Chasing Target Phases

Armin Biere and Mathias Fleury

Johannes Kepler University Linz, Austria
{biere,mathias.fleury}@jku.at

Abstract

We discuss and evaluate the idea of target phases introduced first in CaDiCaL in 2019 and also ported to our latest SAT solver Kissat. Target phases provide a heuristic to choose the value assigned to decision variables. This technique is an extension of phase saving. It extends promising assignments derived by the solver towards full models. Combined with alternatively applying series of Glucose-style and Luby-style restarts, the technique is particularly effective on satisfiable instances.

1 Introduction

In certain applications of SAT solvers, including symbolic execution [17], white-box fuzzing [15], and test case generation [19], most instances are expected to be satisfiable and it is important to find models efficiently. In this regard, the decision heuristic is one of the most critical heuristics in SAT solvers to find models. Most modern versions (VSIDS [31], VMTF [12], or LRB [29]) work on variables and another heuristic, usually phase saving [36], is used to decide whether the variable should be decided positively or negatively. In practice, phase saving is also important in combination with restarts to direct the solver back towards the same region of the search space it was exploring before restarting.

In which direction a variable is set is critical in solving satisfiable instances (and actually the most important property, since it is sufficient to guess a model). However, current implementations are "stubborn" and the initial direction is critical. If it is set to false (following the long tradition started with the SAT solver MiniSat [18]), some problems become either very hard or even impossible to solve as illustrated by some factorization problems [8] that have models where all variables are set to *true*. Other solvers like Lingeling [7] default to set phases to *true* and are able to solve those problems, but they have a hard time in the opposite case.

To be competitive at the SAT Competition and solve some large and easy problems with all-false models, Lingeling at the beginning of the solving process first sets variables to *true* (until the first conflict) and then to *false* (until the second conflict) and after that falls back to *true* again (until the end of the search). We present further developments to improve on that solution. Our solver CaDiCaL since 2018 [10] uses *rephasing* that consists in changing the saved phase to diversify the search direction in regular intervals. This heuristic was never properly described nor benchmarked. Our new SAT solver Kissat [11], submitted to the SAT Competition 2020, includes the ideas implemented in CaDiCaL, but goes further to retain the advantage of phase saving, by rephasing only some variables instead of all.

Restarts are very important for the performance of modern SAT solvers and are done frequently. However, frequent restarts harm performance on satisfiable instances, because the solver never builds assignments that are long enough to be models of the formula. Therefore, some restarts in the SAT solver Glucose are *blocked* [3] to keep searching in a promising direction. We extend that idea with a heuristic called *target phasing* that extends promising assignments to full models. In combination with a change in the restart heuristic, this technique was included in CaDiCaL, which won the satisfiable track (containing only satisfiable instances) at the SAT Race 2019 [25].

After recalling some basics on phase saving and restarts (Section 2), we give details on the different policies for rephasing to diversify the search. The saved phase can be changed in any arbitrary way, because the phase saving heuristic does not affect the correctness of the overall solver. However, we do it only in a limited way to avoid disrupting the search too much (Section 3). These changes do not target specifically satisfiable instances. Our solvers use target phasing, a technique that saves promising models in a way that resembles phase saving with a very different update policy (Section 4). We have implemented our new heuristics in our solvers and also in the SAT solvers Glucose [1] and Spass-Satt [16] with a certain strategy for our heuristics (Section 5). Finally, we benchmark these solvers on the problems used at the SAT Competition 2018 and SAT Race 2019 and show that our heuristics improve the performance on satisfiable instances (Section. 6).

2 Related Work

We use standard notation and refer the reader to the *Handbook of Satisfiability* for an introduction to SAT [14] and the conflict-driven clause learning (CDCL) procedure.

Phase Saving. Most modern decision heuristics [12,29,31] pick a variable and another heuristic is used to determine the direction, or *phase*, it should be set to (i.e., whether the variable should be decided positively or negatively). Picking the right phase is what matters most for satisfiable instances, because the solver only needs to guess one model.

Formerly, some state-of-the-art SAT solvers like zChaff [31] used information based on the number of occurrences [31] with the aim of increasing the number of true clauses under the current assignment. Other SAT solvers like MiniSAT (e.g., in the version submitted to the SAT Competition 2005) always set literals to false. Instead of using information on the clauses, phase saving [36] captures information on the search process and caches how variables are set during propagation or backtracking. This saved value is later used to set the phase when deciding that variable, bringing the SAT solver back to a similar region of the search space. This simple (it only requires an array and is cheap to update) and easy to calculate heuristic has a positive effect on performance and is now standard in most modern SAT solvers.

With phase saving, the solver focuses on the region of the search space explored before. The heuristic is not only important for satisfiable instances, but also for unsatisfiable instances. If the formula is composed of several independent components, phase saving allows the solver to focus on one component instead of working on several components at the same time. In particular, if the problem includes several *disjoint* satisfiable components, the interplay between the decision and phase saving heuristics implies that each component will be solved maintaining the assignment of the previously solved components.

Rephasing The saved phase can be changed arbitrarily, because it is independent of the correctness of CDCL. The SAT solvers PrecoSAT [7] (resp. PicoSAT [7]) use a Jeroslow-Wang score [26] to reset the saved phases on all (resp. irredundant only) clauses in regular intervals following a Luby sequence. The motivation is to adapt the saved phases to the formulas. The SAT solver StrangeNight [5] flips values with a certain probability depending on the depth of the assignment. The motivation here is to avoid the heavy-tail phenomenon. Manthey reported experiments [32, Sect. 3.1] for the SAT solver Riss [6] with negative results. However, to the best of our knowledge, this is the first time that several rephasing heuristics were compared and used.

Restarts. For the efficiency of SAT solvers, restarts are very important. In particular, fast restarts [37] are now widely used. Luby restarts were heavily used because they are the a priori optimal strategy [30]. However, in recent years, mostly Glucose-style restarts [2] are used and are, basically, a requirement for the SAT Competition. Biere and Fröhlich's presentation acts as a survey on various restart heuristics [13], in particular with a simpler implementation of Glucose-style restarts based on exponential moving average. To keep completeness of SAT solvers, either restarts must be delayed more and more (like achieved with Luby restarts), or the number of clauses that are kept during database reduction needs to be increasing (like always implemented in combination with Glucose-style restarts).

Restarts are done extremely frequently and so often actually that parts of the trail can be reused [37] because they would not be altered. In order to find models, the solver must generate long assignments. For Luby-style restarts, this is realized by means of the (non-monotonically) increasing intervals between successive restarts. For Glucose-style restarts these restart intervals are not necessarily increasing and hence there are no guarantees that long assignments will be generated. To overcome that drawback, as implemented in Glucose [3], restarts can be *blocked* and delayed whenever the trail length has increased by a predefined factor since the latest restart [3]. Delaying restarts is only required for satisfiable problems. Another option is to alternate the restart policies [35] and in particular restart less or completely avoid restarting for some time intervals during the search process.

Decision Heuristics. Decision heuristics are responsible for deciding which variable should be decided next. Some variables deemed important (usually variables that appear in the conflict clauses) are *bumped* and their importance is increased, making them more likely to be decided next. VSIDS [31] maps each variable to a score and bumping a variable consists in increasing the score of the variable (or as seen in the options of Glucose, the importance of the other variable is reduced; this is controlled by the *variable decay* factor). If scores are increased a lot, VSIDS focuses on the variables that appeared in the last conflict. On the contrary, no increase means that the initial order of the variables is kept. The VMTF heuristic [12] provides a simpler and more efficient implementation that focuses on the literals involved in the last conflicts. Biere and Fröhlich's presentation is a survey [12], without more recent work like LRB [29].

3 Rephasing

By *rephasing* we mean to globally reset or change saved values and call different variants of such transformations a *rephasing heuristic* (Section 3.1). Several transformations are possible and are scheduled to form a *rephasing strategy* that describes in what order these heuristics are applied and which values they set (Section 3.2).

3.1 Rephasing Heuristics

In our implementations we save the phase during literal propagation unlike in solvers based on MiniSat, which save values during backtracking (with the same effect). During search, every once in a while, in increasing intervals, we execute some operations on the saved values. Such changes are always allowed, because they do not impact the underlying CDCL calculus, its correctness, or the termination. We call *rephasing heuristic* any such operation that changes the saved values.

Rephasing heuristics diversify the exploration of the search space, which can be helpful for satisfiable instances (we could get close to a model). They can also help to variegate the learned

clauses. We conjecture that this, in turn, improves the efficiency of inprocessing (e.g., learning additional important clauses, like glue clauses).

Rephasing to a Fixed Value. Our first rephasing heuristic consists in setting all saved phases to a single value, either the original value (phase '`O`') – remember, unlike MiniSat, our tools default to the value *true* – or the opposite value (the inverted value, phase '`I`'). This alternation is helpful in finding models when most values have a certain sign, but this depends on the order in which literals are decided. For example, modifying the SAT solver Glucose [1] to apply those rephasings does not make solving the factorization problems [8] mentioned earlier completely trivial (some conflicts are still required), but they are now solved extremely fast, while being hard for the default version of Glucose.

Flipping Values. Our second rephasing heuristic consists in flipping the saved values (phase '`F`'), unlike the previous heuristic in which all saved values are set to a single value, namely either *true* or *false*. This allows exploring the "opposite" region of the search space. The motivation behind this heuristic comes from machine learning: Flipping corresponds to diversification with a very different model. It can also support inprocessing by, e.g., simulating hyper binary resolution [23], because a literal can be decided and later its opposite.

Randomizing Values. In order to diversify the exploration of the search space even more, we additionally randomize the saved phase (phase '`#`'). The basic idea is that for satisfiable instances and with some luck, the randomized saved phases will now form an assignment that is close enough to a model of the problem we want to solve. The assignment can then be adapted by means of CDCL to a model. If the randomization is done uniformly, the saved phases will eventually be close to a model. In the same spirit, we also tried to shuffle the scores of the variable decision heuristics (and the VMTF queue) in CaDiCaL, which, however, only produced negative results and is switched off by default.

Local Search. With a limited local search (phase '`W`', standing for walk) we attempt to reduce the number of unsatisfied clauses under the assignment formed by the current saved phases. Our local search implements the ProbSAT strategy [4]. During local search an assignment falsifying the least number of clauses is kept as saved phases and this way used for decisions in the CDCL loop. The idea is that the solver can focus on the unsatisfiable part of the clause set. Beside our solvers, CryptoMiniSat [39] uses local search in a similar way [38].

Best Phasing. The key idea of best phasing (phase '`B`') is that good current assignments are close to models. The solver caches the best assignment found so far (with respect to the length of the trail until the last decision if it generated a conflict). During each backtrack and before each restart, the current partial assignment is saved if it improves the best-so-far found assignment. This heuristic simply replaces the saved phases by the values of the best assignment. For unsatisfiable instances, this corresponds to focusing on the unsatisfiable region of the search space. Note that the length of the trail might not be a perfect measure in the context of techniques like on-the-fly self-subsuming resolution (OTFS) [21,22] or in combination with chronological backtracking [33, 34]. Furthermore, inprocessing needs to be taken into account. In Kissat we actually measure the number of assigned variables plus the number of fixed, substituted, or eliminated variables. The best assignment is reset after each best phasing (in '`B`', and only then).

3.2 Rephasing Strategies

The rephasing heuristics define how to change the saved phases, but they do not have to be applied on all variables and an order has to be defined.

Autarkies. As explained above, one motivation for phase saving is that it caches the phases needed to satisfy some components of an input problem allowing the solver to focus on the unsatisfiable part. If saved values are changed by some of the rephasing heuristics described above, this property does not hold anymore.

To overcome this issue, it is possible to rephase only some variables without changing the phase of satisfied components. One possibility is to compute the largest autarky of the full assignment represented by saved phases using an algorithm originally proposed by Kullmann and described in a publication by Kiesl et al. [27]. Instead of forcing the autark assignment we simply eliminate the clauses touched by the autarky as well as the variables assigned by it.

Rephasing Strategies. We schedule the different rephasing heuristics in geometrically increasing intervals. Consequently, we spend more and more time exploring the search space in any given direction. An extreme case would be to start with the inverted phase '`I`' and an infinitely long interval: We would then explore the search space like MiniSAT without any rephasing. We describe the order in which we apply the heuristics in Section 6.

4 Target Phase for SAT Instances

Although the rephasing heuristics described in the previous section do not target satisfiable instances only, we already see that the best rephasing also makes sense for satisfiable instances. In this section, we go further and describe *target phasing*, which tries to improve solving of satisfiable instances, possibly at the expense of solving fewer unsatisfiable instances.

As already mentioned, fast restarts are important for the performance of SAT solvers, but make solving satisfiable instances harder. To mitigate this issue, restarts can be blocked in Glucose. If the search direction is promising (i.e., the current assignment has become much larger), instead of restarting, the search continues (and the conflict count since the last restart is reset, making it disallow restarts in the next 50 conflicts).

The intuition is that promising assignments should be extended towards full assignments (and hopefully a model of the formula) instead of being discarded by restarting. We refine this idea further in our *target phasing* heuristic that saves promising models separately (similar to best phasing) instead of just extending them.

Target Phasing. The target phasing heuristic follows the idea of extending an assignment to a full model. As for phase saving, an additional implicit (but partial) assignment is kept, with the key difference that the target assignment is updated less frequently during the search.

Target phasing is composed of three parts. First, an implicit target assignment is saved and updated if the current assignment is more promising than the saved one. The current assignment, as represented by the "trail" of the solver, is considered more promising if it assigns more variables (in terms of the size of the "trail") and does not lead to a conflict after propagation exactly as for best phasing. Then the complete current assignment, i.e., the trail, becomes the new target assignment.

Second, when picking the phase to assign to a decision variable, we do not use the saved value (as usually done with phase saving) but instead the value given by the target assignment. If the target phase is unassigned, the solver defaults to the value provided by phase saving.

Third, the target assignment is reset after each rephasing to the initial all-unassigned state. This encourages the solver to find larger and larger target assignments until the next rephasing. The largest one will be recorded as best assignment and reused in the next best rephasing.

Restart Policy. To increase the chance to find a model, the solver must work on relatively long assignments. However, this implies that we learn long clauses, risking encountering heavy-tailed behavior [20] and to miss short proofs. To circumvent this problem, we alternate between *focused mode* (with GLUCOSE-style fast restarts) and *stable mode* (with fewer restarts)[1] in the spirit of [35].

The 2018 version of CADICAL did not restart at all in stable mode. Since 2019, Luby restarts are used with a relatively large base interval (1 024 compared to MINISAT's default value of 100) in CADICAL. The same restart strategy is used for KISSAT. Alternating between these two restart policies allows us to use a shorter minimum of conflicts between two successive restarts, namely 10 instead of the GLUCOSE default of 50, in order to find shorter proofs. Note that CADICAL has a base restart interval of 2 in focused mode and KISSAT even 1 (but increasing logarithmically). The duration of each search mode interval is increased geometrically. In KISSAT the conflict interval is in $\mathcal{O}(n \cdot \log^2 n)$ after n restarts though instead of $\mathcal{O}(2^n)$ [11].

5 Implementation

Our heuristics have been implemented in our SAT solvers CADICAL and KISSAT and ported to two other SAT solvers, namely GLUCOSE and SPASS-SATT.

Default Policies. We schedule the different heuristics in order to find models. We assume that assignments that are saved either as best or as target are good candidates for expansion. Hence, we spend most time on 'B' phases than on any other phase.

In CADICAL the search mode is based on the number of conflicts found so far. In focused mode, the default rephasing policy is '$\texttt{OI(BWOBWI)}^\omega$' (Original, Inverted; then Best, Walk, Original rephasing is repeated). The 'OI' at the beginning speeds up finding models where all literals are set to either *true* or *false* (the phase 'I' starts after the very *first* conflict). In stable mode, the policy is '$\texttt{(IBWFBW\#BWOBW)}^\omega$' (Flipped and # for random rephasing). By default, CADICAL provides a mode to target satisfiable instances, which only uses target phasing and stable mode. However, in these experiments, we keep the alternation between stable and focused mode.

In KISSAT, the default rephasing policy is '$\texttt{(BWOBWIBW\#BWF)}^\omega$'. Unlike CADICAL, KISSAT determines the time spent in focused and stable mode by estimating the number of memory accesses (instead of measuring the time directly in order to be deterministic across runs). More precisely, the number of cache misses is estimated, refining on Knuth's "mems" [28]. Unlike CADICAL, in the satisfiable configuration ('--sat') submitted to the SAT competition, KISSAT alternates between focused and stable mode and always uses target phasing.

All our implementations use the alternation of stable mode with Luby-style restarts and focus mode with GLUCOSE-style restarts. The idea of the first is to be more stable overall. Hence, KISSAT and CADICAL use chronological backtracking [33, 34]. They also use two

[1] The stable mode was initially called "stable phase" [9], which is a confusing name (due to rephasing), so we have decided to rename it, as can already be seen in the system description of this year's SAT Competition [11].

separate decision queues as suggested by Oh [35]. They use VMTF [12] in focused and VSIDS in stable mode. VMTF during focus mode makes the solver more agile whereas VSIDS with a low bumping is more stable.

Implementation in Glucose. We implemented our heuristics in the SAT solver Glucose version 3.0 as used for the Glucose hack track of the SAT Competition. We have not implemented a different decision queue for stable mode. However, to increase stability, we decrease the bonus that bumped variables get (by setting the variable decay to a smaller value). In focus mode, instead, we increase the decay compared to the default value in order to follow more closely the search process. For a decay factor of 0.5, VSIDS is a poor man's version of VMTF: it is as agile as VMTF but does not keep the order of literals when bumping several literals at once, which is important for the performance of VMTF [12]. To have a more balanced alternation between stable and focused mode, we measure time in the same way as Kissat.

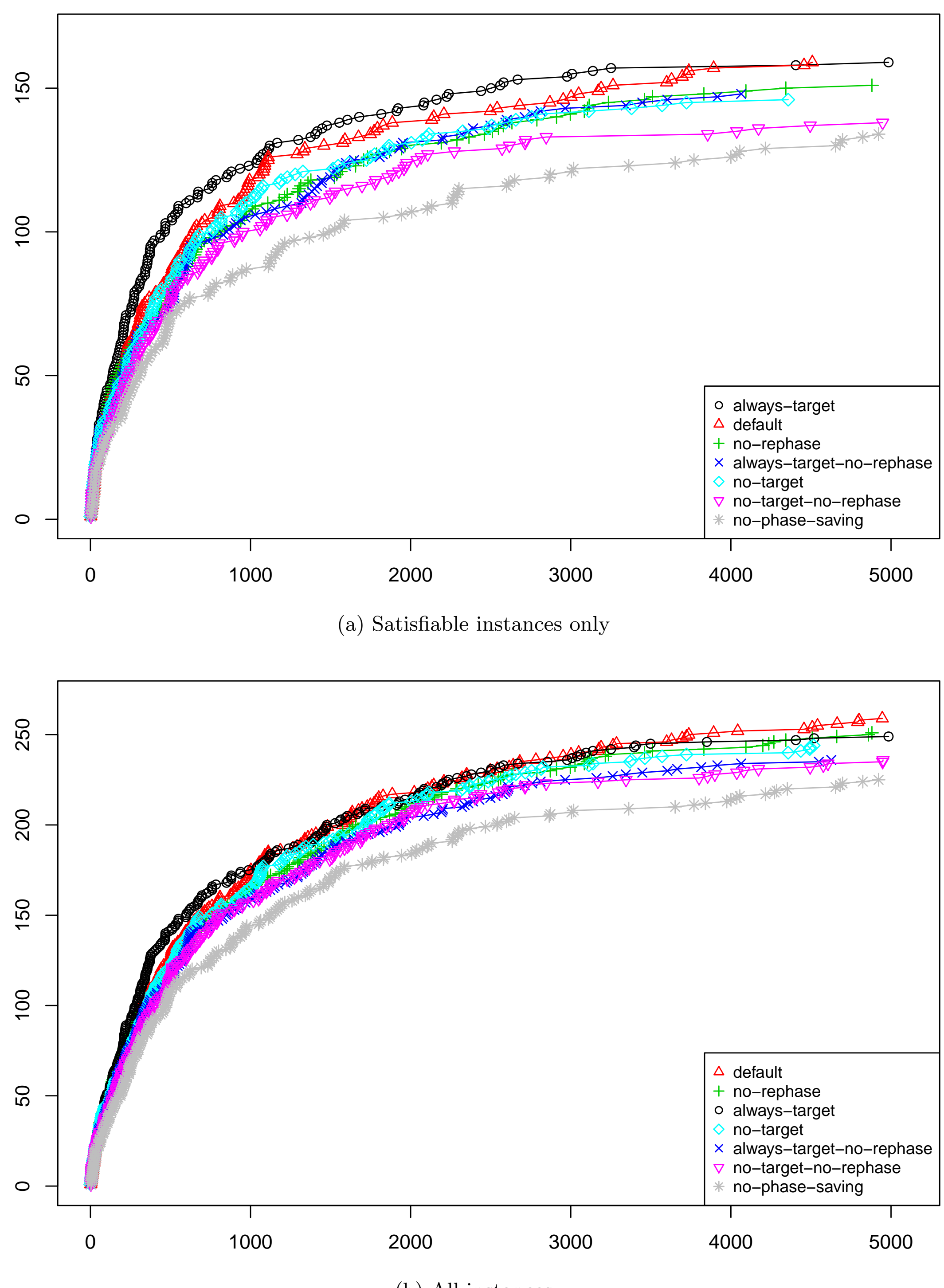
Two details of this implementation effort should be mentioned. First, implementing the alternation of stable and focused mode was easy, but performance significantly dropped to the point that our implementation became much worse than the original implementation of Glucose. We resolved that issue by bumping not only the resolved literals and the literals in the learned clause but also the literals in the *reasons* of the literals in the learned clause, following the idea pioneered by MapleSAT [29], which is also used in our other solvers. Experiments with CaDiCaL confirmed the importance of this heuristic. Second, we had to change the types of data structure to save the target phase. Glucose uses a vector of Booleans, but for target phases, it is necessary to use a vector of tri-states (with third possible *unassigned* value, beside *true* and *false*).

Implementation in Spass-Satt. The (non-open source) SAT solver Spass-Satt is a SAT solver developed for the CDCL($\mathcal{T}$) solver Spass-Satt.[2] It is a new implementation but it mostly follows the ideas of Glucose 2 with a major difference: it has inprocessing and applies subsumption-resolution until fix-point on a regular basis on all clauses including learned clauses, but has no other preprocessing, in particular no variable elimination. We have not implemented random walk (phase 'W') and use the same strategy we have implemented in Glucose without random walk ('$(\texttt{BOBIB\#BF})^\omega$'). Bumping the reasons of the conflict literals was also required for better performance. The implementation is also simpler and the time spent in stable and focused mode is estimated by the number of conflicts like in CaDiCaL, which is a rather unprecise measure. In particular, it is biased towards stable mode.

6 Experiments

Our experiments on the presented SAT solvers were run on an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 7 GB and a time limit of 5 000 s (same time limit as in the SAT Competition). Previous experiments showed that our cluster is slightly slower than StarExec, the cluster used for the SAT Competition. We show the results as a cumulative distribution function (CDF, and not as cactus plot), i.e., as a graph showing the number of solved instances depending on the time. The higher the curve, the better the solver. The log files and source code (except for Spass-Satt) are available.[3]

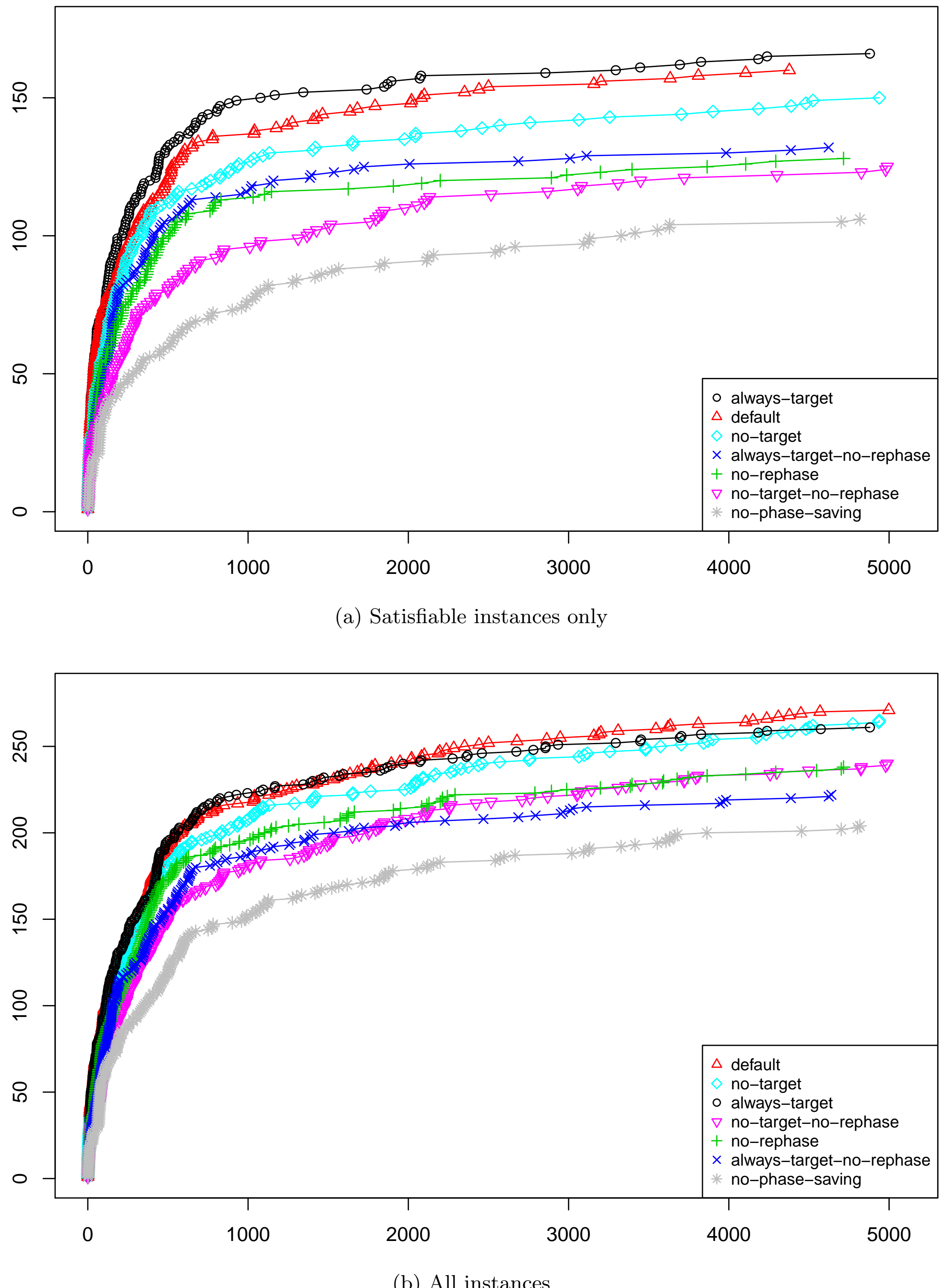
[2] It is in essence an SMT solver supporting only the theory of linear integer and rational arithmetic.
[3] http://fmv.jku.at/chasing-target-phases



(a) Satisfiable instances only

(b) All instances

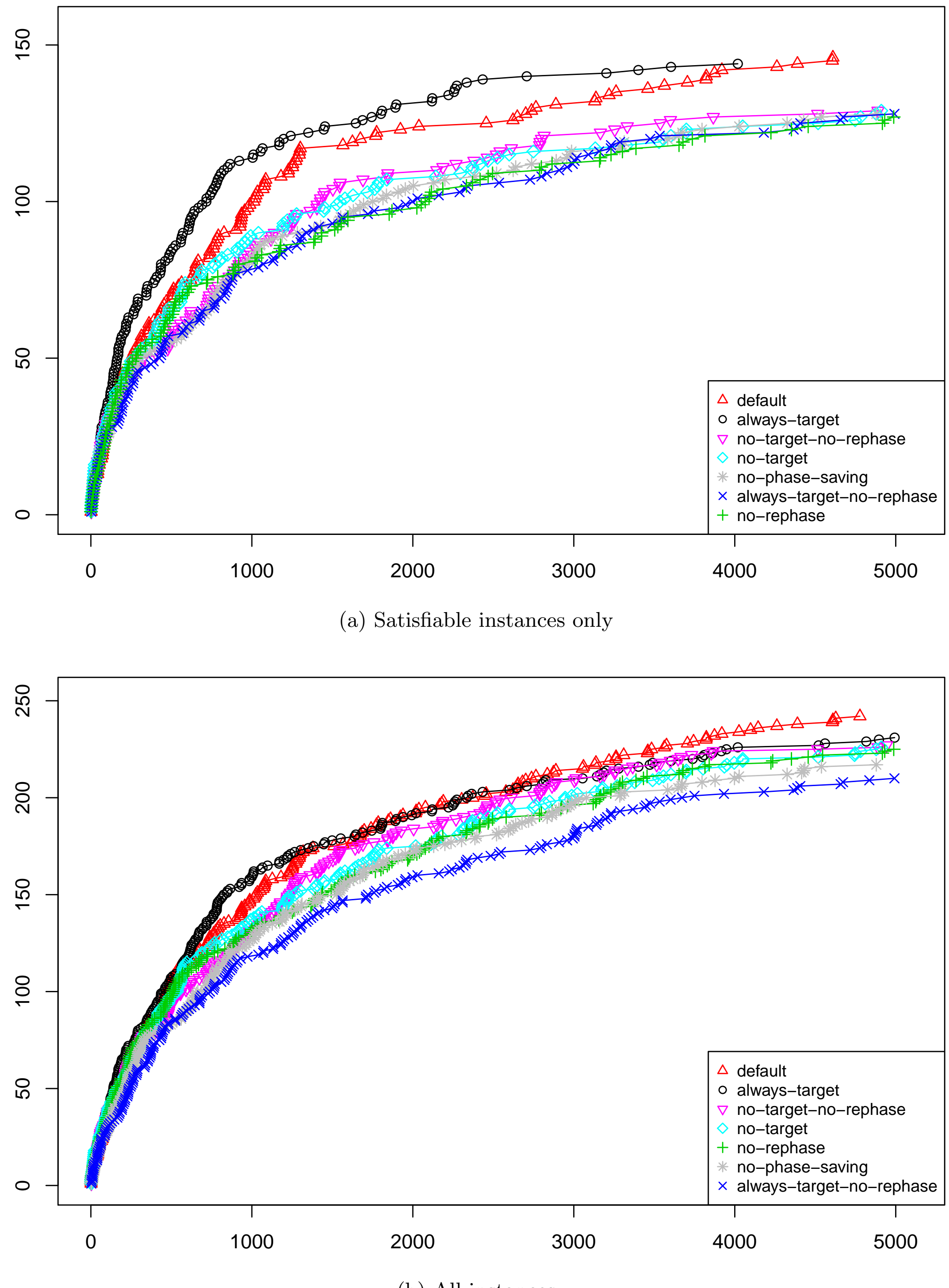
Figure 1: CDF for the solver Kissat on benchmarks from the SAT Race 2019



(a) Satisfiable instances only

(b) All instances

Figure 2: CDF for the solver KISSAT on benchmarks from the SAT Competition 2018



(a) Satisfiable instances only

(b) All instances

Figure 3: CDF for the solver CaDiCaL on benchmarks from the SAT Race 2019

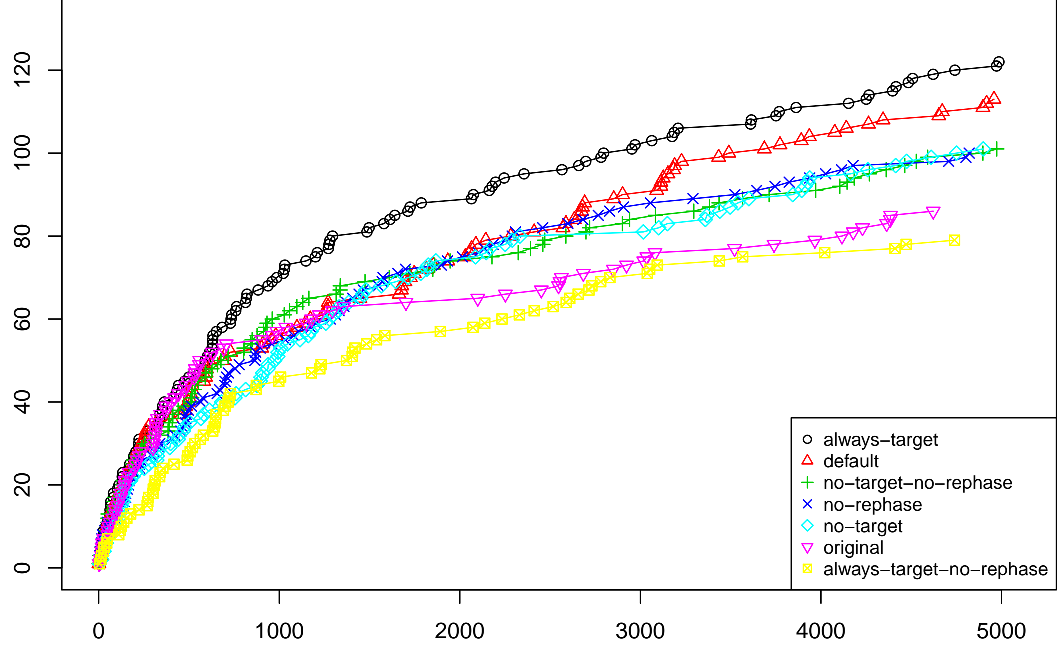


Figure 4: CDF for the solver SPASS-SATT on benchmarks from the SAT Race 2019 (satisfiable instances only)

CaDiCaL and Kissat. We here present our experiments of KISSAT on the problems from the SAT Competition 2018 [24] and SAT Race 2019 [25]. Due to time constraints, CADICAL and the other solvers described below have only been tested on the problems from the SAT Race 2019. We have tested 7 different configurations for CADICAL and KISSAT.

always-target (resp. **no-target**) always (resp. never) uses target phases to set the value of the decision variables. By default, target phasing is only activated during stable mode.

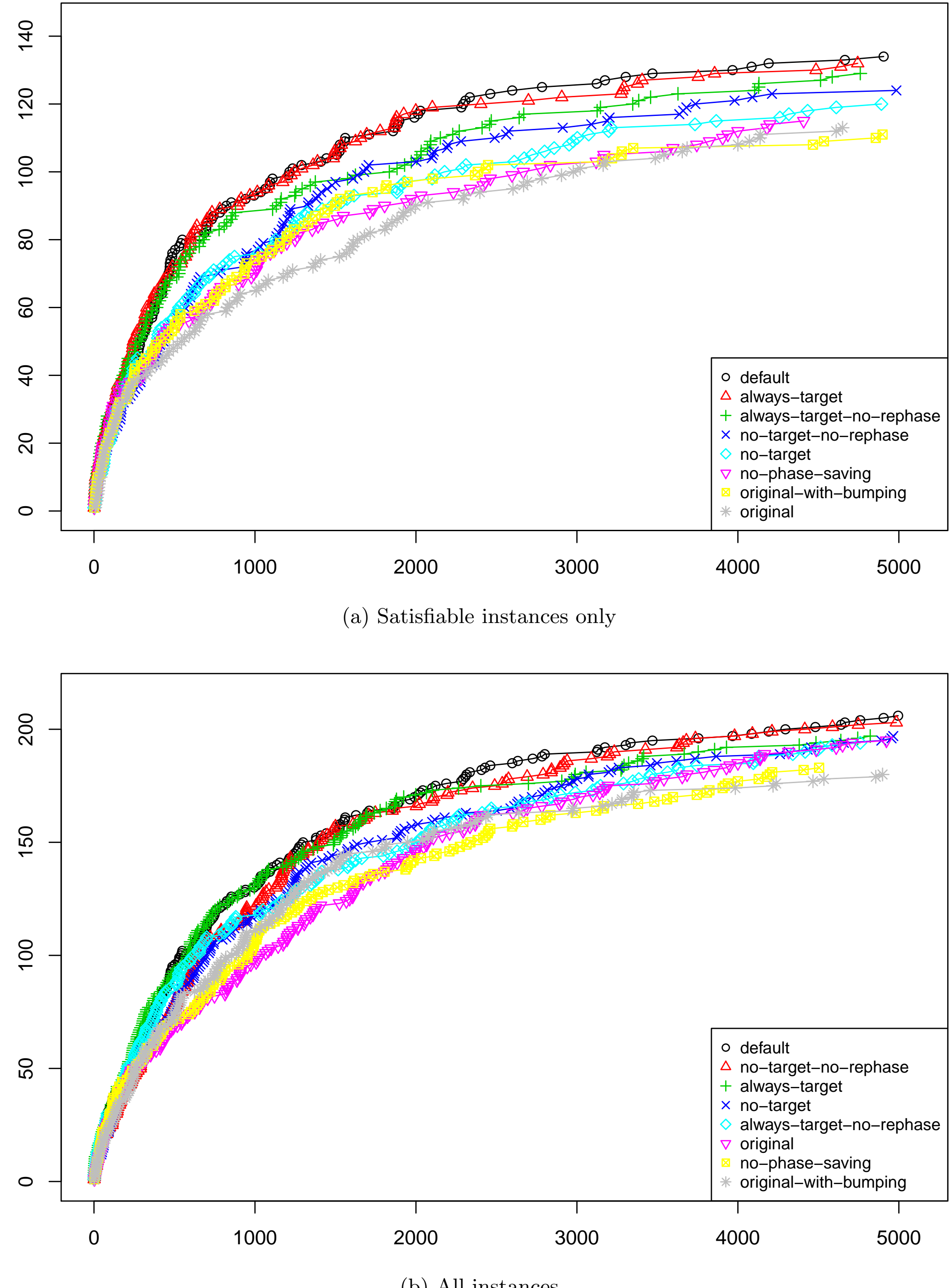
no-rephasing never uses rephasing. It is activated by default.

default is a alternating approach. It uses target phasing in stable mode and standard phase saving in focused mode.

no-phase-saving does not use any phase/target saving, nor rephasing, and simply sets the variable to the inital phase, *true*. It is activated by default.

For instance, the '`no-target`' configuration does not feature any target phasing but uses rephasing, the '`always-target`' configuration always uses target phasing even in focus mode, and the '`no-rephase`' uses target phasing only during stable mode but never rephases the save phases. All these configurations use save phasing to save the values.

The results are presented in Figures 1a and 2a (only satisfiable instances) and in Figures 1b and 2b (all instances) for KISSAT and in Figures 3a and 3b for CADICAL and in Table 1. First it shows that our new solver KISSAT performs better than CADICAL. Second, we see that '`no-phase-saving`' solves the least number of problems confirming previous results. Third, on satisfiable instances, pure phase saving '`no-target-no-rephase`' performs worse than any other configuration, but on all instances, it performs better than '`always-target-no-rephase`'.



(a) Satisfiable instances only

(b) All instances

Figure 5: CDF for the solver GLUCOSE on benchmarks from the SAT Race 2019

Fourth, target phasing without rephasing (configuration '`always-target-no-rephase`') does not perform better than no target phasing without rephasing '`no-rephase`'. Intuitively, this makes sense because target phasing strongly constrains the search in a direction and rephasing resets target phasing making it possible to explore different regions of the search space. Fifth, our default alternating strategy manages to retain the best side of '`no-target`' and '`always-target`': It solves most satisfiable instances and it is not too harmful on unsatisfiable instances. Sixth, '`no-phase-saving`' performs particularly bad in Kissat.

Spass-Satt. Due to time constraints, we have tested the other solvers only on instances from the SAT Race 2019 and in fewer configurations. The results for the SAT solver Spass-Satt on satisfiable instances are presented in Figure 4 and Table 2. Target phasing *alone* does not improve performance and even makes it worse than the original SAT solver (configuration '`original`'), but in combination with rephasing, it works extremely well. Compared to the original version, significantly more problems are solved. As it does not have any preprocessing techniques like variable elimination, it solves less than 50 unsatisfiable instances. Hence, the graph on all instances is not very meaningful because of the overwhelming majority of satisfiable instances solved.

Glucose. The results of Glucose draw a slightly different picture, see Figure 5 and Table 2, with some interesting results. First, unrelated to our heuristics, bumping the reasons of the clauses (configuration '`original-with-bumping`') improves the performance of the original solver (configuration '`original`') until around 4 000 s for satisfiable and 2 000 s for unsatisfiable instances, where the base version of Glucose catches up. Second, the impact of the heuristics is smaller than for Kissat and CaDiCaL. One possible explanation is that further tuning of constants like variable decay during focused and stable mode is required to get more out of our heuristics. Third, rephasing is less helpful than for Kissat, CaDiCaL, and Spass-Satt. Fourth, the '`default`' configuration (the same alternating scheduling as for Kissat) solves most SAT problems, albeit by a very small margin. The performance of the '`no-phase-saving`' is surprising and seems to be due to the stable phases: Deactivating it significantly reduces performance on unsatisfiable problems.

Interpretation. Overall, the performance increases on satisfiable instances thanks to our heuristics. Generally, rephasing with target phasing improves the performance of the solver and makes it more robust to solve problems where a model with only *true* variables exists.[4] Performance on unsatisfiable instances degrades, but the '`default`' alternating approach (target phases during stable mode, usual phase saving during focused mode) achieves a good compromise on a combination of satisfiable and unsatisfiable problems. Rephasing alone does not seem to help as much for Glucose as for the other solvers tested here. We believe that this is due to inprocessing: All tested solvers but Glucose are able to strengthen the clauses and rephasing can help that by being able to simplify more clauses.[5]

Attempts to unify the rephasing strategies is ongoing work and we did not find a simple overall winning strategy yet. Note that these solvers are of course not identical, (e.g., time spent in stable and focused mode, the number and frequency of deleted learned clauses, the details of the variable scoring mechanism, inprocessing is performed). Our experience is however, that using best rephasing every second or third time rephasing is scheduled gives better results.

[4]This kind of problem is unlikely to be selected at the SAT Competition, because very few solvers are able to solve such instances.

[5]This hypothesis was actually the main motivation for implementing these heuristics in Spass-Satt.

Table 1: Summary of the performance of the SAT solvers Kissat and CaDiCaL (best configuration in bold)

	All instances				Satisfiable instances			
	Kissat		CaDiCaL		Kissat		CaDiCaL	
Configuration	Solved	PAR2	Solved	PAR2	Solved	PAR2	Solved	PAR2
`default`	**259**	**1 662 994**	**242**	**1 857 009**	159	263 235	**146**	**323 506**
`alw.-target`	249	1 714 647	231	1 921 490	**159**	**234 467**	144	294 832
`no-target-no-rephase`	236	1 859 306	227	1 979 507	138	450 986	129	473 741
`no-target`	244	1 775 317	226	2 002 909	146	366 564	129	479 070
`no-rephase`	251	1 736 491	225	2 020 048	151	341 314	127	509 767
`no-phase-saving`	225	1 986 220	217	2 084 103	134	522 721	128	498 526
`alw.-target-no-rephase`	236	1 851 424	210	2 165 365	148	359 317	128	508 870

Table 2: Summary of the performance of the SAT solvers Glucose and Spass-Satt (best configuration in bold)

	All instances				Satisfiable instances			
	Glucose		Spass-Satt		Glucose		Spass-Satt	
Configuration	Solved	PAR2	Solved	PAR2	Solved	PAR2	Solved	PAR2
`default`	**206**	**2154525**	159	2 671 578	**134**	**368068**	113	532 083
`alw.-target`	197	2 222 227	**168**	**2582440**	132	383 353	**122**	**436410**
`no-target-no-rephase`	203	2 192 177	148	2 741 503	124	476 546	101	616 097
`no-target`	197	2 259 101	151	2 736 161	120	516 055	101	626 393
`alw.-target-no-rephase`	194	2 282 175	127	2 909 751	129	424 991	79	801 110
`original`	195	2 312 300	137	2 829 600	112	359 317	86	729 944

7 Conclusion

This work presented our rephasing and target phasing heuristics. Our experiments show that rephasing improves performance in general, while target phasing is particularly useful on satisfiable instances. They both significantly improve the performance of our SAT solvers CaDiCaL and Kissat. Our implementation in Glucose and Spass-Satt improved their performance, but also showed that rephasing is more helpful if some inprocessing is executed.

It would be interesting to test our heuristics further (e.g., by implementing them in additional solvers) and to investigate whether inprocessing is necessary for rephasing to be beneficial or whether the performance of our Glucose implementation can be improved without it. Finally, we also tried to assess the importance of autarky reasoning as well as local search, which did not give a clear picture. This will also need to be addressed in future work.

Acknowledgment. This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE) and the LIT AI Lab funded by the State of Upper Austria. Sibylle Möhle suggested many textual improvements.

References

[1] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 399–404. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009), http://ijcai.org/Proceedings/09/Papers/074.pdf

[2] Audemard, G., Simon, L.: Glucose 2.1: Aggressive—but reactive—clause database management, dynamic restarts. In: Workshop on the Pragmatics of SAT 2012 (2012), https://hal.inria.fr/hal-00845494

[3] Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Milano, M. (ed.) Principles and Practice of Constraint Programming–18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7514, pp. 118–126. Springer (2012). doi:10.1007/978-3-642-33558-7_11

[4] Balint, A.: Engineering stochastic local search for the satisfiability problem. Ph.D. thesis, University of Ulm (2014), http://vts.uni-ulm.de/docs/2014/8852/vts_8852_13227.pdf

[5] Balint, A., Belov, A., Heule, M., Järvisalo, M. (eds.): Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, vol. B-2013-1. Department of Computer Science, University of Helsinki, Finland (2013), https://helda.helsinki.fi/bitstream/handle/10138/40026/sc2013_proceedings.pdf

[6] Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT Challenge 2012 solver competition. Artificial Intelligence **223**, 120–155 (2015). doi:10.1016/j.artint.2015.01.002

[7] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. vol. 10 (2010)

[8] Biere, A.: Deep Bound Hardware Model Checking Instances, Quadratic Propagation Benchmarks and Reencoded Factorization Problems Submitted to the SAT Competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2017-1, pp. 40–41. University of Helsinki (2017)

[9] Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2018-1, pp. 13–14. University of Helsinki (2018)

[10] Biere, A.: CaDiCaL at the SAT Race 2019. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Race 2019 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2019-1, pp. 8–9. University of Helsinki (2019)

[11] Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Heule, M., Järvisalo, M., Suda, M., Iser, M., Balyo, T. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions (2020), to appear

[12] Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 405–422. Springer (2015)

[13] Biere, A., Fröhlich, A.: Evaluating CDCL restart schemes. In: Heule, M.J.H., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing—SAT 2015—18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 405–422. EasyChair (2015). doi:10.29007/89dw

[14] Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)

[15] Bounimova, E., Godefroid, P., Molnar, D.A.: Billions and billions of constraints: whitebox fuzz testing in production. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference

on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 122–131. IEEE Computer Society (2013). doi:10.1109/ICSE.2013.6606558

[16] Bromberger, M., Fleury, M., Schwarz, S., Weidenbach, C.: SPASS-SATT – A CDCL(LA) solver. In: Fontaine, P. (ed.) Automated Deduction – CADE 27 – 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11716, pp. 111–122. Springer (2019). doi:10.1007/978-3-030-29436-6_7, https://doi.org/10.1007/978-3-030-29436-6_7

[17] Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[18] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003). doi:10.1007/978-3-540-24605-3_37, https://doi.org/10.1007/978-3-540-24605-3_37

[19] Godefroid, P., de Halleux, J., Nori, A.V., Rajamani, S.K., Schulte, W., Tillmann, N., Levin, M.Y.: Automating software testing using program analysis. IEEE Software **25**(5), 30–37 (2008). doi:10.1109/MS.2008.109

[20] Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reasoning **24**(1/2), 67–100 (Feb 2000). doi:10.1023/a:1006314320276

[21] Hamadi, Y., Jabbour, S., Sais, L.: Learning for dynamic subsumption. In: ICTAI 2009, 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, New Jersey, USA, 2-4 November 2009. pp. 328–335. IEEE Computer Society (2009). doi:10.1109/ICTAI.2009.22

[22] Han, H., Somenzi, F.: On-the-fly clause improvement. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30–July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 209–222. Springer (2009). doi:10.1007/978-3-642-02777-2_21

[23] Heule, M., Järvisalo, M., Biere, A.: Revisiting hyper binary resolution. In: Gomes, C.P., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7874, pp. 77–93. Springer (2013). doi:10.1007/978-3-642-38171-3_6

[24] Heule, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, vol. B-2018-1. Department of Computer Science, University of Helsinki, Finland (2018), https://helda.helsinki.fi//bitstream/handle/10138/237063/sc2018_proceedings.pdf

[25] Heule, M.J., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, Department of Computer Science Report Series B, vol. B-2019-1. Department of Computer Science, University of Helsinki, Finland (2019), https://helda.helsinki.fi//bitstream/handle/10138/308034/sr2019_proceedings.pdf

[26] Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Ann. Math. Artif. Intell. **1**, 167–187 (1990). doi:10.1007/BF01531077, https://doi.org/10.1007/BF01531077

[27] Kiesl, B., Heule, M.J.H., Biere, A.: Truth assignments as conditional autarkies. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis – 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 48–64. Springer (2019). doi:10.1007/978-3-030-31784-3_3

[28] Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–

History of Combinatorial Generation (Art of Computer Programming). Addison-Wesley Professional (2006)

[29] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing–SAT 2016–19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 123–140. Springer (2016). doi:10.1007/978-3-319-40970-2_9

[30] Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47**(4), 173–180 (Sep 1993). doi:10.1016/0020-0190(93)90029-9

[31] Mahajan, Y.S., Fu, Z., Malik, S.: zChaff 2004: An efficient SAT solver. In: Hoos, H.H., Mitchell, D.G. (eds.) Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3542, pp. 360–375. Springer (2004). doi:10.1007/11527695_27

[32] Manthey, N.: Improving SAT solvers using state-of-the-art techniques. Master's thesis, Diploma thesis, Institut für Künstliche Intelligenz, Fakultät Informatik (2010)

[33] Möhle, S., Biere, A.: Backing backtracking. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 250–266. Springer (2019). doi:10.1007/978-3-030-24258-9_18, `https://doi.org/10.1007/978-3-030-24258-9_18`

[34] Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10929, pp. 111–121. Springer (2018). doi:10.1007/978-3-319-94144-8_7, `https://doi.org/10.1007/978-3-319-94144-8_7`

[35] Oh, C.: Between SAT and UNSAT: the fundamental difference in CDCL SAT. In: Theory and Applications of Satisfiability Testing–SAT 2015–18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. pp. 307–323 (2015). doi:10.1007/978-3-319-24318-4_23

[36] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing–SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4501, pp. 294–299. Springer (2007). doi:10.1007/978-3-540-72788-0_28

[37] Ramos, A., van der Tak, P., Heule, M.J.H.: Between restarts and backjumps. In: Sakallah, K.A., Simon, L. (eds.) Theory and Applications of Satisfiability Testing—SAT 2011—14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6695, pp. 216–229. Springer (2011). doi:10.1007/978-3-642-21581-0_18

[38] Soos, M., Biere, A.: CryptoMiniSat 5.6 with YalSAT at the SAT Race 2019. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Race 2019 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2019-1, pp. 14–15. University of Helsinki (2019)

[39] Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 244–257. Springer (2009). doi:10.1007/978-3-642-02777-2_24, `https://doi.org/10.1007/978-3-642-02777-2_24`