An Empirical Evaluation of SAT Solvers on Bit-vector Problems

Bruno Dutertre

SRI International
Menlo Park, CA, U.S.A.
`bruno.dutertre@sri.com`

Abstract

Bit blasting is the main method for solving SMT problems in the theory of fixed size bit vectors. It converts bit-vector problems to equisatisfiable Boolean satisfiability problems that are then solved by SAT solvers. We present an empirical evaluation of state-of-the-art SAT solvers on problems produced by bit blasting with the Yices SMT solver. The results are quite different from common SAT solver evaluations such as the SAT races and SAT competitions, which argues for extending these evaluations to include benchmarks derived from SMT problems.

1 Introduction

Most SMT solvers for the theory of quantifier-free bit-vectors rely on the brute-force approach colloquially known as *bit blasting*. Bit blasting compiles an SMT problem on bit vectors into a Boolean circuit that is then converted to conjunctive normal form (CNF) and solved by a Boolean satisfiability (SAT) solver. Clearly, the efficiency of modern SAT solver is the key reason why this method works at all.

The SAT solving community organizes regular competitions that evaluates SAT solvers on a set of benchmarks. In these competitions, the top SAT solvers are typically very close.[1] There is anecdotal evidence (for example, from results of the SMT solver competitions) that the situation is different on bit-blasting problems. We have experienced this in the 2019 SMT solver competition where we used CaDiCaL and CryptoMiniSAT as backends to the Yices 2 solver. CaDiCaL solved 80 more problems than CryptoMiniSAT.

We explore this issue more thoroughly. We report on an empirical evaluation of sixteen state-of-the-art SAT solvers on CNF problems produced by Yices 2. The solvers we evaluate includes all the solvers that finished in the top three rankings in the 2019 SatRace,[2] winners of earlier SAT Solver Competitions, and other representative solvers. In this evaluation, CaDiCaL is clearly superior to all other solvers. We then empirically investigate the features of CaDiCaL that matter most on our benchmarks.

2 Benchmarks

Our evaluation is based on 15447 CNF problems that we generated from non-incremental benchmarks in the logic QF_BV (quantifier-free, fixed size bit vectors). These benchmarks are available on the SMT-LIB repository.[3] We started from the 41696 benchmarks available in this repository as of July 2019.

---

[1] See `http://www.satcompetition.org`

[2] `http://sat-race-2019.ciirc.cvut.cz`

[3] `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV`

Solver	**Reason for inclusion**
MapleCOMSPS	Winner SAT Competition 2016
MapleCOMSPS_LRB	Accident
MapleLCMDiscChronoBT-DL-v3	1st in UNSAT Track, 1st in SAT+UNSAT at SatRace 2019
MapleLCMDistChronoBT	Winner SAT Competition 2018
MapleLCMDistChronoBT-DL-v1	2nd in SAT Track at SatRace 2019
MapleLCMDistChronoBT-DL-v2.1	2nd in SAT Track at SatRace 2019
MapleLCMdistCBTcoreFirst	3rd SAT+UNSAT Track at SatRace 2019
Maple_LCM_Dist	Winner SAT Competition 2017
PSIDS_MapleCMDistChronoBT	3rd in UNSAT Track at SatRace 2019
cadical-1.2.1	latest CaDiCaL release
cadical-satrace19	1st in SAT Track, 2nd in SAT+UNSAT at SatRace 2019
cryptominisat5	historic
expMaple_CM_GCBumpOnlyLRB	2nd in UNSAT Track at SatRace 2019
glucose-4.2.1	historic
minisat-2.2.0-simp	historic
smallsat	3rd in SAT Track at SatRace 2019

Table 1: SAT Solvers Used in the Evaluation

We processed all these benchmarks with the Yices 2 SMT solver [5] to convert them to the DIMACS CNF format used by SAT solvers. Out of the initial 41696 benchmarks, Yices 2 can solve 18940 problems without conversion to CNFs (i.e., by rewriting and other simplification at the bit-vector level). We obtained then 22756 CNF formulas in total. We then removed trivial formulas and duplicates. There were 325 empty CNFs (trivially SAT), 2685 formulas that contained the empty clause (trivially UNSAT), and 4286 duplicates. This leaves 15450 formulas. Out of them, we decided to remove three extremely large CNF formulas (of more than 5 GB each) and keep the remaining 15457 formulas for our evaluation.

By a duplicate, we mean a CNF file that is syntactically identical to another benchmark; it contains the exact same variables and clauses in the same order. We did not attempt to identify formulas that are identical modulo variable or clause reordering. It may be somewhat surprising that problems in SMT-LIB that are syntactically distinct result in the exact same CNF after bit blasting. We have not investigated this issue very much and we do not know whether this happens with other SMT solvers than Yices 2. But this may suggest removing possibly redundant problems from SMT-LIB.

The formulas resulting from this bit blasting vary widely in size. The smallest formula has two variables and two clauses. The largest formula has more than 9 million variables and 41 million clauses. Such very large examples are present but they are not common. The median number of variables is 9115 and the median number of clauses is 29307. All these CNF benchmarks are available at `https://www.csl.sri.com/~bruno/bit-blasting.html`.

3 Solvers

Table 1 shows the SAT solvers that we selected in our evaluation. We picked solvers that did well at the 2019 SatRace and the winners of the 2016–2018 editions of the SAT Competition. We also added three well-known solvers: MiniSAT [7], Glucose [1], and CryptoMiniSAT [20], and the latest release of CaDiCaL [3]. We also added MapleCOSMPS_LRB by accident (it participated in the SAT Competition in 2016 but did not win).

All the solvers in this table are based on conflict-driven clause learning (CDCL) pioneered by GRASP [13] and Chaff [14]. Except for CaDiCaL and CryptoMiniSAT, all the solvers are derived from MiniSAT [7] via Glucose [1] and COMiniSATPS [16, 17] and still share a lot of code with MiniSAT 2.2.0. They employ techniques introduced by MiniSAT and its successors: learned clause minimization [7], preprocessing with variable and clause elimination [6], glue-based estimates of learned clause quality [2]. In addition to these common bases, recent solvers employ techniques such as vivification of learned clauses [12], chronological backtracking [15], and new branching heuristics [11]. Additional details on each solver and the particular techniques they implement can be found in the SatRace proceedings [10].

CryptoMiniSAT [20] also derives from MiniSAT but it is now very different. Unlike other MiniSAT-derived solvers, CryptoMiniSAT implements the *in-processing* strategy, and includes many more simplification techniques. CaDiCaL does not borrow code from MiniSAT and it also uses in-processing. All other solvers in our list work in two phases. They first simplify the input formula using variable and clause elimination algorithms. After this preprocessing, they switch to pure CDCL search and perform only limited clause simplification during search. The in-processing strategy uses simplification procedures more aggressively. Rather than just performing one round of initial simplification, in-processing solvers apply these simplification procedures periodically, They alternate between search and simplification. This strategy is used by both CaDiCaL and CryptoMiniSAT. Both solvers also implement many more simplification techniques than the others.

4 Experiment

We ran the solvers listed in Table 1 on the 15447 benchmarks produced by Yices 2. For the experiment, we used a set of ten Linux-based servers running Ubuntu 18.04. All servers have 64 GB RAM and have two four-core x86-64 Intel processors (Xeon Gold 5122 Processors, 16.5M Cache, 3.60 GHz). We used a timeout of 20 minutes CPU time per benchmark and did not set a memory limit.

On these servers, solver runtime may be affected by hard-to-predict OS and hardware characteristics. We purposely limited the number of jobs on each server (two jobs per server) to reduce variability. By running the same binary multiple times, we observed variation in runtime of about 4% between the fastest and slowest run. This variability must be taken into account when comparing solver runtimes. Ideally, we should run the same solvers multiple times to get averages but we did not have enough time for this. Our primary performance metrics is the number of solved benchmarks, which is less sensitive to small runtime variability.

We ran all the solvers in Table 1 once on all the benchmarks. All solvers were run with default configurations (i.e., no command-line options), but we disabled generation of DRAT proofs. To perform the experiments, we had to address a few software issues:

- We fixed a division-by-zero bug in the smallsat solver.
- A more significant problem is that many, if not all, of the MiniSAT-derived solvers do not respect the SIGXCPU signal which we used for timeout. We set a runtime limit with the shell `ulimit` command. When the limit is reached, the OS sends the solver the SIGXCPU signal, which by default should terminate the process. Solvers intercept this signal and implement a signal handler that is intended to print statistics when the solver is interrupted. This mechanism seems to have been inherited from MiniSAT where it was working properly but the quality of implementation has not kept up. Many solvers just catch the signal and keep going. They cannot be interrupted by our timeout mechanism

Solver	SAT	UNSAT	TOTAL	Uniq.
cadical-1.2.1	9024	6145	15169	9
cadical-satrace19	9009	6136	15145	8
MapleCOMSPS_LRB	9000	6101	15101	3
MapleLCMDiscChronoBT-DL-v3	8977	6100	15077	3
expMaple_CM_GCBumpOnlyLRB	8922	6097	15019	
cryptominisat5	8898	6114	15012	1
Maple_LCM_Dist	8929	6077	15006	
MapleLCMDistChronoBT	8924	6067	14991	
MapleLCMdistCBTcoreFirst	8924	6064	14988	
MapleLCMDistChronoBT-DL-v1	8920	6065	14985	1
MapleLCMDistChronoBT-DL-v2.1	8919	6064	14983	
smallsat	8924	6048	14972	6
MapleCOMSPS	8912	6056	14968	
PSIDS_MapleCMDistChronoBT	8917	6048	14965	
glucose-4.2.1	8868	6039	14907	
minisat-2.2.0-simp	8897	5739	14636	1
virtual best	9058	6181	15239	

Table 2: Number of Solved Problems. The fourth column shows the total number of problems solved by each solver. The SAT and UNSAT columns show how many of these are satisfiable or unsatisfiable, respectively. The last column shows the number of uniquely solved benchmarks (i.e., that a solver is the only one to solve).

(or by ctrl-C for that matters). We fixed this issue by removing the faulty signal-handling code from all solvers.

- We removed an incorrect warning produced by the DIMACS parser used by most MiniSAT-derived SAT solvers.
- For the solvers that did not provide it, we added an option to disable printing of satisfying assignments on their standard output. This helped reduce the volume of data produced by solvers from gigabytes to more manageable sizes.

We note that many of the MiniSAT-derived solvers do not build very cleanly. Compilation generates a very large number of warnings. Despite this, all the solvers appear to be reliable. We did not notice incorrect results from any solver. There was no disagreement between solvers on any benchmark that was solved by more than one of them.

5 Results

The results of our evaluation are shown in Tables 2 and 3. The first table shows the number of solved instances by each solver. It also includes results for the virtual best solver.[4] The second table shows runtime distributions.

A clear result is that both versions of CaDiCaL are significantly better on our benchmarks than the other solvers. MapleCOMSPS_LRB is in third place and solves 68 fewer problems than CaDiCaL-1.2.1. The winner of last year's SAT race is fourth and solves close to 100 fewer problems than CaDiCaL-1.2.1. Other solvers that did well in this SAT race are further behind.

---

[4] The virtual best solver is obtained by selecting the fastest solver on each problem.

Solver	1 s	10 s	100 s	1000 s	1200 s	timeouts
cadical-1.2.1	11850	2292	664	351	12	278
cadical-satrace19	11872	2256	659	336	22	302
MapleCOMSPS_LRB	10732	3268	690	390	21	346
MapleLCMDiscChronoBT-DL-v3	7753	5373	1612	420	19	370
expMaple_CM_GCBumpOnlyLRB	10006	3646	998	349	20	428
cryptominisat5	11253	2201	1034	498	26	435
Maple_LCM_Dist	7717	5236	1679	348	26	441
MapleLCMDistChronoBT	7733	5262	1636	329	31	456
MapleLCMdistCBTcoreFirst	7715	5266	1649	338	20	459
MapleLCMDistChronoBT-DL-v1	7775	5246	1610	333	21	462
MapleLCMDistChronoBT-DL-v2.1	7768	5303	1559	331	22	464
smallsat	10815	3240	617	273	27	475
MapleCOMSPS	10671	3223	758	294	22	479
PSIDS_MapleCMDistChronoBT	7686	5097	1815	349	18	482
glucose-4.2.1	12188	1778	556	353	32	540
minisat-2.2.0-simp	11713	1709	776	415	23	811
virtual best	13233	1281	449	262	14	208

Table 3: Runtime Distribution. The first column shows the number of problems solved in less than 1 s. The second column shows the number of problems solved within 1 to 10 s, and so forth. The last column shows the number of timeouts, i.e., problems not solved in 20 min.

Apart from MapleLCMDiscChronoBT-DL-v3, solvers from 2019 do not seem particularly better than winners of past SAT competitions. It is also notable that two variants of MapleLCM-DiscChronoBD-DL-v3 do much worse. In fact, the best solver after CaDiCaL is MapleCOM-SPS_LRB, which finished in 7th position in the 2016 SAT Competition. Interestingly, the winner that year was a variant of MapleCOMPSPS_LRD which does not do well on our benchmarks. Also, MapleCOSMPS_LRB does much better than more recent solvers that implement techniques such as chronological backtracking [15] and learned-clause vivification [12]. These new techniques seem to be useful on SAT-Competition benchmarks but their value is less clear here.

One can see that all solvers are significant improvements over MiniSAT. MiniSAT 2.2.0 is last in our table. It times out on 811 problems, which is close to 300 more timeouts than any other solver. It is actually handicapped by its poor results on a specific family of benchmarks due to Bruttomesso and Sharygina [4]. All other solvers work fine on these benchmarks and we know that the benchmarks in question are unsatisfiable and have short resolution proofs. In fact, they can be solved by bounded variable elimination alone. Although MiniSAT implements bounded variable elimination, it puts limits on the procedure that prevents it form solving these benchmarks. In particular, MiniSAT uses a limit on the size of clauses created during variable elimination. This limit is 20 literals by default. When this limit is increased to 400 literals (using command-line option `--cl-lim=400`), MiniSAT solves 185 more problems in total.

Table 3 shows that most of our benchmarks are very easy for all solvers. A very large majority of the problems (75%) are solved in less than 10 s by all solvers. We still see differences between solvers on these easy benchmarks, in particular in the number of instances solved within 1 s. On the other hand, more than 200 problems were not solved at all. We have built a smaller subset of “interesting” benchmarks that removes problems on which all solvers behave similarly. This list includes all problems that are solved by some but not all solvers. We also included

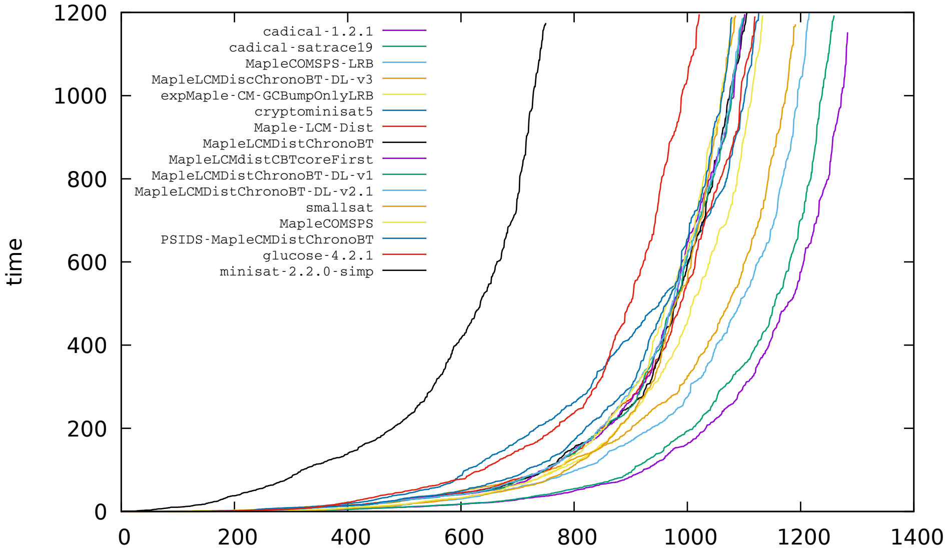

Figure 1: Cactus Plot on Interesting Problems. The legend lists solvers from best (rightmost curve) to worst (leftmost curve).

problems solved by all solvers when the difference between fastest and slowest solvers was large (i.e., more than 100 s). This list of interesting problems contains 1354 instances. Figure 1 shows the traditional *cactus plot* for our solvers on these interesting problems. This plot visually shows what we described previously. Both versions of CaDiCaL are our best solvers, followed by MapleCOMSPS_LRB and MapleLCMDistcChronoBT-DL-v3 then a group of ten solvers that are close to each other. Glucose-4.2.1 is then behind this group and MiniSAT is further to the left.

6 Playing with CaDiCaL

CaDiCaL [3] is one of the many SAT solvers developed by Armin Biere over the years. It is written in C++ and its code is available on GitHub.[5] It started participating in the SAT competitions in 2017. In our study, we used the release 1.2.1 of CaDiCaL which was the latest release available at the time.

As we mentioned previously, CaDiCaL uses in-processing and implements a large number of different simplifications and other specialized procedures. We experimentally investigate which of these features matter most on our benchmarks. For this purpose, we just turn off several options or features that are enabled by default in CaDiCaL then measure performance of the modified CaDiCaL.

compacting: compacting internal variables
chrono: support for chronological backtracking
decompose: elimination of equivalent literals
eagersubsume: apply subsumption to recently learned clauses
elim: bounded-variable elimination
elimgates: recognize clauses that encode and, xor, and if-then-else
lucky: try predefined satisfying assignments
probing: failed-literal probing
rephase: periodically switch preferred variable polarity
scan-index: optimized watched literal search
stabilize: switch between two heuristic modes
subsumption: clause subsumption
ternary: hyper ternary resolution
vivify: clause vivification
walk: random walks

Figure 2: Tested Features in CaDiCaL. Each feature is enabled by default and enables specific CaDiCaL procedures. Except for *scan-index*, all are controlled by command-line options
.

6.1 Overview of Tested Features

Among the many features implemented in CaDiCaL, we selected the subset shown in Figure 2. All of these are enabled by default and activate various simplification and search procedures available in CaDiCaL. Some of these are standard simplifications such as variable elimination and subsumptions [6], and failed-literal probing. Chronological backtracking is based on [15]. Vivification was introduced by Piette et al. [18] and it is used in Luo et al.'s procedure for minimizing learned clauses [12]. It was also proposed by Han and Somenzi under the name *distillation* [9].

Compacting is a data-structure optimization procedure that makes internal tables more compact by removing empty slots and renumbering variables. *Decomposition* computes strongly connected components in the problem's binary implication graph to eliminate variables. It searches for implication cycles of the form: $l_0 \Rightarrow l_1 \Rightarrow \ldots \Rightarrow l_n \Rightarrow l_0$, from which one can deduce that $l_1, \ldots, l_n$ can all be replaced by l_0. *Eager subsumption* keeps track of n most recently learned clauses and when a new clause C is learned, it checks whether C subsumes any of them. Under the name *elimgates* we refer to three options of CaDiCaL that enable code to recognize clausal encodings of common Boolean gates. These clauses can be treated specially during bounded variable elimination [6]. *Lucky* and *walk* refer to two procedures that attempt to quickly find a satisfying assignment before executing the CDCL search procedure. Walk is a local search procedure. Lucky tries several assignments (e.g., set all variables to false or to true) to check whether they satisfy all the clauses. *Rephase* is a heuristic that periodically updates the preferred polarity used when assigning decision variables. Other solvers typically use the caching scheme due to Pipatsrisawat and Darwiche [19]. Rephasing introduces more diversity in this scheme. *Stabilize* enables CaDiCaL to essentially work in two modes in which different heuristics are used for selecting branching variables and controlling restarts. It is related to the distinction between SAT and UNSAT problems observed by Oh [16]. *Ternary* is a form of resolution limited to three (and two) literal clauses.

The *scan-index* feature is different from the others. It is not enabled or disabled by

---

[5] https://github.com/arminbiere/cadical

Disabled Feature	SAT	UNSAT	TOTAL	Impact
elim	9002	6096	15098	−71
stabilize	8961	6141	15102	−67
rephase	8988	6149	15137	−32
scan-index	9026	6123	15149	−20
probing	9022	6130	15152	−17
compacting	9013	6144	15157	−12
vivify	9018	6143	15161	−8
subsumption	9017	6144	15161	−8
ternary	9015	6148	15163	−6
decompose	9023	6142	15165	−4
eagersubsume	9023	6142	15165	−4
	9024	6145	15169	0
lucky	9025	6151	15176	+7
chrono	9021	6156	15177	+8
walk	9024	6154	15178	+9
elimgates	9025	6157	15182	+12

Table 4: Impact of Disabling Features or Options in CaDiCaL-1.2.1. The impact is the difference in numbers of solved problems between the default cadical and cadical with a feature disabled. A negative impact means that the feature is helpful. A positive impact means that cadical does better when the feature is disabled.

command-line options. Unique among all solvers in our list, CaDiCaL implements an optimal procedure for scanning a clause to search for a new watched literal [8]. This requires storing a scan index with the clause and searching from this scan index whenever a current watched literal becomes false. Other solvers use the simpler method of MiniSAT that does not require a scan index. They always scan a clause from the start. We wanted to evaluate the impact of this scan-index procedure. To disable it, we modified the CaDiCaL code to force scanning to start at the beginning of a clause.

Although some of the procedures listed in Figure 2 are also implemented by CryptoMiniSAT and other solvers, implementation details matter. CaDiCaL uses various optimizations that may not be implemented in other solvers. Checking the code is a good idea for full details.

6.2 Results

Table 4 shows the results of our experiment. Each row of the table lists a specific features and gives the number of SAT and UNSAT benchmarks solved when this feature is disabled in CaDiCaL. We also give the total number of solved benchmarks and the difference in number of solved benchmarks compared with the default CaDiCaL 1.2.1.

Based on Table 4, we can see that bounded variable elimination, stabilization, and rephasing have the most impact on performance. Rephasing and stabilization are particularly useful on satisfiable benchmarks. The scan-index procedure, failed-literal probing, and compacting also help. Vivifying, subsumption, hyper ternary resolution, decomposition, and eager subsumption seem to also provide small improvements. Interestingly, we get better results than the default CaDiCaL by disabling several features. The lucky and random walk search make things worse on our benchmarks, as do chronological backtracking and special treatment of clauses that encode logical gates.

Some of these results remain to be more carefully validated. Small variations in number of solved benchmarks should be taken with precaution since CaDiCaL is a randomized solver. We have tested CaDiCaL on our benchmarks with 18 different random seeds. In this test, the number of solved instances varied from 15156 to 15176, with an average of 15167.33 and a standard deviation of 4.59.

Table 4 gives us an initial picture of features of a state-of-the-art SAT solver like CaDiCaL that matter most for our bit-vector benchmarks. This study is far from complete, as CaDiCaL includes many more parameters and options than the ones we tested. We limited our experiments to Boolean features that can be turned on or off. Other aspects that are unique to CaDiCaL (such as the use of the *move-to-front* heuristic) are more difficult to investigate since they require significant code modification.

7 Conclusion

Efficient SAT solvers are key to solving SMT problems in the theory of fixed size bit vectors. Progress in SAT solving is hard to quantify. It is measured empirically on benchmarks used in regular SAT competitions. Unfortunately, it is not clear whether good performance in these SAT competitions correlate with good performance on SMT benchmarks. Our empirical evaluation shows that CaDiCaL is currently the best SAT solver on CNF problems produced with Yices 2, by a significant margin. Other solvers that are close to or better than CaDiCaL on SAT-competition benchmarks are not close in our evaluation. This implies that SAT-competition benchmarks are different and not representative of the SAT problems we produce by bit blasting.

Our initial investigation identified several features and procedures of CaDiCaL that seem to be most beneficial on our benchmarks. Some of these are not difficult to implement and could be easily added to other solvers. Other procedures such as chronological backtracking that have proved effective in SAT competitions do not seem to help on our benchmarks.

Benchmarking is of course a difficult problem and our evaluation is still limited. We have heard that other SMT solvers than Yices work best with CaDiCaL too,[6] but a larger experimental evaluation involving several solvers would be useful. We have not mentioned various potential issues with the SMT-LIB benchmarks (e.g., many hard problems are crafted or do not come from real application domains, there may be too many similar problems), and we have not thoroughly examined SAT solver performance on different benchmark families.

Acknowledgments

This material is based upon work supported in part by NSF grant 1816936, and by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-18-C-4011. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, DARPA, or SSC Pacific.

References

[1] Gilles Audemard and Laurent Simon. GLUCOSE: a solver that predicts learnt clauses quality. In *SAT 2009 Competitive Events Booklet*, pages 7–9, 2009.

---

[6] Personal conversation with Mathias Preiner

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

[3] Armin Biere. CaDiCaL at the SAT race 2019. In *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, pages 8–9, 2019.

[4] Roberto Bruttomesso and Natasha Sharygina. A scalable decision procedure for fixed-width bit-vectors. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD'09)*, pages 13–20, 2009.

[5] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[6] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Practice of Satisfiability Testing (SAT 2005)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[7] Niklas Eén and Niklas Sörensen. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[8] Ian P. Gent. Optimal implementation of watched literals and more general techniques. *Journal of Artificial Intelligence Research*, 48:231–252, 2013.

[9] Hyojung Han and Fabio Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*, pages 582–587, 2007.

[10] Marijn J.H. Heule, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2019. http://hdl.handle.net/10138/306988.

[11] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing (SAT 2016)*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Sprnger, 2016.

[12] Mao Luo, Chu-Min Li, Fan Xio, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 703–711, 2017.

[13] João P. Marques-Silva and Karem A. Sakallah. GRASP - a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference (DAC'01)*, pages 530–535, June 2001.

[15] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing (SAT 2018)*, volume 10929 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018.

[16] Chanseok Oh. Between SAT and UNSAT: The fundamental difference in CDCL SAT. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing (SAT 2015)*, volume 9340 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2015.

[17] Chanseok Oh. Patching MiniSat to deliver performance of modern SAT solver. In *SAT Race Solver Description*, 2015.

[18] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs. Vivifying propositional clausal formulas. In *18th European Conference on Artificial Intelligence (ECAI 2008)*, Frontiers in Artifical Intelligence and Applications, pages 525–529. IOS Press, 2008.

[19] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *Lecture Notes in Computer Science*, pages

294–299. Springer, 2007.

[20] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.