Proceedings of

SAT COMPETITION 2014

Solver and Benchmark Descriptions

Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo (editors)
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 2014 (SC 2014), an open competitive event for SAT solvers, was organized as a satellite event of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014) and FLoC Olympic Games 2014 within the largest event in the history of logic, Vienna Summer of Logic (VSL) 2014, and the Federated Logic Conference (FLoC 2014) in Vienna, Austria. SC 2014 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held 2002-2005 and biannually during 2007-2013, the SAT-Races held in 2006, 2008 and 2010, and SAT Challenge 2012.

SC 2014 consisted of a total of 13 competition tracks, each track being characterized by the combination of

(i) the type of solvers allowed to participate in the track,

(ii) the computational resources provided to each solver, and

(iii) the class of benchmarks used (Application / Hard Combinatorial / Random; SAT / UNSAT / SAT+UNSAT).

In addition to nine main tracks for sequential core solvers and three tracks for parallel core solvers, the competition included a MiniSAT Hack Track following the tradition set forth by previous SAT Competitions. As in SAT Competition 2013, solvers competing in the three main tracks on purely unsatisfiable formulas were required to output actual proofs as certificates for unsatisfiability. Within each track, competing solvers were to solve an unknown set of benchmark instances selected by the organizers. The number of solved instances was used for ranking the solvers under a per-instance timeout of 5000 seconds. Due to the very high computational resource requirements needed for realizing a large-scale solver computation such as SC 2014, each participant was restricted to be a co-author of at most four different sequential solvers, two different parallel solvers, and one MiniSAT Hack Track submission.

There were two ways of contributing to SC 2014: 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 in the competition. Following the tradition put forth by SAT Challenge 2012, the rules of SC 2014 required all contributors (both solver and benchmark submitters) to submit a short, around 2-page long solver/benchmark description as part of their contribution. As a result, we obtained around 50 solver descriptions and 10 benchmark descriptions from at total of around 80 contributors. This book contains all these non-peer-reviewed descriptions in a single volume, providing a way of consistently citing the individual descriptions. We have also included
descriptions of the selection and generation process applied in forming the benchmark instances used in the SC 2014 competition tracks. 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 2014 benchmarks, and also as a future historical reference providing a snapshot of the SAT solver technology actively developed in 2014.

Successfully running SC 2014 would not have been possible without active support from the community at large. Major final decisions on outcomes of SC 2014 were deliberated by a distinguished panel of judges: Pete Manolios, Lakhdar Saïs, and Peter Stuckey. The University of Texas at Austin provided critical infrastructure by offering vast computing resources for running SC 2014. We acknowledge the Texas Advanced Computing Center (TACC, http://www.tacc.utexas.edu) at The University of Texas at Austin for providing grid resources. The real silver medals given as first prizes in each of the competition tracks were provided by The FLoC Olympic Games organization and the SAT Association, for which we are very grateful. We would also like to emphasize that a competition does not exist without participants: we thank all those who contributed to SC 2014 by submitting either solvers or benchmarks and the related description. Finally, M.J. acknowledges financial support from Academy of Finland under grants 251170 (COIN Finnish Centre of Excellence in Computational Inference Research) and 276412.

Dublin, Ulm, Austin, and Helsinki, June 27, 2014

Anton Belov, Daniel Diepold, Marijn J.H. Heule, & Matti Järvisalo
SAT Competition 2014 Organizers
Contents

Preface ......................................................... 3

Solver Descriptions

Balance between intensification and diversification: two sides of the same coin
Chumin Li, Chong Huang, and Ruchu Xu ........................................... 10

BFS-Glucose
Dávid Bartók and Zoltán Ádám Mann ................................................. 12

CBPeneLoPe and ParaCIRMiniSAT
Tomohiro Sonobe and Shuya Kondoh ................................................... 14

CCA2014: A Local Search Solver for the SAT Problem
Chengqian Li, Jianhong Li, and Yuziang Wu ........................................... 16

CCAnr+glucose in SAT Competition 2014
Shaowei Cai, Chuan Luo, and Kaile Su ................................................. 17

CCgscore in SAT Competition 2014
Shaowei Cai, Chuan Luo, and Kaile Su ................................................. 19

CLAS – A Parallel SAT Solver that Combines CDCL, Look-Ahead and SLS Search Strategies
Adrian Balint, Davide Lanti, Ahmed Irfan, and Norbert Manthey ................. 21

CPSparrow
Adrian Balint ................................................................. 22

CryptoMiniSat v4
Mate Soos ................................................................. 23

CryptoMiniSat v4, tuned version
Mate Soos and Marius Lindauer ......................................................... 24

CSCCSat2014 in SAT Competition 2014
Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su ................................... 25

DCCASat+march_rw in SAT Competition 2014
Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su ................................... 27

Dimetheus
Oliver Gableske ................................................................. 29

Glucose in the SAT 2014 Competition
Gilles Audemard and Laurent Simon .................................................... 31

Glucose-Syrup: a parallel version of Glucose with Lazy Clause Exchange
Gilles Audemard and Laurent Simon .................................................... 33
GLUEMINISAT2.2.8
  Hidetomo Nabeshima, Koji Iwanuma, and Katsumi Inoue 35
Glue_lgl_split and GlueSplit_clasp with a Split and Merging Strategy
  Jingchao Chen 37
Yet another Local Search Solver and Lingeling and Friends Entering the SAT Competition
  Armin Biere 39
MiniSAT with Classification-based Preprocessing
  Wenzhang Chen, Adele Howe, and Darrell Whitley 41
miniLoCeG+Glucose
  Sebastian Burg, Oliver Bringmann, and Thomas Kropf 43
MiniSat-ClauseSplit
  Balázs Donát Bessenyei and Zoltán Ádám Mann 44
Minisat bbld
  Jingchao Chen 45
MiniSat_HACK_999ED, MiniSat_HACK_1430ED and SWDiA5BY
  Chanseok Oh 46
MinitSAT
  Norbert Manthey 48
MiPSAT – Probing as Inprocessing in MiniSAT
  Tobias Philipp 49
MiniSAT with Speculative Preprocessing
  Wenzhang Chen, Adele Howe, and Darrell Whitley 50
Ncca+: Configuration Checking Strategy with Novelty Heuristic
  Djamel Habet, Donia Toumi, and André Abramé 52
Nigma 1.1
  Gianfranco Ciardo and Chuan Jiang 53
ntusat and ntusatbreak
  Hsiao-Lun Wang and Jie-Hong Roland Jiang 54
Pcasso – a Parallel CooperAtive Sat SOlver
  Davide Lanti, Ahmed Irfan, and Norbert Manthey 56
PeneLoPe in SAT Competition 2014
  Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette 58
pGlucose
  Souheib Baarir, Julien Sopena, and Julia Lawall 60
pmcSAT
  Ricardo Marques, Luis Guerra e Silva, Paulo Flores, and L. Miguel Silveira 61
probSAT and pprobSAT
  Adrian Balint and Uwe Schöning 63
provoSATeur+glucose
  Sebastian Burg, Oliver Bringmann, and Thomas Kropf 64
Riss 4.27
  Norbert Manthey 65
Riss 4.27 BlackBox
  Enrique Matos Alfonso and Norbert Manthey 68
ROKK1.0.1 Takeru Yasumoto and Takumi Okuwaga 70
Benchmark Descriptions

Generating the Uniform Random Benchmarks
   Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo 80
The Application and the Hard Combinatorial Benchmarks in SAT Competition 2014
   Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo 81
Benchmark for Complete and Stable Semantics for Argumentation Frameworks
   Johannes Peter Wallner 84
Air Traffic Controller Shift Scheduling
   Stojadinović, Mirko 86
Satisfiability through auto-correlation
   Srinivasan Arunachalam and Ilias Kotsireas 87
Generating Clique Coloring Problem Formulas
   Norbert Manthey 89
Erdös Discrepancy Problems
   Marijn J.H. Heule 90
Synthesis of Smaller Equivalent Deterministic Transition-based Generalized Büchi Automata
   Souheib Baarir and Alexandre Duret-Lutz 92
SAT-Encoding of Step-Reduced MD5
   Chu-Min Li and Bing Ye 94
Solving Polarium Puzzles with SAT
   Martin Philipp and Tobias Philipp 96
Too Many Rooks
   Norbert Manthey and Peter Steinke 97
sgen6: A generator for small but difficult unsatisfiable instances
   Ivor Spence 99
A small but hard symbolic simulation benchmark
   Alberto Griggio 101
Description of Z1 Benchmarks for SAT Competition 2014
   Allen Van Gelder 102

Solver Index 105
SOLVER DESCRIPTIONS
Balance between intensification and diversification: a unity of opposites

Chumin LI∗†,
†MIS, Université de Picardie Jules Verne, France
Email: chu-min.li@u-picardie.fr

Chong HUANG*, Ruchu XU‡
‡Huazhong University of Science and Technology, China
Email: *jason_hch@hotmail.com, ‡xrcy0315@sina.com

I. INTRODUCTION

In Stochastical Local Search (SLS) for SAT, intensification refers to search steps improving the objective function, and diversification refers to search steps moving to different areas of the search space. With the introduction of g2wsat [1], the intensification steps are more clearly distinguished from the diversification steps: when there are promising decreasing variables, they are deterministically flipped to improve the objective function, otherwise novelty [2] is called to diversify search, in which intensification steps can be still be made, controlled by a noise. Although very effective, promising decreasing (PD) variable is a too strong notion forbidding many useful intensification steps, because there are few promising decreasing variables in a local search solving a hard SAT instance. Recently, a Configuration Checking [3] (CC) notion is introduced in SLS for SAT and proves to be very useful: an improving variable is deterministically flipped if one of its neighbors has been flipped since the last time x was flipped.

The PD notion and the CC notion are both very effective but are two extremities. In the last work of BalancedZ [4], we proposed a compromise between them called changing decreasing (CD). A variable x is a changing decreasing (CD) variable if \( \text{score}(x) > 0 \) and if it occurs in a clause that has been changed from satisfied to unsatisfied or from unsatisfied to satisfied since the last time x was flipped. On the one hand, a CD variable x is necessarily a CC variable, i.e. one of its neighbors has been flipped since x was flipped. However the inverse is not true. Obviously, CD variables reflect more a changing context than the CC variables. On the other hand, a PD variable is a CD variable in most cases, with one exception: after flipping a variable x with \( \text{score}(x) < 0 \), x becomes a decreasing variable (\( \text{score}(x) > 0 \)). Then, by flipping a variable y, x will be a CD variable if \( \text{score}(x) > 0 \) and it occurs in a clause that has been changed from satisfied to unsatisfied, or from unsatisfied to satisfied. In this case, however, x is not a PD variable.

BalancedZ also employs other techniques to the balance between intensification and diversification according to the 80/20 Rule (i.e. 80% of steps are intensification and 20% of steps are diversification). In this work, we focus on the breaking tie strategy in classical SLS solvers and propose a probability call \( bp \) (break in tie probability) to improve the diversification.

II. MAIN TECHNIQUES

Given a SAT instance to solve, BalancedZ first generates a random assignment and the weight of all clauses being initialized to 1. The objective function of BalancedZ is the sum of weights of all unsatisfied clauses to be reduced to 0. The score of a variable is the decrease of the objective function if the variable is flipped. While the objective function is not 0, BalancedZ modifies the assignment as follows:

1) If there are changing decreasing variables, flip the best one (CD step);
2) Otherwise, if there are decreasing variables with very high score, flip the best one (AD step);
3) Otherwise, randomly pick an unsatisfied clause c and flip the least recently flipped one (Div step);
4) Increase the weight of unsatisfied clauses, and smooth the clause weights under some conditions

where CD refers to Changing Decreasing, AD refers to Aspiration Decreasing, and Div refers to Diversification.

In a CD step, the best variable is defined to be the variable having the highest score, with probability \( 1 - bp \) breaking tie in favor of the variable that most recently becomes a changing decreasing variable.

In a AD step, the best variable is defined to be the variable having the highest score, with probability \( 1 - bp \) breaking tie in favor of the variable that least recently becomes a changing decreasing variable.
The remaining ties of a CD step and a AD step are broken in favor of the least recently flipped variable.

Increasing the weight of unsatisfied clauses increases the score of variables in these clauses, making some steps intensifying. So clause weighting techniques are very important for the performance of a SLS solver. BalancedZ utilizes two clause weighting techniques for random $k$-SAT problem: SWT scheme[5] for $k < 4$; PAWS scheme[6] for $k \geq 4$. PAWS scheme is also applied to the crafted problems. The clause weighting techniques remain the same and more details could be found in [4].

BalancedZ employs an adaptive mechanism to obtain the peak performance after manually tuning different values of $bp$. BalancedZ manages an adaptive noise in the same way of [7], and we consider the value of noise as an indicator underlying the search landscape. In this work, we set $bp = noise/c_1$, where $c_1$ is positive constant.

Last but not least, compared to the microscopic balance mentioned above, BalancedZ accomplishes the task of balancing the intensification and diversification macroscopically as well. Our experimental analysis of solving random $k$-SAT problems with different noises indicates that BalancedZ delivers optimal performance when the ratio of number of steps CD and AD to the number of steps Div is roughly 80% to 20%, which conforms to the 80/20 Rule (Pareto Principle) by coincidence.

III. PARAMETER DESCRIPTION

The parameters of BalancedZ mentioned above are set as follows:

1. $c_1 = 5$, this parameter is not sensitive, but should be set less than 10;
2. $\theta = 1/10$, $\phi = 0.2$, these two parameters are designed for adaptive noise mechanism and more details could be found in [7];

Other parameters include: -maxtries $a$, -seed $b$, allowing to run $a$ times BalancedZ and the random seed of the first run being $b$.

IV. SAT COMPETITION 2014 SPECIFICS

BalancedZ is submitted to two tracks in the competition: Hard-combinatorial SAT track and Random SAT track. BalancedZ is complied by gcc with the following command:

```
gcc -O3 -static BalancedZ.c -o BalancedZ
```

BalancedZ should be called in the competition using:

```
BalancedZ INSTANCE -seed SEED
```

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China. (Grant No. 61272014 and Grant No. 61070235)

REFERENCES

BFS-Glucose

Dávid Bartók
Budapest University of Technology and Economics
Budapest, Hungary

Zoltán Ádám Mann
Budapest University of Technology and Economics
Budapest, Hungary

Abstract—Our program partitions the search space into multiple disjoint parts, and runs a solver instance in each of them. Using a best-first-search approach, it periodically transfers control to the search instance that seems to be the most promising. As a base solver, Glucose is used.

I. INTRODUCTION

The aim of our method is to accelerate a CDCL solver by mitigating the negative effect of bad early branching decisions. For this purpose, the depth-first-search approach underlying CDCL solvers is complemented by a higher-level best-first-search control procedure that makes it possible to quickly jump between different parts of the search space, always focusing on the most promising part.

II. MAIN TECHNIQUES

Our algorithm uses a base solver \( S \), and enhances it with the following techniques:

- First, the search space is divided into a number of disjoint parts. We do this by selecting \( k \) variables and generating all the \( 2^k \) possible value combinations for them. This partitions the search space into \( 2^k \) disjoint parts.
- In each part, an instance of \( S \) is started (\( S_1, S_2, \ldots, S_{2^k} \)). That is, each \( S_i \) is initialized with the appropriate values of the selected variables as assumptions.
- One of the solver instances is run for a pre-defined number of steps.
- Afterwards, a heuristic valuation function is used to determine which solver instance to run next. The chosen instance is again run for a defined number of steps, and so on, the cycle starts again.
- If one of the solver instances finds a solution, then the problem instance is solvable and the program finishes. If a solver instance \( S_i \) returns UNSAT, then the corresponding part of the search space does not contain any solution, hence \( S_i \) will not be run again. If all solver instances have stopped with UNSAT, then the problem instance is unsolvable, and the program finishes.

As base solver, we use Glucose \[1\]. (It would also be possible to use another exact SAT solver.)

The heuristic valuation function uses the following pieces of information for determining the attractiveness of a solver instance: current depth, biggest depth so far, average depth, current decision level, biggest decision level so far, average length of learned clauses, average LBD, number of steps already taken. The valuation function is a linear combination of these values. Every time a solver instance is stopped, its score is computed using the valuation function and stored. Then, we choose the solver instance with highest score to run next.

III. MAIN PARAMETERS

The following parameters play an important role in BFS-Glucose:

- The weights of the components in the heuristic valuation function
- The number of steps after which a solver instance is stopped. This is actually governed by two parameters: an initial value and a factor (greater than 1) by which the value is multiplied every time a new solver instance is started
- The number of solver instances generated at the beginning
- Which variables are used as assumptions
- Whether restarts are used

The values for these parameters were tuned based on experiments with benchmarks of the 2013 SAT Competition.

IV. IMPLEMENTATION DETAILS

The solver is implemented in C++, on top of Glucose 3.0.

V. SAT COMPETITION 2014 SPECIFICS

BFS-Glucose is submitted to the following tracks:

- Sequential, Application SAT track
- Sequential, Application SAT+UNSAT track
- Sequential, Hard-combinatorial SAT track
- Sequential, Hard-combinatorial SAT+UNSAT track
- Sequential, Random SAT track
- Sequential, Random SAT+UNSAT track

Compilation is carried out using g++ in 32-bit version, with O3 optimization.

ACKNOWLEDGMENT

This work was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947) and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

The authors would like to thank Gilles Audemard and Laurent Simon for their work on Glucose. If BFS-Glucose is successful, this is to a large extent due to the strength of Glucose.
REFERENCES

CBPeneLoPe and ParaCIRMiniSAT

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

Shuya Kondoh
Graduate School of Information Science and Technology,
University of Tokyo, Japan

Abstract—We briefly introduce our solvers, CBPeneLoPe and ParaCIRMiniSAT, submitted to SAT Competition 2014. Community Branching (CB) is a novel diversification technique for parallel portfolio-based SAT solvers, and we implemented CB to the latest version of PeneLoPe. We also submit ParaCIRMiniSAT that participated in SAT Challenge 2012.

I. INTRODUCTION

We submit two parallel solvers to SAT Competition 2014. CBPeneLoPe uses our novel diversification technique, Community Branching (CB). ParaCIRMiniSAT is almost the same version of our past solver that participated in SAT Challenge 2012, however we believe that this solver can be still competitive.

In this description, we explain Community Branching and the two solvers.

II. COMMUNITY BRANCHING

Portfolio approach for parallel SAT solvers is known as the standard parallelisation technique. In portfolio, diversification of the search between workers is an important factor in portfolio [1]. The diversification is implemented by setting different parameters for each worker. However, it is difficult to combine the search parameters properly in order to avoid overlaps of search spaces between the workers. For this issue, we propose a novel diversification technique, denominated community branching. In this method, we assign a different set of variables (community) to each worker and force them to select these variables as decision variables in early decision levels. In this manner, we can avoid overlaps of search spaces between the workers more vigorously than the existing method. In order to create communities, we create a graph where a vertex corresponds to a variable and an edge corresponds to a relation between two variables in a same clause, proposed as Variable Incidence Graph (VIG) in [2]. After that, we apply Louvain method [3], one of the modularity-based community detection algorithms, to make the communities of the VIG. The variables in a community have strong relationships, and a distributed search for different communities can benefit the whole search.

The pseudo code of community branching is exhibited in Figure 1. The function “assign_communities” conducts the community detection for the VIG made from the given CNF and learnt clauses. The reason why the learnt clauses are included is that this function can be called multiple times during the search. We should reconstruct the communities along with the transformation of the graph caused by the learnt clauses, which is also mentioned in [2]. The function “community_detection” returns detected communities, in which the Louvain method is used, from the given CNF and the learnt clauses. The variable “coms” is a set of the communities (a two-dimensional array). Then the communities in “coms” are sorted by each size in descending order. We should conduct the search for larger communities preferentially because they can be core parts of the given instance, and these cores should be distributed to each worker. Finally, each community is assigned to each worker (the variable “worker_num” stands for the number of the workers). This procedure is conducted only by a master thread (thread ID 0), and the learnt clauses only in the master thread are used. Note that the number of the communities depends on the community detection algorithm. If the number of the detected communities is greater than the number of the workers, two or more communities are assigned to one worker. In the reverse case, some workers have no communities.

After the assignment of the communities, each worker calls

```c
assign_communities() {
    coms = community_detection(
        convert_CNF_to_VIG(
            given_CNF + learnt_clauses);
        sort_by_size_in_descending_order(coms);
        for(i = 0; i < coms.size(); i++) {
            worker_id = i % worker_num;
            assign(coms[i], worker_id);
        }
    )
}
run_count = 0;
community_branching() {
    if (run_count++ % INTERVAL > 0)
        return;
    [choose one of the assigned communities in rotation]
    for each var in the chosen community
        bump_VSIDS_score(var, BUMP_RATIO);
}
```

Fig. 1. Pseudo code of community branching
the function “community_branching” for every restart. This function chooses a community from the given communities and increases the VSIDS scores of the variables in the chosen community. The variable “run_count” counts the number of the executions of this function, and the main part of this function is executed for every “INTERVAL” restarts. This constant number adjusts the switching of the used community. If this value is small, the worker quickly switches the focusing community. In the main part of this function, one community is chosen, and the VSIDS scores of the variables are increased by proportional to “BUMP_RATIO”. In general, “BUMP_RATIO” is set to 1 at clause learning. In order to force the variables in the community to be selected as decision variables right after the restart, “BUMP_RATIO” should be a large value. In addition, we set an interval for reconstruction of the communities, an interval for the function “assign_communities”. We call it “Community Reconstruction Interval (CRI)” in this description. In particular, the communities are reconstructed for every “CRI” restarts.

III. CBPeneLoPe

We implemented our community branching upon PeneLoPe [4] submitted to SAT Competition 2013 (the download URL is https://bitbucket.org/bhoessen/penelope and the last updated date is denoted as “2013-07-14” at this time). We added some parameter settings to the configuration file (“configuration.ini”) because the default setting of this file was aimed at eight threads. We set “INTERVAL” as one, “BUMP_RATIO” as 100 and “CRI” as 3000 for community branching.

IV. ParaCIRMiniSAT

ParaCIRMiniSAT is same version of the one submitted to SAT Challenge 2012 [5]. We changed some parameters in order to adjust the solver to 12 threads.

REFERENCES

CCA2014: A Local Search Solver for the SAT Problem
Chengqian Li, Jianhong Li, Yuxiang Wu
Department of Computer Science
Sun Yat-sen University, Guangzhou 510006, China
lichengq@mail2.sysu.edu.cn

Abstract—This document briefly describes the stochastic local search solver CCA2014, which is submitted to the SAT Challenge 2014. This solver is an implementation of the CCA algorithm with some minor enhancements.

I. INTRODUCTION

Algorithms for solving the SAT problem are mainly divided into complete algorithms and stochastic local search (SLS) ones. SLS algorithms can’t prove that an instance is unsatisfiable. But SLS algorithms are very efficient on many domains, especially on random instances. CCA2014 is a SAT solver based on the solver CCASat [1] [2] [3] and CCA2013 [4].

II. MAIN TECHNIQUES

CCA2014 is an incomplete SAT solver based on stochastic local search, and it incorporates techniques like tabu, look head and clause weighting.

For 3-SAT instances, we choose a variable to flip from CCD (Configuration Changed Decreasing) variables according to Ncca+ [5]. We prefer to pick a variable in CCD with the biggest score. If there are ties, we will use the number of occurrences of these variables on the unsatisfied clauses to break ties. If there are still ties, we will use the age of variables to break ties.

CCA2014 fixes some bugs of CCA2013 [4]. With these bugs, our solver may have run time error on some instances. And we find that some instances may not fit the DIMACS format of the specific rules for the SAT competition. On huge 3-SAT instance of Sat Competition 2013, some variable may not appear in any clause. This also makes CCA2013 fails on these instances.

III. MAIN PARAMETERS

In our solver, all Parameters are set according to CCASat [1]. For 3-SAT and structured instance, we set \(\gamma = \frac{200 + V(F)}{500}\) and \(\rho = 0.3\), where \(V(F)\) is the number of variables in the instance. For large k-SAT instances, we set \(d = 8\) and \(\beta = 2048\).

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

Our solver has two parts. The first part is to solve 3-SAT and structured instances. We use the CCA algorithm [2] to solve these instances. The second part is to solve K-SAT instances \((K > 3)\). We use the algorithm of CCASat [3] to solve these instances.

V. IMPLEMENTATION DETAILS

We use C++ language to implement our solver. The solver is implemented based on the code of CCASat [1] [2] [3] and CCA2013 [4].

VI. SAT COMPETITION 2014 SPECIFICS

This solver, a 64-bit binary, is submitted to SAT Challenge 2014, for the Random SAT track, Hard-combinatorial SAT track and Application SAT track. It’s compiled by the g++ Compiler with the “-O3” option. It’s running command is: g++ CCA2014.cpp -O3 -o CCA2014 && ./CCA2014 <instance file name> <random seed>.

VII. AVAILABILITY

This solver is not open source. And its not publicly available yet. Its not allowed to use it for commercial purposes, without permission.

ACKNOWLEDGMENT

We would like to thank Prof. Yongmei Liu for introducing us into this research field. We also thank Prof. Kaile Su and Dr. Shaowei Cai for sharing their strategies and source codes.

REFERENCES

CCAnr+glucose in SAT Competition 2014

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

Chuan Luo
School of EECS, Peking University,
Beijing, China

Kaile Su
IIS, Griffith University,
Brisbane, Australia

Abstract—This document describes the SAT solver “CCAnr+glucose”, which is combination of a local search solver and a complete solver.

I. INTRODUCTION

Recently, we proposed a diversification strategy for local search algorithms, which is called configuration checking (CC). The CC strategy is first applied to improving the performance of local search solvers for the minimum vertex cover (MVC) problem [1], [2], [3]. Thanks to its simplicity and effectiveness, the CC strategy has been successfully used to improve stochastic local search (SLS) algorithms for satisfiability (SAT) [4], [5], [6], [7], [8], [9], [10]. Especially, by enhancing CC with an aspiration mechanism, the heuristic of configuration checking with aspiration (CCA) [5] has led to the CCASat solver [7], which is the winner of random SAT track in SAT Challenge 2012.

In SAT Competition 2013, we submitted an SLS solver for SAT called CCAnr [11], which also adopts the CCA heuristic as its search framework. CCAnr shows good performance in solving structured SAT instances. The implementation details of CCAnr can be found in the literature [11].

As the performance of CCAnr is complementary to the performance of complete solvers on solving hard-combinatorial instances, we combine the SLS solver CCAnr with a complete solver glucose [12], and develops a new SAT solver called CCAnr+glucose.

II. MAIN TECHNIQUES

The CCAnr+glucose solver is a combination of the SLS solver CCAnr and the complete solver glucose.

The main procedures of CCAnr+glucose can be described as follows. For solving an SAT instance, the CCAnr+glucose solver first utilizes the SLS solver CCAnr to solve the instance with a cutoff time of $t$ CPU seconds. If the instance is solved by CCAnr with $t$ CPU seconds, then CCAnr+glucose reports the solution which is found by CCAnr. Otherwise, CCAnr+glucose activates the complete solve glucose to solving the instance, and reports the solution if the instance is solved by glucose within the remaining time.

III. MAIN PARAMETERS

The parameter $t$ is set to 1000 CPU seconds when the total cutoff time is given 5000 seconds.

The parameters used in the current version of the CCAnr solver in CCAnr+glucose are the same as the ones in the version which is submitted to SAT Competition 2013 [11].

The parameters used in the current version of the glucose solver in CCAnr+glucose are the same as the ones in the version which is submitted to SAT Competition 2013 [12].

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

Compared to the version of CCAnr submitted to SAT Competition 2013, the current version of CCAnr in CCAnr+glucose uses a more efficient approach to build the neighboring variable list of each variable in the formula.

V. IMPLEMENTATION DETAILS

CCAnr+glucose is implemented in programming language C/C++, and is developed on the basis of CCAnr and glucose.

VI. SAT COMPETITION 2014 SPECIFICS

The CCAnr+glucose solver is submitted to Hard-Combinatorial SAT track, SAT Competition 2014 and Hard-Combinatorial SAT+UNSAT track, SAT Competition 2014.

Generally, the command line for running CCAnr+glucose is

```
./CCAnr+glucose.sh <instance> <seed> <t>
```

Specially, for SAT Competition 2014 where the cutoff time is 5000 seconds, we set the parameter $t$ to be 1000 seconds, and thus the running command is

```
./CCAnr+glucose.sh <instance> <seed> 1000
```

VII. AVAILABILITY

The CCAnr+glucose solver is open source and publicly available for only research purposes.

REFERENCES


CCgscore in SAT Competition 2014

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

Chuan Luo
School of EECS, Peking University,
Beijing, China

Kaile Su
IIS, Griffith University,
Brisbane, Australia

Abstract—This document describes the SAT solver “CCgscore”, which is a local search solver. CCgscore is based on two main ideas, namely the Configuration Checking strategy, and the notion of generalized score which combines score and second-level score as well as age.

I. INTRODUCTION

Local search SAT solvers iteratively pick a variable and flip it, trying to find a satisfying assignment. The essential part of a local search solver is its heuristic to pick a variable, which is denoted as pickVar heuristic.

CCgscore is a local search solver and it has two main ideas. The first one is the configuration checking (CC) strategy. Initially proposed in [1], the CC strategy has been applied to minimum vertex cover problem [2], [3], and then to the SAT problem [4], [5], [6], [7], [8], [9]. This strategy is the main idea in a few state of the art local search SAT solvers such as CCASat [6], CScoreSAT [10] and Ncca+ [9]. According to the CC strategy for SAT, a variable \( x \) is configuration changed (denoted as conf\_changed) iff at least one of its neighboring variables has been flipped after \( x \)'s last flip. The CC strategy usually allows configuration changed variables to be flipped in the greedy mode.

The second idea in CCgscore is a hybrid scoring function called generalized score, denoted by \( gscore \). Recently, we proposed the notion of multilevel properties and showed that the second-level score \( score_2 \) denoted as \( score_2 \) [6], [10] and second-level make denote as \( make_2 \) [11] are particularly effective for solving random \( k \)-SAT with long clauses. The \( gscore \) function is defined as

\[
gscore(x) = score(x) + score_2(x)/\beta + age(x)/\theta
\]

. This hybrid scoring function was first proposed in CScoreSAT [10]. But in CScoreSAT [10], this function is only used when the algorithm gets stuck in local optima. However, in CCgscore, for random \( k \)-SAT with \( k > 3 \), the \( gscore \) function is used as the scoring function during the search process. Note that the value of parameter \( \beta \) varies for greedy mode and random mode in CCgscore.

II. THE CCGSCORE SOLVER

CCgscore distinguishes 3-SAT and \( k \)-SAT with \( k > 3 \), and uses different pickVar heuristics for them. In the following, we describe the two pickVar heuristics for 3-SAT and \( k \)-SAT with \( k > 3 \) respectively.

A. PickVar Heuristic for 3-SAT

The pickVar heuristic for solving 3-SAT instances is described in Algorithm 1.

**Algorithm 1: pickVar-heuristic for 3-SAT**

1. if \( \exists \) conf\_changed variables with \( score(x) > 0 \)
2. then
3. return such a variable with the greatest \( score \);
4. update\_clause\_weights();
5. pick a random unsatisfied clause \( c \);
6. if with probability \( wp \) then
7. return the variable with minimum \( break \), breaking ties by preferring the one with the greatest \( score(x) + age(x)/\beta \);
8. else if \( \exists x \) such that \( score(x) > t \cdot \overline{\gamma} \)
9. return such a variable with the greatest \( score \), breaking ties in favor of the oldest one;
10. else return the oldest variable in \( c \);

**Clause weighting for 3SAT:** When the clause-to-variable ratio is not more than 4.23, the algorithm does not update clause weights. Otherwise, the algorithm updates clause weights using the SWT scheme (Smoothed Weighting based on Threshold) [6]: clause weights of all unsatisfied clauses are increased by one; if the averaged weight \( \overline{\gamma} \) exceeds a threshold \( \gamma \), all clause weights are smoothed as \( w(c_i) := \gamma \cdot w(c_i) + [(1 - \gamma) \cdot \overline{\gamma}] \).

B. PickVar Heuristic for \( k \)-SAT with \( k > 3 \)

The pickVar heuristic for solving \( k \)-SAT instances with \( k > 3 \) very simple, as described in Algorithm 2.

**Algorithm 2: pickVar-heuristic for \( k \)-SAT with \( k > 3 \)**

1. if \( \exists \) conf\_changed variables with \( score(x) > 0 \) or \( score(x) = 0 \& score_2(x) > st \)
2. then
3. return such a variable with the greatest value of \( score(x) + score_2(x)/\beta + age(x)/\theta_1 \);
4. update\_clause\_weights();
5. pick a random unsatisfied clause \( c \);
6. return a variable from \( c \) with the greatest value of \( score(x) + score_2(x)/\beta + age(x)/\theta_2 \);
Clause Weighting for $k$-SAT with $k > 3$: We adopt a clause weighting scheme similar to PAWS [12]. With probability $sp$, for each satisfied clauses whose weight is bigger than 1, decrease the weight by 1. Otherwise, clause weights of all unsatisfied clauses are increased by 1.

III. Main Parameters

For 3-SAT, CCgscore has 4 parameters: $wp$, $t$, $\rho$, and $\gamma$, where the last two are for clause weighting. These parameters are set as follows according to clause-to-variable ratio $r$. When $r \leq 4.225$ of large size: $t = +\infty$, and $wp$ is set to 0.51 for $r \leq 4.2$, 0.5 for $r \in (4.2, 4.21)$, 0.495 for $r \in (4.21, 4.22)$, and 0.492 otherwise. When $r > 4.225$: $wp = 0$, $t = 0.6$, $\rho = 0.3$ and $\gamma = 200 + \frac{1}{0.6}\frac{1}{1 - (1 - \frac{1}{0.492})^4}$. For $k$-SAT with $k > 3$, CCgscore has 5 parameters: $st$, $\beta$, $\theta_1$, $\theta_2$, $sp$, and they are set as follows. For all $k$-SAT, $\beta=13-k$.

4SAT: $st = 27$, $\theta_2 = 1500$;
$\theta_1$ is set to 300000 for $r \leq 9.2$, 500000 for $r \in (9.2, 9.8)$, 100000 otherwise; $sp$ is set to 0.6 + $(r - 9) \cdot 0.1$ for $r < 9.8$ and 0.7 otherwise.
5SAT: $st$ is set to 5 for $r \leq 20$, 8 for $r \in (20, 21.1]$ and 32 otherwise;
$\theta_1$ is set to 100000 for $r \leq 20$, 300000 for $r \in (20, 20.4]$, 40000 for $r \in (20.4, 20.9]$ and 10000 otherwise;
$\theta_2$ is set to 2000 for $r \leq 20.9$, and 1500 otherwise; $sp$ is set to 0.62 + $(r - 20) \cdot 0.15$ for $r \leq 20.5$, and 0.74 + $(r - 20.6) \cdot 0.15$ otherwise.
6-SAT: for $r < 42.5$: $st=7$, $\theta_1=20000$, $\theta_2=2000$, otherwise, $st=10$, $\theta_1=8000$, $\theta_2=1400$; $sp = 0.88$ for $r < 40.5$ and 0.9 otherwise.
7-SAT: $st=20$ for $r < 85.2$ and 30 otherwise, $\theta_1=6000$, $\theta_2=2000$, $sp=0.92$.

IV. Implementation Details

CCgscore is implemented in programming language C++, and is developed on top of CScoreSAT. It is compiled by g++ with the `-O2` option.

V. SAT Competition 2014 Specifics

The CCgscore solver is submitted to “Sequential, Random SAT” and “Parallel, Random SAT” tracks, SAT Competition 2014. The command line of CCgscore for SAT Competition 2014 is described as follows.

```
./CCgscore <instance>
```

Note that in order to serve evaluations with different random seeds, CCgscore can also accept a random seed and runs with the seed.

ACKNOWLEDGMENT

Shaowei Cai would like to thank his wife for all the love and support, and the greatest love from God which supports him all the way.

REFERENCES

CLAS – A Parallel SAT Solver that Combines CDCL, Look-Ahead and SLS Search Strategies

Adrian Balint
Universität Ulm, Germany

Davide Lanti
Free University of Bolzano, Italy

Ahmed Irfan
Fondazione Bruno Kessler, Italy
University of Trento, Italy

Norbert Manthey
TU Dresden, Germany

Abstract—CLAS is a parallel solver that executes the SLS solver SPARROW in parallel to the parallel search space partitioning SAT solver PCASSO. Since PCASSO runs a CDCL solver in parallel to its search space partitioning, this solver represents a portfolio of an SLS solver, a CDCL solver, and a look-ahead based search space partitioning solver. Yet, no information is exchanged between SPARROW and PCASSO.

I. INTRODUCTION

Portfolio SAT solvers are a robust way to solve the diverse formulas that arise from the huge range of applications of SAT, and the crafted instances. In CLAS, three solving approaches are combined: CDCL, look ahead and SLS. The solver SPARROW+CP3 [1] showed a good performance on hard combinatorial benchmarks in the SAT Competition 2013, and PCASSO [2] is a very robust parallel search space partitioning solver. Hence, we combine the two solving approaches in parallel.

II. MAIN TECHNIQUES

SPARROW+CP3 uses the same configuration as the submitted sequential solver [3]. Before SPARROW is executed, COPROCESSOR is used to simplify the formula. One core of the CPU is reserved for SPARROW+CP3.

The remaining 11 cores are used for PCASSO, which also uses COPROCESSOR to simplify the input formula. The simplification techniques are the same as for the SAT solver RISS as submitted to the sequential tracks [4].

III. IMPLEMENTATION DETAILS

The two solvers are executed in parallel by a Python script. SPARROW is implemented in C. PCASSO and COPROCESSOR are implemented in C++. All solvers have been compiled with the GCC C-compiler as 64-bit binaries.

IV. AVAILABILITY

The source code of CLAS is available at tools.computational-logic.org for research purposes.

ACKNOWLEDGMENT

The authors would like to thank Armin Biere for many helpful discussions on formula simplification and the BWGrid [5] project for providing computational resources to tune COPROCESSOR. This project was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 3029-1. Finally, the authors would like to thank the ZIH of TU Dresden for providing the computational resources to develop, test and evaluate CLAS and PCASSO.

REFERENCES

Abstract—We describe some details about the SLS solver CPSparrow, which uses the preprocessor Coprocessor[1] and the SLS solver Sparrow [3] with a new restart policy.

I. INTRODUCTION

The solver CPSparrow is very similar to the solver Sparrow+CP3 [2], which participated at the SAT Competition 2013.

II. MAIN TECHNIQUES

Before solving the instance with the Sparrow SLS solver, the instance is being simplified with Coprocessor. The parameters of Coprocessor are kept identical to Sparrow+CP3 [2]. The simplified formula is then solved with Sparrow, which uses a new restart schedule.

Restarts are performed according to the Luby sequence, while steps are being measured in terms of the number of flips. The Luby base is set to $2^{18}$ flips. In every period the solver starts with a new $sp$ parameter (smoothing probability parameter) according to the following sequence: $(0.0, 0.1, 0.6, 0.1, 0.9, 0.6, 0.8, 0.6, 0)$. The assignment though, is not generated from scratch, but the old one is kept and only the weights of the solver are being reset.

III. FURTHER DETAILS

Sparrow is implemented in C and uses a new XOR implementation scheme for the flip procedure described in detail in [4]. The solver is submitted to the sequential Hard-Combinatorial SAT track. The compile flags for Sparrow are: -Wall -Wextra -static -O3 -funroll-loops -fexpensive-optimizations.

ACKNOWLEDGMENT

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

REFERENCES

[2] Balint, A., Manthey, N.: Sparrow+CP3 and SparrowToRiss In the proceedings of the SAT Competition 2013, Solver and Benchmark Description
I. INTRODUCTION

This paper presents the conflict-driven clause-learning SAT solver CryptoMiniSat v4 (cmsat4). cmsat4 aims to be a modern 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 forl, the previous year’s submission, are explained. Please refer to the previous years’ description for details.

A. Cleaner code

The code has been significantly cleaned up. In particular, it has been refactored to use more descriptive names, smaller functions and uses C++11 constructs that aid in simplifying code.

B. Better data gathering

More data is gathered into an SQL database that allows for interactive display of solving parameters. It also allows for later analysis of the solving, spotting e.g. that certain simplification steps take too long. Every simplification step is recorded and many important factors about clauses, clause cleaning, propagation and conflict analysis are dumped to the database.

C. Bounded variable addition

As per [1] variables are added to simplify the formula. CryptoMiniSat allows for not only 1-literal diff as per the research paper, but also 2-literal diffs. In terms of the algorithm in the research paper this difference introduces almost no change, though makes the implementation somewhat more elaborate.

D. Tuned time-outs

Thanks to the SQL-based query functionality, time-outs could be queried easily and checked. This allowed for fine-tuning of time-outs for weird problems.

E. Multi-threading

An experimental multi-threading system has been added. It only exchanges unit and binary lemmas. The system works even in case of library usage: it cleanly aborts the other threads even if the other threads are solving subcomponents with subsolvers.

F. Better stability

The API has been cleaned up and connected to a number of fuzzers, including e.g. a wrapper for python and a python-based test-suite. This allowed for more rigorous testing to be carried out.

ACKNOWLEDGEMENTS

The author would like to thank in no particular order Horst Samulowitz, Marius T. Lindauer, Martin Maurer, Martin Albrecht, Vegard Nossum, Valentin Mayer-Eichberger, George Katsirelos, Karsten Nohl, Luca Melette, Marijn Heule, Vijay Ganesh, Trevor Hansen and Robert Aston for their help.

REFERENCES

This paper presents a version of CryptoMiniSat v4 (cmsat4) that has had its parameters tuned for the SAT Competition 2014 by Marius Lindauer. No other aspect of the solver has been altered. The original solver’s description is can be found in the competition booklet.

Algorithm Configurators, such as SMAC [1], enable algorithm developers and also users to find automatically well performing configurations of their solver for a specific set of problem instances. Since cmsat4 exposes 82 performance relevant parameters, manual configuration is not feasible. However, SMAC proofed empirically already several times that it can handle such huge configuration spaces.

As instances, we used SAT industrial instances from the last four SAT Competitions and randomly splitted them equally in a training and test set. We used SMAC 2.06 on the training instances with 8 independent runs, a runtime cutoff of 1000 seconds and 400 000 seconds configuration time. To ensure that the best configuration on the training set does not suffer from over-tuning, we assessed its performance on the validation set.

```
--ltclean=0.5 --burst=300 --probemaxm=1600
--blkrestmulip=1.4 --blkrest=1 --printsol=0 --gluehist=100
--sccperc=0.0833 --elimcoststrategy=0 --presimp=0
--viviffastmaxm=400 --calcpolar1st=1 --eratio=0.0554
--lockuip=500 --viviflongmaxm=19 --locktop=0 --incclean=1.1
--clbtwimp=2 --calcreach=1 --flippolf=0 --moreminimbin=438
--cachecutoff=2291 --startclean=10000 --calcpolare1=1
--cleanconflmult=1 --moreminincache=87 --cachesize=2048
--freq=0.0079 --bva2lit=1 --blkrestlen=5000 --morebump=1
--dompickf=219 --perfmult=0.0 --occredmax=200
```

REFERENCES
CSCCSat2014 in SAT Competition 2014

Chuan Luo
School of EECS,
Peking University,
Beijing, China

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

Wei Wu
School of EECS,
Peking University,
Beijing, China

Kaile Su
IIS, Griffith University,
Brisbane, Australia

Abstract—This document describes the SAT solver “CSCCSat2014”, which is based on the local search framework.

I. INTRODUCTION

In 2011, a novel diversification strategy called configuration checking (CC) was proposed for handing the cycling problem [1], and resulted in several state-of-the-art stochastic local search (SLS) algorithms [1], [2], [3] for the minimum vertex cover (MVC) problem. According to its generality, the CC strategy has also successfully applied in the Boolean satisfiability (SAT) problem. The definition of configuration in previous CC strategies for SAT are all based on neighboring variables [4], [5], [6], [7].

Compared to the neighboring variables based configuration checking (NVCC) strategy, recently we proposed an alternative CC strategy, which focuses on clause states and thus is called the clause states based configuration checking (CSCC) strategy [8]. The CSCC strategy has led to several efficient SLS algorithms for SAT, such as FrwCB [9] and DCCASat [10].

Based on FrwCB and DCCASat algorithms, we design a new SLS solver called CSCCSat2014, which calls either FrwCB2014 (an improved version of FrwCB) or DCCASat2014 (an alternative version of DCCASat, which only utilizes the algorithmic settings for random instances) for solving different SAT instances.

II. MAIN TECHNIQUES

The CSCCSat2014 solver is a combination of two SLS solvers FrwCB2014 and DCCASat2014. Compared to original version of FrwCB described in the literature [8], when dealing random \( k \)-SAT instances with \( k > 4 \), FrwCB2014 utilizes the linear make function [11] to break ties. DCCASat2014 is an alternative version of DCCASat, and it only utilizes the algorithmic settings for random instances (described in the literature [10]).

The main procedures of CSCCSat2014 can be described as follows. For solving an SAT instance, CSCCSat2014 first decide the type of this instance. Then based on the properties of the instance, CSCCSat2014 calls either FrwCB2014 or DCCASat2014 to solve the instance.

III. MAIN PARAMETERS

For FrwCB2014 on solving random \( k \)-SAT instances with \( k > 4 \), the parameters of the linear make function [11] can be described as follows: \( w_1 = 3 \) and \( w_2 = 2 \). For FrwCB2014 on solving random \( k \)-SAT instances, \( p \) is set to 0.6 for random 3-SAT, 0.65 for random 4-SAT, 0.58 for random 5-SAT, 0.69 for random 6-SAT, and 0.76 for random 7-SAT.

The parameters of DCCASat2014 on solving random \( k \)-SAT instances can be found in the literature [10].

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

We use \( r \) to denote the clause-to-variable ratio of an SAT instance. For random 3-SAT with \( r \leq 4.24 \), CSCCSat2014 calls FrwCB2014; for random 3-SAT with \( r > 4.24 \), CSCCSat2014 calls DCCASat2014. For random 4-SAT with \( r \leq 9.35 \), CSCCSat2014 calls FrwCB2014; for random 4-SAT with \( r > 9.35 \), CSCCSat2014 calls DCCASat2014. For random 5-SAT with \( r \leq 20.1 \), CSCCSat2014 calls FrwCB2014; for random 5-SAT with \( r > 20.1 \), CSCCSat2014 calls DCCASat2014. For random 6-SAT with \( r \leq 41.2 \), CSCCSat2014 calls FrwCB2014; for random 6-SAT with \( r > 41.2 \), CSCCSat2014 calls DCCASat2014. For random 7-SAT with \( r \leq 80 \), CSCCSat2014 calls FrwCB2014; for random 7-SAT with \( r > 80 \), CSCCSat2014 calls DCCASat2014.

V. IMPLEMENTATION DETAILS

CSCCSat2014 is implemented in programming language C/C++, and is developed on the basis of FrwCB2014 and DCCASat2014.

VI. SAT COMPETITION 2014 SPECIFICS

The CSCCSat2014 solver is submitted to Random SAT track, SAT Competition 2014. The command line of CSCCSat2014 is described as follows.

```bash
./CSCCSat2014 <instance> <seed>
```

VII. AVAILABILITY

The CSCCSat2014 solver is open source and publicly available for only research purposes.

REFERENCES


DCCASat+march_rw in SAT Competition 2014

Chuan Luo
School of EECS,
Peking University,
Beijing, China

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

Wei Wu
School of EECS,
Peking University,
Beijing, China

Kaile Su
IIS, Griffith University,
Brisbane, Australia

Abstract—This document describes the SAT solver “DCCASat+march rw”, which is a combination of a local search solver and a complete solver.

I. INTRODUCTION

Recently, we proposed a diversification strategy for local search algorithms, which is called configuration checking (CC). The CC strategy is first applied to improving the performance of local search solvers for the minimum vertex cover (MVC) problem [1], [2], [3]. According to its simplicity and effectiveness, the CC strategy has been successfully used to improve stochastic local search (SLS) algorithms for satisfiability (SAT) [4], [5], [6], [7], [8], [9], [10]. The DCCASat solver [10] shows good performance on random $k$-SAT instances.

As SLS algorithms are incomplete that they cannot prove an instance to be unsatisfiable, we combine an SLS solver DCCASat2014 (an alternative version of DCCASat, which only utilizes the algorithmic settings for random instances) with an efficient complete solver called march_rw [11], [12], and design a new solver for SAT dubbed DCCASat+march_rw.

II. MAIN TECHNIQUES

The DCCASat+march_rw solver is a combination of the SLS solver DCCASat2014 and the complete solver march_rw.

The main procedures of DCCASat+march_rw can be described as follows. For solving an SAT instance, the DCCASat+march_rw solver first utilizes the SLS solver DCCASat2014 to solve the instance with a cutoff time of $t$ CPU seconds. If the instance is solved by DCCASat2014 within $t$ CPU seconds, then DCCASat+march_rw reports the solution which is found by DCCASat2014. Otherwise, DCCASat+march_rw activates the complete solver march_rw to solve the instance, and reports the solution if the instance is solved by march_rw within the remaining time.

III. MAIN PARAMETERS

We use $total$ to denote the total cutoff time for the whole DCCASat+march_rw solver, and use $r$ to denote the clause-to-variable ratio of an SAT instance. The parameter setting of $t$ for DCCASat2014 in the DCCASat+march_rw solver can be found in Table I.

The parameters of DCCASat2014 in DCCASat+march_rw on solving random $k$-SAT instances can be found in the literature [10].

<table>
<thead>
<tr>
<th>Instance Class</th>
<th>Ratio ($r$)</th>
<th>$t$ for DCCASat2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-SAT</td>
<td>$r &lt; 4.24$</td>
<td>$t = 0.9 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$4.24 \leq r \leq 4.294$</td>
<td>$t = 0.5 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$r &gt; 4.294$</td>
<td>$t = 0.1 \cdot total$</td>
</tr>
<tr>
<td>4-SAT</td>
<td>$r &lt; 9.8$</td>
<td>$t = 0.9 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$9.8 \leq r \leq 10.062$</td>
<td>$t = 0.5 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$r &gt; 10.062$</td>
<td>$t = 0.1 \cdot total$</td>
</tr>
<tr>
<td>5-SAT</td>
<td>$r &lt; 20.5$</td>
<td>$t = 0.9 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$20.5 \leq r \leq 21.734$</td>
<td>$t = 0.5 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$r &gt; 21.734$</td>
<td>$t = 0.1 \cdot total$</td>
</tr>
<tr>
<td>6-SAT</td>
<td>$r &lt; 42$</td>
<td>$t = 0.9 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$42 \leq r \leq 44.74$</td>
<td>$t = 0.5 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$r &gt; 44.74$</td>
<td>$t = 0.1 \cdot total$</td>
</tr>
<tr>
<td>7-SAT</td>
<td>$r &lt; 86$</td>
<td>$t = 0.9 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$86 \leq r \leq 89.58$</td>
<td>$t = 0.5 \cdot total$</td>
</tr>
<tr>
<td></td>
<td>$r &gt; 89.58$</td>
<td>$t = 0.1 \cdot total$</td>
</tr>
</tbody>
</table>

The parameters used in the current version of the march_rw solver in DCCASat+march_rw are the same as the ones in the version (march_rw) which is submitted to SAT Competition 2011 [12].

IV. IMPLEMENTATION DETAILS

DCCASat+march_rw is implemented in programming language C/C++, and is developed on the basis of DCCASat2014 and march_rw.

V. SAT COMPETITION 2014 SPECIFICS

The DCCASat+march_rw solver is submitted to Random SAT+UNSAT track, SAT Competition 2014. The command line of DCCASat+march_rw is described as follows.

```
./DCCASat+march_rw <instance> <seed> <cutoff_time>
```

In SAT Competition 2014, the parameter $cutoff \_time$ is set to 5000 seconds.

VI. AVAILABILITY

The DCCASat+march_rw solver is open source and publicly available for only research purposes.
REFERENCES


Abstract—This document describes the SAT solver Dimetheus as submitted to the SAT Competition 2014. It will first comment on a few basic ideas used in this version, e.g. the application of the SLS search paradigm along with Message Passing. Then, it will elaborate on the parameters used by the solver that are relevant for the competition. And finally, we will comment on a few implementation details.

I. INTRODUCTION

Let us first note, that this article is an excerpt of a more substantial overview for this solver which can be found in [1]. We will focus on the most important aspects of the solver that are most relevant for the SAT Competition 2014.

We begin in Section II by giving an overview of the applied search paradigms and algorithms. We will furthermore cite various papers that should be read in case the reader wants to familiarize himself more with the solver or the source-code.

We then continue in section III by explaining the main parameters used during the competition. This is followed by an overview of some implementation details in Section IV. Here, we will briefly cover the programming language and the compiler compatibility.

And finally, the article is concluded by giving some acknowledgements.

II. MAIN TECHNIQUES

The functionality that the Dimetheus solver employs is rather substantial and separated into various phases of solving (see Figure 1a) which themselves can apply strategies to achieve their respective tasks (see Figure 1b). We cannot possibly explain all the details here and must refer the reader to [1]. However, w.r.t. the SAT Competition 2014, we can be roughly separate the applied functionality into three categories.

First, the solver performs a simple preprocessing ahead of solving the formula. This preprocessing consists of unit propagation, pure literal elimination, subsumption elimination, and failed literal elimination (as described, e.g. in [2]).

Second, it applies Message Passing (MP). The application of MP is commonly done in order to provide biases, which are estimators for the marginals of variables in solutions. Such biases are then used to perform Message Passing Inspired Decimation (MID) as described in [3]. On top of that, the new MP algorithm described in [4] can be used to perform literal occurrence weight learning (LOWL). In contrast to MID, the LOWL approach does not assign variables. It merely influences how MP computes biases. The Dimetheus solver can dynamically decide which approach (either MID or LOWL) it uses in order to solve a formula. The decision which of
the techniques is used is done based on a random forest classification of the formula.

Third, should search be required because MID or LOWL can not determine a satisfying assignment by themselves, the solver will apply SLS. The SLS approach of the solver follows the ProbsAT approach [5].

Due to the incomplete search provided by the SLS module, the Dimetheus solver is incomplete in the way it is executed in the SAT Competition 2014. The reader should also note, that there is a large portion of functionality that is implemented (e.g. a CDCL module), but that is not used during the execution of the solver in the competition. We can thus understand it as a sequential core-engine solver.

III. MAIN PARAMETERS

The parameter settings used during the competition are as follows.

- `--formula STRING` Tells the solver which formula to solve. The STRING gives the full relative path to a formula in DIMACS CNF input format.
- `--seed INT` Tells the solver which seed it is supposed to use in order to initialize its pseudo-random number generator.
- `--classifyInputDomain INT` Tells the solver that it should try to classify the given formula based on the assumption that the formula is uniform random.

An example statement for calling the solver on the command line is as follows (the ordering of the parameters does not matter). `./dimetheus -formula the.cnf --seed 101 --classifyInputDomain 10`. Executing the solver that way will result in a classification of the given formula according to a random forest trained on various types of such formulas. The random forest has been trained on 30 properties of such formulas (e.g. the number of variables, the number of clauses, the literal occurrence distribution etc.). The solver will then know w.h.p. which type of formula it must solve. It will then internally adapt various parameters for MP (i.e. $p, \sigma$ as explained in [3]) and SLS (e.g. $c_1$ as explained in [5]). The details on what these parameters control are rather technical, and we will therefore not explain them here. The interested reader is encouraged to look at the cited papers. However, the reader should know that the parameter configurations that can be applied by the solver have been derived by parameter tuning using the EDACC/AAC system [6].

Finally, the reader should note that calling the solver with the `--h` flag gives a substantial help on how to use it.

IV. IMPLEMENTATION DETAILS

The Dimetheus solver has been implemented from scratch in the programming language C following the C99 standard. The solver can be compiled with the GNU GCC compiled versions 4.4. and above. The compiler used to test the solver was GNU GCC in version 4.6. The solver is designed to work on both 32-bit and 64-bit systems, and supports all common operating systems (e.g. Linux, Windows, Mac) provided that they employ the GNU GCC compiler with the system specific headers.

V. SAT COMPETITION 2014 SPECIFICS

Let us first note, that the Dimetheus solver is submitted in version 2.100 to the uniform random SAT track of the SAT Competition 2014.

The solver is open source and publically available. The license under which the sources are available is Creative Commons Attribution non-commercial no-derivs (CCBYNCND) 3.0.

The latest (tested) sources of the Dimetheus solver can be downloaded from https://www.gableske.net/downloads/dimetheus_latest.tar.gz

A copy of the CCBYNCND 3.0 license under which the solver is published can be downloaded from http://creativecommons.org/licenses/by-nc-nd/3.0/

ACKNOWLEDGMENT

The author would like to thank Jacobo Torán, Dominikus Krüger, Daniel Dipold, and Adrian Balint for fruitful discussions and technical help.

REFERENCES

Glucose in the SAT 2014 Competition

Gilles Audemard
Univ. Lille-Nord de France
CRIL/CNRS UMR8188
Lens, F-62307
audemard@cril.fr

Laurent Simon
University of Bordeaux,
Labri, UMR 5800,
F-33405 Talence Cedex,
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. This short system description is the 2014 SAT Competition companion paper for Glucose.

I. INTRODUCTION

In the design of efficient CDCL-based SAT solvers [1], [2], a lot of effort has been put in efficient Boolean Constraint Propagation (BCP), learning mechanisms, and branching heuristics, their three main components. In [3], a new simple measurement of learnt clause usefulness was introduced, called LBD. This measure was no more based on past clauses activities. It was proved so efficient that, since 2009, Glucose and its updated versions was always one of the award winning SAT solvers in the corresponding competitive events. This year, we proposed a minor revision of Glucose 2.2 (used in the SAT 2012 Challenge, see [4] for details) and increased its aggressive database cleanup strategy once again. The solver is particularly well suited for UNSAT problems but can, thanks to the techniques developed in the 2.2 version, still be competitive on SAT problems too.

This year we submit the same version as 2013. This is due to two main reasons. First, this year we focused on Glucose-Syrup our parallel version of Glucose. Second, we encountered some problems with the output buffer when dumping on the standard output the DRUP certificate for the certified unsatisfiable track and decided to revert to previous version.

II. MAIN TECHNIQUES

During search, each decision is often followed by a large number of unit propagations. We called the set of all literals of the same level a “blocks” of literals. Intuitively, at the semantic level, there is a chance that they are linked with each other by direct dependencies. The underlying idea developed in [3] is that a good learning schema should add explicit links between independent blocks of propagated (or decision) literals. If the solver stays in the same search space, such a clause will probably help reducing the number of next decision levels in the remaining computation. Staying in the same search space is one of the recent behaviors of CDCL solvers, due to phase-saving [5] and rapid restarts.

Let us just recall what is the Literals Blocks Distance (LBD). Given a clause \( C \), and a partition of its literals into \( n \) subsets according to the current assignment, s.t. literals are partitioned w.r.t their decision level. The LBD of \( C \) is exactly \( n \).

From a practical point of view, we compute and store the LBD score of each learnt clause when it is produced. Intuitively, it is easy to understand the importance of learnt clauses of LBD 2: they only contain one variable of the last decision level (they are FUIP), and, later, this variable will be “glued” with the block of literals propagated above, no matter the size of the clause. We suspect all those clauses to be very important during search, and we give them a special name: “Glue Clauses” (giving the name “glucose”).

The LBD measure can be easily re-computed on the fly when the clause is used during unit propagation. We keep here the strategy used in Glucose 1.0: we change the LBD value of a clause only if the new value becomes smaller. However, in the 2.3 version this update is only performed during conflict analysis, and not during propagation.

III. NOVELTIES OF GLUCOSE 2.3

Before Glucose 1.0, the state of the art was to let the clause database size follow a geometric progression (with a small common ratio of 1.1 for instance in Minisat). Each time the limit is reached, the solver deleted at most half of the clauses, depending on their score (note that binary and glue clauses are never deleted). In Glucose 1.0, we already chose a very slow increasing strategy. In this new version, we perform a more accurate management of learnt clauses.

A. Dynamic threshold, revisited

As a basis, we used the 2.2 version of the cleaning process: Every \( 4000 + 300 \times x \), we removed at most half of the learnt clause database \( x \), which this is much more aggressive than the version 1.0, i.e. 20000 + 500 x). Of course, binary clauses, glue clauses and locked clauses are always kept. A locked clause is (1) used as a reason for unit propagation in the current subtree or (2) locked (see [4]). However, in the 2.3 version, thanks to a more focused update of the interesting clauses (propagated clauses LBD scores that are not seen in any conflict are not updated), we were able to use the following more aggressive strategy: the cleaning process is fired every \( 2000 + 300 \times x \). This gives us clause database cleaning after 2000, 4300, 6900, 9800, \ldots etc, instead of 4000, 8300, 12900, 17800, \ldots used in Glucose 2.2. This choice is also related to the new technique described in the next subsection.
The dynamic nature of the threshold used to keep more or less clauses at each cleaning process is kept intact (see version 2.2 description).

B. Starts with a wider search for proofs

The \texttt{var\_decay} constant is 0.95 in Minisat, which was a very good compromise for most of the benchmarks \cite{6}. However, thanks to the work of \cite{7}, it was shown that fixing it to the same value was not always a good choice. Thus, we proposed to use it like some kind of temperature, starting from 0.8 and increasing it by 0.01 every 5000 conflicts until it reaches the 0.95 value (after 75000 conflicts). This idea arose during one of the fruitful discussions we had with George Katsirelos, Ashish Sabharwal and Horst Samulowitz and thus the credits for this idea are clearly shared with them. Adding this rule allowed us to make a small step in Glucose performances (3 additional problems solved on the previous SAT Challenge set of problems) but it may open the door for further improvements.

One may notice that, after a few ten thousands conflicts, the more aggressive strategy used for clause database cleanings (using the constant 2000 instead of 4000) tends to disappear, because the strategy will be mostly dominated by the $300 \times x$ part of the increasing variable. The meaning of that is that we want to quickly drop “bad” (useless) clauses generated when \texttt{var\_decay} was still not properly set.

C. Other embedded techniques

Since the first versions of Glucose, we used the stand alone simplifier "Satelite". In the 2.3 version, we used the built-in "Simp" class of Minisat that simplify the formula in place.

The laziness of the LBD update mechanism is inspired by the study of CDCL solvers proposed in the source code of the work of Long Guo in www.cril.fr/~guo.

IV. SUPPORT FOR UNSAT PROOF CHECKING

Thanks to the effort of Marijn Heule, who implemented the support for DRUP (Delete Reverse Unit Propagation) proof checker into Minisat and Glucose 2.2, it was trivial to port his code into Glucose 2.3. Currently, there are two distinct versions of Glucose (with or without DRUP support) but the next release of Glucose will contain the support for UNSAT proof checking as an argument. More information on the work of Marijn Heule and the DRUP file format can be found at www.cs.utexas.edu/~marijn/drup.

V. MAIN PARAMETERS

Given the fact that auto-tuning of SAT solvers is a classical technique for improving the performances and propose a more scientific approach to fix the parameters, most of the parameters are accessible via command line options. See the description above for the specific parameters we used for Glucose 2.3.

The main objective of Glucose was to target UNSAT Applications problems. However, the blocking-restart strategy introduced in Glucose 2.2 allows to keep a good score on SAT problems too (see \cite{4} for this).

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. Glucose is targeting Application problems.

The main page of Glucose is http://www.labri.fr/perso/lsimon/glucose/.

REFERENCES

**Glucose-Syrup: a parallel version of Glucose with Lazy Clause Exchange**

Gilles Audemard  
Univ. Lille-Nord de France  
CRIL/CNRS UMR8188  
Lens, F-62307  
audemard@cril.fr

Laurent Simon  
University of Bordeaux,  
Labri, UMR 5800, F-33405 Talence Cedex,  
lsimon@labri.fr

---

**Abstract**—In this system description short paper, we briefly describe the Glucose-Syrup solver as it was submitted to the SAT 2014 contest. This solver is a parallel version of Glucose, that uses a new clause exchange policy. Clauses are not sent after conflict analysis but rather identified to be sent when seen at least two times during conflict analysis. Moreover, when a solver receives a clause, it is put “in probation”, i.e., in a 1-watched literal scheme, and thus will not interfere with the solver search strategy until the clauses is found to be empty.

**I. INTRODUCTION**

Parallelizing SAT solvers is an important challenge for the whole SAT community. One of the hardest task for a multi-core SAT solver is identifying which clauses to send and which clauses to get. In Glucose-Syrup, we propose to take an easy (and lazy) technique for that. The techniques developed in this solver are detailed in [1]. This paper only describes technical problems/decisions made for the contest.

**II. MAIN TECHNIQUES**

Glucose is based on Minisat [2] and thus, Glucose-Syrup is another parallel version of Minisat-like solvers. Instances are preprocessed with the native simplifier of Minisat.

**III. MAIN PARAMETERS**

The set of parameters of Glucose-Syrup are adjusted in a special class that aims at tuning each solver/core strategy given the number of cores it will use. Because of limited memory available for the contest, we decided to add a special behavior of Glucose-Syrup for the contest. A maximal amount of memory to be used can be specified, with a maximal number of cores to use. Internally speaking, Glucose-Syrup is reading the initial formula with only one core, simplify it using the satellite component proposed within Minisat. Then, the amount of memory used with one core is measured and the number of cores is chosen automatically. The desired number of cores is adjusted such that, when multiplied by the current memory consumption, it does not exceed 40% of the maximal allowed memory. Then, the mono-core solver used to parse and simplify the formula is cloned into the desired number of solvers. After this initialization phase, the solver is regularly watching the amount of used memory. If the limit is reached, then the clause database cleaning strategy blocks its increasing feature and the solver is then working under almost constant memory constraints.

These two features (simplification, memory bounds) were added in the last days and hours before the deadline. We had a number of unexpected bugs (in the generation of the SAT certificate after simplification) that forced us to send a version of Glucose-Syrup that was “in progress” and thus the code available on the competition web site was not cleaned properly. A lot of unstable options are still in the source code and we strongly encourage users to download the up to date version of Glucose-Syrup on the Glucose web pages given below.

**IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES**

The main point of Glucose-Syrup is the use of the 1-Watched literal scheme to detect empty clauses and then promote them to a 2-Watched literal scheme.

The solver could probably be improved by tuning the parameters of each core. This version was not tuned and was completely built with general consideration without any experimental evidences.

**V. SAT COMPETITION 2013 SPECIFICS**

This solver was submitted to the parallel track of the SAT 2014 competition. Due to the use of cloning techniques, it needs a C++ compiler compliant with C++11. It is compiled with the classical Minisat optimisation parameters (-O4, in 64 bits). The set of parameters are described at the command line with the classical --help argument.

**VI. AVAILABILITY**

The solver is available at [http://www.labri.fr/~lsimon/glucose](http://www.labri.fr/~lsimon/glucose). It is open source, with the same licence as Minisat.

**ACKNOWLEDGMENT**

The authors would like to thank Daniel Le Berre for fruitful discussions. A special thanks also to Marijn Heule for his patience during the too long debugging session of Glucose-Syrup when deployed on the competition cluster.
REFERENCES


GLUE MiniSAT 2.2.8

Hidetomo Nabeshima
University of Yamanashi, JAPAN
Koji Iwanuma
University of Yamanashi, JAPAN
Katsumi Inoue
National Institute of Informatics, JAPAN

Abstract—GLUE MiniSAT is a SAT solver based on MINISAT 2.2 and the LBD evaluation criteria of learned clauses. The main feature of the version 2.2.8 is the on-the-fly lazy simplification techniques which consists of various probing techniques, self-subsuming resolution, on-demand addition of binary resolvents, and clause minimization by binary resolvents. The base solver has features used in GLUCOSE 3.0 such as restart blocking, fluctuation strategy in the initial stage of search, additional maintaining of good learned clauses, etc.

I. INTRODUCTION

GLUE MiniSAT is a SAT solver based on MINISAT 2.2 [1] and the LBD evaluation criteria of learned clauses [2]. We have re-implemented the version 2.2.8 from MINISAT 2.2 to evaluate the performance of the on-the-fly lazy simplification techniques [3], which consists of the following techniques:

1) variable elimination and equivalent literal substitution based on probing techniques [4].
2) original clause and learned clause simplification by self-subsuming resolution with binary resolvents, and
3) on-demand addition of binary resolvents.

The computational cost of these techniques is negligibly small. Hence, these techniques are executed frequently throughout the process of modern conflict-driven clause learning (CDCL) solvers, that is, unit propagation, conflict analysis, removal of satisfied clauses, etc.

The base solver has features used in GLUCOSE 3.0 such as restart blocking [5], fluctuation strategy in the initial stage of search, additional maintaining of good learned clauses, etc.

II. MAIN TECHNIQUES

GLUE MiniSAT 2.2.8 has the on-the-fly lazy simplification techniques [3]. These techniques are defined as operations on binary resolvents, which are extracted from unit propagation process with almost no overhead [6], [7]. In GLUE MiniSAT, the number of binary resolvents to be preserved is restricted and they are maintained in a simple data structure. For each literal \( y \), we hold only one premise literal \( x \) such that \( \phi \models x \rightarrow y \), where \( \phi \) is a given formula. We represent a premise literal of \( y \) as \( \text{premise}[y] \) (that is, \( \phi \models \text{premise}[y] \rightarrow y \)). Initially, \( \text{premise}[y] = y \). In the unit propagation process, if \( y \) is propagated and it has a dominator [6], then the value of \( \text{premise}[y] \) is updated with the dominator. In our experiments, each entry of premise is updated approximately 1000 times on average for solving 1192 application instances of SAT 2009 and 2011 competitions and SAT Challenge 2012 within 1200 CPU seconds. The preserved premise literals are updated frequently during search. That is, we maintain a partial snapshot of binary resolvents which evolves during search. This variation of premise literals contributes to the realization of low cost simplification techniques which are executed on-the-fly.

We can execute probing techniques [4] with a constant time by using the premise literals. For example, the necessary assignment probing can be represented as follows: suppose that \( \phi \) is a formula and \( x, y \) are literals. If \( \phi \models x \rightarrow y \) and \( \phi \models \neg x \rightarrow y \), then \( \phi \models y \). This probing technique requires two premise literals of \( y \). We can get two premise literals of \( y \), that is, the old value of \( \text{premise}[y] \) before updating of it and the new value of it. We denote the old and new values as \( \text{oldpremise}_y \) and \( \text{newpremise}_y \), respectively. Then, we can execute the necessary assignment probing as follows: if \( \text{oldpremise}_y = \text{newpremise}_y \), then \( \phi \models y \). Other probing techniques can be executed in the same way [3].

GLUE MiniSAT executes these on-the-fly probing techniques when an entry of the array \( \text{premise} \) is changed.

We can simplify clauses based on binary self-subsumption by using the array \( \text{premise} \). Given a clause \( C \) and two literals \( x, y \in C \), we define that \( x \) is redundant by \( y \) in \( C \) if \( \text{premise}[y] = x \) or \( \text{premise}[\neg x] = \neg y \), since the resolvent of \( C \) and \( x \rightarrow y \) is \( C \setminus \{x\} \) and it subsumes \( C \). The redundant literals can be eliminated from a clause. Let \( C \) be a clause \( \{w_1, w_2, x_1, \ldots, x_n\} \), where \( w_1 \) and \( w_2 \) mean watched literals and \( x_i \) is an unwatched literal in two watched literal schema [8]. In GLUE MiniSAT, we check whether \( x_i \) is redundant by \( w_1 \) or \( w_2 \) to avoid the updating cost of the list of watched clauses (if \( w_i \) is eliminated, then we should update the list of watched clauses). This binary self-subsumption checking can be incorporated with the scanning-loop of literals in a clause, such as conflict analysis and removal of satisfied clauses.

Suppose that \( \phi \) is a formula and \( \alpha \) is an assignment. \( \text{UP}(\phi, \alpha) \) represents the assignment after the unit propagation process. Let be \( \text{premise}[y] = x \). This means that \( y \in \text{UP}(\phi, \{x\}) \), but the contrapositive \( \neg x \notin \text{UP}(\phi, \{\neg y\}) \) may not hold. This is because the binary resolvent \( x \rightarrow y \) does not exist as a clause in \( \phi \) explicitly. To enhance the unit propagation, for each literal \( p \in \text{UP}(\phi, \alpha) \), if \( \text{premise}[\neg p] \notin \text{UP}(\phi, \alpha) \), then the binary resolvent \( \text{premise}[\neg p] \rightarrow \neg p \) is added to \( \phi \) as a clause. This on-demand addition is executed after the unit propagation process.

Suppose that \( C = \{x_1, \ldots, x_n\} \). If there is a implication chain such that \( \neg x_i \rightarrow \cdots \rightarrow \neg x_j \) (\( i \neq j \)), then \( x_j \) can be eliminated from \( C \). This is a generalization of the above binary self-subsumption checking and a special case of minimization technique used in MINISAT [9].
executes this checking for each learned clause after conflict analysis. For each literal $x_j \in C$, GLUEMINISAT tries to construct a chain \( \cdots \rightarrow y_2 \rightarrow y_1 \rightarrow \neg x_j \) from the end, where $y_1 = \text{premise}[^\neg x_j]_j$ and $y_{k+1} = \text{premise}[y_k]_j \quad (k \geq 1)$.

1. This construction is continued until $y_k \equiv \neg x_i$, where $x_j \in C$. (successful case) or the current assignment of $y_k$ is not true (failed case). The latter condition is introduced as a heuristics to avoid generating a long chain.

### III. OTHER TECHNIQUES

GLUEMINISAT has features which are implemented in GLucose 3.0. The restart blocking [5] helps to catch a chance of making satisfying assignment. This strategy postpones restarting when the local trail size per a conflict is exceedingly greater than the global one.

The variable activity decay heuristics is one of parameters of VSIDS decision heuristics [10] used in MINISAT[1]. When the value of this parameter is small, the activity of recently unused variables decays quickly. This makes easy to move the search space, since the variable selection is not caught in the past activity. GLucose 3.0 increases this parameter by 0.01 from 0.8 until 0.95 whenever 5000 conflicts occur. In our experiments, this strategy is effective for satisfiable instances. GLUEMINISAT also uses this strategy.

GLucose 3.0 protects learned clauses from clause-deletion only once when the LBD value of those clauses decreases and are lower than a certain threshold. In the clause-deletion, basically it removes half of learned clauses in order of LBD values. But protected clauses survive only once. Furthermore, when good learned clauses whose LBD is less than or equal to 3 exist more than half, the limit of number of remained learned clauses is relaxed by adding a constant number. GLUEMINISAT follows this management strategy.

### IV. SAT COMPETITION 2014 SPECIFICS

GLUEMINISAT is submitted to Sequential, Application SAT+UNSAT track and Hard-combinatorial SAT+UNSAT track. The version 2.2.8 does not have a function to output an UNSAT proof. The previous version 2.2.7 which was submitted to SAT 2013 competition can output UNSAT proof when some simplification techniques are disabled.

### V. 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/.

### ACKNOWLEDGMENT

This research is supported in part by Grant-in-Aid for Scientific Research (No. 24300007) from Japan Society for the Promotion of Science and by Kayamori Foundation of Informational Science Advancement.

### REFERENCES


Glue_lgl_split and GlueSplit_clasp with a Split and Merging Strategy

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China

Abstract—In this document, we introduce briefly a new solving policy called split and merging. Using the new policy and the existing core solvers, we developed two SAT solvers, which are named Glue_lgl_split and glueSplit_clasp, respectively.

I. INTRODUCTION

Glue_lgl_split and glueSplit_clasp are a sequential CDCL solver combining two core solvers by a new technique. Glue_lgl_split is submitted to application SAT, SAT+UNSAT track and hard-combinatorial SAT, SAT+UNSAT track. GlueSplit_clasp is submitted to only hard-combinatorial SAT, SAT+UNSAT track.

II. A SPLIT AND Merging Technique

Glue_lgl_split and glueSplit_clasp are based on a split and merging technique, which is described below.

At first, we use a core solver to solve the original problem $F$ within a limited time. This yields an intermediate result $F'$. Let $x$ be a variable of $F'$. At second, we split $F'$ into two subproblems $F' \cup \{x\}$ and $F' \cup \{\neg x\}$, each of them is recursively solved. When the recursive depth reaches some constant value, say 10, we terminate the recursive calling. At this time, if both the sub-problems have not solved yet, we merge them into a new problem, and then continue to solve it by a core solver without the cutoff limit. Here is the pseudo-code of this solving strategy.

```
Algorithm SplitMergeSolve(F, level, abort)
if abort = 0 and level > 0 then cutoff ← ∞
else cutoff ← Φ(level)
(F, r) ← CoreSolve(F, cutoff)
if r=SAT or r=UNSAT or level = α then return (F, r)
var ← GetBranchVariable()
if level = 0 then abort ← 1
if r=SAT or r=UNSAT or level = α then return (F, r)
var ← GetBranchVariable()
if level = 0 then abort ← 1
if r=SAT or r=UNSAT or level = α then return (F, r)
var ← GetBranchVariable()
if level > β then
if r₀=UNKNOWN and r₁=UNKNOWN then r₀ ← CoreSolve(F₀, ∞)
if r₀=SAT then return SAT
if r₀=UNKNOWN then return (F₀, r₀)
F ← Merge(F₀, F₁, lit)
return SplitMergeSolve(F', level + 1, 0)
```

In the above algorithm, we use parameter abort to denote whether the solving process is aborted when the recursive depth reaches $\theta$. When abort=1, it will be aborted. Otherwise it will not. Initially, abort is set to 1. $Φ(level)$ is used to compute the cutoff value for different levels. It may be defined as follows.

$$Φ(level) = \begin{cases} 
600000 & level = 0 \\
100000 & level < 10 \\
50000 & level ≥ 10 
\end{cases}$$

Procedure GetBranchVariable is to select the most promising variable from the variables unassigned so far. This may use a lookahead ACE technique [1]. Parameter $β$ indicates at which level we adopt the merging strategy. It may be set to 2. When the recursive level is greater than $β$, even if now is an abort state, we do not abort and continue to solve the first subproblem until a deterministically solution is found. Otherwise, we adopt the merging strategy.

Procedure Merge is to combine two sub-formulae into one formula. Its basic idea is that if a clause $c$ is shared by two sub-formulae, it is put directly to the new formula. Otherwise, either $(c \lor v)$ or $(c \lor \neg v)$ is put to the new formula, where $v$ is the most promising variable. For example, given $F₀ = \{x₁ \lor x₂, x₃ \lor x₄, x₁ \lor x₃\}$, $F₁ = \{x₁ \lor x₂, x₃ \lor x₄, x₁ \lor x₃\}$ and $v = x₆$. The merging result is $F = \{x₁ \lor x₂, x₃ \lor x₄, x₃ \lor x₄ \land \neg x₆\}$. Here is the pseudo-code of this merging procedure.

```
Algorithm Merge(F₀, F₁, var)
if $F₀ = \emptyset$ then return $F₁$
if $F₁ = \emptyset$ then return $F₀$
F ← ∅
for each clause $c₀ ∈ F₀$ do
if $c₀ \in F₁$ then F ← F ∪ $\{c₀\}$
else F ← F ∪ $\{c₀ \lor \neg var\}$
for each clause $c₁ ∈ F₁$ do
if $c₁ \notin F₀$ then F ← F ∪ $\{c₁ \lor var\}$
return F
```

III. GLUE_LGL_SPLIT

Glue_lgl_split is built on the top of Glucose 2.3 [2] and Lingeling 587f [3]. Here, we made a slight modification on Glucose and Lingeling. We added a bit-encoding phase selection strategy [4] in them. Depending on the feature of instances, we select different solving strategies. For example, for very small instances, say #var < 3200, we use either Glucose or SplitMergeSolve. Before entering SplitMergeSolve, we do a
preprocess by Lingeling. That is, the input of SplitMergeSolve is a preprocessed instance. When the decision level < 10, we use Glucose as the core solver of SplitMergeSolve. Otherwise, we use Lingeling as its core solver. In addition to very small instances, for middle instances with the average LBD (Literal Block Distance) score [5] less than 26, and the average search height less than 150, we use SplitMergeSolve also. However, once SplitMergeSolve reaches an impasse, we cancel this solving strategy, then switch to Glucose. The criterion of an impasse is to test if the number of conflicts required to solve a sub-problem exceeds 1500000. If yes, it is regarded as an impasse. For large instances, we use Glucose. Here is the pseudo-code of glue_lgl_split.

Algorithm Glue_lgl_split(\mathcal{F})
\begin{align*}
& \text{if } \text{var}(\mathcal{F}) < 300 \text{ then } \mathcal{F} \leftarrow \text{Lingeling}(\mathcal{F}, 100000) \\
& \text{if } \text{LBD}(\mathcal{F}) < 20 \text{ then } \\
& \quad \mathcal{F} \leftarrow \text{Lingeling}(\mathcal{F}, 300000) \\
& \quad \text{if } \text{height}(\mathcal{F}) < 150 \text{ then return SplitMergeSolve}(\mathcal{F}) \\
& \text{return Glucose}(\mathcal{F})
\end{align*}

The second input parameter of each core solver above is cutoff, which means that when its number of conflicts reaches cutoff, it terminates and returns an intermediate result.

IV. GLUE_SPLIT_CLASP

The techniques used in GlueSplit_clasp include bit-encoding phase selection, split and merging, and interactive technique etc. It contains two core solvers: Glucose 2.3 [2] and Clasp 2.0-R4092 [6]. When the average LBD score of an instance is less than 20, and its average search height less is than 30, we use SplitMergeSolve to solve it. In the other cases, we solve it, using interactively Glucose and Clasp. However, for large instances, we invoke only Glucose. Here, at any decision level, the core solver of SplitMergeSolve is Glucose. Here is the pseudo-code of glueSplit_clasp.

Algorithm GlueSplit_clasp(\mathcal{F})
\begin{align*}
& \text{if } \text{var}(\mathcal{F}) > 20000 \text{ then return Glucose}(\mathcal{F}) \\
& \mathcal{F} \leftarrow \text{Glucose}(\mathcal{F}, 600000) \\
& \text{if } \text{LBD}(\mathcal{F}) < 20 \text{ and } \text{height}(\mathcal{F}) < 30 \text{ then } \\
& \quad \text{return SplitMergeSolve}(\mathcal{F}) \\
& \text{repeat} \\
& \quad (\mathcal{F}, r) \leftarrow \text{Glucose}(\mathcal{F}, 100000) \\
& \quad \text{if } r \neq \text{UNKNOWN} \text{ return } r \\
& \quad (\mathcal{F}, r) \leftarrow \text{Clasp}(\mathcal{F}, -1) \\
& \quad \text{if } r \neq \text{UNKNOWN} \text{ return } r \\
& \text{end repeat}
\end{align*}

The second input parameter of Clasp is set to -1. It means that when the number of free variables changes, Clasp terminates and returns an intermediate result.

REFERENCES

Yet another Local Search Solver and Lingeling and Friends Entering the SAT Competition 2014

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

Abstract—This paper serves as solver description for the SAT solvers Lingeling and its two parallel variants Treengeling and Plingeling, as well as for our new local search solver YalSAT entering the Competition 2014. For Lingeling and its variants we only list important differences to earlier version of these solvers as used in the SAT Competition 2013. For further information we refer to the solver description [1] of the SAT Competition 2013 or source code.

YalSAT

Recent SAT competitions witnessed a new generation of several efficient local search solvers including Sparrow [2] and ProbSAT [3], which surprisingly were able to solve some non-random instances. Fascinated by this success and the simplicity of the ProbSAT solver [3] we started to implement Yet Another Local Search SAT Solver (YalSAT). At its core it implements several variants of ProbSAT’s algorithm and recent extensions [4]. These variants are selected randomly at restarts, scheduled by a reluctant doubling scheme (Luby).

Beside initializing the assignment at each restart with a randomly picked assignment among previously saved best assignments within one restart round, it is also possible to assign all variables to false, to true or to a random value. At each restart the submitted version further varies the base \( cb_k \) for the exponential score distribution between either using the original base values of ProbSAT, where \( k \) is determined by the maximum length of the clause in the instance, or using the default of \( cb_k = 2 \). Then clause weights are either chosen to be the same for all clauses or are chosen as linear function depending on the clause length (with either larger clauses having larger weight or alternatively smaller clauses). The maximum weight is also picked randomly.

With this set-up we were able to solve a surprisingly large number of satisfiable crafted instances from last year’s competition. For uniform random instances YalSAT is supposed to work almost identical to last year’s version of ProbSAT.

Lingeling

This year’s version \( aqv \) of Lingeling is slightly improved in several ways. Compared to other solvers in the last competition, last year’s version \( aqw \) of Lingeling performed not well on certain unsatisfiable instances (particularly on miter instances). Our analysis showed that this was simply due to our agility based restart heuristic, which skipped too many restarts on these instances. As first measure we increased the agility limit slightly, but then, inspired by restart policies in Glucose [5], incorporated a new restart policy for skipping restarts, called "saturating". It compares the average LBD (glucose level) versus the average decision height. If the latter is relatively small (70% higher at most) restarts are skipped. Using this new restart heuristic allowed us to completely disable agility based restarts without much penalty on satisfiable instance, but with a substantial positive effect on unsatisfiable instances. As compromise the default version still uses agility based restarts, but we submitted an additional version "Lingeling (no agile)" without agility controlled restarts.

A new technique among the enabled techniques is called "tabula rasa". It monitors the number of remaining variables and clauses. If these numbers drop dramatically (below 25% or 50% respectively) all learned clauses are flushed. Finally, the covered clause elimination (CCE) inprocessing has substantially been improved by for instance trying to eliminate large clauses first, focusing on fast clause elimination procedures in early inprocessing rounds, like asymmetric tautology elimination, and then turning to ABCE and full CCE in later rounds.

Plingeling

Based on the average number of occurrences per literal and its standard deviation Plingeling tries to figure out whether the instance is actually a uniform random instance. If this seems to be the case it uses the integration of YalSAT into Lingeling and in essence runs a local search as sub-routine until completion. This is enabled for several worker threads, even all but one if clauses all have the same length.

The soft memory limit is set to one third of the physically available memory, while last year, using half the memory still resulted in last year’s version of Plingeling to occasionally run out of memory. This was due to excessive memory defragmentation when using many threads, e.g. resident set size being more than twice as large as the actual allocated memory.

As described above, last year’s version \( aqw \) of Lingeling turned out not to work well for several unsatisfiable instances due to skipping too many restarts. Thus the third worker thread disables agility based restart skipping.
TREENGELING

As discussed above, local search solvers can be quite competitive on a subset of crafted instances. Thus we integrated YalSAT into Lingeling, which by default is disabled but in Treengeling enabled in the already previously existing single parallel top-level worker thread. It is not run until completion though, but simply scheduled as another inprocessor, limited by the number of memory operations. In this set-up the local search sub-routine exports the best solution found to the CDCL part, by setting default phases accordingly.

Another important change in Treengeling was to use an internal cloning function after simplifying the top-level node for several rounds (10 rounds of inprocessing). This is based on the observation, that heavy preprocessing is useful for crafted instances, even though it requires some warm-up time.

After initial preprocessing the number of variables and clauses is usually substantially reduced. Thus much memory is wasted during cloning the full solver. This for instance includes the clauses on the reconstruction stack as well as the mapping of all original variables to those in reduced instances. Our current solution is to clone the initially simplified formula only internally. Then the root solver can not be reused, but will be needed to reconstruct a solution of the original instance. Further solver instances are cloned from this first internal clone. As a consequence, we have three solver instance after the first look-ahead: the root solver, the first internal clone for the first branch, and its dual for the second branch, both with a unit literal added. In last year’s version there would be only two solver instances at this point.

Finally, if the available cores are not utilized, Treengeling will split more eagerly, to produce more workers, by simply keeping the conflict limit for CDCL small.

LINGELING DRUPLIG

A substantial amount of work went into improving DRUP tracing for version azd of Lingeling, as submitted to the UNSAT tracks. In Lingeling many clauses are implicitly added, deleted or strengthened at various places. In order to find all these places, we implemented a library Druplig, which can be used to dump DRUP traces, but also, in debugging and testing mode, contains a forward online DRUP checker.

This allowed us to reuse our model-based testing and debugging frame work for the incremental API of Lingeling [6] for developing more complete DRUP support. This approach is in our experience much more effective in finding bugs and debugging them, compared to file based fuzzing and delta-debugging [7].

Compared to previous year’s DRUP tracing version, we now also trace clause deletion and further were able to enable many more inprocessing algorithms. Most of the probing based inprocessors can produce traces now. Equivalent literal reasoning is enabled too.

However, no form of extended resolution is traced, e.g. only plain DRUP, no DRAT proofs are supported yet. This means blocked clause addition, cardinality reasoning, gaussian elimination, and variable elimination based on irredundant covers all had to be disabled. Further disabled inprocessors are double look-head based equivalence extraction (lifting), congruence closure for equivalences, as well as unhiding. Even though these resolution based inprocessors can all in principle be mapped to DRUP, the effort for adding trace support is much higher and left as future work.

LICENSE

For the competition version of our solvers we use the same license scheme as introduced last year for our solvers. It allows the use of the software for academic, research and evaluation purposes. It further prohibits the use of the software in other competitions or similar events without explicit written permission. Please refer to the actual license, which comes with the source code, for more details.

REFERENCES

MiniSAT with Classification-based Preprocessing

Wenxiang Chen and Adele Howe and Darrell Whitley
Computer Science Dept., Colorado State University
Fort Collins, CO, USA

Abstract—We present a classification-based approach to selecting preprocessors of CNF formulas for the complete solver MiniSAT. Three different preprocessors are considered prior to running MiniSAT. To obtain training data for classification, each preprocessor is run, followed by running MiniSAT on its resulting CNF, on instances from the last competition. On each instance, the preprocessor leading to the smallest solving time is considered the “best” and used as the classification for the instance. A decision tree is trained to select the best preprocessor according to features computed from the CNF formulas. The decision tree can therefore be used to predict the best preprocessor for new problem instances.

I. INTRODUCTION

Preprocessing techniques simplify a CNF formula in order to speed up SAT solving. Preprocessing techniques are particularly useful for solving real-world SAT instances due to redundant information introduced during the conversion to CNF formula [1], producing a more compact form of the real-world problem, which is less domain-specific [2].

As has been observed by others (e.g., [3]), the value of specific preprocessing techniques is highly variable and depends on its parameterization, the problem instances and the core solver. In a pilot study, we examined the impact of specific CNF modifications on the performance of MiniSAT on application instances from the SAT 2013 competition. We found that different preprocessors (e.g., versions of SatELite) can produce very different CNF formulas. Additionally, we found that some simple truth preserving modifications (e.g., re-arranging the clauses or flipping the polarity of the literals) can significantly impact performance. To leverage the diverse biases of various preprocessors, we apply machine learning techniques to automatically select the preprocessor with the best predicted performance on a per-instance basis.

Three different types of preprocessing methods are used in our solver: 1) principled, 2) bias adjusting and 3) speculative. Principled encompasses those that employ well founded, truth preserving techniques for simplifying the formula, e.g., SatELite [2] and COPROCESSOR [4]. Bias adjusting refers to syntactic modifications to the formula that preserve truth and do not themselves reduce the number of variables or clauses, e.g., re-ordering clauses, grouping unit clauses. Speculative includes techniques that use heuristics to set variable values, remove those variables and the clauses satisfied by them and then use the reduced formula as the input to a core solver. The motivation for speculative pre-processing is that if the variables have been set correctly then the reduced search space should support solving instances more efficiently.

II. MAIN TECHNIQUES

Our solver consists of a set of preprocessors and a single solver. Given an instance, the solver computes its features, selects a preprocessor based on the features, runs the preprocessor, and provides the modified formula to MiniSAT as input. When the selected preprocessor is speculative, additional steps may be added if the heuristically set values lead to the formula being recognized as unsatisfiable; in this case, MiniSAT will be run without other pre-processing for the remaining time.

A. Constituent Preprocessors

Our solver employs three preprocessors (in addition to that built in to MiniSAT), one of each type:

a) Principled: We include the well known SatELite preprocessor. Although MiniSAT-2.2-Simp integrates SatELite, we still found that running SatELite before MiniSAT can be beneficial on some instances, i.e., this strategy solves some instances that MiniSAT by itself cannot solve in the allotted time.

b) Bias Adjusting: We considered a variety of syntactic modifications. The one that appears to best complement the other preprocessors is to invert the signs of all literals, run MiniSAT on the inverted instances and finally invert the returned assignment if a model is found on the inverted instances. MiniSAT has a polarity vector that is initialized to all false, which introduces a bias of setting new branching variables to false. Inverting the instance reverses the impact of this bias, which appears to be helpful for instances in which a majority of variables in the solutions are false.

c) Speculative: We have investigated techniques for heuristically setting variable values based on a Walsh analysis of the formula which we call "Hyperplane Reduction" [5]. We use a simplified form of this for our speculative preprocessor. The ten most frequently occurring variables in an instance are first fixed to false while running MiniSAT for at most 2600 seconds. If MiniSAT return unsatisfiable on the subspace or MiniSAT runs out of time, MiniSAT is rerun with the ten most frequent variables fixed to true. Finally, if both of the speculative assignments are proven to be unsatisfiable, MiniSAT is rerun on the search space without any assumption on assignment.

d) No Additional Preprocessor: We also include the option of not running any of the preprocessors just described; in this case, MiniSAT is run solo.
B. Classifying Problem Instances

We formulate the preprocessor selection as a classification task. In our solver, 43 base features [6] which are used in SATZilla [7] are computed on each instance. The training dataset contains 150 instances from the Core Solvers, Sequential, Application SAT track of SAT competition 2013. The Weka\(^1\) implementation of the C4.5 decision tree with default parameters is used for classification. In the offline training process, each of the preprocessors coupled with MiniSAT-2.2-Simp\(^2\) are evaluated on the training set to obtain runtimes on each training instance. The name of preprocessor resulting in the smallest solving time is used as the class label. The training process generates a decision tree, which is further converted into a script for picking the predicted best preprocessor based on features computed on new instances. Once a preprocessor is selected, it is coupled with MiniSAT to run on the new instance until it returns an answer or times out.

There are two main advantages of using a decision tree as the classifier. First, the classification with decision tree simply goes through a number of binary condition checks on computed features and this can be done quickly with little overhead in runtime. Second, one can manually inspect the built decision tree to gain an understanding of relationship between instance features and performances of preprocessors.

III. MAIN PARAMETERS

We use unmodified MiniSAT as a complete solver along with its default parameters. The parameter to our preprocessing is the number of bits to fix in speculative preprocessor. We use 10 bits for this parameter because it gave us the most consistent results in our empirical tests. It is possible that the number of bits is instance dependent, but we leave this determination to future work.

IV. IMPLEMENTATION DETAILS

The code to find the ten most frequent variables and to generate the instance with these variables fixed to some predefined truth values was implemented in C. GCC 4.8.2 is used to test our code using the “O3” compiler flag to create an optimized 64-bit binary. We used the J48 implementation from Weka 3.6.10 with default parameters of the C4.5 decision tree for classification. We wrote a Python script to convert the output decision tree from Weka to a Python library which can be imported to accomplish preprocessor selection.

The decision tree submitted to the competition was trained on data collected from HP-xw6600-Xeon5450-SAS machines with 8-core CPU at 3.0GHz and 16GB main memory. The configuration is close to what we expect for the competition and so the decision tree should approximate predicted performance.

V. SAT COMPETITION 2014 SPECIFICS

We have submitted our solver to the Sequential, Application SAT track and the Sequential, Hard-combinatorial SAT track of the competition. Although MiniSAT can determine if a speculative preprocessed instance is unsatisfiable, it does not necessarily mean that the original problem is unsatisfiable. Therefore we submitted our entry only to the SAT tracks.

We chose the Application and Hard-combinatorial tracks because we conjecture that the structure of industrial(-like) instances are more likely to have solutions that has a majority of variables set to true (or false).

VI. AVAILABILITY

Our C code and scripts can be downloaded from the following link:
http://www.cs.colostate.edu/~chenwx/~learn.gz

ACKNOWLEDGMENT

This research was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA9550-11-1-0088. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES


\(^1\)Weka software webpage: http://www.cs.waikato.ac.nz/ml/weka/

\(^2\)Source code is available at http://minisat.se/downloads/minisat-2.2.0.tar.gz
miniLoCeG+Glucose

Sebastian Burg
and Oliver Bringmann
and Thomas Kropf
University of Tuebingen
Tuebingen, Germany

Abstract—In this paper we present our, build upon the miniSat2 framework, implementation of our preprocessing technique of Local Counterexample Generation (LoCeG), calles miniLoCeG. LoCeG tries to find conflicts in a subset of clauses from a given instance and enhances the original instance with this knowledge. After the preprocessing we pass the resulting instance to a solver.

I. INTRODUCTION

In an effort to reduce the number of conflicts of a given instance we build the preprocessor miniLoCeG which implements Local Counterexample Generation (LoCeG)[1]. This aims at learning new clauses prior to the solving process. After the preprocessor added these new clauses to the instance this is then passed to a solver of choice. In our case it is the unmodified solver glucose2.1[?].

II. MAIN TECHNIQUES

The main technique used is “Local Counterexample Generation” (LoCeG). Given an Instance \( I := \bigwedge c \in C \) where clause \( c := \bigvee l \in L \) with Literal \( l \rightarrow \{ v \) \( \neg v \) with Variable \( v \rightarrow \top, \bot \).

We select a Variable from this given Instance depending on a heuristic, in our case number of occurrences, and select a number of clauses containing this variable. This results in a subinstance \( I_{v_i} := \bigwedge c \in v \in c. \) This subinstance is then negated \( \neg I_{v_i} \), which is then converted to CNF again. Any satisfying setting\( (S_{v_i}) \) for the variables in this negated subinstance is a Counterexample for the instance \( I \).

These satisfying settings are then negated\( (S_{v_i}^\neg) \) and added to Instance \( I' := I \cup S_{v_i}^\neg. \)

This process is repeated a given number of iterations or until a timeout is reached. It is further prohibited, that the same clauses are used in a following iteration.

III. MAIN PARAMETERS

Besides the usual miniSat2[?] parameters we added the following user controllable parameters to our preprocessor:

1) \(-\)\textit{iterations=XX} This parameter allows the user to set the number of iterations, default value is 20.
2) \(-\)\textit{mode=XX} This parameter changes the filter which decides which clauses shall be learnt in the iteration. These can be:

UNIT Only unit clauses are learnt. This often leaves the instance unchanged
SAT2 Only clauses of maximum length two are learnt.

SAT3 Only clauses of maximum length three are learnt.
ALL All found Counterexamples are learnt.

The default value is ALL.

There are other non-controllable parameters. These are set to what we experienced as good values in development. These are:

1) Number of clauses chosen for the subinstance, this is set to five.
2) The timeout is set to stop the process after a maximum of five minutes.

IV. IMPLEMENTATION DETAILS

MiniLoCeG uses the C++ Framework of Minisat2\footnote{http://www.minisat.se}, which we expanded to fit our needs. We needed to have a solver which tests all possible variable settings for a given mode. For this step of finding counterexamples we have to negate the subinstance. To solve this negated subinstance we convert it to CNF, which is costly in concerns of time and memory. The heuristic for choosing the variable building the subinstance is the number of occurrences in the instance. Other heuristics are possible, but not implemented. Furthermore our preprocessor is not packaged with a solver. The instance has to be written to a file, which then has to be passed to a solver.

V. SAT COMPETITION 2014 SPECIFICS

We submitted our preprocessor with glucose to the Application SAT+UNSAT and Hard Combinatorical SAT+UNSAT track. For the submission we have chosen to set the parameters for the iterations to 200 and the mode to Unit and SAT2. Glucose2.1 is run with all the default values.

VI. AVAILABILITY

The sources and solver description can be found at our homepage for miniLoCeG\footnote{http://www-ti.informatik.uni-tuebingen.de/~burg/loceg.php}.

REFERENCES

**MiniSat-ClauseSplit**

Balázs Donát Bessenyei  
Budapest University of Technology and Economics  
Budapest, Hungary

Zoltán Ádám Mann  
Budapest University of Technology and Economics  
Budapest, Hungary

**Abstract**—This is a MiniSat hack, extending MiniSat with a preprocessor that splits long clauses into shorter ones.

**I. INTRODUCTION**

The main idea of this solver is that short clauses are usually simpler to solve than long clauses. For example, the average-case complexity of random $k$-SAT is increasing in $k$ [1]. In CDCL solvers [2], short clauses are more useful, because they can be used for unit propagation sooner than long clauses. Long clauses pose the risk that the inconsistency of variable assignments that are bound to lead to a conflict will only become apparent after a large number of steps, thus making it impossible to prune useless parts of the search space early on.

**II. MAIN TECHNIQUES**

In order to mitigate the negative effect of long clauses, our solver splits them into shorter ones.

Let $c$ be a long clause and let $(c_1, c_2)$ be a partition of the literals in $c$. Then, splitting $c$ means introducing a new auxiliary variable $z$ and replacing $c$ with the two new clauses $c_1 \lor z$ and $c_2 \lor \neg z$. It can be seen easily that this transformation does not change the satisfiability of the problem instance.

Additionally, we can also generate clauses of length 2. Namely, we can add the clause $\overline{c} \lor \overline{c}$ for all literals $\ell \in c_1$. Again, it can be seen easily that this transformation does not change the satisfiability of the problem instance.

If the length of $c_1$ is $k_1$ and the length of $c_2$ is $k_2$, then our technique splits a clause of length $k_1 + k_2$ into a pair of clauses with lengths $k_1 + 1$ and $k_2 + 1$ (and optionally also introduces $k_1$ new clauses of length 2). If one or both of the resulting clauses is still considered to be too long, it can be split again. In practice, we choose the partition in such a way that the resulting first clause of length $k_1 + 1$ is certainly short enough; if necessary, the second clause can be split again.

**III. MAIN PARAMETERS**

The following parameters play an important role in MiniSat-ClauseSplit:

- Lower and upper threshold on the lengths of clauses that should be split. Too short clauses should not be split because splitting will not decrease the length (e.g., splitting a clause of length 3 would result in a clause of length 2 and one of length 3). Too long clauses should not be split because that would largely increase the number of clauses. Hence, only clauses whose original length is between the two thresholds are split.
- Target clause length for splitting. This is the length of the clauses that will result from splitting, except for the last one which may be shorter.
- Whether the optional clauses of length 2 should be generated.

The values for these parameters were tuned based on experiments with benchmarks of the 2013 SAT Competition.

**IV. IMPLEMENTATION DETAILS**

The solver is implemented in C++, on top of MiniSat [3], version 2.2.0.

**V. SAT COMPETITION 2014 SPECIFICS**

MiniSat-ClauseSplit is submitted to the MiniSat hack track. Compilation is carried out using g++ in 32-bit version, with O3 optimization.

**ACKNOWLEDGMENT**

This work was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947) and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

The authors would like to thank Niklas Een and Niklas Sörensson for their work on MiniSat. If MiniSat-ClauseSplit is successful, this is to a large extent due to the strength of MiniSat.

**REFERENCES**


Minisat_blbd

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China

Abstract—Minisat_blbd is a new kind of hack version of Minisat, which improves its previous version called Minisat_bit.

I. INTRODUCTION
Minisat_blbd is a hack version of MiniSat 2.2.0. It is submitted to sequential, application SAT track, application certified UNSAT track, application SAT+UNSAT track and MiniSAT Hack-track.

Compared with MiniSat, Minisat_blbd makes the following improvements.

1) Maintain the learnt clause database by a LBD technique.
2) Limit to the initial maximum size of database to a small value.
3) Modify the increasing factor of database
4) Clear periodically database
5) Optimize the source code.
6) Use a bit-encoding phase selection strategy [1].
7) Apply multiple restart strategies.

II. MAIN CHANGES
The LBD (Literals Blocks Distance) technique was first introduced by Audemard and Simon [2]. Now it has been used widely in CDCL SAT solvers. The first improvement made in Minisat_blbd is to add LBD computation. According to the LBD value of each clause, Minisat_blbd maintains the learnt clause database. When reducing database, the clauses with the higher LBD value are always deleted. Unlike SINNminisat, the LBD used here is not true LBD [3] that ignores literals assigned at level 0. The LBD here contains the computation of literals assigned at level 0.

The original version of MiniSat set initially the maximum limit size of database to a third of the number of clauses. This is very inefficient for large instances. Therefore, Minisat_blbd modifies it to 30000. In addition, only when database is reduced, the maximum limit grows. The maximum limit adjust of MiniSat depends on the parameter learntsize_adjust_cnt, while Minisat_blbd ignores it. When the size of database reaches 300000, the maximum limit will increase linearly, not proportionally. Notice, MiniSat is always the proportional increase.

In the original version of MiniSat, the increasing factor of database is 1.1. In this hack version, it is modified to 1.08.

In general, CDCL solvers have no database clearing operation. However, here we introduce a clearing operation. Every 13 times the learnt clause database are reduced, we clear them if the number of variables of instances is less than 1000.

The fifth change is to optimize the source code. In the procedure pickBranchLit of MiniSat, the code for testing “next == var_Undef” is unnecessary. Therefore, Minisat_blbd eliminated it.

Like Minisat_bit[4], Minisat_blbd applies also a bit-encoding phase selection strategy [1]. However, the bit-encoding phase selection strategy used here is much simpler. In the procedure pickBranchLit of Minisat, the following statement is added.

\[
\text{if (bitN >= 0) \{}
\text{int dl=decisionLevel();}
\text{if(dl < 12) polarity[next] = (bitN >> (dl \% 4)) & 1;}
\text{\}}
\]

In addition, Minisat_blbd uses alternately the bit-encoding and Minisat phase selection strategy. The alternating interval are set to 30000 conflicts. That is, every other 30000 conflicts Minisat_blbd runs the bit-encoding phase selection strategy for 30000 conflicts.

Finally, based on the feature of instances, Minisat_blbd adopts different restart strategies. When the number of free variables is less than 1500, it uses the geometric restart strategy. Otherwise, it uses the luby restart strategy.

REFERENCES
MiniSat_HACK_999ED, MiniSat_HACK_1430ED and SWDiA5BY

Chanseok Oh
New York University
NY, USA

Abstract—MiniSat_HACK_999ED, MiniSat_HACK_1430ED, and SWDiA5BY all take an unprecedented, extreme measure in managing learnt clauses. MiniSat_HACK_999ED and MiniSat_HACK_1430ED are based on DRUP-patched MiniSat 2.2.0, and the 999ED version qualifies for entering the MiniSAT Hacktrack in the SAT Competition 2014. SWDiA5BY is based on Glucose 2.3 that entered the SAT Competition 2013. All of the solvers are capable of generating DRUP proof for UNSAT instances.

I. INTRODUCTION

The solvers are submitted to the Competition to confirm several hypotheses of the author; one of which is that the element that actually contributes to the progress of solving a problem in LBD-based contemporary CDCL solvers is practically constituted by the accumulated number of learnt clauses having at most a certain small LBD [1] value (or any appropriate metric measuring necessity of such clauses) in almost all cases. Another hypothesis is that it is the current set of clauses that drives the way that the next sets of learnt clauses are generated by the means of conflicts and variable selection and; that having direct influence on them, the current set determines the characteristics and the evolution of the future sets in a chain reaction. In this regard, the goal of the presented solvers is to contain the momentum of self-accelerating divergence.

II. MINISAT_HACK_999ED

The solver is based on MiniSat 2.2.0 with the DRUP patch provided by the Competition. The edit distance (see the Competition website for the definition, which is basically the number of different characters) from original MiniSat is 999, hence the name MiniSat_HACK_999ED.

A. Learnt Clause Management: Core and Local

Core learnt clauses Clauses with LBD less than or equal to 5, classified as core, are kept indefinitely, \(^1\) unless removed when satisfied at decision level 0 by trivial simplification. Note that the solver does not need to store LBD values, since only done is the classification.

Local learnt clauses The number of other clauses, classified as local, is maintained roughly between 15,000 and 30,000, and the clauses are maintained by the MiniSat-based clause activity scores. Specifically, the solver tries to halve the size of the database at every 15,000 conflicts, starting from practically 30,000 clauses.

It is extremely unlikely that instances that can be solved reliably with resolution-based solvers will not produce any core learnt clauses with the aforementioned LBD threshold, or even with a smaller one, though the rate may slow down over time. Even for such rare cases, it has been observed that, interestingly, this scheme can sometimes solve the problems.

B. Restart Strategy and Restart Blocking Strategy

The restart and restart blocking strategies are essentially same as those of Glucose [2], although it has been observed that the unmodified MiniSat-style restart (luby series) strategy alone yields much more superior results for some cryptographic instances where Glucose often gets stuck forever. The decision to use the Glucose-style restart strategy is mainly because that the author had no time and resource to evaluate the MiniSat-style restart in full scale, and the initial intent was to get a more accurate result of comparison with original Glucose.

C. Other Minor Tweaks

In contrast to original MiniSat or Glucose, the behavior of MiniSat_HACK_999ED (particularly long-term) is relatively stable and insensitive against changes to other participating factors; as long as the core learnt clauses are kept to grow indefinitely, the eventual behavior and the time to solve the same problem are largely consistent and determined by the growing number of the core clauses in most cases (except for some cryptographic instances). Often, a single tweak makes no profound or noticeable effect, or if backed by good theoretical justification, improves the overall performance consistently.

Nevertheless, below is the list of tweaks that have been implemented in the solver. Some of them are inspired by or adopted from Glucose.

- To prevent learnt clauses that were just generated or participated in recent conflict analyses from being dropped immediately by the fast-paced periodic clause database reduction and to give higher priority to them, they are marked so as to be protected at the next clause reduction. This measure seems to give observable and consistent improvement.
- If a local clause participates in a conflict analysis and the then LBD value is below 5, it becomes a core clause to be kept indefinitely thereafter.
• Similar to Glucose, during conflict analysis, the VSIDS scores of variables at the current decision level is additionally bumped if they are propagated by core learnt clauses [3], although its effect with MiniSat_HACK_999ED has not been thoroughly evaluated.

• The variable decay factor initially starts at 0.8 and grows slowly up to 0.95, as in the implementation of Glucose 2.3 [4]. The effect has not been evaluated at all.

• Preprocessing is turned off if the number of input clauses is over 4,800,000, following the hack of Glucose 2.3 that entered the SAT Competition 2013 [4].

• Clause activity is bumped only for local clauses; otherwise, it would unnecessarily trigger re-scaling of the activity scores of the local clauses.

III. MINISAT_HACK_1430ED

This version is identical to MiniSat_HACK_999ED except that, similar to the decision of Lingeling that entered the SAT Competition 2013 [5], it falls back to the original restart and reduction strategies of MiniSat if the number of variables of a problem is at most 1,000. Otherwise, it behaves exactly same as MiniSat_HACK_999ED. This may additionally prove whether the 999ED version performs better or worse than original MiniSat on those instances.

The edit distance is 1430, and thus this variation does not qualify for the MiniSAT Hack-track.

IV. SWDIASBY A26

SWDiASBY implements the main idea of the core and local learnts on top of Glucose 2.3. A notable difference from MiniSat_HACK_999ED is that it strictly maintains the number of local clauses between 10,000 and 20,000, mainly because it uses a different clause protection mechanism. Implementation-wise, there are also other differences for a few reasons, but only in a minor way.

Note that it does not use LBD but the traditional MiniSat-based clauses activity scores for database reduction of local clauses.

A. Other Minor Tweaks

• Unlike original Glucose, decision level 0 is excluded when computing LBD during conflict analysis.

• If a local clause participates in a conflict analysis and the then LBD value is less than or equal to 5, it becomes a core clause.

• Clause activity is bumped only for local clauses.

V. SWDIASBY A30

The Apr 30 version (A30) is a variation based on A26 and uses an even smaller LBD threshold of 4 for core learnts, which shows surprisingly good results on some very hard instances that neither original Glucose nor A26 can ever seem to solve it. The author hopes to prove that even clauses with such a small LBD of 5 often have an adverse and irrecoverable impact. A30 also adds a few tweaks.

A. Added Tweaks

The author emphasizes that the effects of the following tweaks have not been evaluated at all.

• The solver adjusts the LBD threshold dynamically, ranging from 4 to 10. If the rate of discovering the core learnts is too low or stalling, the threshold is increased dynamically. If increased, the restart policy becomes more imposing temporarily. The threshold also decreases over time.

• If the number of core learnts reaches 1,000,000, 10% of the clauses are dropped according to the decreasing order of LBD (1st criterion) and the increasing order of time they were added (2nd criterion). However, it is anticipated that few instances will reach this limit with the given time-out in the competition.

VI. AVAILABILITY

All of the presented solvers do not add any additional license to the base solvers.

REFERENCES

Abstract—The solver MINIT SAT initializes the variable activities and the polarities for the decision heuristic.

I. MINIT SAT

MINISAT 2.2, as well as many solvers that are based on this solver do not initialize the activities and the polarities of the decision heuristic, although many heuristics have been proposed [1]. In this solver modification, the decision heuristic is initialized as follows:

The activity of the variable \( v \) is set to the corresponding integer value \( v \). This way, the decision heuristic of the solver will pick the very last variable first. According to widely used formula generation processes, the Boolean variables of a problem are usually the first \( n \) variables, and all higher values are used by auxiliary variables. Hence, MINISAT tends to decide auxiliary variables first. However, after some time of search, the changes to the activities of the variables during search become stronger, so that these values are overwritten. For short solving timeouts, this initialization seems to be helpful.

Furthermore, MINISAT 2.2 always picks the negative of a variable where no information for polarity caching [2] is present yet [1]. MINIT SAT sets the initial polarities for a variable \( v \) as follows:

\[
\text{pol}(v) = \begin{cases} 
\top & \text{if } \sum_{C \in F_v} 2^{-|C|} + \sum_{C \in \neg F_v} 2^{-|C|} > 0 \\
\bot & \text{otherwise}
\end{cases}
\]

This equation assigns a positive polarity, if the positive literal of \( v \) is more constrained in the formula according to the Jeroslow-Wang heuristic [3], and therefore focus on producing conflicts more than looking for a solution. If a variable is constrained equally for the positive and the negative variable, then the negative polarity is used.

II. IMPLEMENTATION DETAILS

The Levenshtein distance to the patched version of MINISAT 2.2 is 461 characters, including the comments inside the code.

REFERENCES

Abstract—We submit the solver MiPiSAT to the MiniSAT Hack-track: The solver extends MiniSAT 2.2 by inprocessing capabilities. In particular, we implemented failed literals and necessary assignments as inprocessing. The technique is applied on variables with high activity and therefore also exploits information that is not available for preprocessors.

I. INTRODUCTION

Simplifying the formula before giving it to the SAT solver has become an important part of the solving chain [1]. However, we observed the addition of necessary assignments and failed literals as a preprocessing step can decrease the solved instances within a time bound in MiniSAT 2.2 [2]. Inprocessing [3] is to apply formula simplifications during search, and it is an attractive idea since it allows to apply computationally expensive formula simplification techniques. Additional information can be taken into account when formula simplifications are applied as inprocessing, such as the variable activity and learned clauses. In particular, probing can benefit of additional clauses as unit propagation can infer more literals. Moreover, probing can be restricted to relevant variables, i.e. literals having a high VSIDS score [4].

II. PROBING AS INPROCESSING

In this solver modification, failed literals and the detection of necessary assignments [5] is performed every 32 times before the solver makes a top level decision. Moreover, probing is limited to the 100 variables with highest activity.

III. IMPLEMENTATION DETAILS

We implemented probing using three methods: The method SOLVER::PS collects all literals that are inferred by unit propagation given a single literal. It is called by the method SOLVER::P two times for a literal $x$ and its negation $\neg x$. If unit propagation detects an inconsistency given the literal $x$, we enqueue its negation as a unit clause. Otherwise, we collect the commonly implied literals and enqueue them. Finally, we apply unit propagation. If unit propagation derives a conflict, we terminate the computation with the answer UNSAT. The method SOLVER::POH then calls SOLVER::P on the first 100 variables in the orderheap data structure.

The Levenshtein distance to the patched version of MiniSAT 2.2 is 980 characters.

IV. CONCLUSION

The chosen parameter were guessed and we believe that further improvements can be obtained by parameter tuning.

ACKNOWLEDGEMENTS

I want to thank Norbert Manthey for the intensive support during the implementation and the introduction into the MiniSAT 2.2 architecture.

REFERENCES

MiniSAT with Speculative Preprocessing

Wenxiang Chen and Adele Howe and Darrell Whitley
Computer Science Dept., Colorado State University
Fort Collins, CO, USA

Abstract—We present a new preprocessing technique utilizing subspace averages to perform a reduction by heuristically fixing truth assignments. We use the Walsh transform to efficiently compute the average evaluation of solutions in subspaces of the search space that we refer to as hyperplanes. A hyperplane contains all solutions that have the same truth assignments over some subset of the \( n \) variables in the given formula. The formula is reduced by fixing the values from the hyperplane with the best average and reduce the original formula by eliminating clauses satisfied by the variables with consistent truth assignments across all solutions in the hyperplane. We then run the MiniSAT complete solver on this reduced instance.

I. INTRODUCTION

A strategy that has performed well in previous competitions is to incorporate the use of a preprocessor, typically SatELite [1], to reduce both the number of variables and clauses of the original CNF formula. We introduce a speculative form of preprocessing in which a heuristic is applied to select the truth assignment of frequently occurring variables; these assignments allow the formula to be reduced by removing clauses satisfied by the assignments.

In our submission, we use unmodified versions of MiniSAT and SatELite. For our solver, we use MiniSAT 2.2.0 solver [2], the most recent available from the MiniSAT web page\(^1\). We also employ the SatELite [1] preprocessor.

Our contribution is the introduction of an speculative preprocessing step that we call hyperplane reduction, which further reduces the search space from what SatELite does. The key step of hyperplane reduction is identifying a promising set of variable values. We define a heuristic based on Walsh analysis, which permits efficient computation of the average evaluation across a region of the search space. The region is defined as a hyperplane which is a maximal set of solutions that share the same truth assignment over some subset of \( n \) variables. For all \( k \)-bounded pseudo-Boolean optimization problems, we can convert the evaluation functions into a polynomial form in \( O(n) \) time [3]. This allows us to quickly and exactly compute hyperplane averages, the average evaluation of all solutions in the hyperplane.

We find the hyperplane averages of the hyperplanes corresponding to each possible assignment of the ten variables that appear most often in the reduced formula produced by SatELite. We then reduce the formula further by first removing the clauses satisfied by the partial assignment over the ten variables that corresponds to the highest average hyperplane. We next remove the ten variables from the remaining clauses in which they appear. The resulting formula is then passed to MiniSAT.

The SAT space is known to be deceptive [3]. Thus, we know that the hyperplane with the best average may not contain a globally optimal solution. It is therefore possible that our reduced problem may be unsatisfiable even though the original formula is satisfiable. This is why we call our preprocessor “speculative”. To allow for this possibility, we limit the runtime of MiniSAT to 2400 seconds on the hyperplane reduced formula. If this upper limit is reached, or MiniSAT finds the formula to be unsatisfiable before the time limit, MiniSAT runs on the SatELite reduced formula until it terminates (either by deciding the satisfiability of the problem or reaching some maximum time limit).

II. MAIN TECHNIQUES

The hyperplane values are set to maximize the expected value of solutions. Walsh coefficients can be used to efficiently compute the average evaluation of solutions in a hyperplane [3]. Functions on binary variables such as MAXSAT can be decomposed into an orthogonal basis:

\[
f(x) = \sum_{i=0}^{2^n-1} w_i \psi_i(x)
\]

where \( w_i \) is a real-valued weight known as a Walsh coefficient and \( \psi_i \) is a Walsh function. The Walsh function

\[
\psi_i(x) = (-1)^{x^T i}
\]

generates a sign: if \( i^T x \) is odd \( \psi_i(x) = -1 \) and if \( i^T x \) is even \( \psi_i(x) = 1 \).

Since MAXSAT is a linear combination of subfunctions, Rana et al. [3] show that the Walsh coefficient associated with each clause can be directly computed. Each clause contributes at most \( 2^k \) nonzero Walsh coefficients where \( k \) is the length of a clause.

To manage the computation of the best hyperplane, we restrict the number of variables in the hyperplane and the maximum size of a clause. To maximize the potential impact on the size of the search space remaining, the selected variables are the ten most frequently occurring variables. We have experimented with other ways of choosing variables, but have found this to be the most effective. Although many application instances have variable clause lengths with some large clauses, we restrict the maximum clause length to six to reduce the number of coefficients to calculate. Empirically we have found that the clause length restriction exerts small effects on the evaluation because only one of the variables need be true.

\(^1\)http://minisat.se/Main.html
To further reduce the required computation, we use an equation that includes only the clauses influenced by the hyperplane variables. Let \( h \) be a particular hyperplane, \( \alpha(h) \) be a mask used to select \( 2^{o(h)} \) (\( o(h) \) is the number of bits fixed by \( h \)) relevant coefficients; \( \beta(h) \) is used to extract the 1 bits from the respective coefficients. The average evaluation of \( h \) is:

\[
\text{Avg}(h) = \sum_{j \subseteq \alpha(h)} w_j \psi_j(\beta(h))
\]

As an example, for hyperplane \( 11 \ast 00 \ast *(\ast \text{ indicates variables not in } h, 0s \text{ and } 1s \text{ are particular variable settings for the variables in the hyperplane}), \alpha(h) = 110110 \text{ and } \beta(h) = 1100000.

The hyperplane with the highest average is chosen as the best. The original problem is reduced by eliminating the fixed variables and any clauses that are satisfied by them. For example using hyperplane \( 11 \ast 00 \ast (\ast \text{ i.e., } x_1 = 1, x_2 = 1, x_4 = 0, x_5 = 0 \text{ are fixed}), \) if the instance was \((x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor x_3 \lor x_5) \land (\neg x_2 \lor \neg x_4 \lor x_6), \) it would be reduced to a single clause with one literal \( x_3. \)

The drawback of this method is that it is heuristic – the partial assignment may not correspond to a model for the original formula or may not correspond to a model that is easily found within a time limit. However, we find that the best hyperplane often does contain a satisfying solution or can be found to be unsatisfiable very quickly.

### III. Main Parameters

We use unmodified MiniSAT as a complete solver along with its default parameters. The two parameters to our preprocessing is the number of bits to fix in the hyperplane and maximal lengths of clauses considered in the hyperplane average computation. We use 10 bits for the number to fix because it gave us the most consistent results in our empirical tests. It is possible that the number of bits is instance dependent, but we leave this determination to future work. For the second parameter, the maximal clause length is set to six.

### IV. Implementation Details

We used C to implement the calculation of Walsh coefficients and the hyperplane averages. We used GCC 4.8.2 to test our code using the “O3” compiler flag to create an optimized 64-bit binary.

A bash script wrapper performs the reduction and calls SatELite and MiniSAT on the reduced problem. If a satisfiable solution is found in the hyperplane reduced problem, we add the truth assignments fixed by our reduction code to the satisfying solution provided by MiniSAT and report this solution to the original problem.

### V. SAT Competition 2014 Specifics

We have submitted our solver to the Sequential, Application SAT track and the Sequential, Hard-combinatorial SAT track of the competition. Although MiniSAT can determine if a hyperplane reduced instance is unsatisfiable, it does not necessarily mean that the original problem is unsatisfiable. Therefore we submitted our entry only to the SAT tracks.

We chose the Application and Hard-combinatorial tracks because we conjecture that there is more variance in the hyperplane averages in structured instances than random instances. We believe that this structure allows our algorithm to be more effective on these types of problems. More details about this conjecture were previously discussed in the context of incomplete solvers based on local search for solving MAXSAT [4].

### VI. Availability

Our hyperplane reduction code along with the wrapper script for calling MiniSAT and SatELite can be downloaded from the following link:

http://www.cs.colostate.edu/~chenwx/~simple.gz

### ACKNOWLEDGMENT

This research was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA9550-11-1-0088. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

### REFERENCES


**Abstract**—This document describes the local search SAT solver Ncca+ based on the CCASat solver and Novelty-like heuristic.

I. MAJOR SOLVING TECHNIQUES

Ncca+ is based on CCASat which is first described in [1]. CCASat is the winner of last SAT Challenge 2012 on the random SAT track. The evolutions introduced by Ncca+ are:

1) For 3-SAT random instances: when selecting a CCD (Configuration Changed Decreasing) variable [2], CCASat breaks ties in the favor of the oldest variable in the case of equal score of the candidate variables. In Ncca+, the breaking ties is done according to the number of occurrences of the CCD variables on the falsified clauses.

2) For \( k \)-SAT (\( k > 3 \)) random instances: In this case, Ncca+ works as follows:
   - If the set of CCD variables is not empty, select the one with the highest score, breaking ties in the favor of the oldest one or the one of the highest subscore and occurring the most in the falsified clauses.
   - In other cases, update clause weights and with probability \( wp \) do as Novelty(\( p \)) else (with probability \( 1–wp \)) choose a variable in a randomly selected variable as done in CCASat. The values of \( p \) and \( wp \) are adaptively adjusted during the search [3], [4]. Accordingly, the SD variables [2] are not considered for such instances.

The rest of the solver is similar to CCASat regarding to the smoothing scheme of the clause weights.

II. PARAMETER DESCRIPTION

The parameters of Ncca+ are similar to those of CCASat. The adaptive noise settings (\( p \) and \( wp \) values) are based on the ones used in [5] with \( \phi = 5 \) and \( \theta = 2 \).

III. SPECIAL ALGORITHMS, DATA STRUCTURES AND FEATURES

The code of Ncca+ is based on CCASat code\(^2\). There is no additional data structure.

IV. IMPLEMENTATION DETAIL

1) The programming language used is C++
2) The solver is based on CCASat with the additional features explained above.

V. SAT CHALLENGE 2012 SPECIFICS

1) The solver is submitted in "Core Solvers, Sequential, Random SAT" track.
2) The used compiler is g++
3) The optimization and compilation flags used are "-O3 -static".

VI. AVAILABILITY

Our solver will be publicly available from the SAT competition website.

ACKNOWLEDGMENT

We would like to thank the authors of CCASat for making available the source code of their solvers.

REFERENCES

Nigma 1.1

Chuan Jiang
Department of Computer Science
Iowa State University
Ames, Iowa, USA

Gianfranco Ciardo
Department of Computer Science
Iowa State University
Ames, Iowa, USA

Abstract—This document describes the SAT solver Nigma 1.1, a CDCL SAT solver with partial backtracking. Nigma 1.1 is improved with a more aggressive clause deletion strategy and failed literal probing.

I. INTRODUCTION

Nigma 1.1 is an improved version of Nigma 1.0, which entered the SAT competition 2013 [1]. We improved Nigma with a more aggressive clause deletion strategy and failed literal probing.

II. MAIN TECHNIQUES

Backtracking is a basic technique of search-based SAT solvers. Guided by conflict analysis, a SAT solver computes a backtracking level and discards the part of assignment trail between the conflicting level and the backtracking level. It is widely assumed that the assertion level, which is the second-highest level among the literals in the learnt clause or 0 if the learnt clause contains only one literal, is chosen as the backtracking level.

The solver Nigma is featured with partial backtracking, a conservative backtracking strategy. This technique was inspired by the observation that CDCL (Conflict Driven Clause Learning) solvers often redo many discarded variable assignments as the popular branching heuristics [2][3] have a potential tendency to keep the solver in the same search space. By backtracking to a level between the conflicting level and the assertion level, the solver discards the assignment trail as little as possible and retains as many sub-solutions as possible. Then, a generalized version of the regular propagation is invoked to assign the conflicting variable at the assertion level and resolve the resulting complications: unusual implication, inappropriate watched literal, spurious conflict, and wrong decision level. For implementation details of partial backtracking, please refer to [4]. We keep the triggering condition for partial backtracking used in Nigma 1.0: the solver backtracks partially only if it is going back more than ten levels.

Compared with Nigma 1.0, Nigma 1.1 has been improved by employing an aggressive clause deletion strategy and failed literal probing.

A. Aggressive Clause Deletion Strategy

We observed that the number of decisions per conflict becomes smaller when partial backtracking is enabled, which indicates that the solver identifies conflicts more frequently. Since one conflict results in one learnt clause in general, an aggressive clause deletion strategy is employed to maintain a high efficiency in the propagation. Nigma removes almost half of the learnt clauses with glucose-like heuristics [5] each time the number of learnt clauses reaches $4000 + 2000 \times x$ ($x$ denotes the number of learnt clause deletion happened before).

B. Failed Literal Probing

We added failed literal probing [6] into the arsenal of optional preprocessing techniques. By identifying a failed literal, the solver asserts the value of the corresponding variable at the top level and halves the search space immediately. A timeout limit can be set for the probing procedure to avoid spending too much time on complicated cases.

III. SAT COMPETITION 2014 SPECIFICS

We submit Nigma 1.1 to the tracks for Sequential, Application SAT+UNSAT/SAT/certified UNSAT and the tracks for Sequential, Hard-combinatorial SAT+UNSAT/SAT/certified UNSAT. It is compiled by GCC in 64-bit with the -O3 flag.

IV. AVAILABILITY

Nigma is an open-source SAT solver under MIT license. The source code can be downloaded at http://sourceforge.net/projects/nigma/.

V. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grant CCF-1442586.

REFERENCES

Abstract—This document is brief description of solver ntusat and ntusatbreak entering the SAT Competition 2014.

I. INTRODUCTION

ntusat is based on Simpsat in SAT Competition 2012 which is an enhanced version of CRYPTOMINISAT 2.9.2 [1] with new technique for gaussian elimination described in [2]. ntusatbreak is ntusat with preprocessing technique described in [3] and [4] of SAT Competition 2013. The additional technique compared to Simpsat in 2012 is pure literal detection and learning [5]. Adding pure literal implication clauses from time to time could make the theory not functional equivalent but reduce the search space to help the solver solve faster. The other technique is to find more XOR-clauses by improving XOR-recovery. If no XOR-clauses found, do the clause resolution procedure to find more information. The detail technique would be described in algorithm and datastructure part.

II. MAIN TECHNIQUES

ntusat and ntusatbreak are CDCL solver with simplification techniques subsumption, block literal, and inprocessing technique literal probing, hyper-binary resolution applied by CRYPTOMINISAT 2.9.2. Since there are XORclauses extraction and equivalent variable replacement, part of the pure literal detection technique is different from MINIPURE [5] last year. If any variable of literal $\beta$ is in the XORclause, $\beta$ would not be detected as pure literal. If literal $\alpha$ is replaced by literal $\beta$, then all the clause ID in vector litcontain [5] of $\alpha$ should be moved to litcontain($\beta$). For other inprocessing techniques which causes variable elimination, clause subsumption and clause elimination, the clause watch and litcontain datastructure should also be checked.

III. MAIN PARAMETERS

Most of parameters are the same as SIMPSAT in SAT Competition 2012. Since improved XOR-recovery from time to time is time consuming, NTUSAT do not apply that technique after 700 seconds. The pure literal detection and learning is applied every eight restarts. Empirically, for some cases, adding to much pure literal implication clauses would make it harder to find solution for SAT cases. If the difference between max and min decision level of variables in pure literal implication clauses is less than 25, the clauses is added to the database as irredundant clauses. Otherwise, the solver drop the clause.

Add-XORs(add new learnt xor for each restart)

input: Solver data: watch, original clauses $\phi$
output: NEWXOR and LEARNTCLAUSE

begin
01 for each original and learnt clause $C$ size;5 and $C$ not found
02 match-clauses := $\phi$;
03 for each literal $L$ in $C$
04 candidate-clauses := $\phi$ watch($\neg L$)+watch(L);
05 +cache($\neg L$)+cache(L);
06 match-clauses := match-clauses+;
07 check-all-literal(candidate-clauses);
08 end
09 if foundallxor($C$, match-clauses);
10 newxor.push($C$);
11 else
12 learntclause=Getmoreinfo($C$,match-clauses);
13 end

end

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

During the simplification procedure, NTUSAT would detect XORclauses by Figure 1. If there is no XOR-clauses found, the solver would apply the resolution method to find more information. For example, the clause set $\Omega = \{ (x_1 \lor x_2 \lor x_3), (x_1 \lor x_2 \lor x_3), (x_1 \lor x_2 \lor x_3), (x_1 \lor x_2 \lor x_3) \}$ these clauses can not form a XOR clause, but these clauses imply $x_1 \Rightarrow x$. The other clause set $\Gamma = \{ (x_1 \lor x_2 \lor x_3), (x_1 \lor -x_2 \lor x_3), (-x_1 \lor x_2 \lor x_3), (-x_1 \lor x_2 \lor x_3) \}$ can neither form a XOR clause, but $x_2 = x_3$ can be implied from the set. The detail of the algorithm is in Figure 2. The solver generates clauses by resolution which would reduce the clause size by 1. In line 2, the solver implements this process until size is equal to three and in this loop the generating clauses would be binary clauses. In line 15, the vector candidate-learnt will store the binary clauses of the same variables, e.g. $\{ (-a \lor b), \} \}$ would be push into candidate-learnt (a,b) $= \{ 1, 0, 0 \}$. The sign of the variables in binary bits form is equal to 1 and $(-a, -b)$ is equal to 0. In line 22, k.size() means how many binary clauses with the same variables. k.size() equal to 4 means conflict, k.size() equal to 3 means the two variables are assigned. k.size() equal to 2 means the two variables are equivalent or one variable is assigned. The solver could get more information even there is no XOR-clauses found.
Getmoreinfo

input: C, match-clauses
output: learntclauses

begin
01 even := φ, odd := φ, candidatelearnt := φ;
02 for sz = match-clauses.size down to 3
03 for each clause i in match-clauses and C
04 match-clauses.popup();
05 if xor all the literal in i is true
06 even.pushback(i);
07 else
08 odd.pushback(i);
09 end
10 for each i in even
11 for each j in odd
12 if sign of literals in i, j has only one difference
13 learnt := resolve(i, j);
14 if sz equal 3
15 candidate-learnt[learnt].pushback(signs of learnt);
16 else
17 match-clauses.pushback(learnt);
18 end
19 end
20 end
21 for each k in candidate-learnt and k.size() > 0
22 learntclause.pushback(learn clauses according to k.size());
23 end
end

Fig. 2. Algorithm: GETMOREINFO

V. IMPLEMENTATION DETAILS

ntusat and ntusatbreak are implemented in C++. Both of them are based on SIMPSAT and CRYPTO MINISAT 2.9.2. ntusatbreak applies saucy and breakID as preprocessor before solving.

VI. SAT COMPETITION 2013 SPECIFICS

Both of the solvers are submitted to hard combinatorial SAT and UNSAT , Application SAT and UNSAT , Random SAT and UNSAT. The compiler version is 4.7.1. , O3 used for compiling and 64-bit binary. There is no command-line option. All the parameters are locked in the solver. More information about parameters is in Main Parameters part.

VII. AVAILABILITY

Solver NTUSAT is available upon request for research purpose. Send email to the authors. For preprocessor of BreakID [3] and Saucy http://vlsicad.eecs.umich.edu/BK/SAUCY/

ACKNOWLEDGMENT

The author thanks to Paul T. Darga, Mark Lifton and Hadi Katebi for the symmetry detection tool Saucy and embedded breaking clause tool Shatter in Saucy 3.0. Also thanks to K.N.M. Soos who is the author of CRYPTO MINISAT series and the discussion with C.-S. Han for new ideas , who is the author of SIMPSAT.

REFERENCE

[1] K. N. M. Soos and C. Castelluccia., “Extending sat solvers to cryptog-
tographic problems.” Proc. Int’l Conf. on Theory and Applications of
elimination in a simplex way.” Proc.Int’l Conf. on Computer Aided
PCASSO – a Parallel CooperAtive Sat SOlver

Davide Lanti
Free University of Bolzano
Italy
Ahmed Irfan
Fondazione Bruno Kessler, Italy
University of Trento, Italy
Norbert Manthey
TU Dresden
Germany

Abstract—The SAT solver PCASSO is a parallel SAT solver based on partitioning the search space iteratively and using clause sharing among the nodes.

I. INTRODUCTION

The major difference to last years version of PCASSO is the update of the formula simplifier COPROCESSOR. The remaining configuration of the tool remained the same. We only adapted the internal data structures to the structures of RISS to match this years version 4.27 [1].

PCASSO proceeds by creating and solving partitions of the input instance. Partitions are created through partition functions, where a partition function is a function \( \phi \) such that, given a formula \( F \) and a natural number \( n \in \mathbb{N}^+ \), \( \phi(F, n) \) := \( \{ F_1, \ldots, F_n \} \), where \( F := F_1 \lor \cdots \lor F_n \) and each pair of partitions is disjoint: \( i \neq j \in [1, n] \), \( F_i \land F_j \equiv \bot \). Without loss of generality we assume that partitions \( F_1, \ldots, F_n \) are always of the form \( F \land K_1, \ldots, F \land K_n \), where \( K_1, \ldots, K_n \) are sets of clauses, called partitioning constraints. By iteratively applying the partition function to a formula \( F \), a partition tree is produced. Nodes in the partition tree are tagged with their positions: the root node \( F \) is tagged with the empty position \( \epsilon \); the i-th successor (from left to right) of a node \( F^p \) at position \( p \) is the node \( F^{pi} \) (see Figure 1). Please notice that, as positions are strings, the standard prefix relation among strings (\( < \)) is defined for positions as well.

II. MAIN TECHNIQUES

The partition function used in PCASSO is tabu scattering, which is an extension of scattering [2]. The idea of scattering is to define each partitioning constraint as conjunctions of cubes [3], where a cube is a formula \( Q := \{ C_1, \ldots, C_n \} \) such that \( |C_i| = 1 \), for each \( 1 \leq i \leq n \). Observe that the negation of a cube \( Q := \{ I_1, \ldots, I_n \} \) is the clause \( \{I_1, \ldots, I_n\} \). More precisely, given a formula \( F_0 \) and an integer \( n \), the \( n \) partitions \( F_1, \ldots, F_n \) are created by using \( n - 1 \) cubes \( Q_1, \ldots, Q_{n-1} \) and applying them according to the following schema: \( F_1 := F_0 \land Q_1; F_m+1 := F_0 \land (\bigwedge_{i=1}^{m} Q_i) \land Q_{m+1}(1 \leq m < n - 1) \); and finally \( F_n := F_0 \land (\bigwedge_{i=1}^{n-1} Q_i) \). Tabu scattering adds the restriction to scattering that a variable used in one cube must not be used in the cubes for creating the remaining partitions. Using tabu scattering, we diversify the search more. PCASSO uses lookahead techniques [4] for choosing the literals (in cubes). We also use the following reasoning techniques: failed literals, necessary assignments, pure literals, and add learned clauses to the partition constraints. Techniques like constraint resolvent, double lookahead, and adaptive pre-selection heuristics are also used as proposed in the literature [4].

To describe the node-state of a node \( F^p \) at a certain point of execution we use a triple \((F^p, s, r)\) where \( s \in \{\top, \bot, ?\} \) (\( \top \) indicates that an incarnation found a model for the node, whereas \( \bot \) indicates that an incarnation proved unsatisfiability of \( F^p \); finally, \( ? \) indicates that the node has not been solved yet) and \( r \in \{\square, \blacklozenge\} \) (indicating whether an incarnation is running on \( F^p \) or not, respectively). Given the notion of node-state, PCASSO exploits the overlapped solving strategy if two incarnations are allowed to run at the same time on nodes \( F^p, F^q \) such that \( p \leq q \). In order to solve an unsatisfiable node \( F^p \), either \( F^p \) has to be directly solved by some incarnation or each child node \( F^{pi} \) has to be solved. There is no limit on the solving time for each node. Per variable, VSIDS activity and progress saving are shared from parent to child nodes.

Learned clauses are shared between incarnations to intensify the search. A learned clause is considered unsafe if it belongs to partitioning constraints, or it is obtained by a resolution derivation involving one or more unsafe clauses. A clause that is not unsafe is called safe. The main intuition here is that sharing unsafe clauses might affect the soundness of the procedure, and thus it should be forbidden. Concerning PCASSO, a learned clause is shared in a sub-partition tree if it is safe in that sub-partition tree. This information can be calculated by tagging each clause with the position of the sub-tree where the clause is valid (position-based tagging [5]). We propose a dynamic learned clause sharing scheme, that is based on LBD scores [6]. A learned clause is eligible for sharing by an incarnation if the LBD score of this clause in the incarnation is lower than a fraction \( \delta \) of the global LBD average of the incarnation. In PCASSO, we use \( \delta = 0.5 \).

PCASSO uses different restart policies and different clause cleaning policies for the nodes, depending whether the node is root, leaf or middle (not root and not leaf). PCASSO can have a scenario where there is only one unsolved node at some partition level. We call this scenario the only child scenario. Consider that if the only child scenario happens at some level of the partition tree, then there are two cases: i) the parent node is looking into the search space which has been solved by one of its children already, ii) the parent node is looking into the same search space where its unsolved children are looking. In either case, we have the risk of doing redundant work. We propose an approach to get out of this
scenario by reintroducing the solving limit in a node that has only one unsolved child (AVOID). To be on safe side, we do not apply this limit for the root node. The introduced limit grows with the level of the node (level \( + 4096 \) conflicts). Since in the only child scenario all learned clauses can be shared among the two participating nodes, we can also exploit this situation, by enabling this sharing. In the extreme case, this configuration is very similar to portfolio solvers, where all clauses can be shared without restrictions. When clauses are tagged by position-based tagging [5], additional information can be obtained by performing a conflict analysis on solved unsatisfiable nodes. Consider a node \((F^p, \bot, \blacksquare)\), and let \( \{\}^q \) be the empty clause derived by the incarnation that solved \(F^q\). Then, from the main theorem in [5], we conclude that \( \{\}^q \) is the semantic consequence of the node of position \(q\) in the partition tree. Observe that \(q\) is a prefix \(p: q \leq p\). Consequently, not only the node at position \(p\) can be marked as unsatisfiable, but also the node \(F^q\) as well as all its child nodes. As a result, more incarnations can be terminated and start solving different partitions. We call this kind of technique *conflict driven node killing*. A similar approach is reported in [7].

### III. MAIN PARAMETERS

The major parameters of the solver specify the number of threads that should be used, the number of partitions that should be created for each node, and how sharing should be performed. For the competition, we use 12 threads, and produce 8 partitions. Furthermore, we share learned clauses according to their LBD value. Finally, the treatment of the only-child scenario can be specified as well.

There are only minor magic constants that control the run time of the look-ahead procedures that are applied during partitioning, whose values have been chosen according to the literature [4].

### IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

Each node in the partition tree is associated to a pool of shared clauses, where a pool is implemented as a vector of clauses. This permits to decouple the life of a shared clause from the life of the incarnation where the shared clause has been learned. Instead of tagging each clause with a position, clauses are tagged with integers representing a level in the partition tree (root node has level zero). Observe that this is sufficient to simulate the position based approach —that is, each incarnation working over a node \(F^p\) can only access the pools placed at nodes of positions \(q \leq p\). Concurrent access to pools is regulated by standard POSIX Read-Write locks. COPROCESSOR is used as preprocessor [8].

### V. IMPLEMENTATION DETAILS

PCASSO is built on top of a simplified version of RISS without inprocessing and without the modifications to the CDCL algorithm, so that the implementation of clause sharing remains as simple as possible. The formula simplifier COPROCESSOR is used as an external tool before PCASSO is executed.

### VI. AVAILABILITY

The source code of PCASSO and COPROCESSOR is available for research at tools.computational-logic.org.

### REFERENCES

PeneLoPe in SAT Competition 2014

Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, Cédric Piette
Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{audemard,hoessen,jabbour,lagniez,piet}@cril.fr

Abstract—This paper provides a short system description of our updated portfolio-based solver called PeneLoPe, based on ManySat. Particularly, this solver focuses on collaboration between threads, providing different policies for importing and importing learnt clauses between CDCL searches. Moreover, different restart strategies are also available, together with a deterministic mode.

I. OVERVIEW

PeneLoPe [2] is a portfolio parallel SAT solver that uses the most effective techniques proposed in the sequential framework: unit propagation, lazy data structures, activity-based heuristics, progress saving for polarities, clause learning, etc. As for most of existing solvers, a first preprocessing step is achieved. For this step -which is typically sequential- we have chosen to make use of SatElite [6].

In addition, PeneLoPe includes a recent technique for its learnt clause database management. Roughly, this approach follows this schema: each learnt clause \( c \) is periodically evaluated with a so-called \( psm \) measure [3], which is equal to the size of the set-theoretical intersection of the current interpretation and \( c \). Clauses that exhibit a low \( psm \) are considered relevant. Indeed, the lower, the higher the related clause is about to unit-propagate some literal, or to be falsified. On the opposite, a clause with a large \( psm \) value has a lot of chance to be satisfied by many literals, making it irrelevant for the search in progress.

Thus, only clauses that exhibit a low \( psm \) are selected and currently used by the solver, the other clauses being frozen. When a clause is frozen, it is removed from the list of the watched literals of the solver, in order to avoid the computational over-cost of maintaining the data structure of the solver for this useless clause. Nevertheless, a frozen clause is not erased but it is kept in memory, since this clause may be useful in the next future of the search. As the current interpretation evolves, the set of learnt clauses actually used by the solver evolves, too. In this respect, the \( psm \) value is computed periodically, and sets of clauses are frozen or unfrozen with respect to their freshly computed new value.

Let \( P_0 \) be a sequence where \( P_0 = 500 \) and \( P_{i+1} = P_i + 500 + 100 \times i \). A function "updateDB" is called each time the number of conflict reaches \( P_i \) conflicts (where \( i \in [0, \infty) \)). This function computes new \( psm \) values for every learnt clauses (frozen or activated). A clause that has a \( psm \) value less than a given limit \( l \) is activated in the next part of the search. If its \( psm \) does not hold this condition, then it is frozen.

Moreover, a clause that is not activated after \( k \) (equal to 7 by default) time steps is deleted. Similarly, a clause remaining active more than \( k \) steps without participating to the search is also permanently deleted (see [3] for more details).

Besides the \( psm \) technique, PeneLoPe also makes use of the \( lbd \) value defined in [4]. \( lbd \) is used to estimate the quality of a learnt clause. This new measure is based on the number of different decision levels appearing in a learnt clause and is computed when the clause is generated. Extensive experiments demonstrates that clauses with small \( lbd \) values are used more often than those with higher \( lbd \) ones. Note also that \( lbd \) of clauses can be recomputed when they are used for unit propagations, and updated if it becomes smaller. This update process is important to get many good clauses.

Given these recently defined heuristic values, we present in the next Section several strategies implemented in PeneLoPe.

II. DETAILED FEATURES

PeneLoPe proposes a certain number of strategies regarding importation and exportation of learnt clauses, restarts, and the possibility of activating a deterministic mode.

Importing clause policy: When a clause is imported, we can consider different cases, depending on the moment the clause is attached for participating to the search.

- no-freeze: each imported clause is actually stored with the current learnt database of the thread, and will be evaluated (and possibly frozen) during the next call to updateDB.
- freeze-all: each imported clause is frozen by default, and is only used later by the solver if it is evaluated relevant w.r.t. unfreezing conditions.
- freeze: each imported clause is evaluated as it would have been if locally generated. If the clause is considered relevant, it is added to the learnt clauses, otherwise it is frozen.

Exporting clause policy: Since PeneLoPe can freeze clauses, each thread can import more clauses than it would with a classical management of clauses, where all of them are attached. Then, we propose different strategies, more or less restrictive, to select which clauses have to be shared:

- unlimited: any generated clause is exported towards the different threads.
- size limit: only clauses whose size is less than a given value (8 in our experiments) are exported [8].
- lbd limit: a given clause \( c \) is exported to other threads if its \( lbd \) value \( lbd(c) \) is less than a given limit value \( d \) (8...
by default). Let us also note that the \( lbd \) value can vary over time, since it is computed with respect to the current interpretation. Therefore, as soon as \( lbd(c) \) is less than \( d \), the clause is exported.

**Restarts policy**: Beside exchange policies, we define two restart strategies.

- **Luby**: Let \( l_i \) be the \( i \)th term of the Luby serie. The \( i \)th restart is achieved after \( l_i \times \alpha \) conflicts (\( \alpha \) is set to 100 by default).
- **LBD** [4]: Let \( LBD_d \) be the average value of the LBD of each learnt clause since the beginning. Let \( LBD_{100} \) be the same value computed only for the last 100 generated learnt clause. With this policy, a restart is achieved as soon as \( LBD_{100} \times \alpha > LBD_d \) (\( \alpha \) is set to 0.7 by default). In addition, the VSIDS score of variables that are unit-propagated thank for a learnt clause whose \( lbd \) is equal to 2 are increased, as detailed in [4].

Furthermore, we have implemented in PeneLoPe a deterministic mode which ensures full reproducibility of the results for both runtime and reported solutions (model or refutation proof). Large experiments show that such mecanism does not affect significantly the solving process of portfolio solvers [7]. Quite obviously, this mode can also be unactivated in PeneLoPe.

## III. Fine Tuning Parameters of PeneLoPe

PeneLoPe is designed to be fine-tuned in an easy way, namely without having to modify its source code. To this end, a configuration file (called `configuration.ini`, an example is provided in Figure 1) is proposed to describe the default behavior of each thread. This file actually contains numerous parameters that can be modified by the user before running the solver. For instance, besides export, import and restart strategies, one can choose the number of threads that the solver uses, the \( \alpha \) factor if the Luby techniques is activated for the restart strategy, etc. Each policy and/or value can obviously differ from one thread to the other, in order to ensure diversification.

**Acknowledgment**

PeneLoPe has been partially developed thank to the financial support of CNRS and OSEO, under the ISI project “Pajero”.

**References**

Abstract—This document describes the SAT solver “pGlucose”, a new kind of parallel hybrid solver, combining portfolio with divide and conquer techniques.

I. Solver Description

The presented parallel solver is based on an existing implementation of a CDLC like solver, namely glucose. It is a multi-threaded solver combining portfolio with divide and conquer techniques.

The solving process starts by setting up a portfolio phase. This phase consists in $T$ threads that execute the same CDCL algorithm, but start at different decision points. The threads exchange some of their learned clauses, through a shared non blocking database. The shared clauses are chosen with respect to the Literals Blocks Distance (LBD) criterion defined in [1]. The portfolio phase stops either by answering the problem (SAT/UNSAT) or by meeting the switch criterion indicating that the divide and conquer phase must start. This is reached when $T_{q}$ ($\leq T$) threads learn $p$ percent of new clauses (with respect to the number of original clauses).

The divide and conquer phase then starts when the portfolio fails to answer the problem. It uses in a global sub-problems queue ($Q$), and $T$ threads trying to resolve the sub-problems stored in $Q$, plus a controller thread. Initially, $Q$ contains only one problem, the original one.

While no answer is found, each idle thread will try to pick a problem from $Q$. If a thread finds an other remaining thread in an idle state, it decomposes the picked problem in $n$ sub-problems (where $n$ is a parameter) and push $n − 1$ of them back into $Q$. This is used to ensure a workload equilibrium between the threads during the whole resolution process.

Each thread can perform a local restart according the strategy defined in the underling sequential solver (i.e., glucose). A global restart is scheduled every $2^{r}\times baseline$ loop of the controller thread, where $r$ is the actual number of global restarts and $baseline$ is constant.

The first thread that answers the problem will kill all the other threads.

At the beginning of the divide and conquer phase, and in order to capitalise on the portfolio phase, the thread with the $best$ progress is taken as a reference, and its configuration is copied into all other $T−1$ threads (the most important point here is to copy the heap of the literals).

II. Main Parameters

By default, the parameters discussed above are set to the following values:

1) $T$: is set to the number of cores in the underling machine (automatically detected).
2) $LBD$: is set to 2.
3) $T_{q}$: is set to $\frac{2}{3} + T$.
4) $n$: is set to 2.
5) $p$: is set to 5%.
6) $baseline$: is set to 10000.

III. Implementation Details

1) The programming language used is C++.
2) The solver is based on glucose 2.3 with the additional features explained above.

IV. SAT Competition 2014 Specifics

1) The solver is submitted in “Parallel, Hard-combinatorial SAT+UNSAT” and “Parallel, Random SAT” traks.
2) The used compiler is g++.
3) The optimization flag used is -O3. The compilation options are the same as the used existing solver.

V. Availability

Our solver is not yet publicly available.

ACKNOWLEDGEMENTS

We would like to thank the authors of glucose for making available the source code of their solver.

REFERENCES


\textsuperscript{2}We take the best ratio between the number of assigned variables and the total number of variables of all threads.
pmcSAT

Ricardo Marques, Luis Guerra e Silva, Paulo Flores and L. Miguel Silveira
INESC-ID / IST-TU Lisbon
Rua Alves Redol, 9, 1000-029 Lisbon, Portugal
{rsm,lgs,pff,lms}@algos.inesc-id.pt

Abstract—This document describes the SAT solver pmcSAT, a conflict-driven clause learning (CDCL) portfolio solver that launches multiple instances of the same basic solver using different heuristic strategies, for search-space exploiting and problem analysis, which share information and cooperate towards the solution of a given problem.

I. INTRODUCTION

pmcSAT is a portfolio-based multi-threaded, multi-core SAT solver, built on top of GLUCOSE [2], a state-of-the-art SAT sequential SAT Solver. The general strategy pursued in pmcSAT is to launch multiple instances of the same solver, with different parameter configurations, which cooperate to a certain degree by sharing relevant information when searching for a solution. This approach has the advantage of minimizing the dependence of current SAT solvers on specific parameter configurations chosen to regulate their heuristic behavior, namely the decision process on the choice of variables, on when and how to restart, on how to backtrack, etc.

II. MAIN TECHNIQUES

The solver uses multiple threads (twelve currently), which explore the search space independently, following different paths, due to the way each thread is configured.

In order to ensure that each thread follows divergent search paths, we defined distinct priority assignment schemes, one for each thread of pmcSAT. Note that the priority of a variable will determine its relative assignment order.

Below are described the different priority schemes that were used.

- **Thread #0** - All the variables have the same priority, therefore this thread mimics the original VSIDS heuristic.
- **Thread #1** - The first half of the variables read from the file have higher priority than the second half.
- **Thread #2** - The first half of the variables read from the file have lower priority than the second half.
- **Thread #3** - The priority is sequentially decreased as the variables are read from the file.
- **Thread #4** - The priority is sequentially increased as the variables are read from the file.
- **Thread #5** - The priority is sequentially decreased in chunks as the variables are read from the file.
- **Thread #6** - The priority is sequentially increased in chunks as the variables are read from the file.
- **Thread #7** - Higher priority is alternately given as the variables are read from the file.
- **Thread #8** - The priority is increased according to its number of occurrences in the file.
- **Thread #9** - The priority is decreased according to its number of occurrences in the file.
- **Thread #10** - The priority is decreased according to the number of variables that have the same number of common variables.
- **Thread #11** - The priority is increased according to the number of variables that have the same number of common variables.

Although each pmcSAT thread exploits independently the search space, this is not just a purely competitive solver. All the threads cooperate by sharing the learnt clauses resulting from conflict analysis, leading to a larger pruning of the search space.

To reduce the communication overhead introduced by clause sharing, and its overall impact in performance, we designed data structures that eliminate the need for read and write locks. These structures are stored in shared memory, which is shared among all threads.

Each thread owns a queue, where the clauses to be shared are inserted. Associated to this queue is a pointer, which marks the last inserted clause, manipulated by the source thread, while every other targeted thread owns a pointer that indicates the last read clause from the queue. Therefore, this data structure eliminates the need for a locking mechanism.

A more detailed explanation of the techniques used in this solver can be found in [3].

III. MAIN PARAMETERS

The internal parameters of pmcSAT are the same as in GLUCOSE 3.0, with the addition of the following:

- The learnt clauses size condition to be exported. For the SAT competition the clause size limit was set to 8, i. e., only learnt clauses with less than 8 literals are exported and shared with other threads.

IV. IMPLEMENTATION DETAILS

1) The programming language used is C++, using pthread for parallel computing.
2) The solver was implemented on top of GLUCOSE 3.0.

V. SAT COMPETITION 2013 SPECIFICS

1) The solver was submitted to all Parallel Tracks: Application SAT+UNSAT, Hard-Combinatorial SAT+UNSAT, Random SAT and Open Track.
2) The compiler used is g++.
3) The optimization flag used is "-O3"
4) 64-bit binary.
5) The only command-line parameter is the input file

VI. AVAILABILITY

More information about the pmcSAT solver, including its source code, can be found on the ALGOS research group publicly available website:

http://algos.inesc-id.pt/algos/software.php

ACKNOWLEDGMENT

The authors would like to thank the authors of GLUCOSE and MiniSAT for making available the source code of their solvers.

This work was partially supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under project “ParSat: Parallel Satisfiability Algorithms and its Applications” (PDTC/EIA-EIA/103532/2008) and project PEst-OE/EEI/LA0021/2011.

REFERENCES

probSAT and pprobSAT
Adrian Balint
Ulm University
Ulm, Germany
Uwe Schöning
Ulm University
Ulm, Germany

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

I. INTRODUCTION

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

II. MAIN TECHNIQUES

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

Algorithm 1: ProbSAT

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

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

\[
\begin{align*}
    f(x,a) &= (c_b)^{-\text{break}(x,a)} \\
    f(x,a) &= (\epsilon + \text{break}(x,a))^{-c_b}
\end{align*}
\]

III. PARAMETER SETTINGS

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

<table>
<thead>
<tr>
<th>( k )</th>
<th>( fct )</th>
<th>( cb )</th>
<th>( \epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>2.06</td>
<td>0.9</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3.88</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4.6</td>
<td>-</td>
</tr>
<tr>
<td>( \geq 7 )</td>
<td>1</td>
<td>4.6</td>
<td>-</td>
</tr>
</tbody>
</table>

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

IV. PPROB SAT

The solver pprobSAT starts \( n \) instances of probSAT in parallel and returns once one of the solvers have found a solution. The last two instantiations of probSAT use restarts after \( 10^7 \) flips, \( 10^8 \) flips respectively.

V. FURTHER DETAILS

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

The solver probSAT is submitted to the sequential Random SAT track. The solver pprobSAT is submitted to the Parallel Random SAT track. The compile flags are: -Wall -Wextra -static -O3 -funroll-loops -fexpensive-optimizations.

The solvers will be available online1

ACKNOWLEDGMENT

This project was funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 3029-1.

REFERENCES


1https://github.com/adrianopolus/probSAT
Abstract—In this paper we present our implementation of our preprocessing technique of Cluster Vivification, which tries to find conflicts in a subset of clauses from the instance. This implementation is based on miniSat. After the preprocessing we pass the resulting instance to glucose2.1 for solving.

I. INTRODUCTION

In an effort to reduce the number of conflicts of a given instance we built the preprocessor provoSATeur which implements Cluster Vivification. These clusters are built upon a connected subset of clauses and with that may represent a subproblem. This aims at learning new clauses prior to the solving process by finding conflicts in cluster. Conflicts in these clusters are not influenced by the rest of the instance and therefore valid for the whole instance.

After the preprocessor added these new clauses to the instance this is then passed to a solver of choice. In our case we prefer the unmodified solver glucose2.1[1] from the satcompetition 2012.

II. MAIN TECHNIQUES

In clause vivification[2] a literal from a chosen clause are consequently with its negating value, trying to provoke a conflict.

We expanded this method to not only work on one clause but to employ this technique on a number of clauses making up a subinstance. For this we select a connected set of clauses for a subinstance \( I_s \subseteq I \) from instance \( I := \bigwedge c \in C \), where clause \( c := \bigvee l \in L \) with Literal \( l \rightarrow v, \neg v \) for variable \( v \rightarrow \top, \bot \).

We then select literals which are then set to a negating value. This is repeated until either a conflict occurs or there are no unsatisfied clauses with more than one unset literals left.

If a conflict occurs, this is saved to the instance as new learnt clause.

We call this technique cluster vivification, as it vivifies not only depending on a single clause but on a number of clauses which are connected.

After the conflicts are added to the instance, the instance is written to a file and passed to the solver.

III. MAIN PARAMETERS

Besides the usual miniSat2[3] parameters we added the following user controllable parameter to our preprocessor, \( -\text{iterations=XX} \). This parameter allows the user to set the number of iterations, default value is 200.

There are other non-controllable parameters. These are set to what we experienced as good values in development. These are:

1) Number of clauses chosen in the subinstance, this is set to ten.
2) The timeout is set to stop the process after a maximum of five minutes

IV. IMPLEMENTATION DETAILS

provoSATeur uses the C++ Framework of miniSat2\(^1\), which we expanded to fit our needs. We needed to have a solver which tests all possible variable settings for a given mode. As heuristic for the selected variable to build the subinstance we select the variable with the the highest occurrence. Other heuristics are possible, but not yet implemented or selectable. Furthermore our preprocessor does not solve the instance by itself yet. The instance has to be passed to an other solver. The timeout usually occurs before the number of iterations is reached, depending on the size of the instance.

V. SAT COMPETITION 2014 SPECIFICS

We submitted our preprocessor with glucose to the Application SAT+UNSAT and Hard Combinatorical SAT+UNSAT track. For the submission we have chosen to set the parameters for the iterations to 200. Glucose2.1 is run with all the default values.

VI. AVAILABILITY

The sources and solver description can be found at our website\(^2\) The license is derived from miniSat.

REFERENCES


\(^1\)http://www.minisat.se
\(^2\)http://www-ti.informatik.uni-tuebingen.de/~burg/provoSATeur.php
Abstract—The solver RISS combines the improved Minisat-style solving engine of GLUCOSE 2.2 with a state-of-the-art preprocessor COPROCESSOR and adds further modifications to the search process.

I. INTRODUCTION

The CDCL solver RISS in version 4.27 was first build on the MINISAT search engine [1], and next incorporated the improvements that have been proposed for GLUCOSE 2.2 [2], [3]. Afterwards, more search algorithm extensions have been added. RISS is equipped with the preprocessor COPROCESSOR [4], that implements most of the recently published formula simplification techniques, ready to be used as processing as well by taking care of learned clauses. The aim of the solver is to provide a huge portfolio of options on the one hand, to be able to adopt to new SAT applications, with an efficient implementation on the other hand.

II. MAIN TECHNIQUES

RISS uses the modifications of the CDCL algorithm that are proposed in [5], namely local look-ahead, all-unit-UIP learning and on-the-fly probing. An additional learned clause minimization technique, based on hidden literal elimination is added. Another addition to the solver is the ability to perform restricted extended resolution (RER) during search as described by Audemard et al. [6]. However, the used implementation differs to reduce the overhead of the procedure: introduced variables are not deleted, and new learned clauses are only rewritten with the most recent extension of a literal. During unit propagation, lazy hyper binary resolution [7] can be used, and on-the-fly clause improvement [8] is available during conflict analysis. During learning, usual first-UIP clauses can be learned [9], but alternatively the first bi-asserting clause can be learned instead [10]. For large learned clauses with a high LBD value, a so called decision clause is learned, which simply negates the current decision literals [11], because this clause is assumed to be removed from the formula again soon. The activity decay for variables that are used in clause learning steps can be adopted dynamically, similarly to the method used in GLUCOSE 2.3 [12]. When learned clauses are removed again, then the set of the heuristically worst clauses can be kept. Finally, the algorithm proposed by Wieringa [13] to strengthen learned clauses in parallel to the actual CDCL search has been sequentialized, so that search can be interleaved with such an clause strengthening technique.

For the initialization of the activities of variables, as well as the polarities for polarity caching [14], several opportunities are offered, among them the Jeroslow-Wang heuristic [15]. For scheduling restarts, the scheme presented for GLUCOSE 2.2 is used by default, but also the geometric series of MINISAT can be used [1], or the LUBY series [16]. Partial restart are also available [17].

The built-in preprocessor COPROCESSOR incorporates the following formula simplification techniques: Unit Propagation, Subsumption, Strengthening (also called self-subsuming resolution) – where for small clauses all subsuming resolvents can be produced, (Bounded) Variable Elimination (BVE) [18] combined with Blocked Clause Elimination (BCE) [19], (Bounded) Variable Addition (BVA) [20], Probing [21], Clause Vivification [22], Covered Clause Elimination [23], Hidden Tautology Elimination [24], Equivalent Literal Substitution [25], Unhiding (Unhide) [26], Adding Binary Resolvents [27], a 2SAT algorithm [28], and a walksat implementation [29]. The preprocessor furthermore supports parallel subsumption, strengthening and variable elimination [30].

Since the last version of RISS [31], the following simplification techniques have been added: The implementation of Unhide now supports finding equivalent literals and can remove redundant binary clauses. Structural hashing is performed during equivalent literal elimination [32], and the Tarjan algorithm to find strongly connected components in the binary implication graph is now implemented iteratively. The Fourier-Motzkin method (FM) for reasoning on cardinality constraints, similar to the procedure by Biere [11], is used as a preprocessing step, where the cardinality constraints can be extracted syntactically for degrees less than 3, and for higher degrees a semantic method based on unit propagation is used [33]. Furthermore, an algorithm is embedded, which is able to derive new clauses from a special set of cardinality constraints: when at-most-one constraints for rows, and at-least-one constraints for columns encode a possibly over constraint matching problem, then the at-least-one constraints for the rows can are deduced and added in form of clauses. Another deduction systems that is stronger than resolution is available in COPROCESSOR: by retrieving XOR gates from the CNF, with the help of the Gaussian elimination equivalent literals and unit clauses can be found. The elimination of resolution asymmetric tautologies is implemented in a first naive version, which performs the same steps as the theoretical description: building the resolvent and testing this resolvent for being an asymmetric tautology by running full unit propagation [34]. For an extension \( x \leftrightarrow (a \land b) \), BVA also replaces the disjunction \((a \lor b)\) in the formula with \(\tau\), and adds the full extension. Furthermore, BVA can be search for
other gate types than AND-gates: XOR-gates and If-THEN-ELSE-gates can be used additionally. Finally, the counter technique of covered literal addition [23], which is used to eliminate covered clauses, is added to COPROCESSOR: during the computation of BCE, covered literal elimination (CLE) is performed [35], which removes literals from a clause, if these literals can be added by covered literal addition again [23].

RISS is able to output proofs for unsatisfiable formulas in the DRUP format [36], [37], also when look-ahead or the all-units-learning modifications are enabled. Furthermore, most of the techniques inside COPROCESSOR are able to produce proofs. If a technique introduces fresh variables or cannot be simulated by unit propagation (easily), DRAT proofs are printed instead [38], as required for example by RER, BVA or CLE. Inside the solver the generated proof can be verified during its construction. Techniques that do not support producing proofs yet are Gaussian elimination and FM. Furthermore, for some probing based simplification techniques proofs are not produced. For more details on the generation of proofs and simplification techniques see [35].

III. MAIN PARAMETERS

RISS offers all the parameters that are available in GLUCOSE 2.2. Furthermore, all the techniques that are mentioned above can be enabled or disabled, and the number of execution steps per technique can be limited, as well as variants can be produced. The total number of parameters for the solver is 486, where 190 of these parameters are Boolean, and the remaining parameters have either floating point or integer domains.

For the SAT Competition 2014 the formula size limits for the formula simplification techniques have been set, so that these techniques do not consume too much run time. Next, a set of well performing techniques was determined by a local-search like selection procedure. Based with this configuration, the parameters for the search algorithm have been tuned. Finally, the search configuration has been combined with the techniques of the formula simplification.

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

In GLUCOSE 2.2, binary clauses are handled specially during propagation: both literals of the clause can be retrieved from the watch list, so that the actual clause is not touched. Therefore, GLUCOSE 2.2 introduces an extra watch list for binary clauses. RISS keeps all clauses in a single watch list, but applies that same idea by modifying the elements in the watch list so that they know whether the watched clause is binary or not. This modification reduces the memory consumption of the solver. Another reduction of the memory consumption is achieved by merging multiple bit arrays into a single array and using bit operations. Especially when new variables are introduced by BVA or RER, the memory fragmentation is lower with this modification.

V. IMPLEMENTATION DETAILS

RISS and COPROCESSOR are implemented in C++. All simplification techniques inside COPROCESSOR are implemented in separate classes to increase the structure of the code.

VI. SAT COMPETITION 2014 SPECIFICS

RISS is submitted as a 64-bit binary to the SAT and SAT+UNSAT tracks for the categories Application and Crafted. The compilation uses the flag “-O3”.

The submitted configuration of RISS 4.27 uses the following techniques, where FM and variable renaming is disabled for the certified unsatisfiable tracks: BVE, FM, five iterations of UNHIDE, CLE and variable renaming to compact the representation of the formula during search.

Furthermore, the version DRAT uses the same configuration as RISS 3.6 [31] for all application tracks and crafted tracks, because the used techniques BVE, BVA, Unhide, and Local Look-Ahead, support the DRAT proof format.

VII. AVAILABILITY

RISS, as well as the formula simplifier COPROCESSOR are available for research. The collection additionally contains the parallel search space splitting SAT solver PCASSO, the portfolio SAT solver PRISS that can produce DRUP proofs [39], and a CNF feature extraction. The framework can be downloaded from http://tools.computational-logic.org.

ACKNOWLEDGMENT

The author would like to thank the developers of GLUCOSE 2.2 and MINISAT 2.2. Furthermore, many thanks go to Marijn Heule for discussions on the proof formats DRUP and DRAT, as well as to Armin Biere for discussions on formula simplification techniques. The computational resources to develop, evaluate and configure the SAT solver have been provided by the BWGrid [40] project and the ZIH of TU Dresden.

REFERENCES


[40] bwGRID (http://www.bw grid.de/), “Member of the german d-grid initiative, funded by the ministry of education and research (bundesministerium für bildung und forschung) and the ministry for science, research and arts baden-württemberg (ministerium für wissenschaft, forschung und kunst baden-württemberg),” Universities of Baden-Württemberg, Tech. Rep., 2007-2010.


Riss 4.27 BlackBox

Enrique Matos Alfonso and Norbert Manthey
Knowledge Representation and Reasoning Group
TU Dresden, Germany

Abstract—The solver Riss BlackBox uses the highly configurable SAT solver RISS in version 4.27 as the base solver and selects a configuration per input formula. For this process, a large set of CNF features is computed. Then, a set of random decision forests, one forest for each configuration, selects a predefined configuration of the solver.

I. INTRODUCTION

Motivated by the tool SATZILLA [1] and by the fact that application formulas and crafted formulas contain structure we decided to classify the formulas and map the classes to solver configurations. With RISS in version 4.27 [2] a solver is available that is not only competitive in its default configuration, but that furthermore offers plenty of techniques that are especially good in solving formulas that cannot be solved by the robust default configuration. However, these specialized techniques might consume too much run time on too many formulas, so that choosing among the available techniques or solver configurations is the better choice. A more detailed explanation is provided in [3].

II. MAIN TECHNIQUES

The configurations of RISS have been preset, such that the combined solver can solve a huge amount of formulas in a timeout of one hour on the used computing resources.

The extracted features originate from sequences that can be extracted from the degrees of the nodes in a graph. The used graphs are for example the clause-variable graph, the variable-clause graph (both for positive and negative literals). Furthermore, we consider the binary implication graph, and the graphs that are build based on AND-gates, partially represented AND-gates as well as EXACTLY ONE-constraints. Further sequences are created based on the clause size, the RW-heuristic [4], [5], and a symmetry approximation similar to the coloring idea in [6]. Then, for each sequence the mean, minimum, max, standard deviation, the value rate, mode and the entropy is considered as a feature. Furthermore, we use the values of the 25 % and the 75 % quantile. Finally, for each sequence we compute its derivation, and use the same statistical values of the derivation as features as well. Instead of measuring the run time to construct each feature, we use counters that are incremented for each major calculation step, so that their value stays independent from the used architecture, but correlate with the run time.

With these features we trained a classifier that returns the most promising configuration. The classifier uses a random decision forest for each configuration, and returns the configuration where the probability of an correct answer is most likely. If no configuration is predicted by the classifier, the configuration that performed best on all training instances is used, because this configuration is assumed to be most robust.

III. MAIN PARAMETERS

RISS BlackBox does not offer any parameters, because the configuration of the SAT solver is chosen automatically. However, during training the classifier and for extracting the features, several options are available, namely which features to compute, how to label a configuration as good for a certain configuration, and how to set up and train the classifier.

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

The implementation of the feature extraction aims at being as fast as possible. Therefore, we trade space for time by not checking the duplicate entries in the adjacency lists of the constructed graphs. Only after the graph has been constructed completely we eliminate these duplicates by sorting the list and iterating over the list exactly once. This way, much more memory is used, but only a single cleanup and sort operation is required for each list.

V. IMPLEMENTATION DETAILS

The feature extraction, as well as the communication with the classifier is implemented into RISS. For the machine learning part we use WEKA [7] as external tool, which is not part of the solver framework itself.

VI. SAT COMPETITION 2014 SPECIFICS

RISS BlackBox is submitted as a 64-bit binary to the SAT and SAT+UNSAT tracks for the categories Application and Crafted. The extracted features do not include the graph based features of the clause graph, the resolution graph, and the clause-variable graph. The compilation uses the flag “-O3”.

Since not all techniques of RISS are able to produce DRAT proofs [7], for the certified unsatisfiability tracks these configurations are excluded from the portfolio. Since RISS default configuration is also not able to produce DRAT proofs, the used fall-back configuration is the most robust configuration which can produce DRAT proofs.

VII. AVAILABILITY

The feature extraction and classification code is part of RISS, which is available for research. The tool can be downloaded from http://tools.computational-logic.org. The machine learning tool WEKA is not part of the framework, because it is available under the GPL.
ACKNOWLEDGMENT

The authors would like to thank the developers of WEKA for providing an easy entry to machine learning. Furthermore, the authors thank the ZIH of TU Dresden for providing the computational resources for setting up and evaluating the solver configurations for this solver, and training the classifier.

REFERENCES

I. INTRODUCTION

ROKK is a SAT solver based on MiniSat2.2.0[1]. ROKK add two new strategies to MiniSat. One is a learnt clause management strategy. It manages number of learnt clause using Periodic ReduceDB strategy and compares learnt clauses with TrueLBD which is a kind of LBD and Linear Activity. The other is a restart strategy which mixes luby Restart and LBD+DLV Restart.

II. LEARNT CLAUSE MANAGEMENT

SAT solver make a lot of learnt clauses while it’s running. Number of learnt clauses increase rapidly. Learnt clauses are absolutely essential for solving problems but too many learnt clauses make solver slow.

So, strong SAT solver has good clause management system. For example, MiniSat remove almost half learnt clauses in ReduceDB method. Interval of each ReduceDB increase exponentially. In the other hand, ROKK’s approach is simple. Remove all unnecessary clauses as soon as possible so SAT solver can be faster and faster.

A. Periodic ReduceDB

ROKK implement an aggressive removing clauses strategy called Periodic ReduceDB. It call ReduceDB method at fixed intervals. ReduceDB method is a heavy method so this interval become 10,000. (this is still very short.)

B. Criterion of learnt clause

ROKK combine two criteria of clause importance, True LBD and Linear Activity. ROKK regard a clause as important when both its TrueLBD and Linear Activity are low. It means even if its TrueLBD is 2 (it means best value), it will be removed when its Linear Activity is very high and vice versa.

1) True LBD: True LBD(TLBD) is a kind of LBD[?]. TLBD is different from LBD in the manner of updating its value. It ignores literals assinged at level 0. And ROKK use two kind of TLBDs. One is Newest TLBD which value is the latest TLBD value in propagation. The other is Initial TLBD, the first value when a clause is generated.

2) Linear Activity: Linear Activity of a clause show how many conflicts happened after last propagation of the clause.

III. RESTART STRATEGY

ROKK’s learnt clause management strategy is very strong, but it is specialized in SATISFIABLE problems. So, ROKK need some essences to solve more UNSATISFIABLE problems. Then, ROKK mixes two restart strategies, luby Restart and LBD+DLV Restart.

IV. PREPROCESSING

ROKK use SatElite[4] to simplify the clauses.

ACKNOWLEDGMENT

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

REFERENCES

[5] Takeru Yasumoto, Takumi Okugawa. ZENN0.1.0, SAT 2013 Competition System Description.
ROKKminisat

Takeru Yasumoto
yasumoto.kyushu@gmail.com
Takumi Okugawa
okugawa.takumi@gmail.com

I. INTRODUCTION

ROKKminisat is based on MiniSat 2.2.0[1]. The ROKKminisat adds one learnt clause management strategy to MiniSat. It manages number of learnt clause with Periodic ReduceDB strategy and compares learnt clauses with TrueLBD which is a kind of LBD and Linear Activity.

II. MAIN TECHNIQUES

A. Periodic ReduceDB

SAT solver make a lot of learnt clause while it’s running and number of learnt clauses increase rapidly. Learnt clauses are important for solving problems but too many learnt clauses make solver very slow. So, SAT solver need to remove them when they are too much.

ROKKminisat implement an aggressive removing clauses strategy called Periodic ReduceDB. This approach is very simple. It call remove clauses which is not important at fixed intervals (each 10,000 conflicts happened).

B. Criterion of learnt clause

ROKKminisat combine two criteria of clause importance, True LBD and Linear Activity. ROKKminisat judge a clause is important when both its TrueLBD and Linear Activity are low. And even if its TrueLBD is 2 (it means best value), it will be removed when its Linear Activity is very high.

1) True LBD: True LBD is a kind of LBD[2]. True LBD is different from LBD in the manner of updating its value. It ignores literals assinged at level 0.

2) Linear Activity: Linear Activity of a clause show how many conflicts happened after last propagation of the clause.

ACKNOWLEDGMENT

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

REFERENCES

Abstract—This document describes the SAT solver RSeq2014, a two-engine SAT solver based on Sattime2013 and Relback.

I. INTRODUCTION

RSeq2014 is a two-engine SAT solver which combines Sattime2013 with Relback. Sattime2013 is a Stochastic Local Search (SLS) algorithm based on Sattime [1]. In the SAT Challenge 2012, Sattime2012 was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [2]. Sattime2013 was the second best solver in the random SAT category of the SAT Competition 2013 [3].

Relback is a CDCL-based solver due to D. Habet and C.M. LI, which is implemented by modifying the backtracking of the Glucose solver of G. Audemard and L. Simon [4]. In SAT challenge 2012, Relback was one of the best single-engine solver in Hard Combinatorial SAT+UNSAT category [2]. We believe that each solver has its own superiority in solving different problems. Sattime and Relback should be complementary to solve different problems. In order to solve a SAT instance, RSeq2014 calls Sattime and Relback sequentially: Sattime is started first with a time limit. If the time limit is exceeded and a solution is not found, Sattime will be killed then Relback is started to continue solving the simplified instance. The starting process is controlled by a Unix shell script.

II. MAIN PARAMETERS

Sattime2013 is a new version of Sattime introduced in SAT Competition 2013 [3]. Sattime2013 uses the following parameters: -cutoff a, -tries b, -seed c, -nbsol d, allowing to run b times Sattime2013 for at most a steps each time, the random seed of the first run being c, to search for d solutions of the input instance. In the version submitted to the competition, a=2000000000, b=1000, and d=1.

Relback is the same version as in SAT Challenge 2012 (see the description of Relback in [2]).

III. SAT COMPETITION 2014 SPECIFICS

RSeq2014 is submitted to the sequential SAT and SAT+UNSAT category of Application and Hard-combinatorial instances. Sattime2013 is compiled using the gcc compiler using the ”-O3 -static” flag. Relback is compiled using g++ with the optimization flag ”-O3”.

RSeq2014 should be called in the competition using:

```
./RSeq2014.sh INSTANCE -seed SEED cutofftime
```

where cutofftime is the time limit within which Sattime2013 can run. In the competition, cutofftime is equal to 1200 seconds.

REFERENCES

Description of Sattime2014r

Chu Min LI
Huazhong University of Science and Technology, China; MIS, Universit de Picardie Jules Verne, France

Chong Huang
Huazhong University of Science and Technology, China

Yu Li
Universit de Picardie Jules Verne, France

Abstract—This document describes the SAT solver “Sattime2014r”, a stochastic local search algorithm for SAT exploiting the satisfying history and the falsifying history of the unsatisfied clauses during search to select the next variable to flip in each step.

I. INTRODUCTION

Sattime [1] is a stochastic solver local search solver based on G2WSAT [2]. In SAT competition 2011, Sattime was the best mono-core solver in the crafted sat category and was ranked 4th after three portfolio solvers [3]. In SAT challenge 2012, Sattime2012 was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [4]. In SAT competition 2013, Sattime2013 was improved with more greediness to solve the crafted instances but won a silver medal in the random category [5]. Combined with the complete solver Relback, it was the second best solver in the crafted sat category (full results) [5]. Sattime2014r is a new version of Sattime improved from Sattime2013 by extending the noise mechanism of Novelty [6] to the third best variable of an unsatisfiable clause.

II. MAIN TECHNIQUES

During local search, clauses may frequently be satisfied or falsified. Modern SLS algorithms often exploit the falsifying history of clauses to select a variable to flip, together with variable properties such as score and age. The score of a variable \( x \) refers to the decrease in the number of unsatisfied clauses if \( x \) is flipped. The age of \( x \) refers to the number of steps done since the last time when \( x \) was flipped.

Novelty and Novelty based SLS algorithms such as Novelty+ [7] and Novelty++ [2] consider the youngest variable in a randomly chosen unsatisfied clause \( c \), which is necessarily the last falsifying variable of \( c \) whose flipping made \( c \) from satisfied to unsatisfied. If the best variable according to scores in \( c \) is not the last falsifying variable of \( c \), it is flipped, otherwise the second best variable is flipped with probability \( p \), and the best variable is flipped with probability \( 1-p \). TNM [8], [9] extends Novelty by also considering the second last falsification of \( c \), the third last falsification of \( c \), and so on... If the best variable in \( c \) most recently and consecutively falsified \( c \) several times, TNM considerably increases the probability to flip the second best variable of \( c \).

Another way to exploit the falsifying history of clauses is to define the weight of a clause to be the number of local minima in which the clause is unsatisfied, so that the objective function is to reduce the total weight of unsatisfied clauses.

Sattime uses a new heuristic by considering the satisfying history of clauses instead of their falsifying history, and by modifying Novelty as follows: If the best variable in \( c \) is not the most recent satisfying variable of \( c \), flip it. Otherwise, flip the second best variable with probability \( p \), and flip the best variable with probability \( 1-p \). Here, the most recent satisfying variable in \( c \) is the variable whose flipping most recently made \( c \) from unsatisfied to satisfied. The intuition of the new heuristic is to avoid repeatedly satisfying \( c \) using the same variable.

In previous versions of Sattime, as well as all Novelty-based solvers, only the best two variables in an unsatisfied clause \( c \) are considered when selecting the next variable to flip. In Sattime2014r, the third best variable is considered when the best and the second best variables are the two most recent falsifying variables of \( c \). Concretely, given a SAT instance \( \phi \) to solve, Sattime2014r first generates a random assignment and while the assignment does not satisfy \( \phi \), it modifies the assignment as follows:

1) If there are promising decreasing variables, flip the oldest one;
2) If there are enforced decreasing variables, flip the oldest one;
3) Otherwise, pick an unsatisfied clause \( c \);
4) With probability \( wp \), flip randomly a variable in \( c \); With probability \( 1-wp \), sort the variables in \( c \) according to their score (breaking tie in favor of the least recently flipped one). Consider the best, second best and the third best variables in \( c \). If the best and the second best variables are the two most recent falsifying variables of \( c \) and the best variable does not have a positive score, flip the third best variable with probability \( p' \). Otherwise, if the best variable is not the most recent satisfying variable of \( c \), then flip it. Otherwise, with probability \( p \), flip the second best variable, and with probability \( 1-p \), flip the best variable.
Probability $p$ is adapted according to the improvement in the number of unsatisfied clauses during search according to [10], and $wp=p/10$. Probability $p'=p/5$. The intuition of $p'$ is to break a possible trap formed by the best two variables in a local minimum.

The notion of promising decreasing variable was defined in [2], referring to those variables whose score is positive and became positive not by flipping themselves. For example, let $x$ be a variable and $\text{score}(x)<0$, after flipping $x$, $\text{score}(x)$ becomes positive, i.e., decreasing, then $x$ is not promising. If $\text{score}(x)\leq 0$, but after flipping another variable $y$, $\text{score}(x)$ becomes positive, $x$ is promising and will keep to be promising as long as its score is positive. Promising decreasing variables have the highest priority to be flipped.

In previous versions of Sattime, the unsatisfied clause $c$ in Step 3 is randomly selected. In Sattime2014r, it is selected using $\text{flip} \ mod \ \text{obj}$, where $\text{flip}$ is the number of flips done so far and $\text{obj}$ is the number of unsatisfied clauses. This technique is borrowed from ProbSAT (see the source code of ProbSAT in sat competition 2013).

The subscore of a variable $x$ is the increase in the number of clauses satisfied by two literals when $x$ is flipped [11]. In [11], subscores are used to break ties for $k$-SAT ($k > 3$) when several variables have the highest score. In Sattime2014r, subscores are only used for 4-SAT and 5-SAT.

III. MAIN PARAMETERS

As Sattime2013, Sattime2014r uses Hoo’s adaptive noise mechanism that uses two parameters, $\Phi$ and $\Theta$. The performance of Sattime is not very sensitive to the variation in the value of these parameters, In Sattime2014r as in Sattime2013, $\Phi=10$ and $\Theta=5$.

Other parameters include: -cutoff $a$, -tries $b$, -seed $c$, -nbSol $d$ allowing to run $b$ times Sattime2014r for at most $a$ steps each time, the random seed of the first run being $c$ to search for $d$ solutions of the input instance. In the version submitted to the competition, $a=2000000000$, $b=1000$, and $d=1$.

IV. SPECIAL ALGORITHMS, DATA STRUCTURES, AND OTHER FEATURES

Sattime2014r, as well as all the other versions of Sattime, uses the same data structures as Satz [12], [13]. It uses a preprocessing inherited from Satz to simplify the input formula by propagating all unit clauses and detecting all failed literals in the input formula, which may prove the unsatisfiability of the input instance.

V. IMPLEMENTATION DETAILS

Sattime2014 is implemented in C and is based on g2wsat [2].

VI. SAT COMPETITION 2014 SPECIFICS

Sattime2014r is an single engine solver and is submitted to all the sequential tracks on Application, Hard-combinatorial, and Random instances. Because of the preprocessing, Sattime2013 may prove the unsatisfiability of an instance.

Sattime2014r is compiled as 64-bit binary using the intel compiler as follows:

```
icc sattime2014r.c -03 -static -o sattime2014r
```

Sattime2014r should be called in the competition using sattime2014r INSTANCE -seed SEED -nbSol 1 to solve the input instance INSTANCE, where SEED can be any positive integer. If “-seed SEED” is not specified, sattime2014r also works, but it will be difficult to reproduce the same execution of sattime2014r for the input instance.

VII. AVAILABILITY

The codes sources of Sattime2014r will be available for research purpose after the competition 2014 at http://home.mis.u-picardie.fr/~cli/EnglishPage.html

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 61072353)

REFERENCES


satUZK is a conflict-driven clause learning solver for the boolean satisfiability problem (SAT).

satUZK has successfully participated in the SAT Challenge 2012 [1] and SAT Competition 2013 [2]. This solver description gives an overview of the changes from the SAT Competition 2013 version.

General changes

- We turned off our pre- and inprocessing techniques for this submission. Instead we are using SatELite [3] as an external preprocessor. We do not use any preprocessing for the certified UNSAT tracks.
- We changed our MiniSAT-like clause database reduction policy to a more aggressive one. The new deletion policy is similar to the one discussed in [4]. Inactive clauses are not deleted immediately if their LBD is low. Instead they are frozen (i.e. removed from watched lists) and can be reactivated later. Clauses are reactivated if their PSM is low, that is if they lie in the part of the search space currently processed by the solver.

Parallel version

We built a parallel version of SatUZK based on our sequential one. The parallel version runs multiple SatUZK instances concurrently and exchanges learned clauses between them.

The clause exchange policy is based on [5]. Each solver exports learned clauses if their LBD is smaller or equal 8. Imported clauses are frozen and can be activated during the usual clause database reduction runs.

In addition to clause exchange we implemented a dedicated clause reducer thread. This thread receives learned clauses from the solvers and tries to shorten them by applying a Disillation-like [6] technique. Reduced clauses are sent back to the solvers. This idea was introduced in [7].

REFERENCES

Description of SattimeGlucoseSeq

Chu Min LI∗†, Hua Jiang∗ and Ru Chu Xu∗
∗Huazhong University of Science and Technology, China
†MIS,Université de Picardie Jules Verne, France

Abstract—This document describes the SAT solver SattimeGlucoseSeq, a two-engine SAT solver based on Sattime2013 and Glucose2.3.

I. INTRODUCTION

SattimeGlucoseSeq is a two-engine SAT solver combining Sattime2013 with Glucose2.3. Sattime2013 is a Stochastic Local Search (SLS) algorithm based on Sattime2012 [1]. In SAT challenge 2012, Sattime2012 was the best local search solver in the crafted (hard combinatorial problems) category and the second best mono-core solver in the random category [2]. Glucose is a CDCL-based solver due to D. Habet and C.M. LI, which is implemented by modifying the backtracking of the Glucose solver of G. Audemard and L. Simon [3]. In SAT challenge 2012, Relback was the best single-engine solver in Hard Combinatorial SAT+UNSAT category [2].

We believe that each solver has its own superiority in solving different problems. Sattime and Glucose should be complementary to solve different problems. In order to solve a problem instance, SattimeGlucoseSeq calls Sattime and Glucose sequentially: Sattime is started first with a time limit. If the time limit is exceeded and a solution is not found, Sattime will be killed and Glucose is started to continue solving the instance. The starting process is controlled by a unix shell script.

II. MAIN PARAMETERS

Sattime2013 is a new version of Sattime. Please see the description of Sattime2013 in this book. Sattime2013 uses the following parameters: -cutoff a, -tries b, -seed c, -nbsol d, allowing to run b times Sattime2013 for at most a steps each time, the random seed of the first run being c, to search for d solutions of the input instance. In the version submitted to the competition, a=2000000000, b=1000, and d=1.

Glucose is the same version as in SAT Competition 2013. See the description of glucose in this book.

III. SAT COMPETITION 2014 SPECIFICS

SattimeGlucoseSeq is submitted to the sequential SAT and SAT+UNSAT category of Application, Hard-combinatorial and Random instances. Sattime2013 is compiled using the gcc compiler using the -O3 -static flag. Glucose is compiled using g++ with the optimization flag -O3.

SattimeGlucoseSeq should be called in the competition using:

./SGSeq.sh INSTANCE -seed SEED -tmp TMPDIR

where "cutofftime" is the time limit within which Sattime can run. "TMPDIR" is a temporary directory in which SatElite writes temporary files. In the competition, cutofftime is equal to 1000 seconds.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 61072035).

REFERENCES

SPARROWToRiss

Adrian Balint
Universität Ulm, Germany
Norbert Manthey
Knowledge Representation and Reasoning Group,
TU Dresden, Germany

Abstract—The solver SPARROWToRiss combines the SLS solver SPARROW with the CDCL solver RISS. For both solvers a separate formula simplification is executed with COPROCESSOR. Then, SPARROW is run for at most 900 seconds, or until 500 million flips are executed. If SPARROW cannot solve the formula within these limits, then the formula is passed to RISS, which tries to solve the formula.

I. INTRODUCTION

SLS solvers showed remarkable performance on the satisfiable crafted problems in the competitions from the last years. Motivated by these results we have analyzed the utility of different preprocessing techniques for the SLS solver SPARROW in [1]. The best found technique together with SPARROW represents the basis of the SLS part in our solver SPARROWToRiss. The selected configuration uses bounded variable elimination and allows to remove variables, even if the number of clauses in the formula increases. Along this fact, we enabled two more simplification techniques that reduce the number of variables without reducing the size of clauses: equivalent literal substitution [2], unhiding tautology elimination [3] and a probing approximation based on the binary implication graph [4], [5].

As SPARROW is not able to prove the unsatisfiability of a formula, we append a CDCL solver, namely RISS, after limiting the execution of SPARROW to 5 ⋅ 10^6 flips and 900 seconds CPU time. The CDCL solver RISS uses the MINISAT search engine [6], more specifically the extensions added in GLUCOSE 2.2 [7], [8]. Furthermore, RISS is equipped with the preprocessor COPROCESSOR [9], that implements most of the recently published formula simplification techniques, ready to be used as inprocessing as well. The latest version includes some new simplification techniques, as for example the Fourier-Motzkin algorithm [10], which is an approximation of the cutting planes deduction system [11], or covered literal elimination (CLE) to remove redundant literals from clauses [12]. CLE is the counter technique to covered literal addition [13].

II. MAIN TECHNIQUES AND CONFIGURATION

SPARROW is a clause weighting SLS solvers that uses promising variables and probability distribution based selection heuristics [14]. Compared to the original version, the submitted version updates weights of unsatisfied clauses in every step where no promising variable can be found. SPARROW schedules restarts along a Luby series and for each restart the smoothing parameter is changed, so that SPARROW effectively acts as a portfolio solver of different configurations where the solver itself takes care of scheduling the different strategies. This idea is improved further by keeping the current assignment during a restart and only resets the weights.

The used configuration dynamically adopts the decay value for bumping active variables. Furthermore, the information for the decision heuristic are initialized: the activities of the variables increase linearly with the number of the corresponding variable. The polarities are set to the opposite value that would be selected with the Jeroslow-Wang heuristic [15]. Random decisions are made with a probability of 0.5%. After 32 decisions on decision level 1, RISS performs local look-ahead with the first decision variable and adds informations about necessary assignments and equivalent literals [16].

During unit propagation, RISS performs lazy hyper binary resolution. When a learned first UIP clause is a unit clause, then the used all-unit-UIP learning [16] continues to generate the next UIP clause and in case this clause is a unit clause, this clause is kept and the process is continued. If a non-unit clause is the next UIP, then the procedure is aborted and this clause is not used. After a learned clause is minimized with the procedures used in MINISAT 2.2 and GLUCOSE 2.2, the clause is further minimized along the ideas of unhiding literal elimination [3]. Therefore, a binary implication graph (BIG) is maintained during search. The final minimized clause C' is then used for on-the-fly probing [16]: all literals x that are commonly implied in the BIG by all literals y_i of the y_i ∈ C are added as unit clauses. More details about the techniques available in RISS can be found in [5].

The combination of SPARROW and RISS does not forward information from SPARROW to RISS as last years version. Both the parameters for SPARROW and RISS have been tuned on the instances of hard combinatorial instances of the SAT Challenge 2012 and the SAT Competition 2013.

III. IMPLEMENTATION DETAILS

SPARROW is implemented in C. The solver RISS is implemented in C++. Furthermore, we integrated COPROCESSOR into the system, allowing inprocessing techniques to be executed during search – however, this feature is not used in the competition. All solvers have been compiled with the GCC C++ compiler as 64 bit binaries.

IV. AVAILABILITY

The source code of SPARROWToRiss is available at tools.computational-logic.org for research purposes. At the same place the latest version of RISS can be found. The
latest version of SPARROW is available at https://github.com/adrianopolus/Sparrow.

ACKNOWLEDGMENT

The authors would like to thank Armin Biere for many helpful discussions on formula simplification and the BWGrid [17] project for providing computational resources to tune COPROCESSOR. This project was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1. Finally, the authors would like to thank TU Dresden for providing the computational resources to develop, test and evaluate RISS.

REFERENCES

BENCHMARK DESCRIPTIONS
Generating the Uniform Random Benchmarks

Anton Belov
anton@belov-mcdowell.com
Dublin, Ireland

Marijn J. H. Heule
Department of Computer Science,
The University of Texas at Austin, United States

Daniel Diepold
eXXcellent solutions
Ulm, Germany

Matti Järvisalo
HIIT & Department of Computer Science
University of Helsinki, Finland

INTRO

This description explains how the benchmarks of the uniform random tracks of SAT Competition 2014 were generated. The benchmarks in the tracks consist 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 were 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 also varies. The huge 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 threshold to relatively close, while for each \( k \) the number of variables is the same. The used parameter values are detailed in Table I.

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>
<thead>
<tr>
<th>( k )</th>
<th>medium (30)</th>
<th>huge (15)</th>
</tr>
</thead>
</table>
| 3     | \( r = 4.267 \)  
\( n \in \{5000, 5200, \ldots, 10800\} \)  
\( r \in \{3.96, 4.00, \ldots, 4.24\} \)  
\( n = 1,000,000 \) | \( r \in \{8.5, 8.6, \ldots, 9.9\} \)  
\( n = 500,000 \) |
| 4     | \( r = 9.941 \)  
\( n \in \{1250, 1300, \ldots, 2700\} \) | \( r \in \{17, 17.2, \ldots, 19.8\} \)  
\( n = 250,000 \) |
| 5     | \( r = 21.117 \)  
\( n \in \{250, 260, \ldots, 540\} \) | \( r \in \{31.0, 31.5, \ldots, 38.0\} \)  
\( n = 100,000 \) |
| 6     | \( r = 43.37 \)  
\( n \in \{125, 130, \ldots, 270\} \) | \( r \in \{60, 61, \ldots, 74\} \)  
\( n = 50,000 \) |
| 7     | \( r = 87.79 \)  
\( n \in \{75, 78, \ldots, 162\} \) | \( \) |

---

\(^1\)The observed clause-to-variable ratio for which 50\% of the uniform random formulas are satisfiable. For most algorithms, the closer a formula is generate near the phase-transition ratio, the harder it is to solve.

GENERATING THE UNSATISFIABLE BENCHMARKS

This year no unsatisfiable random benchmarks were generated since no solvers were submitted to the random certified UNSAT track. Some solvers were submitted to the random SAT+UNSAT track, but none of these solvers were able to solve any of the unsatisfiable benchmarks during the test phase. By comparison, the unsatisfiable benchmarks used during the test phase were easy for some solvers that participated in SAT Competition 2013. However, much smaller (easier) unsatisfiable benchmarks would have been needed to ensure that some of the solver submitted this year would actually be able to solve unsatisfiable instances. We decided against this, and dropped the random SAT+UNSAT track.
The Application and the Hard Combinatorial
Benchmarks in SAT Competition 2014

Anton Belov
anton@belov-mdowell.com
Dublin, Ireland

Marijn J. H. Heule
Department of Computer Science,
The University of Texas at Austin, United States

The benchmarks for the Application and the Hard Combinatorial tracks of SAT Competition 2014 were drawn from a pool containing benchmarks that either (i) were used in the past seven competitive SAT events (SAT Competitions 2007, 2009, 2011, 2013; SAT Races 2008, 2010; SAT Challenge 2012); (ii) were submitted to these events but not used; or (iii) are new benchmarks submitted to SAT Competition 2014 (the descriptions for these benchmarks are provided in these proceedings). The main factor that influenced the benchmark selection process of SAT Competition 2014 is the fact that, as with the previous SAT competitions, the solvers participating in the competition are ranked using the solution-count ranking system. Thus the primary requirement is that the selected set of benchmarks should contain as few as possible benchmarks that would not be solved by any of the submitted solvers. At the same time, the set should contain as few as possible benchmarks that would be solved by all submitted solvers. In order to level out the playing field for submitters who do not have the resources to tune their solvers on all benchmark sets used in the previous competitions, an additional requirement is that the selected set should contain as many new benchmarks, i.e., benchmarks that were not used in the previous SAT competitions, as possible. Finally, the selected set should not contain a dominating number of benchmarks from the same application domain and the same source. To accommodate this latter requirement, we assigned the benchmarks in the pool to buckets, where the assignment is guided by the combination of the specific application or a specific combinatorial problem the benchmark originates from and the benchmark submitter.

Ideally the empirical hardness of the benchmarks in the pool for SAT Competition 2014 would be evaluated using a selection of top-performing solvers from SAT Competition 2013. However, given the restricted computational resources available for the competition, we opted to reuse the evaluation data for the old benchmarks in the pool from the benchmark ranking experiments performed for SAT Competition 2013. Thus, the empirical hardness of the benchmarks in the pool was evaluated using a selection of five well-performing SAT solvers from SAT Challenge 2012. The solvers were selected from the set of the state-of-the-art (SOTA) contributors [1] in the corresponding tracks of SAT Challenge 2012, with the preference given to solvers that solved a higher number of benchmarks in the Challenge uniquely. The selected solvers for each track are as follows. Application track: glucose, Lingeling, simpsat, linge_dyphase, ZENN. Hard Combinatorial track: clasp-crafted, glucose, Lingeling, simpsat, sattime2012.

To accommodate for the faster execution environment used for SAT Competition 2014, the CPU runtimes for the old benchmarks was scaled by the factor of 0.8.

The benchmarks rating for the tracks was defined as follows:

easy — benchmarks that were solved by all 5 solvers in under 500 seconds (1/10-th of the Competition’s timeout). These benchmarks are extremely unlikely to contribute to the solution-count ranking of SAT solvers in the competition, as all reasonably efficient solvers are expected to solve these instances within the 5000 seconds timeout enforced in the Competition.

too-hard — benchmarks that were not solved by any solver within 10000 seconds (2 times the timeout used in the Competition). These benchmarks are likely to be unsolved by all solvers in the Competition, and as such are also useless for the solution-count ranking, and any other ranking that takes into account the execution time of the solvers, e.g. the careful ranking [2].

hard — the remaining benchmarks, i.e. the benchmarks that were solved by at least one solver within 10000 seconds, and were not solved by at least one solver within 5000

\footnote{The description files that accompany benchmark set distributions contain all information, including the assignment to buckets.}
seconds. These benchmarks are expected to be the most useful for ranking the top-performing solvers submitted to the Competition.

Once the hardness of the benchmarks in the pool was established, 300 benchmarks for each track were selected from the pool. The selection process was controlled by the following constraints:

(i) the ratio of SAT to UNSAT benchmarks should be exactly 50-50;
(ii) no more than 10% of the selected set should come from the same bucket;
(iii) the ratio of new to used benchmarks should be as high as possible;
(iv) the ratio of medium to hard benchmarks should be as close to 50-50 as possible — however, in order to reduce influence of the solvers used for the rating of the benchmarks, 20% of the selected benchmarks were selected among the medium, hard and too-hard benchmarks in the pool without the consideration of their rating;
(v) the performance of the 5 solvers used for the evaluation of the benchmarks should be as uniform as possible — this is to avoid a potential bias towards a particular evaluation solver in the set (the potential negative effects of such bias are discussed in [3]).

The details for the selected sets are provided in Tables I and II on the following page.

REFERENCES

### Table I: Detailed counts of the Application benchmark set

<table>
<thead>
<tr>
<th>bucket</th>
<th>count SAT</th>
<th>UNSAT</th>
<th>UNKNOWN</th>
<th>new</th>
<th>old</th>
<th>easy</th>
<th>medium</th>
<th>hard</th>
<th>too-hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>2d-strip-packing</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>argumentation</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>5</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>bio</td>
<td>11</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>crypto-aes</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>crypto-des</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>crypto-gos</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>crypto-md5</td>
<td>21</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>crypto-sha</td>
<td>29</td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>0</td>
<td>1</td>
<td>18</td>
<td>10</td>
</tr>
<tr>
<td>crypto-vmpc</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>diagnosis</td>
<td>28</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>28</td>
<td>0</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>fpga-routing</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hardware-bmc</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>hardware-bmc-ihm</td>
<td>18</td>
<td>1</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>hardware-cec</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>hardware-manolios</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>hardware-velev</td>
<td>27</td>
<td>18</td>
<td>9</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>19</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>planning</td>
<td>19</td>
<td>11</td>
<td>8</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>14</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>scheduling</td>
<td>30</td>
<td>18</td>
<td>12</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>9</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>scheduling-pesp</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>software-bit-verif</td>
<td>9</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>software-bmc</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>symbolic-simulation</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>termination</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>300</strong></td>
<td><strong>130</strong></td>
<td><strong>150</strong></td>
<td><strong>0</strong></td>
<td><strong>136</strong></td>
<td><strong>164</strong></td>
<td><strong>110</strong></td>
<td><strong>156</strong></td>
<td><strong>34</strong></td>
</tr>
</tbody>
</table>

### Table II: Detailed counts of the Hard Combinatorial benchmark set

<table>
<thead>
<tr>
<th>bucket</th>
<th>count SAT</th>
<th>UNSAT</th>
<th>UNKNOWN</th>
<th>new</th>
<th>old</th>
<th>easy</th>
<th>medium</th>
<th>hard</th>
<th>too-hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>VanderWaerden</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>autocorrelation</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>automata-synchronization</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>clique-width</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>cliquecoloring</td>
<td>21</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>connuCsp-sa</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>counting-ph</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>edgematching</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ensemble-computation</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>erdos-discoverability</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>factoring</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>fixed-shape-forced</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>frb</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>games-hidoku</td>
<td>12</td>
<td>2</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>games-parity</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>games-pebbling</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>graph-isomorphism</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>greenio</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>grid-coloring</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>labs</td>
<td>10</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>modcircuits</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>planning</td>
<td>30</td>
<td>29</td>
<td>1</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>puzzle-queens-with-rooks</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>puzzle-polariun</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ramseycube</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>rbsat</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>sgen</td>
<td>16</td>
<td>10</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>software-bit-verif</td>
<td>27</td>
<td>10</td>
<td>17</td>
<td>0</td>
<td>17</td>
<td>10</td>
<td>0</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>zeronedesign</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>300</strong></td>
<td><strong>150</strong></td>
<td><strong>150</strong></td>
<td><strong>0</strong></td>
<td><strong>218</strong></td>
<td><strong>82</strong></td>
<td><strong>0</strong></td>
<td><strong>44</strong></td>
<td><strong>207</strong></td>
</tr>
</tbody>
</table>
Benchmark for Complete and Stable Semantics for Argumentation Frameworks

Johannes Peter Wallner
Institute of Information Systems
Vienna University of Technology
Vienna, Austria
Email: wallner@dbai.tuwien.ac.at

Abstract—In the last decades argumentation emerged as a distinguished and important topic within Artificial Intelligence (AI). Many formalizations of argumentation and their applications are based on the simple, yet expressive argumentation frameworks (AFs) due to Dung in 1995. The associated reasoning tasks usually involve computing jointly acceptable sets of arguments satisfying certain criteria, called semantics of AFs. Unfortunately almost all reasoning problems on AFs are intractable. In fact some are even located at the second level of the polynomial hierarchy, which calls for efficient solving procedures. In this benchmark we provide generators creating random AFs and translating them together with conditions for either complete or stable semantics to a Boolean formula. Although interesting in their own regard, these encodings are also important for recent approaches based on incremental SAT for solving computationally more demanding problems of the preferred and semi-stable semantics of AFs.

I. INTRODUCTION

Argumentation is nowadays an important field within Artificial Intelligence (AI) [1], which originated from considerations in philosophy, law and formal logic. Applications of argumentation include E-health tools, legal reasoning support, multi-agent systems and more. Formalizations of argumentation typically make use of an abstract representation of discourses. A particularly influential formal model in this regard are Dung’s argumentation frameworks (AFs) [7]. AFs can simply be represented as directed graphs, with vertices being abstract arguments and directed edges denoting attacks between arguments. Semantics of AFs specify criteria to jointly accept sets of arguments. Reasoning tasks on AFs face a high computational complexity [8], with even basic tasks being intractable and some are even hard for the second level of the polynomial hierarchy. Here we focus on the semantics called complete, stable, preferred [7] and semi-stable [4]. The associated reasoning tasks for first two semantics are NP or coNP hard, while the last two semantics have reasoning problems hard for a class in the second level of the polynomial hierarchy.

Implementations of AFs nevertheless need to cope with the high intrinsic complexity. Several directions for implementation were investigated [6]. Among them reduction based approaches are very promising which translate argumentation problems to e.g. SAT, quantified Boolean formulae, answer-set programming or constraint satisfaction problems. We present here generators for SAT encodings for complete and stable semantics based on results from [2]. Although interesting in their own regard, these can also be used as a basis for the computation of the preferred and semi-stable semantics in an incremental SAT, respectively minimal correction sets (MCSes) [10] scheme, developed in [5], [9] and [11].

The generators for this benchmark randomly create AFs and translate them together with conditions for either complete or stable semantics to a Boolean formula as used in [9] and [11] to compute preferred, respectively semi-stable semantics. Other interesting types of random generation models for AFs were proposed e.g. in [3].

II. BACKGROUND

Definition 1. An argumentation framework (AF) is a pair $F = (A, R)$ where $A$ is a set of arguments and $R \subseteq A \times A$ is the attack relation. The pair $(a, b) \in R$ means that $a$ attacks $b$.

An argumentation framework can be represented as a directed graph, as shown in the following example.

Example 1. Let $F = (A, R)$ be an AF with $A = \{a, b, c, d\}$ and $R = \{(a, b), (b, a), (a, c), (b, c), (c, d)\}$. The corresponding graph representation is depicted in Fig. 1.

A basic property of semantics for AFs is a conflict-free set of arguments and a further refinement, an admissible set of arguments. Intuitively, a set is admissible if there are no attacks between arguments inside the set and each attack from outside is dealt with by an attack on this attacker from inside.

Definition 2. Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is conflict-free in $F$, if there are no $a, b \in S$, such that $(a, b) \in R$. We say that an argument $a \in A$ is defended by a set $S \subseteq A$ in $F$ if, for each $b \in A$ such that $(b, a) \in R$, there exists a $c \in S$ such that $(c, b) \in R$.

For an AF $F = (A, R)$ a set $S$ is admissible if it is conflict-free and each argument in $S$ is defended by $S$. An admissible set $E$ in $F$ is complete if every $s \in A$ which is defended by $E$ in $F$ is in $E$. Maximal admissible sets/complete extensions w.r.t. subset-inclusion are called preferred extensions and accept as many arguments as possible, without violating
TABLE I. COMPUTATIONAL COMPLEXITY OF REASONING IN AFs.

<table>
<thead>
<tr>
<th></th>
<th>com</th>
<th>stb</th>
<th>prf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cred$_\sigma$</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>NP-complete</td>
</tr>
<tr>
<td>Skept$_\sigma$</td>
<td>P-complete</td>
<td>coNP-complete</td>
<td>$\Pi_2^P$-complete</td>
</tr>
</tbody>
</table>

admissibility. A set $E$ which is conflict-free in $F$ is stable if for each argument $a \in A \setminus E$ there is an attacker in $E$, i.e. there is a $b \in E$ s.t. $(b, a) \in R$.

In Example 1 the set $\{a, d\}$ is admissible and complete, since it is clearly conflict-free, each argument is defended and all defended arguments are inside the set. It is further preferred, since there is no superset which is also complete. This set is also stable, since it is conflict-free and attacks all arguments outside.

Important reasoning tasks are the credulous acceptance of an argument, which asks if a given argument is in at least one $\sigma$ extension for a given AF, with $\sigma$ a semantics on AFs. The skeptical acceptance of argument asks whether the given argument is in all $\sigma$ extensions of the given AF. Furthermore the enumeration problem deals with enumerating all extensions of a given AF and semantics. We denote the complete, stable and preferred semantics by com, stb and prf respectively and summarize the complexity of credulous (Cred$_\sigma$) and skeptical reasoning (Skept$_\sigma$) in Table I. The complexity class $\Pi_2^P$ denotes the complementary class of $\Sigma_2^P$, which can be represented by a polynomial time non-deterministic Turing machine with access to a coNP oracle.

A. Encodings for Argumentation Problems

We briefly sketch the idea of the encodings for the enumeration problem of complete and stable semantics, s.t. the satisfying assignments of the constructed Boolean formula from a given AF are in a 1-1 correspondence to the complete respectively stable extensions. Our encodings are slightly adapted from [2] and are used in the incremental SAT solving procedure for preferred and other semantics. In particular we take the complete encodings from [9] and the stable encoding from [11]. Let $F = (A, R)$ be an AF.

$$\bigwedge_{(a,b)\in R} (\neg a \lor \neg b) \land \bigwedge_{(b,c)\in R} (\neg c \lor \bigvee_{(a,b)\in R} a)$$

This formula encodes all admissible sets in the sense that the set of variables set to true in a model corresponds to an admissible set. Note that we use arguments directly as Boolean variables. Encoding complete semantics is achieved via introducing further auxiliary variables.

III. BENCHMARK DESCRIPTION

In this benchmark we include a generator, which works as follows. It first randomly generates an AF, by fixing a number of arguments and inserting an attack between two arguments $x$ and $y$ with $x \neq y$ with a given probability $p$. This probability also determines the expected edge density of the directed graph. This AF generator is written in Java and also receives a random seed for generation. After that a simple C++ program parses the AF and returns the Boolean formula (in DIMACS) for complete, respectively stable semantics.

The generated instances for complete semantics are always satisfiable. This follows from the fact that complete extensions always exist (see also [2] and [7]). The SAT encoding for stable semantics may be satisfiable or unsatisfiable, since stable extensions are not guaranteed to exist [7].

According to earlier performance analyses [11] with a minimal correction set algorithm and further preliminary analyses, we expect that at least 400 arguments should be present in the AF with our generation method to exclude easy SAT instances. In earlier performance analyses using MCSes we experienced a significant number of timeouts starting with AFs with about 300 arguments. We expect that newer state-of-the-art SAT solvers are capable of solving larger instances and thus provide a pre-generated set of instances with $\{300, 400, 500\}$ arguments. The second parameter, the expected edge density, is also important for difficulty of the instance. In earlier tests, we generated AFs within $p \in [0.1, 0.2, 0.3, 0.4]$ and a probability of 0.1 resulted in harder instances than the other parameter choices. Thus in the pre-generated set of instances we set this parameter to 0.1. For each parameter choice we generated 20 instances.

ACKNOWLEDGMENT

The author thanks Wolfgang Dvořák for providing the AF generator. This work is supported by the Austrian Science Fund (FWF) through project P1102.

REFERENCES

Air Traffic Controller Shift Scheduling

Mirko Stojadinović
Faculty of Mathematics
University of Belgrade
Belgrade, Serbia
Email: mirkos@matf.bg.ac.rs

I. INTRODUCTION

Air Traffic Controller (ATCo) Shift Scheduling Problem (ATCoSSP) is a type of scheduling problem with an objective to make a shift schedule, so that at each working hour every position is filled by sufficient number of controllers with adequate skills. Many hard constraints need to be satisfied, e.g., each controller must not take more than the specified number of consecutive working/rest shifts, all controllers need to have some minimum leisure time between working shifts, etc. Soft constraints represent staff preferences. Controllers may prefer different shifts (e.g., some prefer morning shifts while others do not like to work early), they may prefer to take consecutive working shifts as rarely as possible, etc. The goal is to make a schedule that minimizes the number of unsatisfied staff preferences and this can be achieved by specifying objective function that needs to be minimized.

The problem was solved using different approaches and solvers, but the reduction to SAT outperformed other solving methods. The detailed description of the problem as well as the instances submitted to SAT Competition 2014 are available from: http://jason.matf.bg.ac.rs/~mirkos/Atco.html. The submitted instances were generated while making shift schedules for one air traffic control center.

II. THE PROBLEM ENCODINGS AND OPTIMIZATION TECHNIQUES

We developed three encodings of the problem. The first two encodings formulate problem as a Constraint Optimization Problem [6], where the first encoding uses linear arithmetic constraints and global constraints [1], while the second uses linear arithmetic constraints only. The third encoding formulates the problem as a Pseudo Boolean problem using only Boolean cardinality constraints [4]. In each encoding, objective function is given as a linear expression. A weight is associated with each unsatisfied controllers preference and coefficients in linear expression are these weights. Objective function is made equivalent to integer variable, and the goal is to find the minimum value of this variable. The encodings can be easily translated to several satisfiability input formats.

Two different optimization techniques are used for solving this problem. Each technique runs solver on instances that differ only in values of optimization variable. The first technique uses a variant of binary search to determine the next value of this variable. The second is adapted for this problem and uses a two-step approach that can significantly speed up the solution process by overcoming the main difficulty: the great number of variables and constraints. It finds initial solution and iteratively searches for a better solution by fixing some parts of the initial solution and performing optimization of its other parts.

III. THE BENCHMARK

When generating ATCoSSP schedule, several instances that differ in value of objective variable need to be solved. We used both mentioned optimization techniques for solving the problem. The first and the second encoding were translated to SAT instances using order encoding [6] implemented in Sugar. Boolean cardinality constraints of the third encoding are translated to SAT using sequential counters [4], and the encoding of at-most-one constraint was done in a way described by Chen [2] and by Klieber [3]. For each combination of optimization technique and encoding we selected 8 SAT instances that are not easily solved. This way 2 × 3 × 8 = 48 instances were generated. From these instances 37 are satisfiable and 11 are unsatisfiable.

ACKNOWLEDGMENT

This work was partially supported by the Serbian Ministry of Science grant 174021.

REFERENCES

Satisfiability through auto-correlation

Srinivasan Arunachalam
Institute of Quantum Computing
Department of Combinatorics and Optimization
University of Waterloo
Email: srinivasan1390@gmail.com

Ilias Kotsireas
Department of Physics and Computer Science
Wilfrid Laurier University
Email: ikotsire@wlu.ca

Abstract—This document describes a family of hard satisfiable 3-SAT benchmarks generated via sequences with constant auto-correlation. We give a brief and self-contained description of the combinatorial problem and then describe the corresponding generated 3-SAT instances.

Index Terms—3-Satisfiability, sequences with constant auto-correlation, combinatorial designs

I. INTRODUCTION

Several different kinds of hard combinatorial problems have been encoded as SAT problems previously, see for instance chapter 17 of [1]. One such class of problems emanate from the realm of combinatorial designs [3], which have a wide range of applications. In this SAT Competition 2014 submission we provide for the first time a family of hard satisfiable 3-SAT instances that arise from sequences with constant auto-correlation. We hope that the SAT solver designers could employ this formulation of hard satisfiable 3-SAT instances to improve their optimized implementations.

II. AUTO-CORRELATION

The auto-correlation function associated with a finite sequence $A = [a_1, \ldots, a_n]$ is defined by $P_A(s) = \sum_{i=1}^n a_ia_{i+s}$, for $s = 1, \ldots, n-1$, where the subscript $i+s$ is taken modulo $n$ when it is greater than $n$. Two sequences $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_n]$ are said to have constant auto-correlation if they satisfy the property $P_A(s) + P_B(s) = c$, for $s = 1, \ldots, n-1$. It turns out that if the values of $A$, $B$ are restricted to the domain $\{-1, +1\}$ and the constant $c = 2$ then two necessary conditions for the existence of such sequences are (i) $n$ should be odd and (ii) the Diophantine equation $\alpha^2 + \beta^2 = 4n - 2$ is solvable. Considering $n = 3$, it follows that $\alpha = 1$, $\beta = 3$ and a feasible solution is $A = [1, 1, -1]$, $B = [1, 1, 1]$. It is less obvious that such sequences exist for $n = 93$, but it turns out that they actually do exist. It is widely believed that the necessary conditions stated above, are also sufficient, i.e. that for every odd $n$ such that $\alpha^2 + \beta^2 = 4n - 2$, there exist sequences of length $n$ with elements from $\{-1, +1\}$ that have constant auto-correlation.

III. ENCODING

The auto-correlation sequences in the previous section can be reduced to an instance of Boolean satisfiability by a series of polynomial-time reductions. Our motivation to provide a 3-SAT encoding of the combinatorial problem described in the previous section is to investigate whether modern SAT solvers can be competitive with the current methods used by researchers to find sequences with constant auto-correlation. We are confident that SAT solves will eventually be proved to be competitive with the evolution in the solvers being best described by the statement [2] “From 100 variables, 200 clauses (early 90s) to 1,000,000 variables and 5,000,000 clauses in 15 years”.

IV. BENCHMARKS

In this first table, we have described 5 SAT instances directly translated from the problem of auto-correlation sequences. Alongside we provide solutions to these well-known problems of tractable size. The second table encodes the 11 unknown benchmark instances submitted for the SAT competition which involve larger number of variables and clauses compared to the well known original problems. Note that in each benchmark we are concerned with the satisfying assignment for the first $2N$ variables, since they encode the solution to the constant auto-correlation sequence.

<table>
<thead>
<tr>
<th>Instance size (N)</th>
<th>No. of clauses</th>
<th>No. of Variables</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>5986</td>
<td>1856</td>
<td>$A = {1 1 0 1 1 1 0 1 1 0 1 0 0 0 1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$B = {1 1 1 1 0 1 1 1 1 1 0 1 0 0}$</td>
</tr>
<tr>
<td>23</td>
<td>14187</td>
<td>4342</td>
<td>$A = {0 1 0 1 0 0 0 0 0 1 1 1 1 0 1 1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$B = {0 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1}$</td>
</tr>
<tr>
<td>31</td>
<td>25725</td>
<td>7808</td>
<td>$A = {0 0 0 0 0 1 1 1 0 1 1 0 1 1 0 0 1 1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$B = {0 0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1}$</td>
</tr>
<tr>
<td>33</td>
<td>29422</td>
<td>8912</td>
<td>$A = {0 0 0 0 0 1 0 0 1 1 1 1 0 1 1 0 1 1 0 1 1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$B = {0 0 0 0 0 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 0 1 0 0 1 0 1 1}$</td>
</tr>
<tr>
<td>43</td>
<td>49635</td>
<td>14960</td>
<td>$A = {0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 0 1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$B = {0 0 0 0 0 1 1 1 0 1 0 0 1 1 1 1 0 0 0 1 1 1 0 1 0 0 0 1 1 1 0 1 1 0 1 1 1 1 1 1}$</td>
</tr>
</tbody>
</table>

TABLE I

WELL-KNOWN INSTANCES AND SOLUTIONS FOR AUTO-CORRELATION
By considering higher values of $N$, we have compiled the following table consisting of harder instances which would likely take longer to solve on SAT solvers. It can be easily seen that the average hardness of the generated instances is approximately 3.2.

<table>
<thead>
<tr>
<th>Instance</th>
<th>No. of clauses</th>
<th>No. of Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>64904</td>
<td>19548</td>
</tr>
<tr>
<td>63</td>
<td>106600</td>
<td>32000</td>
</tr>
<tr>
<td>69</td>
<td>128440</td>
<td>38504</td>
</tr>
<tr>
<td>72</td>
<td>137418</td>
<td>41184</td>
</tr>
<tr>
<td>75</td>
<td>151489</td>
<td>45374</td>
</tr>
<tr>
<td>77</td>
<td>159688</td>
<td>47816</td>
</tr>
<tr>
<td>85</td>
<td>194644</td>
<td>58224</td>
</tr>
<tr>
<td>93</td>
<td>232720</td>
<td>69560</td>
</tr>
<tr>
<td>99</td>
<td>264574</td>
<td>79098</td>
</tr>
<tr>
<td>111</td>
<td>331825</td>
<td>99118</td>
</tr>
</tbody>
</table>

In conclusion, we believe that the SAT generation through auto-correlation sequences will pose a significant challenge for the well-known optimized implementations of SAT solving and is a worthy benchmark for testing the power of future SAT solvers.

V. ACKNOWLEDGMENT

The authors acknowledge generous support for their research from NSERC.

REFERENCES


\[^{1}\text{This instance corresponds to an sequence with auto-correlation constant } c = 0\]
Generating Clique Coloring Problem Formulas

Norbert Manthey
Knowledge Representation and Reasoning Group,
TU Dresden, Germany

Abstract—The clique coloring problem assigns colors to the nodes in a graph such that in each maximal clique there are at least two different colors. Depending on the number of cliques and the number of colors, the resulting instances are either satisfiable or unsatisfiable.

I. INTRODUCTION

The presented instance family is discussed in details by van Gelder in [1]. Here, only the corresponding generator for the mentioned clique coloring instances is provided. The most interesting question is how to add extended resolution to a SAT solver, such that these formulas can be solved more efficiently. The formula generator is submitted, so that a benchmark of interesting instances, Van Gelder recommends a ratio of

As long as \( n > s \), the formulas are unsatisfiable. To obtain interesting instances, Van Gelder recommends a ratio of \( n \in \{2s, 3s, 4s\} \) and to set \( t = s - 1 \).

II. CLIQUE COLORING

Let \( G \) be a graph on \( n \) vertices, where there are \( s \) maximal cliques, and let \( t \) be a number with \( t < s \). Then the question is whether the graph \( G \) can be colored with \( t \) colors, such that in each maximal clique there are at least two different colors. For \( t = n - 1 \) and \( s = n \), this is the well known pigeon-hole principle [1].

The formula, which is generated with the parameters \( n \), \( s \) and \( t \) is constructed as follows. A Boolean variables \( y_{p,j} \) is satisfied if the slot \( p \) of a clique is mapped to the vertex \( j \). The variable \( z_{i,k} \) is satisfied when the vertex \( i \) has the color \( k \). Finally, the variable \( x_{i,j} \) is satisfied, if there is an edge between the vertexes \( i \) and \( j \) in the graph. The indexes have the following domains: \( i, j \in \{1, \ldots, n\} \), \( k \in \{1, \ldots, t\} \) and \( p, q \in \{1, \ldots, s\} \). Then, the clique coloring problem is encoded with the formula

\[
\text{(1)} \quad 
\bigwedge_{1 \leq p \leq s} \bigwedge_{1 \leq q \leq s} \bigwedge_{1 \leq j \leq n} \neg y_{p,j},
\]

\[
\text{(2)} \quad \bigwedge_{1 \leq p \leq s} \bigwedge_{1 \leq q \leq s} \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq n} (y_{p,i} \lor y_{q,j})
\]

\[
\text{(3)} \quad \bigwedge_{1 \leq p \leq s} \bigwedge_{1 \leq q \leq s} \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq n} (y_{p,i} \lor y_{q,j} \lor x_{i,j})
\]

\[
\text{(4)} \quad \bigwedge_{1 \leq k \leq t} \bigwedge_{1 \leq j \leq n} \bigwedge_{1 \leq j \leq n} (z_{i,k} \lor z_{j,k} \lor x_{i,j})
\]

\[
\text{(5)} \quad \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq k \leq t} \bigwedge_{1 \leq k \leq t} z_{i,k}.
\]

The formula (1) ensures that the variables \( y \) build a function, (2) ensures that \( y \) is injective. Next, (3) ensures that there is an edge between any two vertices, and (4) encodes that \( z \) is injective for edges. Finally, (5) ensures that \( z \) is a function. The resulting formula has \( s \) clauses of size \( n \), \( n \) clauses of size \( t \), \( s^2 + n \) binary clauses, and \( nt^2 + s^2 \) ternary clauses.

III. THE GENERATOR

The generator takes the three input variables \( s, n \) and \( t \), and then outputs the CNF formula following the above formulas.

IV. HARDNESS OF THE INSTANCES

With the increasing number of parameters \( s, n \) and \( t \) the instances become harder for CDCL solvers. For the unsatisfiable instances the clique coloring problem contains two nested pigeon hole formulas, so that their hardness increases faster than for plain pigeon hole problems.

REFERENCES

Around 1932 mathematician Paul Erdős conjectured that for any infinite ±1-sequence \( \langle x_1, x_2, \ldots \rangle \) and any integer \( C \), there exist integers \( k \) and \( d \) such that:

\[
\left| \sum_{i=1}^{k} x_i \cdot d \right| > C
\]

It is easy to validate that the conjecture holds for \( C = 1 \), but the case \( C = 2 \) has been a longstanding open problem. It was recently solved by Konev and Lisitsa \([?]\). The discrepancy of a ±1-sequence of length \( k \) can be compute as

\[
\max_{d=1,\ldots,k} \left| \sum_{i=1}^{k} x_i \cdot d \right|
\]

The discrepancy of an infinite ±1-sequence is the largest discrepancy of all its subsequences. Any ±1-sequence of length 12 contains a subsequence of discrepancy two or larger, while and any ±1-sequence of length 1161 contains a subsequence of a discrepancy three or larger \([?]\). It is not known whether the conjecture also holds for \( C = 3 \). In other words, it is not known whether there exists an upper bound on the length of a ±1-sequence such that any subsequence has discrepancy three or less. The benchmark family described here encode the problem of finding a large ±1-sequence of that type.

**SAT Encoding**

The SAT encoding for finding large ±1-sequences such that any subsequence has discrepancy three or less is based on the state machine shown in Fig. 1. Any sequence starts in the state 0. The next two elements in the sequence are used to determine the next state. In case elements are +1 and -1 (the order does not matter), then the state does not change. In case both elements are +1, the state changes from 0 to +2. Similarly, if both elements are -1, the state changes from 0 to -2. If we are in state +2 or -2, we need to enforce that +1, +1 (or -1, -1, respectively) is not allowed.

We use a single boolean variable \( x_i \) for each element \( i \) in the sequence. Positive literals \( x_i \) mean that the element \( i \) is +1, while negative literals \( \overline{x}_i \) mean that element \( i \) is -1. For the states +2 and -2 we introduce boolean variables \( y_j \) and \( z_j \) for each time step \( j \). Assigning \( y_j \) to true means that we are in state +2 at time \( j \), which assigning \( z_j \) to true means we are in state -2 at time \( j \). We are in state 0 when both \( y_j \) and \( z_j \) are false. Notice that \( y_j \) and \( z_j \) cannot true at the same time.

![State machine forcing a ±1-sequence to discrepancy three or less.](image)

**Submitted Problems**

The family of benchmarks that was submitted to SAT Competition 2014 contains instances of various sizes asking whether there exists a ±1 sequence with discrepancy three. The smallest formula has length 8,000. This formula is easy for most SAT solvers. Starting from length 11,000, the formulas appear to be very hard. However, with the right heuristics, it is possible to solve much larger instances. For example a ±1 sequence of length 200,000 with a discrepancy three can be found using the right heuristics. Fig. 2 shows such a sequence. The largest instance of the family has length 200,000. Hence all benchmarks in the family are satisfiable.

**References**

Fig. 2. A ±1 sequence of length 200,000 with discrepancy three shown as a 400 × 500 grid with white squares for -1 and black squares for +1.
Synthesis of Smaller Equivalent Deterministic Transition-based Generalized Büchi Automata

Souheib Baarir
Université Paris Ouest Nanterre la Défense, Nanterre, France;
Sorbonne Universités, UPMC Univ. Paris 6, UMR 7606, LIP6, Paris, France

Alexandre Duret-Lutz
LRDE, EPITA, Le Kremlin-Bicêtre, France

I. INTRODUCTION

The four SAT problems we submitted are intermediate steps from a larger benchmark designed for our FORTE’14 paper [1] where we tested a SAT-based minimization procedure for Deterministic Transition-based Generalized Büchi Automata (DTGBA).

Transition-based Generalized Büchi Automata (TGBA) recognize infinite words labeled by Boolean formulas over some atomic propositions. On the examples of Figure 1, the accepting runs are all those that visit infinitely often at least one transition of each acceptance set (i.e., of each color). These examples accept only the runs where either a and b are infinitely often true, or c and d are infinitely often true. Figure 1 demonstrates that the number of states can be reduced by augmenting the number of acceptance sets.

This type of automata are used for instance in model checking [2]. In this context the valuations typically represent the possible states of some system to verify. The user specifies the expected behaviors of the system using some Linear-time Temporal Logic formula that is then translated into a TGBA recognizing all unwanted behaviors, and this TGBA is then synchronized with the system to find counterexamples.

II. SAT-BASED ITERATIVE MINIMIZATION

At SAT’10, Ehlers [3] presented a SAT-based minimization procedure for Deterministic Büchi Automata (DBA), i.e., with a single acceptance set made of states (not transitions). The procedure is iterative: given a reference DBA \( A \) of \( n \) states, he uses Boolean variables to encode a candidate DBA \( B \) with \( n - 1 \) states, adds more variables to encode the synchronous product \( A \otimes B \) and all its cycles, and adds clauses to ensure that \( A \) and \( B \) are equivalent by making sure that for each cycle of \( A \otimes B \), its projection on \( B \) is accepting iff its projection on \( A \) is accepting. If the problem turns out to be unsatisfiable, it means \( A \) is minimal. Otherwise the solution to the problem contains an encoding of \( B \) and the procedure can be started again using \( B \) as a reference automaton, now trying to find an automaton with \( n - 2 \) states. His tool implementing this procedure can be found at http://www.react.uni-saarland.de/tools/dbaminimizer/.

In our FORTE’14 paper [1], we generalized this SAT encoding to support generalized acceptance (i.e., multiple colors) and transition-based acceptance. We also improved the original encoding by limiting the cycle constrains to the SCCs of \( A \). Also we setup a framework around this SAT-based minimization procedure in order (1) to avoid it for some subclasses of automata where polynomial algorithms are available, and (2) to simplify \( A \) as much as we can. The tools implementing all of this are available in Spot 1.2.4 (http://spot.lip6.fr/) and are documented at http://spot.lip6.fr/userdoc/satmin.html.

III. BENCHMARK

For our FORTE’14 paper [1] we prepared a benchmark we were took several LTL formulae, translated them into DTGBA using different constructions, and finally attempted to build the minimal DTGBA and minimal DBA using the SAT-based iterative minimization described above. We used glucose 2.3 [4] for our experiments.

In many cases, the iterations where the SAT problem is satisfiable (a smaller equivalent DTGBA exists) were fast, and most of the time was spent on the last iteration, proving the problem unsatisfiable.

Our benchmark setup kills the minimization procedure when it took more than 2 hours (for all iterations, not just one) on an Intel Xeon E7-2860 with 512GB of RAM. In practice we either had cases that would terminate within 30 minutes or cases that would not terminate within 2 hours: nothing in between.

The four problems we submitted correspond to iterations that glucose failed to solve within our time constraints. They are detailed in Table I. We suspect that they are unsatisfiable.

IV. REBUILDING THESE PROBLEMS

For people willing to work on more problems with similar encoding, we now describe how to obtain the files listed in Table I. We assume Spot (http://spot.lip6.fr/) is installed and is configured to use some installed SAT-solver: it is configured by default to use glucose, but this can be changed using the SPOT_SATSOLVER environment variable, see http://spot.lip6.fr/userdoc/satmin.html for details. The tool ltl2dstar (http://ltl2dstar.de) should also be installed.

We shall demonstrate the procedure using GF(\( a \leftrightarrow XXXb \)) as an example:

1) Convert the LTL formula into a deterministic Rabin automaton using ltl2dstar, telling it to use Spot’s ltl2tgba as translator. /path/to/ltl2tgba has
to be replaced by the absolute location of the ltl2tgba binary installed by Spot.

```sh
dltilfilt -1f ’GF(a<>XXb)’ | ltl2dstar \
--ltl2nba=spin:/path/to/dstar2tgba -D --stats \
--ltl2nba=spin:/path/to/dstar2tgba -D --stats \
```

2) (optional) Make sure this DRA can be translated into a DTGBA (since DTGBA are less expressive than DRA or non-deterministic TGBA):

```sh
dstar2tgba -D --stats \n"states=\$s,accsets=\$a,\$d=\$d" out.dra
```

The output gives the $s_0$ displayed in Table I:

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>17</td>
<td>s_a</td>
<td>8</td>
<td>DTGBA</td>
</tr>
<tr>
<td>accsets</td>
<td>1</td>
<td>s_a</td>
<td>8</td>
<td>DTGBA</td>
</tr>
<tr>
<td>filename</td>
<td>logfile.csv</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>608</td>
<td>130202</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3) Start the SAT-based minimization, enabling logging and without erasing temporary files.

```sh
SPOT_TMPKEEP=1 SPOT_SATLOG=logfile.csv \nltl2tgba -D -x sat-minimize out.dra
```

To build a DTGBA with more than one acceptance condition, use for instance `-x sat-minimize,sat-acc=2`, and to build a DBA instead of a DTGBA add option `-B`.

While this command is running, monitor the creation of temporary files named sat-*.cnf (input) and sat-*.out (output). Statistics are appended to logfile.csv after each successful iteration.

For instance while the SAT solver is still running we might have:

```sh
% cat logfile.csv
16,14,54,56,102224,5954762,298,11,247,13
13,12,45,48,58318,2213213,119,2,89,5
11,11,44,44,35684,1178894,64,1,164,2
10,10,40,40,24890,756597,43,2,88,3
9,9,34,36,16758,433433,27,2,80,1
8,8,32,32,10808,249998,19,0,47,1
```

Column 1 is the target number of states for the input encoding, while column 2 is the number of states actually reachable in the result. As all these lines correspond to past iterations the SAT-solver presently running is looking for a 7-state automaton, and its input (whose filename can be found by looking at the process list, or by checking the modification time of the sat-*.cnf files) corresponds to the first problem of Table I.

## References

SAT-Encoding of Step-Reduced MD5

Chu-Min Li
MIS, Université de Picardie Jules Verne
Amiens, France
chu-min.li@u-picardie.fr

Bing Ye
Huazhong University of Science and Technology
Wuhan, China
yb19891229@126.com

I. INTRODUCTION

MD5 is a hash function widely used in cryptography. It takes a text (of any length) as input and gives a digest of 128 bits as output. It is now easy to find a collision in MD5 (i.e., find two texts giving the same digest [1][2]). However, to the best of our knowledge, no one is able to make a second-preimage attack to MD5 (i.e., given a digest, find a text giving that digest). In recent years, many researchers encode MD5 into SAT and attempt to make the second-preimage using a SAT solver (see e.g. [3][4]). Since the whole problem is too hard to solve (using any method), people attack a reduced MD5. The best result in our knowledge is the attack using a CDCL SAT solver [4] that is able to find a second-preimage of a digest obtained using 29 MD5 steps. Note that the complete MD5 consists of 64 steps.

Although we are far from a successful second-preimage attack of the complete MD5, Reduced MD5 provides us very useful benchmarks to evaluate SAT solvers for the following reasons: (1) these SAT instances can be guaranteed to be satisfiable by construction, so they are suitable especially for evaluating incomplete solvers as those based on stochastic local search; (2) the hardness of these instances can be controlled by increasing or decreasing the number of steps used in MD5; (3) finally, many SAT instances of similar difficulty are available because we can generate $2^{128}$ digests of 128 bits in total, each digest corresponding to a particular SAT instance.

In the following, we first briefly describe how we encode basic operations into SAT, and then present the instances submitted to the SAT competition.

II. SAT ENCODING OF MD5

Each step of MD5 consists of three basic operations: non-linear boolean functions, cyclic left-shifting, addition of two operands and addition of four operands. Encoding MD5 into SAT consists in encoding these basic operations into CNF clauses in each step.

A. Encoding non-linear functions

There are four non-linear functions:

$$F(X,Y,Z) = (X \land Y) \lor (\neg X \land Z)$$
$$G(X,Y,Z) = (X \lor Z) \lor (Y \land \neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \lor \neg Z)$$

For example, the first equation can be easily transformed into a DNF subformula:

$$(x \land y \land F) \lor (x \land \neg y \land \neg F) \lor (\neg x \land z \land F) \lor (\neg x \land \neg z \land \neg F)$$

which can be transformed into a CNF subformula:

$$(x \lor z \lor \neg F) \land (x \lor \neg z \lor F) \land (\neg x \lor y \lor \neg F) \land (\neg x \lor \neg y \lor F)$$

B. Encoding cyclic left-shifting

For cyclic left-shifting, we don’t need any tricks. It suffices to shift the index of variables.

C. Encoding the addition of two operands

The addition of two operands can be illustrated as follows.

$$x_3 \land x_2 \land x_1 + y_3 \land y_2 \land y_1 = z_3 \land z_2 \land z_1$$

Suppose $r_i$ is the carry from bit $i$ to bit $i+1$, $i(1 \leq i \leq 3)$. The relations between these variables are the following ($r_1$ is defined to be 0):

$$z_i = x_i \land y_i \land r_i$$
$$r_{i+1} = (x_i \lor y_i) \land (x_i \land r_i) \lor (y_i \land r_i)$$

Using an auxiliary variable $w_i$, equation $z_i = x_i \land y_i \land r_i$ can be transformed into two equations: $z_i = w_i \land w_i \land w_i \land w_i$, which are easily transformed into CNF clauses. For example, equation $z_i = x_i \lor w_i$ is equivalent to the following four clauses:

$$x_i \lor w_i \lor \neg z_i$$
$$x_i \lor \neg w_i \lor z_i$$
$$\neg x_i \lor w_i \lor z_i$$
$$\neg x_i \lor \neg w_i \lor \neg z_i$$

D. Encoding the addition of four operands

The addition of four operands can be illustrated as follows.

$$w_3 \land w_2 \land w_1$$
$$x_3 \land x_2 \land x_1$$
$$y_3 \land y_2 \land y_1$$
$$z_3 \land z_2 \land z_1$$
$$s_3 \land s_2 \land s_1$$

Suppose $c_1$ is the carry from bit $i-1$ to bit $i$ and $c_2$ is the carry from bit $i-2$ to bit $i$. So $s_i$ can be driven by $s_i = w_i \lor x_i \lor y_i \lor z_i \lor c_1 \lor c_2 (c_1, c_2$ are defined to be 0). If two or three or five or six of $\{w_1, x_1, y_1, z_1, c_1, c_2\}$
equal to 1, \( c_{1i+1} \) equals to 1, else \( c_{1i+1} \) equals to 0. If four or six of them equal to 1, \( c_{2i+2} \) equals to 1, else \( c_{2i+2} \) equals to 0. Clauses can be deduced according to the above fact.

Our encoding of the addition is different from the encoding presented in [4]. It uses more variables but fewer clauses. In fact, the encoding of the complete MD5 presented in [4] uses 12,721 variables and 171,235 clauses, while our encoding uses 20,006 variables and 165,414 clauses.

### III. Instances Submitted to SAT Competition 2014

MD5 instances submitted to previous satcompetitions such as gus-md5-11.cnf and md5_48_3.cnf encode the problem of finding a collision and do not guarantee the satisfiability. In this submission, we submit instances encoding the second-preimage attack of step-reduced MD5 which are guaranteed to be satisfiable. We choose the numbers of MD5 steps according to the known performance of the state-of-the-art SAT solvers on these instances to generate them.

CDCL solvers such as Cryptominisat3.3, Glucose2.2 and MIniSat2.2.0 solve our reduced MD5 instances up to 28 steps. We submit 5 instances for each number of steps in \{23, 24, 25, 26, 27, 28, 29, 30, 31, 32\}. These 50 instances are generated from the following 5 texts:

- abcd
- GOOD EVENING
- ChuminLi&BingYe
- abcdefghijklmnpqrstuvwxyzabcdefghijklmnpqrstuvwxyzabc
- ABCDEFGHIJKLMNOPQRSTUVWXYZABC

For each text and each number of steps, we compute a digest using standard MD5 steps and then generate an instance. A solution of an instance corresponds to a text giving the same digest (using the same number of steps). We hope the new solvers in the competition 2014 are able to solve all these instances. Nevertheless, the performance of the state-of-the-art stochastic local search solvers for these instances is poor according to our experimental results, and the largest 15 instances (for 30, 31 and 32 steps) are presumably very challenging for all solvers.

**Acknowledgment**

This work is supported by the National Natural Science Foundation of China (Grant No. 61272014 and Grant No. 6107235).

**References**

Solving Polarium Puzzles with SAT

Martin Philipp
Carl von Ossietzky Universität Oldenburg, Germany
martin.philipp@uni-oldenburg.de

Tobias Philipp
International Center for Computational Logic
Technische Universität Dresden, Germany
tobias.philipp@tu-dresden.de

Abstract—Polarium is a popular Nintendo DS puzzle consisting of a grid surrounded by a border. The task is to build a path through the puzzle, meeting the following conditions: 1. it must not cross itself, 2. it must be one single line, 3. for each row which does not belong to the border, the path must contain either all white or all black cells. We submit CNF files to the application SAT+UNSAT track that encode the Polarium puzzle where we minimize the required time to enter the solution in the original game, and the required steps.

I. SPECIFICATION OF THE POLARIUM PUZZLE

A Polarium puzzle \( P \) is an \( m \times n \) grid of white, black, and gray cells, represented as \( P_{i,j} \in \{ \text{black, white, gray} \} \), where \( i \in \{ 1, \ldots, m \} \) and \( j \in \{ 1, \ldots, n \} \). A solution is a finite sequence of coordinates \((x_k, y_k) \mid 1 \leq k \leq K \) representing which cells are visited. Visiting a cell flips a white into a black cell and vice versa except for gray cells that do not change their color. The resulting puzzle \( P' \) is defined by the sequence of coordinates \( I \) and the Polarium puzzle \( P \) follows: For every \( i \in \{ 1, \ldots, m \} \) and \( j \in \{ 1, \ldots, n \} \) it holds that 1. if \( P_{i,j} = \text{gray} \) then \( P'_{i,j} = \text{gray} \), 2. if \( P_{i,j} = \text{white} \) (black, resp.) and the coordinate \((i, j) \in I \), then \( P'_{i,j} = \text{black} \) (white, resp.), 3. if \( P_{i,j} = \text{white} \) (black, resp.) and every coordinate \((i', j') \in I \neq (i, j) \) then \( P'_{i,j} = \text{white} \) (black, resp.). The image below illustrates a Polarium puzzle of size 5 x 5: On the left hand side, the gray border is printed with dotted cells and black cells are printed with stripes. One solution is illustrated with the line. The right side of the image shows the resulting board. Valid solutions do not allow black and white cells in the same row.

II. BOOLEAN VARIABLES & CNF ENCODING

We use the following Boolean variables: \( \text{step}(i, j) \) states that the path uses the cell \((i, j) \), \( \text{dfr}(i) \) states that the row \( i \) uses all black or all white cells of a row, \( \text{sp}(i, j) \) states that the cell \((i, j) \) is the start of the solution, \( \text{ep}(i, j) \) states that the cell \((i, j) \) is the end of the solution, \( \text{hc}(i, j) \) states that the cells \((i, j) \) and \((i + 1, j) \) are both crossed by the solution, \( \text{vc}(i, j) \) states that the cells \((i, j) \) and \((i, j + 1) \) are both crossed by the solution. For a Polarium puzzle \( P \) of size \( m \times n \), we add the constraint relating the step with the \( \text{hc} \) and \( \text{vc} \) variables:

\[
\text{step}(i, j) \rightarrow (\text{hc}(i, j) + \text{hc}(i + 1, j) + \text{vc}(i, j) + \text{vc}(i, j + 1) +\]

\[
\text{sp}(i - 1, j - 1) + \text{ep}(i - 1, j - 1) \neq 2 \rightarrow \neg \text{hc}(i, j) \land \neg \text{hc}(i - 1, j) \land \neg \text{vc}(i, j) \land \neg \text{vc}(i, j + 1) \land \neg \text{sp}(i - 1, j - 1) \land \neg \text{ep}(i - 1, j - 1)) \text{ for every } i, j.
\]

Additionally, we add constraints that ensure that the solution does not cross itself: The basic idea to prevent this is to start from one cell which is on the path, follow the path and observe that all cells have been reached. We use an additional propositional variables \( \text{visited}(s, i, j) \), where \( s \in \{ 1, \ldots, K \} \) represents a step in the solution path. The variables \( \text{visited}(s, i, j) \) is then true if the \( s \)’th step of the solution visits the cell \((i, j) \). Moreover, the clause \( (\neg \text{visited}(s, i, j) \lor \text{step}(i, j)) \) and the clause \( (\neg \text{step}(i, j) \lor \text{visited}(1, i, j) \lor \text{visited}(2, i, j) \lor \cdots \lor \text{visited}(K, i, j)) \) are added for every \( i, j \). Finally, we relate the variables \( \text{hc} \) and \( \text{vc} \) with the visited variable, and add for every \( s \in \{ 1, \ldots, K \} \):

\[
\text{visited}(s, i, j) \rightarrow ((\text{visited}(s - 1, i, j) \land \text{vc}(i, j - 1)) \lor (\text{visited}(s - 1, i + 1, j) \land \text{vc}(i, j))) \lor (\text{visited}(s - 1, i, j - 1) \land \text{hc}(i, j - 1)) \lor (\text{visited}(s - 1, i, j + 1) \land \text{hc}(i, j))).
\]

III. OPTIMIZATION OF SOLUTION LENGTH AND TIME

Minimizing the path length is done by adding the at-most-k constraint \( \leq k \{ \text{step}(i, j) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n \} \) using the encoding in [1]. A more challenging problem is to minimize the needed time to enter the solution in Nintendo DS. The game represents each cell as a 16 x 16 pixels graphic, and the cursor can be moved up to 40 pixels in horizontal and vertical direction per frame (1/60th second). After each frame, the cursor position is read out and is valid as long as the cursor is on the same row or column as in the previous frame. For the first frame, we may assume that the cursor already is on the right position. We use the new variables \( \text{tt}(i, j) \) to store if there is one frame where the cursor position belongs to the cell \((i, j) \). We also use the variables \( \text{trt}(i, j) \) and \( \text{ttt}(i, j) \) to store the cursor’s location with the intended meaning: \( \text{tt}(i, j) \land \neg \text{trt}(i, j) \land \neg \text{ttt}(i, j) \rightarrow \text{cursor} \at (i \times 16 + 4, j \times 16 + 4); \)

\[
\text{tt}(i, j) \land \text{trt}(i, j) \land \neg \text{ttt}(i, j) \rightarrow \text{cursor} \at (i \times 16 + 12, j \times 16 + 4); \)
\[
\text{tt}(i, j) \land \neg \text{trt}(i, j) \land \text{ttt}(i, j) \rightarrow \text{cursor} \at (i \times 16 + 4, j \times 16 + 12); \)
\[
\text{tt}(i, j) \land \text{trt}(i, j) \land \texttt{ttt}(i, j) \rightarrow \text{cursor} \at (i \times 16 + 12, j \times 16 + 12). \]
\]

Then, we search for the smallest \( k \) such that \( \leq k \{ \text{tt}(i, j) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n \} \) is satisfiable.

IV. CONCLUSION

Translating Polarium into CNF is an attractive idea, but finding solutions that require the minimal time to enter seems to be still challenging.

REFERENCES

Too Many Rooks

Norbert Manthey and Peter Steinke
Knowledge Representation and Reasoning Group
TU Dresden, Germany

Abstract—Unsatisfiable matching problems like the pigeon hole problem are well understood, and it is known that todays SAT solvers cannot solve these problems easily. A satisfiable variant is the n-Queens problem. The presented benchmark family turns the n-queens problem into the unsatisfiable variant with one additional piece on the board. Since for this task it is sufficient to use rooks, the problem is encoded for rooks only.

I. INTRODUCTION

For determining the power of a proof system, usually unsatisfiable instances are considered. One of the most famous families is the pigeon hole problem, where \( n+1 \) pigeons are mapped to \( n \) holes, such that each hole contains at most one pigeon. The proposed benchmark implements the same idea for chess boards: placing \( n+1 \) rook pieces on an \( n \times n \) chess board, such that the pieces cannot attack each other.

More formally, for an \( n \times n \) chess board the formula enforces that

1) On each column there is at most one rook (AMO(COL)).
2) On each row there is at most one rook (AMO(ROW)).
3) On the whole board there are \( n+1 \) rooks (ALK(BOARD)).

As explained before, the two hints that can be used are the following:

1) On each column there has to be at least one rook (ALO(COL)).
2) On each row there has to be at least one rook (ALO(ROW)).

Essentially, since the argumentation for the two hints holds for the case for placing \( n \) rooks on an \( n \times n \) board, the argumentation holds consequently for any higher number of rooks as well.

II. ROOKS ON A CHESS BOARD

On an \( n \times n \) chess board, a rook that is placed on a field \((x, y)\) can attack any other piece that is placed on the same row or the same column, i.e. which either has the same x-coordinate, or the same y-coordinate. To place \( n \) rooks on such a board, on each column and each row there has to be one rook, resulting in the first two conditions. From this fact, one can also conclude, that on each row there has to be at least one rook. Adding this information might reduce the difficulty, as proposed for Hamiltonian Cycle problems in [1]. To complete the above description, \( n+1 \) rooks have to be present on the whole board, resulting in the third condition.

III. ENCODING

The chess board is encoded with the Boolean variables \( x_{c,r} \), where \( 1 \leq r \leq n \) and \( 1 \leq c \leq n \). The variables \( x_{c,r} \) is satisfied when a rook is placed on the field \((r, c)\).

Then, for each column and row the at-most-one constraints are encoded with the pairwise encoding:

\[
\text{AMO(COL)} = \bigwedge_{r} \bigwedge_{1 \leq c < c' \leq n} (\overline{x_{c,r}} \lor \overline{x_{c',r}})
\]

\[
\text{AMO(ROW)} = \bigwedge_{c} \bigwedge_{1 \leq r < r' \leq n} (\overline{x_{c,r}} \lor \overline{x_{c,r'}})
\]

Enforcing the number of \( n+1 \) pieces on the board is encoded with

\[
\text{ALK(BOARD)} = \sum_{1 \leq c \leq n} x_{c,r} \geq k
\]

This cardinality constraint is encoded with cardinality networks [2].

Finally, the at-least-one hints for the row and the column can be encoded with single clauses:

\[
\text{ALO(COL)} = \bigwedge_{r} \bigvee_{1 \leq c \leq n} x_{c,r}
\]

\[
\text{ALO(ROW)} = \bigwedge_{c} \bigvee_{1 \leq r \leq n} x_{c,r}
\]

IV. THE GENERATOR

To generate the instances, a generator is used, which takes three parameters \( n \), ALO(COL) and ALO(ROW). The parameter \( n \) specifies the size of the board, and the latter two parameters specify whether the hints for the row and the column should be encoded as well. The implementation of the generator uses the PBLib [3] to generate the cardinality constraint formulas.

V. HARDNESS OF THE INSTANCES

Depending on the size \( n \) the instances become harder for CDCL solvers. When the hints are added, then the hardness can be reduced slightly. The more hints are added, the simpler are the instances.
REFERENCES


sgen6: A generator for small but difficult unsatisfiable instances

Ivor Spence
School of Electronics, Electrical Engineering and Computer Science
Queen’s University Belfast
i.spence@qub.ac.uk

I. INTRODUCTION

This is a development of the previous generators sgen1[1] and sgen2[2]. Both satisfiable and unsatisfiable instances can be generated, but it is the unsatisfiable ones which are of interest. The method which was previously used to generate satisfiable benchmarks is modified to permit unsatisfiable benchmarks. Using the measure of the shortest time to solution by any solver it is expected that these benchmarks will be amongst the most difficult for their size.

II. UNSATISFIABLE INSTANCES IN PREVIOUS VERSIONS

In order to generate unsatisfiable instances in previous versions cardinality constraints required \((n+1)/2\) variables to be positive and also \((n+1)/2\) to be negative where \(n\), the number of variables, was an odd number. Clearly this was impossible to satisfy. Such instances have proved to be the most difficult for their size at all the SAT competitions since they were proposed, but a recent version of the SAT4j solver by Armin Biere and others [3] explicitly tests for cardinality constraints and is able to solve them very quickly.

III. SATISFIABLE INSTANCES IN PREVIOUS VERSIONS

In order to generate satisfiable instances previously three partitions of the variables were used, all in groups of size five. Clauses generated from the first partition permitted at most one variable from each group to be positive and clauses from the second and third partitions both required at least one variable per group to be positive. Thus the cardinality constraints were that the number of positive variables was at least \(n/5\) and at most \(n/5\). From a cardinality viewpoint therefore the instances could have been satisfiable. The deciding factor was whether it was possible to choose \(n/5\) positive variables in such a way that exactly one was chosen from each group in all three partitions.

For example, given the three partitions

\[
\begin{align*}
\{1,2,3,4,5\}, & \{6,7,8,9,10\}, \{11,12,13,14,15\} \\
\{1,2,6,7,11\}, & \{3,4,8,12,13\}, \{5,9,10,14,15\} \\
\{1,4,5,6,10\}, & \{2,3,11,12,13\}, \{7,8,9,14,15\}
\end{align*}
\]

it is possible to choose the positive variables to be \(\{3, 6, 15\}\) with one positive variable in each group in each partition. The partitions were explicitly chosen in such a way as to ensure that this was possible, and these were amongst the most difficult satisfiable instances for their size.

IV. UNSATISFIABLE INSTANCES

In sgen6 generated instances are of the same form as the previous satisfiable ones, but the partitioning process is no longer constrained to ensure satisfiability. For example using the partitions

\[
\begin{align*}
\{1,2,3,4,5\}, & \{6,7,8,9,10\}, \{11,12,13,14,15\} \\
\{3,7,9,12,15\}, & \{2,4,6,10,14\}, \{1,5,8,11,13\} \\
\{5,7,9,12,15\}, & \{1,3,6,10,14\}, \{2,4,8,11,13\}
\end{align*}
\]

it is not possible to choose three variables which are in different groups in each partition. These partitions lead to the unsatisfiable benchmark in Table I. The first 30 clauses in this benchmark guarantee that at most one variable per group is positive and the final six guarantee that at least one variable per group is positive.

For small values of \(n\) most instances are satisfiable and the example in Table I is the smallest known unsatisfiable one. As \(n\) increases the likelihood of satisfiability decreases until at approximately \(n = 200\) most instances are unsatisfiable. The instances submitted for the competition contain from 100-240 variables (600-1440 literals) and all have been tested as unsatisfiable. They are not easily solvable by constraint checking and the larger ones (while still very small by most standards!) are expected to be difficult for any solver.

V. RESULTS

For most complete solvers these new benchmarks are slightly easier than the unsatisfiable benchmarks generated by previous versions of sgen, but they are very difficult for the cardinality-based solvers which find previous unsatisfiable benchmarks to be easy. Measured by the shortest time by any known solvers, these are expected to be more difficult.
<table>
<thead>
<tr>
<th>Clauses</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 -2 0</td>
<td>Clauses corresponding to the first partition.</td>
</tr>
<tr>
<td>-1 -3 0</td>
<td>Within each group at most one variable can be positive.</td>
</tr>
<tr>
<td>-1 -4 0</td>
<td></td>
</tr>
<tr>
<td>-1 -5 0</td>
<td></td>
</tr>
<tr>
<td>-2 -3 0</td>
<td></td>
</tr>
<tr>
<td>-2 -4 0</td>
<td></td>
</tr>
<tr>
<td>-2 -5 0</td>
<td></td>
</tr>
<tr>
<td>-3 -4 0</td>
<td></td>
</tr>
<tr>
<td>-3 -5 0</td>
<td></td>
</tr>
<tr>
<td>-4 -5 0</td>
<td></td>
</tr>
<tr>
<td>-6 -7 0</td>
<td></td>
</tr>
<tr>
<td>-6 -8 0</td>
<td></td>
</tr>
<tr>
<td>-6 -9 0</td>
<td></td>
</tr>
<tr>
<td>-6 -10 0</td>
<td></td>
</tr>
<tr>
<td>-7 -8 0</td>
<td></td>
</tr>
<tr>
<td>-7 -9 0</td>
<td></td>
</tr>
<tr>
<td>-7 -10 0</td>
<td></td>
</tr>
<tr>
<td>-8 -9 0</td>
<td></td>
</tr>
<tr>
<td>-8 -10 0</td>
<td></td>
</tr>
<tr>
<td>-9 -10 0</td>
<td></td>
</tr>
<tr>
<td>-11 -12 0</td>
<td></td>
</tr>
<tr>
<td>-11 -13 0</td>
<td></td>
</tr>
<tr>
<td>-11 -14 0</td>
<td></td>
</tr>
<tr>
<td>-11 -15 0</td>
<td></td>
</tr>
<tr>
<td>-12 -13 0</td>
<td></td>
</tr>
<tr>
<td>-12 -14 0</td>
<td></td>
</tr>
<tr>
<td>-12 -15 0</td>
<td></td>
</tr>
<tr>
<td>-13 -14 0</td>
<td></td>
</tr>
<tr>
<td>-13 -15 0</td>
<td></td>
</tr>
<tr>
<td>-14 -15 0</td>
<td></td>
</tr>
<tr>
<td>3 7 9 12 15 0</td>
<td></td>
</tr>
<tr>
<td>2 4 6 10 14 0</td>
<td>Clauses corresponding to the second partition.</td>
</tr>
<tr>
<td>1 5 8 11 13 0</td>
<td>Within each group at least one variable has to be positive.</td>
</tr>
<tr>
<td>3 7 9 12 15 0</td>
<td></td>
</tr>
<tr>
<td>1 3 6 10 14 0</td>
<td></td>
</tr>
<tr>
<td>2 4 8 11 13 0</td>
<td>Clauses corresponding to the third partition.</td>
</tr>
<tr>
<td>3 7 9 12 15 0</td>
<td></td>
</tr>
<tr>
<td>1 3 6 10 14 0</td>
<td></td>
</tr>
<tr>
<td>2 4 8 11 13 0</td>
<td>Within each group at least one variable has to be positive.</td>
</tr>
</tbody>
</table>

**TABLE I**

**UNSATISFIABLE BENCHMARK**

**REFERENCES**


A small but hard symbolic simulation benchmark

Alberto Griggio  
Fondazione Bruno Kessler  
Trento, Italy  

The benchmark comes from an encoding of some path conditions in a symbolic simulator for the analysis of x86 binaries, based on the Insight framework (http://insight.labri.fr). The tool uses an SMT solver as a backend. The SAT encoding has been generated by the MathSAT SMT solver (http://mathsat.fbk.eu) from an SMT formula in the theory of bit-vectors. According to Gérald Point, the developer who produced the SMT instance, the benchmark represents the symbolic execution of the following x86 instructions:

```assembly
mov %eax, %ecx
mov $0x51eb851f, %edx
imul %edx
mov %edx, %eax
shr $0x1f, %eax
sar $0x5, %edx
add %eax, %edx
imul $064, %edx, %eax
sub %eax, %ecx
cmp %ecx, %esi
```

The benchmark checks that the sign flag (SF) of the last cmp instruction is set.
I. Summary

Z1 is an abbreviation for “Zero-One”, which in turn is an abbreviation for “partially balanced zero-one design.” The idea has been described in a SAT 2010 paper [VGS10].

The benchmarks submitted to SAT Competition 2014 use a generalization of that generation method. All benchmarks are unsat. Bipartite regular graphs may be 3-regular or 4-regular. In all cases they have no 2-cycles or 4-cycles, as these are observed to generate easier problems.

The directory and file name have N-D in the name, where N is the partition size of the bipartite graph and D is its degree. The propositional variables are the edges in the graph, which can be naturally arranged into rows and columns. For D = 4, one extra edge is added. For D = 3, one column is deleted.

The clauses enforce cardinality constraints on the rows and columns. For D = 4, each normal row requires at least two true variables, and each normal column requires at most two true variables. The row with the extra edge requires at least three true variables, and the column still requires at most two true variables.

For D = 3, each row requires at least one true variable, and each column requires at most one variable, but there are only N–1 columns.

There are four series of instances, as described in the next section.

II. Tar File Contents

The directory Z1-024-4 has 10 instances. The directory Z1-025-4 has 12 instances. The directory Z1-070-3 has 10 instances. The directory Z1-072-3 has 13 instances. In the listing the first number is the file size in bytes, so readers can see these files are quite small.

0 May 7 18:21 2014 ./Z1-024-4/
2826 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-7907777-1-00.cnf
2826 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-10183999-1-00.cnf
2814 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-7880777-1-00.cnf
2827 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-5728555-1-00.cnf
2824 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-7780777-1-00.cnf
2827 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-3351333-1-00.cnf
2827 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-4351333-1-00.cnf
2827 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-5647555-1-00.cnf
2827 Jan 17 09:12 2014 ./Z1-024-4/edges-024-4-5613555-1-00.cnf
2827 Jan 17 09:12 2014 ./Z1-025-4/
2959 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-5714555-1-00.cnf
2956 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-7932777-1-00.cnf
2956 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-1237111-1-00.cnf
2956 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-10062999-1-00.cnf
2956 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-10169999-1-00.cnf
2953 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-1136111-1-00.cnf
2956 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-3460333-1-00.cnf
2953 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-10169999-1-00.cnf
2953 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-7780777-1-00.cnf
2956 Jan 17 09:44 2014 ./Z1-025-4/edges-025-4-7780777-1-00.cnf
REFERENCES

Solver Index

BalancedZ, 10
BFS-Glucose, 12
CBPeneLoPe, 14
CCA2014, 16
CCAnr+glucose, 17
CCgscore, 19
CLaS, 21
CPSparrow, 22
CryptoMiniSat v4, 23
CryptoMiniSat v4 tuned, 24
CSSCSat2014, 25
DCCASat+march_rw, 27
Dimetheus, 29
Glucose, 31
Glucose-Syrup, 33
Glue_lgl_split, 37
GLUEMINISAT 2.2.8, 35
GlueSplit_clasp, 37

Lingeling, 39
Lingeling (druplig), 39
Lingeling (no agile), 39

MCP, 41
miniLoCeG+Glucose, 43
MiniSat blbd, 45
MiniSat-ClauseSplit, 44
MiniSat_HACK_1430ED, 46
MiniSat_HACK_999ED, 46
MinitSAT, 48
MiPiSAT, 49
MSP, 50

Ncca+, 52
Nigma 1.1, 53
ntusat, 54
ntusatbreak, 54

ParaCIRMiniSAT, 14
Pcasso, 56
PeneLoPe, 58
pGlucose, 60
Plingeling, 39
pmcSAT, 61

pprobSAT, 63
probSAT, 63
provoSATeur+glucose, 64

Riss 4.27, 65
Riss 4.27 BlackBox, 68
ROKK 1.0.1, 70
ROKKminisat, 71
RSeq2014, 72

Sattime2014r, 73
SattimeGlucoseSeq, 76
satUZK, 75
satUZKcert, 75
satUZKp, 75
satUZKpl, 75
SparrowToRiss, 77
SWDiA5BY, 46

Treengeling, 39
YalSAT, 39
Benchmark Index

Air Traffic Controller Shift Scheduling, 86
Application Tracks, 81
Argumentation Frameworks, 84
Auto-Correlation, 87
Clique Coloring, 89
Erdős Discrepancy Conjecture, 90
Hard Combinatorial Tracks, 81
Minimization of Deterministic Transition-based Generalized Büchi Automata, 92
Partially Balanced Zero-One Designs, 102
Polarium Puzzles, 96
Random k-SAT Tracks, 80
sgen6, 99
Step-Reduced MD5, 94
Symbolic Simulation, 101
Too Many Rooks, 97
Author Index

Abramé, André, 52
Alfonso, Enrique Matos, 68
Arunachalam, Srinivasan, 87
Audemard, Gilles, 31, 33, 58
Baarir, Souheib, 60, 92
Balint, Adrian, 21, 22, 63, 77
Bartók, Dávid, 12
Belov, Anton, 80, 81
Bessenyei, Balázs Donát, 44
Biere, Armin, 39
Bringmann, Oliver, 43, 64
Burg, Sebastian, 43, 64
Cai, Shaowei, 17, 19, 25, 27
Chen, Jingchao, 37, 45
Chen, Wenxiang, 41, 50
Ciardo, Gianfranco, 53
Diepold, Daniel, 80, 81
Duret-Lutz, Alexandre, 92
Flores, Paulo, 61
Gableske, Oliver, 29
Griggio, Alberto, 101
Guerra a Silva, Luis, 61
Habet, Djamal, 52, 72
Heule, Marijn J.H., 80, 81, 90
Hoesen, Benoit, 58
Howe, Adele, 41, 50
Huang, Chong, 10, 73
Inoue, Katsumi, 35
Irfan, Ahmed, 21, 56
Iwanuma, Koji, 35
Järvisalo, Matti, 80, 81
Jabbour, Said, 58
Ji, Chengqian, 16
Jiang, Chuan, 53
Jiang, Hua, 76
Jiang, Jia-Hong Roland, 54
Kondoh, Shuya, 14
Kotsireas, Ilias, 87
Kropf, Thomas, 43, 64
Lagniez, Jean-Marie, 58
Lanti, Davide, 21, 56
Lawall, Julia, 60
Li, Chumin, 10, 72, 73, 76, 94
Li, Jianhong, 16
Li, Yu, 73
Lindauer, Marius, 24
Luo, Chuan, 17, 19, 25, 27
Mann, Zoltán Ádám, 12, 44
Manthey, Norbert, 21, 48, 56, 65,

68, 77, 89, 97
Marques, Ricardo, 61
Nabeshima, Hidetomo, 35
Oh, Chanseok, 46
Okugawa, Takumi, 70, 71
Philipp, Martin, 96
Philipp, Tobias, 49, 96
Piette, Cédric, 58
Schöning, Uwe, 63
Silveira, L. Miguel, 61
Simon, Laurent, 31, 33
Sonobe, Tomohiro, 14
Soos, Mate, 23, 24
Sopena, Julien, 60
Speckenmeyer, Ewald, 75
Spence, Ivor, 99
Steinke, Peter, 97
Stojadinović, Mirko, 86
Su, Kaile, 17, 19, 25, 27
Toumi, Donia, 52
van der Grinten, Alexander, 75
Van Gelder, Allen, 102
Wallner, Johannes Peter, 84
Wang, Hsiao-Lun, 54
Whitley, Darrell, 41, 50
Wotzlaw, Andreas, 75
Wu, Wei, 25, 27
Wu, Yuxiang, 16
Xu, Ruchu, 10, 76
Yasumoto, Takeru, 70, 71
Ye, Bing, 94