Assessing Progress in SAT Solvers Through the Lens of Incremental SAT *

Stepan Kochemazov[1], Alexey Ignatiev[2], and Joao Marques-Silva[3]

[1] ITMO University, St. Petersburg, Russia
veinamond@gmail.com
[2] Monash University, Melbourne, Australia
alexey.ignatiev@monash.edu
[3] IRIT, CNRS, Toulouse, France
joao.marques-silva@irit.fr

Abstract. There is a wide consensus, which is supported by the hard experimental evidence of the SAT competitions, that clear progress in SAT solver performance has been observed in recent years. However, in the vast majority of practical applications of SAT, one is expected to use SAT solvers as oracles deciding a possibly large number of propositional formulas. In practice, this is often achieved through the use of incremental SAT. Given this fundamental use of SAT solvers, this paper investigates whether recent improvements in solver performance have an observable positive impact on the overall problem-solving efficiency in settings where incremental SAT is mandatory or at least expected. Our results, obtained on a number of well-known practically significant applications, suggest that most improvements made to SAT solvers in recent years have no positive impact on the overall performance when solvers are used incrementally.

1 Introduction

Boolean Satisfiability (SAT) solving can only be viewed as one of the most important successes of computer science. SAT was the first decision problem to be proved NP-complete in the early 70s [4]. As a result, and unless P = NP, SAT being NP-complete implies in theory and in practice that the worst-case running time of SAT algorithms grows exponentially with the number of variables. This was indeed the case until the early 90s, with SAT solvers capable at best of deciding formulas with a few hundred variables and a few thousand clauses. However, since the mid 90s, and building on the well-known DPLL algorithm [5, 6], a stream of new algorithmic improvements led to what is now known as CDCL (conflict-driven clause learning) SAT solvers[4]. CDCL SAT solving revolutionized the size and complexity of the formulas that SAT solvers can decide efficiently in practice. Indeed, it is well-known that in many applications, modern CDCL SAT solvers routinely decide formulas with a few million variables and tens of millions clauses. As a direct consequence of these algorithmic improvements,

* Stepan Kochemazov is supported by the Ministry of Science and Higher Education of Russian Federation, research project no. 075-03-2020-139/2 (goszadanie no. 2019-1339). Joao Marques-Silva is supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future - PIA3" under Grant agreement n^o ANR-19-PI3A-0004.

[4] It is generally accepted that the term CDCL was coined by L. Ryan [40].

the last two decades have witnessed an ever increasing range of highly significant practical applications [3, Ch. 04]. (The techniques used in SAT solvers have also found widespread use in other automated reasoners, including SMT [3, Ch. 33], ASP [9], CP/LCG [37], ILP and even theorem provers [3].) This success makes SAT one of the few NP-complete problems that has achieved widespread practical deployment. Among the improvements made to DPLL-style SAT solvers, it is generally accepted that clause learning [23,25,26] played a fundamental role, not only because of its remarkable ability to prune the search space for practical formulas, but also because it enables other techniques to become very effective in practice. Other improvements of notice include search restarts [10], branching heuristics [34], watched literals [34], phase saving [39] and literal block distance [2]. (A detailed account can be found in different chapters of the recent SAT handbook [3].) Furthermore, it is generally accepted that the hard experimental evidence offered by the SAT competitions supports the following assertion: "*There has been regular performance improvements in SAT solvers over the years*".

In terms of practical uses of SAT, incremental SAT is by far the most often used option. The most widely used approach for instrumenting incremental SAT was proposed originally in the MiniSat solver [7], using the so-called activation (or selection variables). Since then, a wealth of practical applications of SAT have resorted to incremental SAT solving (e.g. [3, Ch. 04] and references therein). The importance of incremental SAT is underscored for example by specific optimizations in the engineering of recent SAT solvers [1,11,17]. As yet another example, the PySAT framework for prototyping with SAT solvers makes extensive use of incremental SAT solving [12].

This paper seeks to understand how significant the recent progress made in SAT solvers, as documented by the results from the SAT competitions, is to practical incremental SAT solving. Although there is a well-known incremental track in the SAT competitions, the problem we address is somewhat different. First, our investigation is not limited to the applications considered in the incremental track of the SAT competition; indeed we consider applications of well-known importance, but which are not contemplated in the incremental track. Second, we do not seek to find the best SAT solver, but instead to assess whether specific algorithmic improvements made to SAT solvers contribute visibly to improve solver performance, specifically when the goal is incremental SAT solving[5]. Our results indicate that, contrary to the conclusions obtained from the results of the SAT competition, most of the improvements made to SAT solvers in recent years do not contribute in a visible way to improving the performance of SAT solving. Furthermore, based on the prominent role of incremental SAT in practical problem solving, one overall recommendation of this work is that the assessment of future SAT solvers should take into account their observed performance improvements with respect to incremental SAT solving.

The paper is organized as follows. Section 2 briefly introduces the notation and definitions used throughout. Section 3 analyzes the results of the SAT competition, aiming to draw the general conclusions that justify our assertion above. Section 4 outlines how a SAT solver was instrumented to enable the proposed study. Section 5 presents the experimental results we have obtained. Section 6 concludes the paper.

[5] Similarly, in the area of Satisfiability Modulo Theories (SMT) reasoning [3], it is generally accepted that not all optimizations made to SAT and SMT solvers find widespread use.

2 Preliminaries

Definitions & notation. The standard definitions used in SAT solving are assumed throughout the paper [3]. A SAT solver decides the decision problem of propositional logic (for formulas represented in conjunctive normal form (CNF)). For satisfiable formulas, a SAT solver returns a model, i.e. an assignment that satisfies the formula. For unsatisfiable formulas, most CDCL SAT solvers will return a non-minimal explanation for unsatisfiability. Most modern CDCL SAT solvers offer an incremental interface, without exception inspired by the incremental interface of MiniSat [7]. Incremental SAT solving finds an ever increasing range of practical applications (a sample of which are documented e.g. [3, Ch. 04]).

Related work. A number of papers have investigated improvements to the performance of SAT solvers from different perspectives [8, 15]. However, none has investigated how improvements made to SAT solvers impact incremental SAT solving. The importance of incremental SAT solving is demonstrated by its ubiquitous use in e.g. the PySAT framework [12]. Before PySAT was developed, incremental SAT was extensively used in a wide range of applications of SAT (a brief account is available from [3, Ch. 04]). The importance of incremental SAT explains a number of recent optimizations made to SAT solvers [1, 11, 17].

3 Motivation

It is natural to view the winners of recent SAT Competitions as state of the art in CDCL SAT solving. They incorporate the most promising CDCL heuristics aimed at improving solvers' performance over a wide variety of benchmarks originating from different application domains. There are two SAT solvers that played a special role in SAT competitions over the years. They are MiniSat [7] and Glucose [2]. It will not be an exaggeration to say that from 2005 to 2015 the list of competition winners in all categories tended to include at least one version (or sometimes *hack*) of either one or the other.

In 2015 the COMiniSatPS solver [36] combined the Luby series restarts [21] from MiniSat and Glucose-style restarts into a single whole, where the solver switched between two modes, each exploiting one of the restart strategies in conjunction with VSIDS activity values specific to each mode.

In 2016 MapleCOMSPS [20] supplanted Glucose as the source of many of the winners of SAT Competitions 2016 to 2020. The key novel feature of MapleCOMSPS was the use of *learning rate branching* heuristic (LRB) [19] instead of VSIDS joined with Luby restarts in COMiniSatPS. It also replaced the scheme for switching between modes employed by COMiniSatPS by a simple variant where the LRB+Luby restarts mode is used exclusively during the first 2500 seconds.

In 2017 MapleCOMSPS has been extended with an expensive *inprocessing* [14] technique for improving the quality of learnt clauses, termed *learned clause minimization* [18, 22, 38], with the resulting solver called MapleLCMDist.

In 2018 the latter was augmented with the *chronological backtracking* [31, 35] heuristic aimed at improving the solver behavior in specific cases, in form of MapleL-

CMDistChronoBT. (The chronological backtracking scheme mimics the organization of backtracking used in the GRASP SAT solver [25].)

The winner of SAT Race 2019 implemented on top of MapleLCMDistChronoBT the so-called *duplicate learnts heuristic* [16] aimed at detecting and exploiting repeatedly learned clauses, resulting in the MapleLCMDistChronoBT-DL-v3 solver. It also changed the parameters of LBD-based separation between tiers of learnt clauses and used a new scheme for switching between solver modes, which is reminiscent to the one used in MapleCOMSPS but is less frequent.

Finally, Relaxed_LCMDCBDL_newTech [41], that took the 2nd place at SAT Competition 2020, incorporated into MapleLCMDistChronoBT-DL-v3 the stochastic local search (SLS) component, complemented with *rephasing* technique[6], and a novel approach that modifies the activity values of branching heuristic in a CDCL solver based on some of the statistics accumulated by the SLS component. It also modified the scheme for switching between solver modes.

There is a clear agreement that the performance of SAT Competition winners improved significantly over the years[7], thanks to the several major heuristics listed above, among others. In this context, it is very surprising that many applications of SAT solvers still employ the time-tested MiniSat and Glucose first introduced back in 2003 [7] and 2009 [2], respectively. For example, if we look at the participants of the recent MaxSAT Evaluations [28–30], it turns out that the majority of them employ either Glucose (different versions), MiniSat 2.2 or COMiniSatPS as the underlying SAT solver. Of course, there are many possible reasons for this, varying from unwillingness of developers to replace the core components of working tools under pretext that they already work well enough, to the fact that many SAT Competition winners of recent years do not provide incremental interface out of the box. However, whatever the reasons behind this are, it is important to *question* and *evaluate* whether the apparent progress in CDCL SAT solvers indeed translates into the benefits in their practical applications.

Of course, it is impossible to cover all possible use cases of SAT solvers in a single study. Therefore, in this paper we concentrate our attention on the ones that can be employed incrementally, in particular, in maximum satisfiability (MaxSAT) solving and minimal unsatisfiable subset (MUS) extraction.

4 Setup and Its Rationale

In order to evaluate whether or not the improvements made to SAT solvers in recent years contribute to their performance in the incremental setting, it is first necessary to choose a solver (or solvers) that could serve as (a) strong representative(s) of the solver "generation". It would be ideal to exploit the winner of the most recent SAT Competition that implements all the recently proposed CDCL heuristics, can be easily modified to enable or disable some of the heuristics whenever needed, and that can be embedded into various tools that could apply it incrementally. Unfortunately, the winner of the SAT Competition 2020, the Kissat solver does not support the incremental mode according to the data available at the moment of writing, and thus is not eligible for

[6] http://fmv.jku.at/chasing-target-phases/

[7] http://fmv.jku.at/kissat/

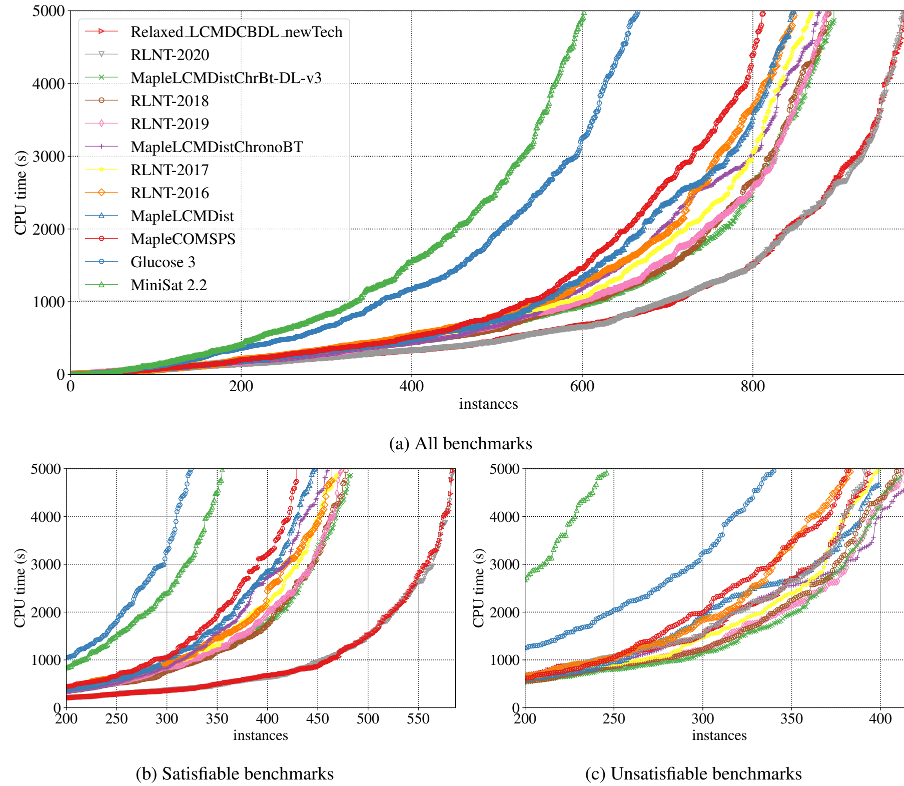
the experiments that we need to perform. However, Relaxed_LCMDCBDL_newTech that took the 2nd place in the main track of SAT Competition 2020 satisfies all the aforementioned criteria. Despite not supporting incremental SAT out of the box, it uses the MiniSat codebase and thus can be easily upgraded.

For our experiment we prepared a variant of Relaxed_LCMDCBDL_newTech, which we hereinafter refer to as RLNT[8]. Compared to the original, RLNT supports incremental mode, has several small issues fixed and also allows to separately enable or disable some of its major heuristics. In particular, we are interested in testing the implementations of *stochastic local search* and *rephasing* components (**SLS**) introduced in Relaxed_LCMDCBDL_newTech, the *duplicate learnts* (**DL**) heuristic that appeared in MapleLCMDistChronoBT-DL-v3, the *chronological backtracking* (**CB**) that became a signature of MapleLCMDistChronoBT, the *DISTANCE* (**DIST**) and *learnt clause minimization* heuristics (**LCM**) first introduced in MapleLCMDist. These were modified in order to be enabled or disabled via preprocessor conditional inclusive directives (e.g. `#define SLS` and `#ifdef SLS`). As it happens, these heuristics represent the vital development steps signifying the progress of SAT solvers in the last 4 SAT Competitions and so can serve as *inherent characteristics* of the corresponding generations of SAT solvers.

The listed heuristics employed in RLNT were not specifically adapted to the incremental usage separately. Instead, the variables that govern scheduling of the procedures that switch between solver modes, apply learnt clause minimization, apply rephasing, and so on — these are all reset to initial values with each new call to the SAT solver. The motivation for this adjustment is to alleviate the increase in intervals between, e.g. learnt clause minimization, so that each call of a SAT solver preserves the accumulated knowledge, but still uses all the heuristics as often as it would in the *"standard"* non-incremental mode. In line with this, the call to the SLS component is scheduled to happen at the start of each SAT solver invocation since this is the way it is used in the original implementation. Due to the fact that it mainly affects rephasing, and also that SLS subsolver calls take negligible amount of time, this implementation should not introduce any adverse effects on the solver's performance. It should be noted that the changes between, say, the SAT competition 2016 winner MapleCOMSPS and Relaxed_LCMDCBDL_newTech certainly cannot be summarized to just the 5 heuristics listed above. There have been also small changes to the handling of conflict clauses with small literal block distance, and to the strategy employed to switch between solver modes that combine branching heuristics with restart strategies (LRB+Luby restarts and VSIDS + glucose restarts). It is natural to assume that these changes are worthwhile, but it should be checked experimentally anyway. Therefore, in the following experiments we opt to use the following SAT solvers:

- MapleCOMSPS – SAT Competition 2016 winner.
- MapleLCMDist – SAT Competition 2017 winner.
- MapleLCMDistChronoBT – SAT Competition 2018 winner.
- MapleLCMDistChronoBT-DL-v3 – SAT Race 2019 winner.
- Relaxed_LCMDCBDL_newTech – SAT Competition 2020 2nd place.
- RLNT-2020 – RLNT with SLS, DL, CB, LCM and DIST enabled.

[8] https://github.com/veinamond/RLNT



(a) All benchmarks

(b) Satisfiable benchmarks

(c) Unsatisfiable benchmarks

Fig. 1: Evaluation of considered solvers over benchmarks from the main tracks of SAT Competitions 2017-2020.

- RLNT-2019 – RLNT with DL, CB, LCM and DIST enabled.
- RLNT-2018 – RLNT with CB, LCM and DIST enabled.
- RLNT-2017 – RLNT with LCM and DIST enabled.
- RLNT-2016 – RLNT with SLS, CB, DL, LCM and DIST disabled.
- Glucose 3.0.
- MiniSat 2.2.

Thus, our conjecture is that RLNT-2019 should be functionally “equivalent” to the winner of SAT Race 2019, RLNT-2018 to the winner of SAT Competition 2018, etc. (The results of the following section confirm this.) To offset the newer solvers we will use Glucose 3.0 and time-tested MiniSat 2.2, which both are often employed in many practical applications up to these days, although having been released back in 2013 and 2008, respectively.

4.1 SAT Competition Main Track Benchmarks

In this experiment we used the benchmarks from the main tracks of the SAT Competitions 2017–2020; thus, the total number of benchmarks considered is 1550. The exper-

Table 1: The detailed statistics on the performance of considered solvers over benchmarks from the main tracks of SAT Competitions 2017-2020.

	SCR	SAT	UNSAT	PAR-2
Relaxed_LCMDCBDL_newTech	980	585	395	4199
RLNT-2020	978	586	392	4203
MapleLCMDistChrBt-DL-v3	896	484	412	4772
RLNT-2018	890	479	411	4811
RLNT-2019	889	474	415	4822
MapleLCMDistChronoBT	879	461	418	4933
RLNT-2017	870	470	400	4961
RLNT-2016	849	465	384	5096
MapleLCMDist	849	449	400	5107
MapleCOMSPS	813	430	383	5303
Glucose 3	666	325	341	6238
MiniSat 2.2	603	356	247	6606

iments were performed on the nodes of the computing cluster [27], equipped with two 18-core Intel Xeon E5-2695 v4 CPUs and 128 GB RAM. All the competitors worked in 36 simultaneous threads with the time limit of 5000 seconds. As the evaluation criteria, we used the Solution Count Ranking (SCR) and Penalized Average Runtime (PAR-2) following the metrics used in the SAT Competitions.

The results of the evaluation are presented in the form of cactus plots in Figure 1 and as a more detailed statistics in Table 1. From the presented results, it is easy to conclude that in accordance with the SAT Competition criteria, the RLNT configurations perform as well as (or *better* than) the corresponding SAT Competition winners with negligible deviations. This confirms that the rationale behind the selected baseline solver as well as the implementation choices made is reasonable. Also, one can easily observe the trend according to which the recent competitions favor satisfiable benchmarks over unsatisfiable, thus making the solvers which are stronger on satisfiable benchmarks look better. The particularly distinctive difference between Relaxed_LCMDCBDL_newTech (and RLNT-2020) and the remaining group is thanks to the SLS component that appears to be solely responsible for being able to tackle at least 80 benchmarks. Finally, the performance of Glucose 3 and MiniSat 2.2 when contrasted with that of more modern solvers appears to be an issue. It is especially so if we look at the performance of MiniSat 2.2 on unsatisfiable benchmarks compared to that of the competition.

Now let us see whether or not the overall picture will change when we move into the incremental context.

4.2 SAT Competition Incremental Track Benchmarks

In this series of experiments we used the benchmarks and applications from the Incremental Track of SAT Competition 2020. The solvers participating in this track have to support the *IPASIR*[9] incremental interface. In the course of the evaluation, the solvers

[9] https://github.com/biotomas/ipasir

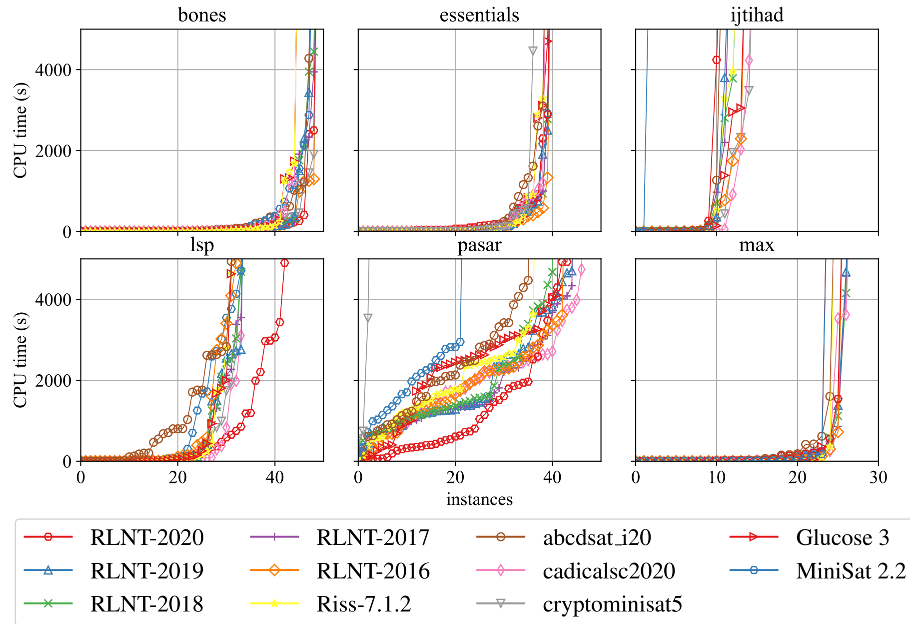


Fig. 2: Evaluation of considered solvers over benchmarks from the incremental track of SAT Competitions 2020.

are compiled into an incremental library together with specific IPASIR-based applications that aim to cover various practical domains that may employ incremental solvers. In 2020, the incremental track included the applications for (a) finding backbones of a CNF SAT formula, (b) finding variables essential for the satisfiability of a formula, (c) finding the longest simple path in a graph, (d) a simple MaxSAT solver, (e) Ijtihad QBF solver and (f) the PASAR solver for solving planning instances. For each application, there were 50 instances (which overlapped in the case of finding backbones and the variables essential for satisfiability). Due to high requirements to the execution environment, in particular for it to support C++ 17, we used PCs with 16-core AMD Ryzen 3950x CPUs and 32 GB RAM running Ubuntu 20.04, as the computing platform. The solvers were launched in 16 threads.

The results of this experiment are summarized in Table 2 and cactus plots in Figure 2. It needs to be noted, that since incremental track is significantly less popular than the main track (at least looking at the number of participants of each), it is less polished and is harder to reproduce. In particular, the outputs and the success criterion for each application have to be parsed by hand, the number of instances is small, and the majority of instances are too simple. Moreover, there are frequent problems when the built application produces a core-dump and it is unclear whether the application itself is to blame or the solver was not built properly. This is the reason, for example, of the poor performance of the CryptoMiniSat5 solver in the *pasar* application or of MiniSat 2.2 in both *ijtihad* abd *pasar* applications: the majority of launches ended in a core-dump. (It

Table 2: The detailed statistics on the performance of considered solvers over benchmarks from the incremental track of SAT Competitions 2020. The best results for each application are marked with bold. Column S refers to the number of solved instances, P2 – to the PAR-2 score.

	bones		essentials		lsp		max		ijtihad		pasar	
	S	P2	S	P2	S	P2	S	P2	S	P2	S	P2
RLNT-2020	49	350	40	2219	**43**	**1987**	26	4869	11	6879	44	2325
RLNT-2019	48	580	40	2170	34	3546	27	4753	12	6845	45	2733
RLNT-2018	49	482	40	2168	34	3577	**27**	**4739**	13	6393	41	3254
RLNT-2017	49	441	40	2174	34	3542	27	4745	12	6845	45	2553
RLNT-2016	**49**	**322**	**40**	**2086**	33	3837	26	4826	14	6369	43	2902
Glucose 3	45	1108	40	2273	32	3845	24	5211	14	6895	42	3387
MiniSat 2.2	48	635	40	2180	34	3811	24	5212	2	9600	22	6380
Riss-7.1.2	45	1108	39	2388	32	3844	25	5013	13	7270	37	3907
abcdsat_i20	48	627	39	2450	32	4205	25	4696	11	7830	36	4295
cadicalsc2020	45	1085	39	2323	34	3381	27	4756	15	6729	**47**	**2400**
cryptoMiniSat5	49	333	37	2737	34	3495	26	4478	**15**	**5966**	3	9496

is unclear to us how the organizers treated such situations in their evaluation.) Nevertheless, the results we obtained more-or-less follow the ones available at SAT Competition 2020 web page.[10]

One conclusion to be drawn from the presented data is that the RLNT configurations perform in the incremental setting as well as the participants of the incremental track of SAT Competition 2020, and in several cases outperform them. It means that the RLNT solver and its configurations can be viewed as the state-of-the-art representatives of the modern SAT solvers. Thus, we are justified to use them for the following in-depth evaluation presented below. Another conclusion is that Glucose 3, although it is not a winner in any of the subtracks, is on par with most of the competitors for all the considered benchmarks. Of particular interest is the fact that in contrast to SAT Competition Main Track benchmarks, in the incremental track environment, MiniSat 2.2 is on par with its peers (with the exception of *pasar* and *ijtihad* applications where it has likely suffered from some implementation issue). Finally, one can observe that the incremental track of the SAT Competition 2020 does not provide a solid number of benchmarks that could demonstrate the performance differences (if any) among the competitors of the incremental track, leaving much to be desired. All these points bring us to the need to evaluate the progress in SAT solving in a thorough evaluation from the perspective of two well-known practical use-case scenarios for incremental SAT, which is covered next.

5 Experimental Evidence

This section details the experimental results obtained with the use of the developed configurations of RLNT in the two concrete practical settings of (1) maximum satisfiability (MaxSAT) solving and (2) minimal unsatisfiable subset (MUS) extraction,

[10] https://satcompetition.github.io/2020/results.html

where incremental calls to a SAT oracle are of crucial importance. Concretely, in all the following experiments we tested the 5 configurations of RLNT (2016–2020) and compared them to the Glucose 3 SAT solver [1], which has been widely used in various incremental settings. Finally, we additionally considered the *"good old"* MiniSat 2.2 [11] solver [7] to see how it stands against more advanced SAT solvers.

All the SAT solvers are integrated in the PySAT framework [12] and are used in a *unified* fashion through the same API. The conducted experiments involve testing three practical problem solvers: (1) an award-winning core-guided MaxSAT solver RC2 [13, 28–30] [12] (namely, competition configurations *RC2-A* and *RC2-B*), (2) a linear search SAT-UNSAT algorithm for MaxSAT [33] [13] (in the following referred to as *LSU*), and (3) a simple deletion-based MUS extractor [12, 24][14] (referred to as *MUSx*). All the problem solvers used are a part of the PySAT framework.

Note that the rationale behind the choice of the problem solvers is to test the performance of the underlying SAT oracles when dealing with (1) *mostly unsatisfiable* oracle calls, (2) *mostly satisfiable* oracle calls, and (3) *mixed* (satisfiable and unsatisfiable) oracle calls. Hereinafter, given a problem solver $*$, its configuration that exploits the Glucose 3 (resp. MiniSat 2.2) solver is marked as $*_{\mathrm{G3}}$ (resp. $*_{\mathrm{M22}}$) while the configurations using one of the RLNT solvers are marked by the corresponding year, as $*_{\mathrm{year}}$.

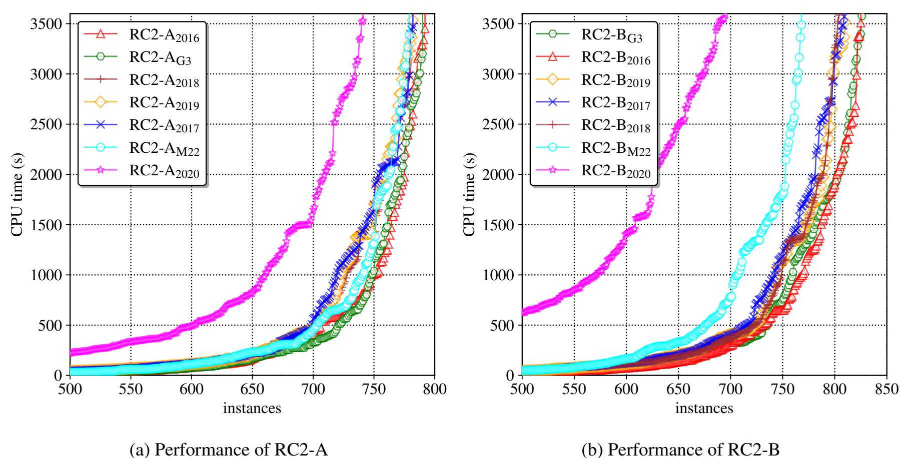
Our experimental setup replicates the setup of the annual MaxSAT Evaluations [28–30]. In particular, the experiments were performed on the *StarExec cluster*[15]. Each process was run on an Intel Xeon E5-2609 2.40GHz processor with 128 GByte of memory, in CentOS 7.7. The memory limit for each individual process was set to 32 GByte. The time limit used was set to 3600s for each individual process to run.

5.1 RC2 MaxSAT & Mostly Unsatisfiable Calls

The RC2 MaxSAT solver [13] belongs to the large family of core-guided MaxSAT solvers [33] and provides an efficient implementation of the OLL/RC2 algorithm [32]. For this reason, each iteration performed by the solver involves calling a SAT oracle incrementally given an unsatisfiable formula that is slightly modified at each iteration of the algorithm. The solver proceeds until the final iteration, which determines the working formula to be satisfiable. The solver can also be instructed to apply a few additional heuristics [13], some of which may increase the number of satisfiable oracle calls; however, unsatisfiable oracle calls made by RC2 still prevail. Note that the competition configurations RC2-A and RC2-B make use of the Glucose 3 SAT solver. Also note that this part of the experiment tested RC2 on the complete set of benchmarks (both unweighted and weighted) from the MSE'20.

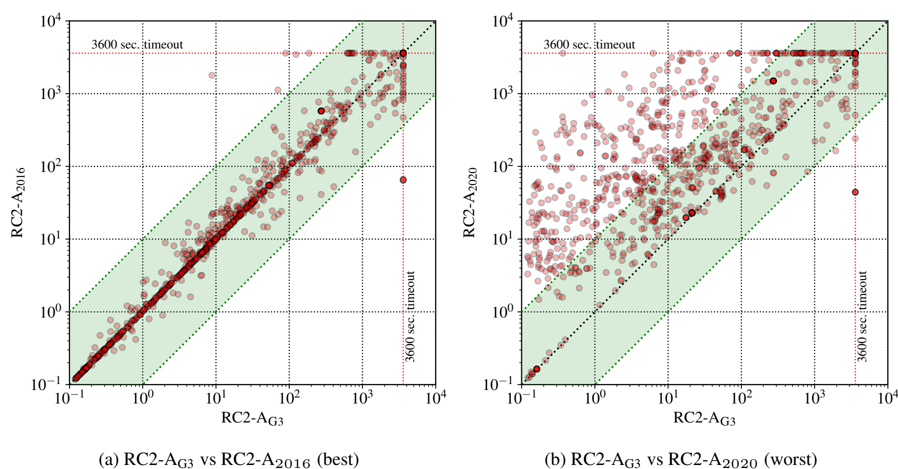
Figure 3 shows two cactus plots depicting the performance of the RC-A and RC2-B solvers on the MSE'20 benchmarks when using either Glucose 3 or one of the variants of RLNT as an underlying SAT oracle. According to Figure 3a, in total, the best per-

[11] https://github.com/niklasso/minisat
[12] https://pysathq.github.io/docs/html/api/examples/rc2.html
[13] https://pysathq.github.io/docs/html/api/examples/lsu.html
[14] https://pysathq.github.io/docs/html/api/examples/musx.html
[15] https://www.starexec.org/



(a) Performance of RC2-A (b) Performance of RC2-B

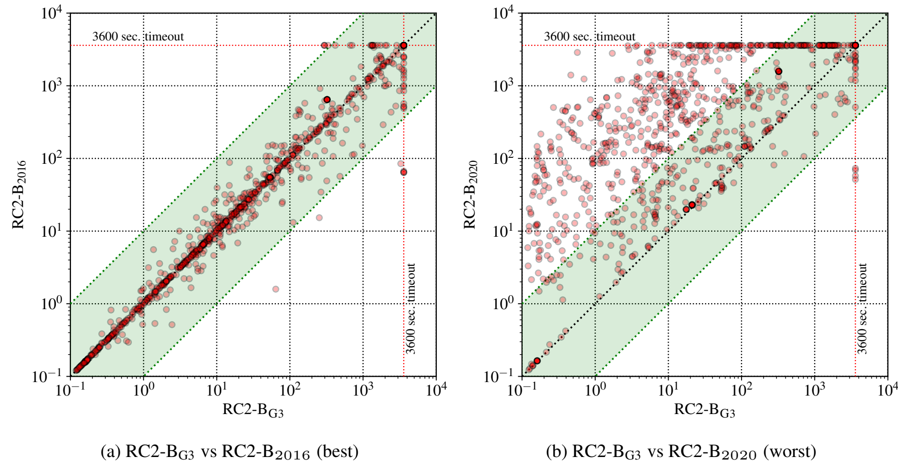
Fig. 3: RC2 with various SAT solvers on MSE'20 unweighted and weighted benchmarks.



(a) RC2-A_{G3} vs RC2-A_{2016} (best) (b) RC2-A_{G3} vs RC2-A_{2020} (worst)

Fig. 4: Performance of RC2-A_{G3} compared to RC2-A with *best* and *worst* RLNT.

formance of RC2-A is achieved when using RLNT-2016. It solves 792 instances and in average spends 1293.2 seconds per instance. The default, RC2-A_{G3} is not far away with 790 instances solved and the average time spent being 1290.8 seconds. The worst performance is demonstrated when RLNT-2020 is in use; here, the average time used per instance is 1549.8 seconds and the number of instances successfully solved is 741. As an additional remark, the MiniSat 2.2 based version is not far behind the top performing competitors – it solves 779 instances and spends 1327.5 seconds per instance on average. As can be seen in Figure 3b, similar results are obtained by RC2-B. The worst performance is shown by RC2-B_{2020}, which solves 695 benchmarks and spends 1805.8 seconds per formula on average. The default configuration RC2-B_{G3} outper-



(a) RC2-B_{G3} vs RC2-B_{2016} (best)

(b) RC2-B_{G3} vs RC2-B_{2020} (worst)

Fig. 5: Performance of RC2-B_{G3} compared to RC2-B with *best* and *worst* RLNT.

forms the other competitors with 826 instances solved in 1222.7 seconds on average while RC2-B_{2016} comes second with 825 instances solved in 1213.3 seconds on average. RC2-B_{M22} solves 768 instances with the average time of 1385.1 seconds. The scatter plots shown in Figure 4 and Figure 5 detail the performance comparison of the default version of RC2-A_{G3} and RC2-B_{G3} against the best- and worst-performing competitor running RLNT. As can be observed, there is no clear winner in the pair RC2-$*_{G3}$ vs RC2-$*_{2016}$ while for the lion's share of benchmarks the default versions of the solver working on top of Glucose 3 significantly outperform RC2-$*$ with the most advanced RLNT-2020.

5.2 LSU MaxSAT & Mostly Satisfiable Calls

The LSU MaxSAT algorithm performs a linear search strategy iterating over the possible numbers of satisfied soft clauses and decreasing this number as long as the underlying solver reports the current formula to be satisfiable. As a result, all but one iterations of the algorithm involve satisfiable oracle calls. Similarly to RC2, we used the MSE'20 benchmarks for testing the performance of LSU. One difference, however, is that our implementation of LSU supports only unweighted formulas, i.e. the weighted formulas are discarded.

The performance of LSU is summarized in the cactus plot shown in Figure 6a. Observe that although the version with Glucose 3 is outperformed by a few other competitors, it is not too far behind. Concretely, it solves 317 benchmarks, each within 1699 seconds on average. The best performing LSU_{2019} solves 327 instances, with the average running time of 1714.2 seconds. The worst configuration is LSU_{2020}, which can cope with 312 formulas in 1771.9 seconds on average. Finally, observe that LSU_{M22} also solves 312 instances and the average time spent per formula is 1751.3 seconds. The scatter plots shown in Figure 7a and Figure 7b detail performance comparison of LSU_{G3} against LSU_{2019} and LSU_{2020} (as best- and worst-performing configurations of

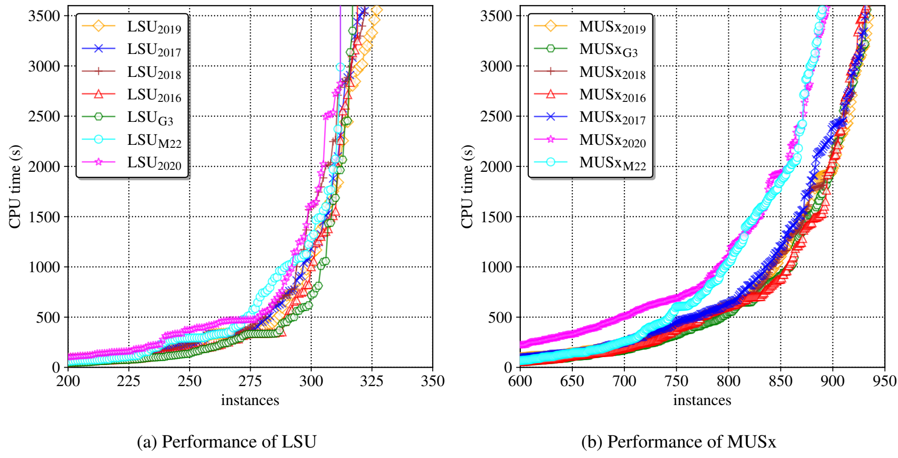


Fig. 6: Performance of LSU and MUSx with various SAT solvers.

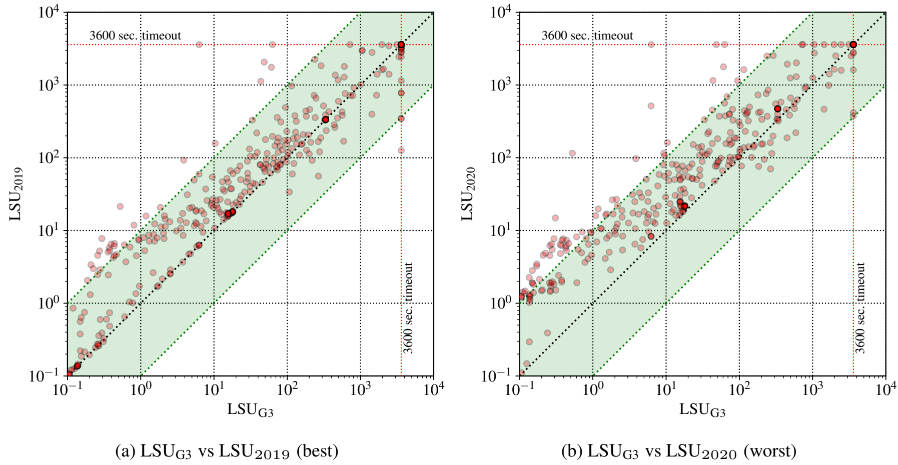
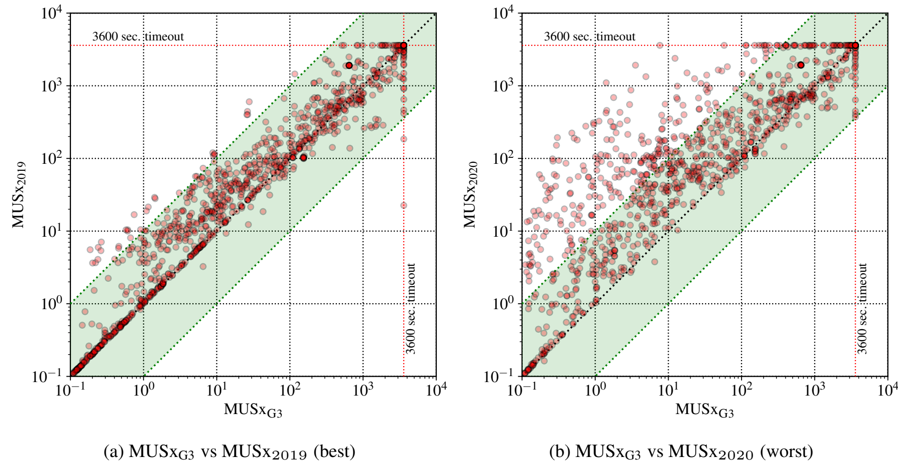


Fig. 7: Performance of LSU_{G3} compared to LSU with *best* and *worst* RLNT.

RLNT). According to these plots, Glucose 3 tends to be significantly faster than both RLNT-2019 and RLNT-2020, although RLNT-2019 manages to solve more instances overall.

5.3 MUS Extraction & Mixed Oracle Calls

The MUS extractor MUSx implements the simple deletion-based algorithm, which is bootstrapped with an unsatisfiable core of a formula and iterates over all clauses of the core trying to incrementally get rid of them one-by-one to get an MUS [12, 24]. Therefore and depending on whether the target clause belongs to an MUS, the outcome of the corresponding SAT oracle call may vary. Hence, this part of the experiment aims



(a) $MUSx_{G3}$ vs $MUSx_{2019}$ (best)

(b) $MUSx_{G3}$ vs $MUSx_{2020}$ (worst)

Fig. 8: Performance of $MUSx_{G3}$ compared to MUSx with *best* and *worst* RLNT.

at representing a practical scenario where the outcomes of incremental SAT solver calls are mixed. As the standard MUS benchmarks date back to 2011 and most of them are not challenging enough, we opted to generate a large collection of new MUS benchmarks based on the MSE'20 benchmark set. Concretely, we ran RC2 and dumped the working formulas representing the two last unsatisfiable oracle calls. (In practice, these calls are typically the hardest for a SAT solver.) The generation procedure resulted in 1103 formulas in total. Note that in order to make a fair comparison, for each benchmark MUSx was bootstrapped with an initial unsatisfiable core, which was always obtained by Glucose 3. This was done to ensure that the reduction phase computes exactly the same MUS guaranteed by the same initial unsatisfiable core as well as the same order of clauses to traverse.

Figure 6b overviews the performance of MUSx given the competing SAT engines. $MUSx_{2019}$ outperforms the competitors and successfully deals with 934 formulas, within 802.2 sec. on average. $MUSx_{G3}$ comes second with 933 instances solved spending 779.1 sec. on average. Although the version based on MiniSat 2.2 is slower than all the other configurations for this family of benchmarks (it solves 890 instances with the average time per instance being 926.2 seconds), once again, the worst RLNT-based configuration uses RLNT-2020. It solves 893 benchmarks spending 983.5 seconds per benchmark. Scatter plots comparing the performance of $MUSx_{G3}$ against the best- and worst-performing RLNT-based configurations $MUSx_{2019}$ and $MUSx_{2020}$ are shown in Figure 8a and Figure 8b, respectively. Observe that $MUSx_{G3}$ is much faster than both competitors, which is especially clear in the case of $MUSx_{2020}$.

5.4 Final Remarks

Observe that none of the tested configurations of RLNT brings any consistent (and significant) performance improvements to the considered problem solvers. Motivated by this observation, we decided to measure and report the PAR-2 metric for all the tested

Table 3: PAR-2 measure for each of the tested SAT solvers.

	MiniSat 2.2	Glucose 3	RLNT-2016	RLNT-2017	RLNT-2018	RLNT-2019	RLNT-2020
RC2-A	2536.8	2466.3	**2462.5**	2544.6	2536.7	2543.9	2875.7
RC2-B	2628.1	2287.7	**2281.3**	2406.0	2406.1	2387.5	3272.8
LSU	3391.1	3307.4	3302.7	3306.3	3318.7	**3259.7**	3411.7
MUSx	1621.4	**1333.9**	1345.7	1386.3	1361.8	1353.8	1665.7
Overall	2303.2	2061.9	**2056.3**	2151.8	2153.2	2134.3	2754.0

SAT solvers per each of the performed experiments as well as across all benchmarks, which is presented in Table 3. As the table suggests, the best overall performance is demonstrated by the solvers with RLNT-2016 “on board” although its advantage over Glucose 3 is negligible. This enables us to conclude that most of the heuristics recently proposed for SAT solvers have no significant (or none at all) positive impact on the performance of practical problem solvers in settings when SAT oracles are to be used incrementally.

6 Conclusions

This paper studies improvements made to SAT solvers in recent years, and analyzes their impact on performance when the SAT solvers are used for solving incremental SAT. Based on Relaxed_LCMDCBDL_newTech a new SAT solver RLNT was developed, to allow the activation/deactivation of specific heuristics and to allow incremental SAT uses. Thus, RLNT is able to be executed under a vast number of possible configurations. The experimental results, on the SAT competition problem instances, demonstrate that RLNT is on par with the best performing SAT solvers. As for the incremental SAT track, the experimental results suggest that recent improvements made to SAT solvers offer no clear gains. Furthermore, the experimental results on two well-known applications of incremental SAT, confirm that most recent improvements have no observable contribution to improving SAT solving performance in incremental settings.

The conclusions drawn from the experimental results can be challenged if other uses of incremental SAT are considered. We feel that MaxSAT and MUS extraction are fairly representative, since a large number of SAT calls is usually required, both with satisfiable and unsatisfiable outcomes. Further validation of our conclusions would require considering additional applications that build on incremental SAT solving. Moreover, the results presented in the paper represent a first step towards a deeper understanding of the interplay between incremental SAT and optimizations used for improving the efficiency of SAT solvers. Additional experiments and analyzes will enable a more comprehensive understanding of this interplay. From a SAT practitioner’s perspective, we believe this work demonstrates the need for a discussion within the SAT community on the improvements made to SAT solvers in light of (practical) incremental SAT solving, including more focus on this issue in the annual SAT Competitions. We also believe this work can serve to start such a discussion.

References

1. Audemard, G., Lagniez, J., Simon, L.: Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In: SAT. pp. 309–317 (2013)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI. pp. 399–404 (2009)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability – 2nd Edition. IOS Press (2021)
4. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC. pp. 151–158 (1971)
5. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
6. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. pp. 502–518 (2003)
8. Fichte, J.K., Hecher, M., Szeider, S.: A time leap challenge for SAT-solving. In: CP. pp. 267–285 (2020)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2012)
10. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: AAAI. pp. 431–437 (1998)
11. Hickey, R., Bacchus, F.: Speeding up assumption-based SAT. In: SAT. pp. 164–182 (2019)
12. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: A Python toolkit for prototyping with SAT oracles. In: SAT. pp. 428–437 (2018)
13. Ignatiev, A., Morgado, A., Marques-Silva, J.: RC2: an efficient MaxSAT solver. J. Satisf. Boolean Model. Comput. **11**(1), 53–64 (2019)
14. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: IJCAR. LNCS, vol. 7364, pp. 355–370 (2012)
15. Katebi, H., Sakallah, K.A., Marques-Silva, J.: Empirical study of the anatomy of modern SAT solvers. In: SAT. pp. 343–356 (2011)
16. Kochemazov, S., Zaikin, O., Semenov, A.A., Kondratiev, V.: Speeding up CDCL inference with duplicate learnt clauses. In: ECAI. pp. 339–346 (2020)
17. Lagniez, J., Biere, A.: Factoring out assumptions to speed up MUS extraction. In: SAT. pp. 276–292 (2013)
18. Li, C., Xiao, F., Luo, M., Manyà, F., Lü, Z., Li, Y.: Clause vivification by unit propagation in cdcl sat solvers. Artif. Intell. **279** (2020)
19. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: SAT. pp. 123–140 (2016)
20. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In: Proc. of SAT Competition 2016. vol. B-2016-1, pp. 52–53 (2016)
21. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993)
22. Luo, M., Li, C., Xiao, F., Manyà, F., Lü, Z.: An effective learnt clause minimization approach for CDCL SAT solvers. In: IJCAI. pp. 703–711 (2017)
23. Marques-Silva, J.: Search algorithms for satisfiability problems in combinational switching circuits. Ph.D. thesis, University of Michigan (1995)
24. Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: SAT. pp. 159–173 (2011)
25. Marques-Silva, J., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD. pp. 220–227 (1996)

26. Marques-Silva, J., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5), 506–521 (1999). https://doi.org/10.1109/12.769433, https://doi.org/10.1109/12.769433
27. Irkutsk Supercomputer Center of SB RAS, http://hpc.icc.ru
28. MaxSAT Evaluation 2018. https://maxsat-evaluations.github.io/2018/
29. MaxSAT Evaluation 2019. https://maxsat-evaluations.github.io/2019/
30. MaxSAT Evaluation 2020. https://maxsat-evaluations.github.io/2020/
31. Möhle, S., Biere, A.: Backing backtracking. In: SAT. pp. 250–266 (2019)
32. Morgado, A., Dodaro, C., Marques-Silva, J.: Core-guided MaxSAT with soft cardinality constraints. In: CP. pp. 564–573 (2014)
33. Morgado, A., Heras, F., Liffiton, M.H., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: A survey and assessment. Constraints An Int. J. **18**(4), 478–534 (2013)
34. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. pp. 530–535 (2001)
35. Nadel, A., Ryvchin, V.: Chronological backtracking. In: SAT. pp. 111–121 (2018)
36. Oh, C.: Between SAT and UNSAT: The fundamental difference in CDCL SAT. In: SAT. pp. 307–323 (2015)
37. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints An Int. J. **14**(3), 357–391 (2009)
38. Piette, C., Hamadi, Y., Saïs, L.: Vivifying propositional clausal formulae. In: ECAI. p. 525–529 (2008)
39. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: SAT. pp. 294–299 (2007)
40. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's thesis, School of Computing Science, Simon Fraser University
41. Zhang, X., Cai, S.: Relaxed backtracking with rephasing. In: Proc. of SAT Competition 2020. vol. B-2020-1, pp. 15–16 (2020)