmarks

Jan Elffers and Jakob Nordström

Let us start by making a general comment. The parameter ranges for the combinatorial benchmarks were chosen to insure reasonable CNF sizes and to allow most of them to be solved under some settings of the CDCL algorithm. In particular, we want to emphasize that all of these instances have short resolution proofs that can in principle be found by CDCL without any preprocessing, and often even without any restarts given an appropriate (fixed) variable order. More specifically, all formulas except the relativized pigeonhole principle formulas have resolution proofs in size linear in the number of clauses (with reasonably small multiplicative constants), and this last formula family has proofs of size scaling like n^k, which is still a reasonably small polynomial for small, constant k.

Thus, one way of viewing our contributed bencharmks is as a challenge to CDCL to be competitive with resolution: Can CDCL solvers produce proofs of unsatisfiability that are somewhat close to the short proofs that do exist, as the theoretical results in [1], [2] suggest, or are there formulas that are very easy in theory but very hard in practice?

In what follows, we give a brief description of each family and provide pointers to references that discuss them in more detail. Many, though not all, of these instances were generated using the tool *CNFgen* [3], [4].

I. Tseitin Formulas (tseitingrid⟨r⟩)

Tseitin formulas [5] encode systems of linear equation over $\mathrm{GF}(2)$ (i.e., XOR constraints) generated from connected graphs $G = (V, E)$ with *charge function* $\chi : V \to \{0, 1\}$. Every edge $e \in E$ corresponds to a variable x_e, and for every vertex $v \in V$ there is an equation $\sum_{e \ni v} x_e \equiv \chi(v) \pmod 2$ encoded in CNF, yielding an unsatisfiable formula if and only if $\sum_{v \in V} \chi(v) \not\equiv 0 \pmod 2$. When G has bounded degree and is well-connected, the formula is exponentially hard for resolution [6].

We study Tseitin formulas on long, narrow grid graphs, where every vertex has edges horizontally and vertically to its 4 neighbours, and where edges "wrap around" so that the topmost and bottommost rows are connected, as are the rightmost and leftmost columns. For the right settings of parameters, Tseitin formulas over such graphs have been proven to exhibit strong size-space trade-offs for resolution [7], [8]. These parameter settings are not appropriate for practical experiments, however. Instead, we fix the number of rows to a small constant $r \in \{4, 5, 6, 7\}$ and vary the number of columns n to scale the size of the instances. Formula sizes and number of variables in this family scale linearly with n, as does the size of minimal resolution proofs. Even tree-like resolution, corresponding to DPLL without clause learning, can refute these formulas efficiently, although at the cost of a small polynomial blow-up depending on r.

II. Ordering Principle Formulas (pop)

Ordering principle formulas claim that there is a finite set $\{e_1, \ldots, e_n\}$ with an ordering $\preceq$ such that no element e_j is minimal with respect to this ordering, where variables $x_{i,j}$, $i \neq j \in [n]$, encode $e_i \preceq e_j$. One can generate two variants of formulas in this family encoding that the ordering is partial (pop) and total (lop), respectively, where the latter formula is a strict superset of the former. For the SAT competition, only pop formulas were used.

These formulas were conjectured to be hard in [9] but were later shown to have resolution proofs of size linear in the formula size [10]. Ordering formulas have clauses of size $n-1$, but if they are converted to 3-CNF in some appropriate way, they can be shown to require large width [11]. Combining this with the size-width lower bounds in [12], it follows that ordering principle formulas are exponentially hard for tree-like resolution and DPLL.

III. Pebbling Formulas (peb-pyrofpyr-neq-3)

Pebbling formulas [12] are generated from directed acyclic graphs (DAGs) $G = (V, E)$, with vertices $v \in V$ identified with variables x_v, and contain clauses saying that (a) sources s are true (a unit clause x_s) and (b) truth propagates through the DAG (clauses $\bigvee_{i=1}^{\ell} \overline{x}_{u_i} \vee x_v$ for each non-source v with predecessors $u_1, \ldots, u_\ell$) but (c) sinks z are false (a unit clause $\overline{x}_z$). As just described, pebbling formulas are trivially refuted by unit propagation, but they become more interesting if we replace every variable x by some suitably chosen Boolean function $f(x_1, \ldots, x_d)$ for new variables $x_1, \ldots, x_d$, where f should have the property that no single variable can fix the value of f. Strong space lower bounds and size-space trade-offs for these formulas were shown in [13], [14].

Although from a theoretical point of view any function f that satisfies the properties above yields formulas with similar properties, in practice there can be significant differences. A fairly extensive experimental evaluation of pebbling formulas with different substitution functions was performed in [15]. Guided by that work, we use not-all-equal (NEQ) over 3 variables as the substitution function in our experiments. As the base graph, we used "pyramids of pyramids," which are defined as follows. A *pyramid graph* of height n consists of $n + 1$ layers $i = 0, 1, \ldots, n$ with $i + 1$ vertices in layer i, and with edges from vertices j and $j + 1$ in layer i to vertex j in layer $i - 1$. In a *pyramid of pyramids*, each vertex in the pyramid of height n is itself blown up to a pyramid of height n.

All pebbling formulas, even after substitution with Boolean functions of constant arity, have resolution proofs of size linear in the formula size and of constant width, but are exponentially hard for tree-like resolution [16] (for the right kind of graphs and substitution functions, such as the ones we use here).

IV. Stone Formulas (stone-width3chain-nmarkers)

Stone formulas are also generated from DAGs and are similar in flavour to pebbling formulas, but here we think of every vertex of the graph as containing a stone or marker, where every marker has colour red or blue and (a) stones on sources are blue and (b) a non-source with all predecessors blue also has a blue stone, but (c) sinks have red stones. This family of formulas have been used to separate general resolution from so-called *regular* resolution [17], which is a subsystem of resolution that is strong enough to capture the DP algorithm using variable elimination [18]. Stone formulas have also been investigated as candidates for showing that CDCL without restarts cannot simulate the full power of resolution, but the results so far have been inconclusive. It was shown in [19] that a model of CDCL without restarts can decide these formulas in principle, but this model of CDCL seems too general to yield any compelling practical conclusions.

The parameters for which the theoretical results in [17] hold yield far too large formulas to be manageable in practice (a graph over n vertices requires $m = 3n$ stones, and so we instead generate scaled down versions for which there are no theoretical guarantees. We use the chain/road graphs (as described in [15], [20]) of length n with n stones. These formulas have resolution proofs in size linear in the formula size. As mentioned above, for the right kind of parameters they are hard for regular resolution, and hence also for tree-like resolution, but the concrete parameter values that we use are not strong enough to provide such theoretical lower bound guarantees.

V. Subset Cardinality Formulas (fixedbandwidth-eq)

These formulas are variations of benchmarks suggested in [21], [22]. To generate subset cardinality formulas, consider a $0/1$ $n \times n$ matrix $A = (a_{i,j})$ and identify positions $a_{i,j} = 1$ with variables $x_{i,j}$. If we write $R_i = \{j \mid a_{i,j} = 1\}$ and $C_j = \{i \mid a_{i,j} = 1\}$ to denote the positions of 1s/variables in row i and column j, respectively, the subset cardinality formula over A encodes the cardinality constraints $\sum_{j \in R_i} x_{i,j} \geq |R_i|/2$ and $\sum_{i \in C_j} x_{i,j} \leq |C_i|/2$ for all $i, j \in [n]$. In the case when all rows and columns have $2k$ variables, except for one row and column that have $2k + 1$ variables, the formula is unsatisfiable, but is hard for resolution if the positions of the variables are "scattered enough" [23]. It can be shown (see [24]) that this holds even if we strengthen the cardinality constraints to equalities $\sum_{j \in R_i} x_{i,j} = \lceil |R_i|/2 \rceil$ and $\sum_{i \in C_j} x_{i,j} = \lfloor |C_i|/2 \rfloor$. The clauses in the latter formula are a strict superset of the clauses in the former. We refer to these two variants of the formulas as `geq` and `eq` versions, respectively. For the SAT competition only `eq` versions were used. Formula sizes and number of variables scale linearly with n.

We want to generate *easy* instances of these formulas, however. It is not hard to show—as observed in [22]—that if one fixes a $0/1$ pattern for the first row and just shifts this pattern down the diagonal, then the resulting formulas have linear-size resolution proofs. The formulas remain easy even for tree-like resolution, albeit with a polynomial blow-up. We use the fixed bandwidth pattern 11010001 with 1s in positions 2^i for $i = 0, 1, 2, 3$ suggested in [22], and then flip one 0 somewhere to 1 to obtain an unsatisfiable instance.

VI. Even Colouring Formulas (ecgrid$\langle r \rangle$)

Even colouring formulas [25] are defined on connected graphs $G = (V, E)$ with all vertices of constant, even degree. Edges $e \in E$ correspond to variables x_e, and for all vertices $v \in V$ constraints $\sum_{e \ni v} x_e = \deg(v)/2$ assert that there is a $0/1$-colouring such that each vertex has an equal number of incident 0- and 1-edges. The formula is satisfiable if and only if the total number of edges is even. For suitably chosen graphs these formulas are empirically hard for CDCL. We are not aware of any formal resolution size lower bounds, and a naive application of the standard lower bound techniques does not work. It does not obviously seem out of reach to establish exponential lower bounds with some extra effort to develop a variation of existing techniques, however.

We generate our even colouring formulas not for hard graphs, however, but again for grids with "wrap-around" edges, where we subdivide one edge into a degree-2 vertex to get an odd number of edges. There are reasons to believe that for grids with similar parameters as those in [8] one should get obtain strong size-space trade-offs, just as for Tseitin formula, but we have no formal proof for this. As for our Tseitin benchmarks, we instead fix the number of rows to be a small constant ($r = 6$ for the formulas used) and then scale the formulas by varying the number of columns. Formula sizes and number of variables scale linearly with n. Again, the formulas have linear-size resolution proofs and polynomial-size tree-like resolution proofs.

VII. Relativized PHP Formulas (rphp$\langle k \rangle$)

Relativized pigeonhole principle (PHP) formulas, which have been studied in [26], [27], are a variant of the well-known pigeonhole principle formulas with a twist to scale down the hardness from exponential to polynomial. These formulas claim that k pigeons (where we let k be a small constant) can fly into $k-1$ holes via n "resting places," where n is the parameter used to scale the formulas. There are clauses enforcing that pigeons fly into the resting places in a one-to-one fashion and continue from resting places to holes in a one-to-one fashion. For constant k, formula sizes and number of variables scale like n^2. It was shown in [27] that these formulas require resolution proofs of size roughly n^k, and such proofs can be found even in tree-like resolution.

REFERENCES

[1] A. Atserias, J. K. Fichte, and M. Thurley, "Clause-learning algorithms with many restarts and bounded-width resolution," *Journal of Artificial Intelligence Research*, vol. 40, pp. 353–373, Jan. 2011, preliminary version in *SAT '09*.

[2] K. Pipatsrisawat and A. Darwiche, "On the power of clause-learning SAT solvers as resolution engines," *Artificial Intelligence*, vol. 175, pp. 512–525, Feb. 2011, preliminary version in *CP '09*.

[3] "CNFgen formula generator and tools," https://github.com/MassimoLauria/cnfgen.

[4] M. Lauria, J. Elffers, J. Nordström, and M. Vinyals, "CNFgen: a generator of crafted CNF formulas," 2016, manuscript in preparation.

[5] G. Tseitin, "On the complexity of derivation in propositional calculus," in *Structures in Constructive Mathematics and mathematical Logic, Part II*, A. O. Silenko, Ed. Consultants Bureau, New York-London, 1968, pp. 115–125.

[6] A. Urquhart, "Hard examples for resolution," *Journal of the ACM*, vol. 34, no. 1, pp. 209–219, Jan. 1987.

[7] P. Beame, C. Beck, and R. Impagliazzo, "Time-space tradeoffs in resolution: Superpolynomial lower bounds for superlinear space," in *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC '12)*, May 2012, pp. 213–232.

[8] C. Beck, J. Nordström, and B. Tang, "Some trade-off results for polynomial calculus," in *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC '13)*, May 2013, pp. 813–822.

[9] B. Krishnamurthy, "Short proofs for tricky formulas," *Acta Informatica*, vol. 22, no. 3, pp. 253–275, Aug. 1985.

[10] G. Stålmarck, "Short resolution proofs for a sequence of tricky formulas," *Acta Informatica*, vol. 33, no. 3, pp. 277–280, May 1996.

[11] M. L. Bonet and N. Galesi, "Optimality of size-width tradeoffs for resolution," *Computational Complexity*, vol. 10, no. 4, pp. 261–276, Dec. 2001, preliminary version in *FOCS '99*.

[12] E. Ben-Sasson and A. Wigderson, "Short proofs are narrow—resolution made simple," *Journal of the ACM*, vol. 48, no. 2, pp. 149–169, Mar. 2001, preliminary version in *STOC '99*.

[13] E. Ben-Sasson and J. Nordström, "Short proofs may be spacious: An optimal separation of space and length in resolution," in *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*, Oct. 2008, pp. 709–718.

[14] ——, "Understanding space in proof complexity: Separations and trade-offs via substitutions," in *Proceedings of the 2nd Symposium on Innovations in Computer Science (ICS '11)*, Jan. 2011, pp. 401–416.

[15] M. Järvisalo, A. Matsliah, J. Nordström, and S. Živný, "Relating proof complexity measures and practical hardness of SAT," in *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, ser. Lecture Notes in Computer Science, vol. 7514. Springer, Oct. 2012, pp. 316–331.

[16] E. Ben-Sasson, R. Impagliazzo, and A. Wigderson, "Near optimal separation of tree-like and general resolution," *Combinatorica*, vol. 24, no. 4, pp. 585–603, Sep. 2004.

[17] M. Alekhnovich, J. Johannsen, T. Pitassi, and A. Urquhart, "An exponential separation between regular and general resolution," *Theory of Computing*, vol. 3, no. 5, pp. 81–102, May 2007, preliminary version in *STOC '02*.

[18] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[19] S. R. Buss and L. Kołodziejczyk, "Small stone in pool," *Logical Methods in Computer Science*, vol. 10, Jun. 2014.

[20] S. M. Chan, M. Lauria, J. Nordström, and M. Vinyals, "Hardness of approximation in PSPACE and separation results for pebble games (Extended abstract)," in *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS '15)*, Oct. 2015, pp. 466–485.

[21] I. Spence, "sgen1: A generator of small but difficult satisfiability benchmarks," *Journal of Experimental Algorithmics*, vol. 15, pp. 1.2:1.1–1.2:1.15, Mar. 2010.

[22] A. Van Gelder and I. Spence, "Zero-one designs produce small hard SAT instances," in *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, ser. Lecture Notes in Computer Science, vol. 6175. Springer, Jul. 2010, pp. 388–397.

[23] M. Mikša and J. Nordström, "Long proofs of (seemingly) simple formulas," in *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, ser. Lecture Notes in Computer Science, vol. 8561. Springer, Jul. 2014, pp. 121–137.

[24] ——, "A generalized method for proving polynomial calculus degree lower bounds," in *Proceedings of the 30th Annual Computational Complexity Conference (CCC '15)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 33, Jun. 2015, pp. 467–487.

[25] K. Markström, "Locality and hard SAT-instances," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 1-4, pp. 221–227, 2006.

[26] A. Atserias, M. Müller, and S. Oliva, "Lower bounds for DNF-refutations of a relativized weak pigeonhole principle," in *Proceedings of the 28th Annual IEEE Conference on Computational Complexity (CCC '13)*, Jun. 2013, pp. 109–120.

[27] A. Atserias, M. Lauria, and J. Nordström, "Narrow proofs may be maximally long," *ACM Transactions on Computational Logic*, vol. 17, pp. 19:1–19:30, May 2016, preliminary version in *CCC '14*.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Community Attachment Instances

Benchmarks Description

Jesús Giráldez-Cru and Jordi Levy
Artificial Intelligence Research Institute (IIIA-CSIC)
Bellaterra, Catalonia, Spain
{jgiraldez,levy}@iiia.csic.es

***Abstract*—We describe a family of random SAT instances generated with the *Community Attachment* model. This model allows the generation of highly modular instances, a very characteristic feature of most application SAT benchmarks.**

Introduction

It is well-known that random k-CNF and industrial SAT instances have a very distinct nature. While random formulas can be easily generated on demand, the set of industrial instances, which encode real-world problems, is reduced. Moreover, solving a set of industrial benchmarks often has a high cost. This has important impacts on the process of developing and testing new SAT solving techniques. For this reason, the generation of random instances that realistically model application problems has been proposed as an important challenge [1], [2], [3].

An important feature shared by the majority of real-world SAT instances is the community structure. It has been shown that most industrial SAT formulas exhibit a clear community structure, or high modularity Q [4]. This means that, representing formulas as graphs, we can find a partition of the formula into communities with many edges between nodes of the same community (i.e., many clauses relating variables of the same community), and few edges connecting distinct communities. This property is very characteristic in real-world problems in contrast to randomly generated instances, where modularity is very low. In the context of SAT, it has been shown that the community structure is correlated with the runtime of CDCL SAT solvers [5], [6]. Moreover, it has been also used to improve the performance of some solvers [7], [8], [9], [10].

Community Attachment Model

Recently, it has been proposed a new model of generation of random SAT instances, called *Community Attachment*. This model allows the generation, with high probability, of random SAT instances with clear community structure. Notice that this kind of instances are very unlikely to be produced by the classical random model. This model is published in:

- [11] J.Giráldez-Cru and J. Levy. Generating SAT instances with community structure. Artificial Intelligence 2016. DOI: http://dx.doi.org/10.1016/j.artint.2016.06.001.
- [12] J.Giráldez-Cru and J. Levy. A modularity-based random SAT instances generator. Proc. of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15), pp. 1952–1958

As in the classical random model, the Community Attachment model is parametric in the number of variables n, the number of clauses m, and the clause size k. Additionally, it is also parametric in a probability P and a partition C of the set of variables. In this model all variables of a clause belong to the same community with probability P, and with probability $1-P$ they all belong to distinct communities. In particular, the probability P is taken as:

$$P = Q + \frac{1}{c} \tag{1}$$

where Q is the modularity, and $c = |C|$ is the number of communities.

In the previous equation, when the value of Q is high, the (expected) modularity of the instance is very close to this value. Therefore, for a high value of modularity, the resulting formula is more adequate to model industrial problems than classical random k-CNF formulas. Interestingly, this model also generates SAT instances very similar to the ones produced by the classical random model when the value of the modularity is low.

We recall that industrial SAT instances are characterized by a high modularity $Q > 0.7$ in most cases, while the modularity of random SAT formulas is very small $Q \approx 0.3$. Moreover, the number of communities c is usually in the interval $(10, 100)$ [4].

This generator is publicly available in http://www.iiia.csic.es/~jgiraldez/software. We address the reader to references [12], [11] for further details about the Community Attachment model.

Set of Submitted Instances

Using the Community Attachment model described in the previous section, we generate a family of pseudo-industrial random SAT instances using the following parameters:

- Number of variables $n = 2200$
- Number of clauses $m = 9086$
- Clause size $k = 3$
- Modularity $Q = 0.8$
- Number of communities $c = 40$

We remark that the phase transition point of this family ($Q = 0.8$) was experimentally found for this clause/variable ratio $m/n = 9086/2200 = 4.13$.

Finally, we remark that when the value of modularity Q used to generate the family of instances is very high, there exist a small subset of instances having a very small refutation. This happens because most of the clauses relate variables of the same community, and hence it is more likely to find a small unsatisfiable set of clauses which only contains variables of one or few communities. Therefore, it is recommendable to filter this family in order to remove those *easy* instances.

REFERENCES

[1] B. Selman, H. A. Kautz, and D. A. McAllester, "Ten challenges in propositional reasoning and search," in *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI'1997)*, 1997, pp. 50–54.

[2] H. A. Kautz and B. Selman, "Ten challenges redux: Recent progress in propositional reasoning and search," in *Proc. of the 9th International Conference on Principles and Practice of Constraint Programming (CP'2003)*, 2003, pp. 1–18.

[3] R. Dechter, *Constraint Processing*. Morgan Kaufmann, 2003.

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

[5] Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, and L. Simon, "Impact of community structure on SAT solver performance," in *Proc. of the 17th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'14)*, 2014, pp. 252–268.

[6] Z. Newsham, W. Lindsay, V. Ganesh, J. H. Liang, S. Fischmeister, and K. Czarnecki, "SATGraf: Visualizing the evolution of SAT formula structure in solvers," in *Proc. of the 18th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'15)*, 2015, pp. 62–70.

[7] R. Martins, V. M. Manquinho, and I. Lynce, "Community-based partitioning for maxsat solving," in *Proc. of the 16th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'13)*, 2013, pp. 182–191.

[8] T. Sonobe, S. Kondoh, and M. Inaba, "Community branching for parallel portfolio SAT solvers," in *Proc. of the 17th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'14)*, 2014, pp. 188–196.

[9] M. Neves, R. Martins, M. Janota, I. Lynce, and V. M. Manquinho, "Exploiting resolution-based representations for MaxSAT solving," in *Proc. of the 18th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'15)*, 2015, pp. 272–286.

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

[11] J. Giráldez-Cru and J. Levy, "Generating SAT instances with community structure," *Artificial Intelligence*, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2016.06.001

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

[13] A. Biere, "Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver Lingeling," in *Proc. of Pragmatics of SAT (POS'2014)*, 2014.

[14] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI'2009)*, 2009, pp. 399–404.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

SAT-Encodings of Sorting Networks

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

***Abstract*—We suggest some benchmarks based on a propositional encoding of sorting networks.**

I. Introduction

Sorting networks are a representation of data-oblivious sorting algorithms [1]. These algorithms perform a sequence of comparisons which only depends on the length of the input, and is independent of the actual data to sort. This makes them attractive for parallel implementations, as comparisons on disjoint elements of the input can be performed in parallel. Figure 1 shows a sorting network for 5 inputs. The horizontal lines, denoted channels, transport values from the left-hand side to the right-hand side. Whenever two of them are connected by a vertical line, denoted comparator, the values on the channels are compared, and swapped, if necessary. The depth of a sorting network is the number of parallel sorting steps required to sort every input.

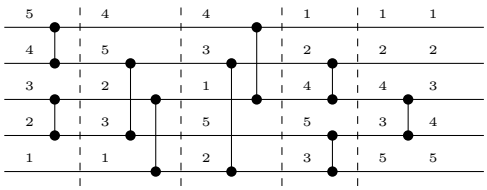


Fig. 1. A sorting network on 5 channels, sorting the input $(5, 4, 3, 2, 1)$.

Although optimal asymptotic bounds on the depth of sorting networks are known, it is hard to find concrete bounds on the optimal depth. Recent work [2], [3], [4] has used SAT-solvers to engage this problem. This is made possible by the zero-one-principle, which claims that it is sufficient to consider binary input sequences [1].

II. Encoding & Optimisations

A good encoding for sorting networks with bounded depth was given by Bundala and Zavodny in [2], together with a symmetry break on the first two layers. This was sufficient to prove optimal bounds on 11 to 16 channels. Codish, Cruz-Filipe and Schneider-Kamp introduced symmetry breaks on the last two layers [3]. Ehlers and Müller suggested an improved encoding, and used symmetries on the first two layers to minimise the number of variables in the SAT formulas, proving optimality for 17 channels [4].

III. Hardness

The set of benchmarks follows this history. We consider networks on 13, 16 and 17 channels. The prefix of the benchmarks names are "snw_n_d", where n denotes the number of channels, and d the number of layers. The formulas with infixes "13_8", "16_8" and "17_9" are unsatisfiable, all others are satisfiable. Furthermore, we use the following infixes to indicate which optimisations was used.

- **CCS**: Symmetry breaks due to Codish, Cruz-Filipe and Schneider-Kamp [3] were enabled.
- **Enc**: Improved encoding as in [4] used, which allows for more propagations, and reduces the number of clauses.
- **preOpt**: Symmetries were used to minimise the number of variables in the encoding [4].
- **pre**: A preprocessing step (Failed Literal Branching) was applied to the formula [4]. Interestingly, this easy preprocessing seems to have a big impact on solver performance.

The hardness of the resulting formulas depends on the set of optimisations used, ranging from trivial to hard.

We used different solvers when trying to find new bounds. Interestingly, Glucose and Lingeling were competitive on easy instances, whereas they were outperformed by MiniSAT 2.20 on harder instances. Therefore, it will be interesting to see how the solvers in the SAT Competition perform on them. The formula "17_9" corresponds to the easiest case we had to solve in [4], it took MiniSAT 2.20 roughly 6 hours to solve it.

Acknowledgment

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

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[2] D. Bundala and J. Zavodny, "Optimal sorting networks," in *LATA 2014, Madrid, Spain, March 10-14, 2014. Proc.*, ser. LNCS, A. H. Dediu, C. Martín-Vide, J. L. Sierra-Rodríguez, and B. Truthe, Eds., vol. 8370. Springer, 2014, pp. 236–247.

[3] M. Codish, L. Cruz-Filipe, and P. Schneider-Kamp, "Sorting networks: The end game," in *LATA 2015, Nice, France, March 2-6, 2015, Proc.*, ser. LNCS, A. H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, Eds., vol. 8977. Springer, 2015, pp. 664–675.

[4] T. Ehlers and M. Müller, "New bounds on optimal sorting networks," in *CiE 2015, Bucharest, Romania, June 29 - July 3, 2015. Proc.*, ser. LNCS, A. Beckmann, V. Mitrana, and M. I. Soskova, Eds., vol. 9136. Springer, 2015, pp. 167–176.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

An Interlocking Safety Proof Applied to the French Rail Network

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

I. Introduction

SNCF Reseau is the owner of the French rail network and manages the circulation of all trains on the national grid.
The system is characterised by critical systems that interact in a non-deterministic environment where qualified operators apply operating rules and regulations.

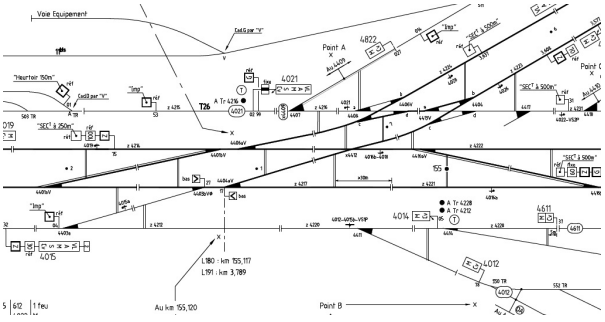
Global system safety is therefore based on both:

- The correctness and adequacy of operational rules and regulations; and
- The absence of safety failures in critical systems (interlocking systems, centralized control systems, etc.).

In order to continue to ensure a high level of safety, SNCF has for many years been interested in the use of formal methods. In particular, SAT solver technology has become a very active field of academic research. Interest in these techniques is partly due to their effectiveness in solving industrial problems, and partly to their ease of use. Proof engines that are based on such techniques make it possible to largely automate the proof of a safety property. Either the proof exists (in which case the property is valid), or a counter example is found (if a system execution leads to the violation of a property). These counter-examples are very useful for system development.

II. The Benchmark

This benchmark provides a safety proof for an interlocking system of approximately 100 routes, 25 signals and 35 points.



It is based on:

- A model of interlocking;
- An environment model that expresses safety properties (trains, points, signals, procedures, etc.);
- Two generic safety properties designed to ensure that:
  - There are no derailments by moving points; and
  - There are no head-on collisions between two trains.

The first step of the proof verifies that no property is falsified. This is achieved by a BMC with a depth of 10 (in this example). This corresponds to the files "sncf_ixl_bmc_depth*.cnf". Assuming the previous step completes successfully, the induction proof is run on the set of safety properties. The induction proof is divided into two stages corresponding to the files:

- "sncf_ixl_proof_induction_base.cnf" (base case);
- "sncf_ixl_proof_induction_step.cnf" (inductive step).

If all instances return UNSAT then safety properties are proven for the entire interlocking model.

Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, University of Helsinki 2016. ISBN 978-951-51-2345-9.

Industrial Combinational Equivalence Checking Benchmark Suite

Valeriy Balabanov
Calypto Systems Division, Mentor Graphics, Fremont, USA
balabasik@gmail.com

***Abstract*—This document describes the benchmark suite submitted to the main track of SAT competitions 2016.**

I. Introduction

Combinational equivalence checking (CEC) is a problem of verifying functional equivalence of two combinational circuits. The need in CEC may emerge for various reasons. Most commonly the golden specification (*spec*) is verified against the synthesized/simplified version of itself. Boolean satisfiability problem (SAT) and CEC could be interchangeably reduced to each other, and therefore CEC benchmarks are of a direct relevance to the SAT community. For more recent advancements in the state of the art in combinational equivalence checking please refer to [1], [2].

II. Benchmark suite description

This benchmark suite contains 45 CEC problems represented in the "dimacs" format. Most of the problems in the suite come from the domain of the bit-vector arithmetics, have medium to hard complexity, and are believed to be unsatisfiable. Benchmarks have 36000 variables and 135000 clauses on average (ranging from 1377 to 170852 variables, and 5289 to 659936 clauses).

III. Availability

Benchmarks are publicly available from the competitions website. They could be used by the research community for non-profit purposes.

References

[1] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12-16, 2001*, 2001, pp. 114–121. [Online]. Available: http://dx.doi.org/10.1109/DATE.2001.915010

[2] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *2006 International Conference on Computer-Aided Design, ICCAD 2006, San Jose, CA, USA, November 5-9, 2006*, 2006, pp. 836–843. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233679

Solver Index

Benchmark Index

Author Index