Journal of Artificial Intelligence Research 74 (2022) 1515-1563 Submitted 01/2022; published 08/2022

Better Decision Heuristics in CDCL through Local Search and Target Phases

Shaowei Cai SHAOWEICAI.CS@GMAIL.COM
Xindi Zhang ZHANGXD@IOS.AC.CN
State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences
Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences
Beijing, China

Mathias Fleury FLEURY@CS.UNI-FREIBURG.DE
Armin Biere BIERE@CS.UNI-FREIBURG.DE
Albert-Ludwigs-University Freiburg, Freiburg, Germany
Johannes-Kepler-University Linz, Linz, Austria

Abstract

On practical applications, state-of-the-art SAT solvers dominantly use the conflict-driven clause learning (CDCL) paradigm. An alternative for satisfiable instances is local search solvers, which is more successful on random and hard combinatorial instances. Although there have been attempts to combine these methods in one framework, a tight integration which improves the state of the art on a broad set of application instances has been missing. We present a combination of techniques that achieves such an improvement. Our first contribution is to maximize in a local search fashion the assignment trail in CDCL, by sticking to and extending *promising* assignments via a technique called *target phases*. Second, we *relax* the CDCL framework by again extending *promising* branches to complete assignments while ignoring conflicts. These assignments are then used as starting point of local search which tries to find improved assignments with fewer unsatisfied clauses. Third, these improved assignments are imported back to the CDCL loop where they are used to determine the value assigned to decision variables. Finally, the *conflict frequency* of variables in local search can be exploited during variable selection in branching heuristics of CDCL. We implemented these techniques to improve three representative CDCL solvers (Glucose, MapleLcm DistChronoBT, and Kissat). Experiments on benchmarks from the main tracks of the last three SAT Competitions from 2019 to 2021 and an additional benchmark set from spectrum allocation show that the techniques bring significant improvements, particularly and not surprisingly, on satisfiable real-world application instances. We claim that these techniques were essential to the large increase in performance witnessed in the SAT Competition 2020 where Kissat and Relaxed_LcmdCbDl_NewTech were leading the field followed by CryptoMiniSAT-Ccnr, which also incorporated similar ideas.

1. Introduction

The satisfiability problem (SAT) asks to determine whether a given propositional formula is satisfiable or not. Propositional formulas are usually represented in conjunctive normal form (CNF). A growing number of problem domains are successfully tackled by SAT solvers, including electronic design automation (EDA) (Silva & Sakallah, 2000), particularly hardware verification (Prasad, Biere, & Gupta, 2005) and model checking (Vizel, Weissenbacher, & Malik, 2015; Biere & Kröning, 2018), mathematical theorem proving (Heule, Kullmann, & Marek, 2016), AI planning (Kautz & Selman, 1992), and spectrum allocation (Newman, Fréchette, & Leyton-Brown, 2018), among others. Additionally, SAT solvers are also often used as a core component of more complex tools such as solvers for satisfiability modulo theory (SMT) (Barrett, Sebastiani, Seshia, & Tinelli, 2021), which form a crucial component of state-of-the-art program analysis and software verification.

Many approaches have been proposed to solve SAT, but conflict-driven clause learning (CDCL) and local search are the most popular ones. Since their inception in the mid-90s, CDCL-based SAT solvers have been applied, in many cases with remarkable success, to a number of practical applications, because CDCL solvers are so effective in practice.

The local search paradigm is an incomplete method only able to solve satisfiable instances. Local search solvers begin with a complete assignment and iteratively modify it, typically by flipping the value of a single variable, until a model is found or a resource limit (usually time) is reached. Although local search solvers usually have worse performance than CDCL on practical instances, they can be more successful on random and hard combinatorial instances (Li & Li, 2012; Cai, Luo, & Su, 2015; Biere, Fazekas, Fleury, & Heisinger, 2020). Many techniques, including clause learning (Fang & Ruml, 2004) and unit propagation (Hirsch & Kojevnikov, 2005), have been tried to improve local search algorithms but they are still not competitive. Recent studies show that given a promising initial solution for local search helps to improve the performance on some benchmarks (Zhang, Sun, Zhu, Li, Cai, Xiong, & Zhang, 2020; Cai, Luo, Zhang, & Zhang, 2021). In this paper we go one step further and the two components exchange information.

It is usually believed that one limitation of CDCL solvers is that they frequently restart (in order to find short proofs) and therefore they usually work with partial short assignments (Ryvchin & Strichman, 2008; Oh, 2015). We argue that this makes finding complete assignments harder. Comparatively, local search solvers explicitly work on complete assignments.

There have been several attempts to combine both approaches. However, in previous hybrid solvers, both solvers, the CDCL and the local search solver, are opaque to each other, at most exchange some partial information in one direction and therefore usually see each other as a black box. These early hybrid solvers invoke the respective solver according to different situations (Mazure, Sais, & Grégoire, 1998; Habet, Li, Devendeville, & Vasquez, 2002; Letombe & Marques-Silva, 2008; Balint, Henn, & Gableske, 2009; Audemard, Lagniez, Mazure, & Sais, 2010) as discussed in Section 8 on related work.

This work is devoted to a tighter cooperation of CDCL and local search for SAT, with CDCL acting as the main solver and local search mainly used as a tool to improve branching heuristics in the CDCL solver. Occasionally the local search solver finds satisfying assignments too (particularly for random formulas where CDCL performs worse) but, of course,

cannot determine unsatisfiability on its own. In contrast to earlier work, the solvers work hand in hand and information flows frequently in both directions. Our main overall goal is thus to decrease the running time of CDCL solvers on real-world application instances, especially for satisfiable instances, by teaming it up with a local search solver as a "sidekick".

Our first two contributions are inspired by the concept of "promising branches" originating in the Glucose solver, where it was used to schedule, actually prohibit, solver restarts. In Section 3 we explicitly expand such promising branches in two different ways during CDCL solving, instead of just avoiding restarts, as proposed in Glucose. Our vehicle to improve models is to change the heuristic to determine values assigned to selected decision variables. Current state of the art relies on saving the previous value to which a variable was assigned (for instance through propagation) and reuse that value as *saved phase* (Pipatsrisawat & Darwiche, 2007) in case the variable is selected as decision. We present two different mechanisms which in regular intervals try to find improved sets of values.

Our **first contribution** and first mechanism to expand promising branches is called *target phases* and tries to maximize the size of the partial assignment (the trail size) explored during a CDCL run consistent under unit-propagation, forcing the CDCL solver to repeat the phases which led to the previously largest assignment. A larger assignment consistent under unit-propagation is considered an improvement, recorded, and then used for selecting values assigned to decision variables. Targeting these recorded phases forces the CDCL solver to stay close to this assignment with the hope to reach larger and larger assignments, thus gradually increasing the size of the assignment (the trail) until a full consistent thus satisfying assignment is found.

While target phases follow the local search principle to optimize a global criterion locally, our **second contribution** and second mechanism explores these promising branches directly by calling a local search solver on an extension of the current assignment, which is "relaxed", thus complete but not necessarily unit-propagation consistent. Promising branches of sufficient length are extended to a complete assignment by the default CDCL decision heuristic mechanism. We use unit-propagation to complete the model, while ignoring all conflicts along the way, because unit propagations are hard to find for local search. Then, a local search solver is called to find a model nearby. If the local search cannot find a model within a given time limit the CDCL search process resumes.

In order to make use of the effort spent in a failed local search attempt, which did not find a model (the usual case), our **third contribution** consists of saving the best assignment found during local search as new improved set of values and reuse it for the phase selection heuristics during assigning values to decision variables. This use of local search can also be considered as a "rephasing" technique (Section 4), which resets saved phases in regular intervals and thus implements a diversification strategy.

Besides the phases of the best assignment, statistics gathered during local search can provide additional information useful for guiding CDCL. As **fourth contribution** we propose to enhance the CDCL variable selection heuristic by giving more focus to those variables with high activity during local search. The idea is that variables for which it is "difficult" to find a consistent value should be given higher preference to be selected as decision during CDCL, as this might settle their value in a satisfying assignment early on or guide the solver to a short proof of unsatisfiability. As an approximation of this difficulty, we propose to use the variables' *conflict frequency* during local search, that is, its frequency of appearing in

unsatisfied clauses. This information is used to modify the variables' *activity* in the VSIDS heuristic and the variables' *learning rate* in the LRB heuristic (Section 5).

To counteract the effect of losing satisfying assignments during rephasing of parts of the formula, that is, for instance of some disconnected component, we further propose to find autarkies (Section 6) during rephasing. The idea is that a partial assignments which satisfies all clauses it touches allows to remove the touched clauses. We also discuss how to reconstruct models of the original formula from models of the simplified formula.

All these ideas are primarily trying to improve promising partial assignments further and they are indeed, according to our experiments, successful in substantially reducing solving time on satisfiable instances. They can further be implemented and incorporated into a SAT solver in such a way that solving time on unsatisfiable formulas in general degrades only slightly if at all, yielding a clear overall performance gain.

We have implemented our proposed techniques in four state-of-the-art CDCL solvers, including the latest version of Glucose (Audemard & Simon, 2009) (from the SAT Competition 2019), and the winners of the Main track of SAT Competition 2019 and 2020, namely MapleLcmDistChronoBT-DL (Kochemazov, Zaikin, Kondratiev, & Semenov, 2019), and Kissat and CaDiCaL (Biere et al., 2020). The experimental results clearly show that these techniques enable solving a remarkable number of additional instances in the main track benchmarks of the last three SAT Competitions from 2019 to 2021 (following the evaluation guidelines set out by the SAT Practitioner Manifesto (Biere, Järvisalo, Le Berre, Meel, & Mengel, 2020)). Moreover, the improved versions of the three CDCL solvers also give better results on an additional real-world benchmark arising from a spectrum repacking problem in the context of bandwidth auction.

As the experiments clearly show, exploration of promising branches either through target phases or through local search are very helpful to solve satisfiable instances, with a slight degradation on unsatisfiable instances (usually solving 2 or 3 fewer unsatisfiable instances on SAT Competition Benchmarks). Using conflict frequency of variables to enhance the CDCL branching strategy has to a large extent positive effects on satisfiable instances too and gives improvements on few unsatisfiable instances. Overall, our proposed techniques significantly improve the performance of CDCL solvers, leading to a remarkable increase in the total number of solved instances, which we also claim is the main reason for the large jump in performance of the top solvers in the SAT Competition 2020, where Kissat and Relaxed_LcmdCbDl_NewTech were leading the field followed by CryptoMiniSAT-Ccnr, which also incorporated similar ideas. In the latest SAT Competition 2021 variants of Kissat were dominating.

This work combines, on the one hand, our previous (partially unpublished) work on target phases (Biere, 2019), rephasing (Biere, 2017a, 2018), and using local search (Biere, 2019; Soos & Biere, 2019) to improve CDCL assignments which was presented at the workshop on Pragmatics of SAT in 2020 (POS'20) (Biere & Fleury, 2020) and, on the other hand, our paper published at SAT'20 on a deeper integration of local search into CDCL (Cai & Zhang, 2021), including "relaxed" CDCL, local search based rephasing and using conflict frequency to enhance branching heuristics. This publication received a best paper award at SAT'20 and its ideas can be dated back to our ReasonLS solver in the SAT Competitions 2018 (Cai & Zhang, 2018) and four relaxed CDCL solvers in the SAT Competitions

2019 (Cai & Zhang, 2019), and an earlier work on MaxSAT solvers that pass assignments between a decimation algorithm and a local search algorithm (Cai, Luo, & Zhang, 2017).

We discovered that both lines of works, while having been developed independently, have the same underlying ideas and give similar quite remarkable improvements. This is also the reason we decided to join forces on writing this article in order to provide a more complete understanding of the ideas. Compared to our previously reported empirical results we have further implemented both line of works in Glucose and provide more details.

2. Preliminaries

In this section we define the notion of formulas we need (Section 2.1). CDCL is a procedure to solve satisfiability problems in CNF (Section 2.2). In particular we recall how decisions work in most implementations. Unlike CDCL that builds a partial assignment, local search solvers work on adapting full assignments (Section 2.3). Finally, we give the setup for the experiments and the SAT solvers we used for our experiments (Section 2.4)

2.1 Preliminary Definitions and Notations on Formulas

Let $V = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables, a *literal* is either a variable x or its negation $\neg x$. A *clause* is a disjunction of literals. A conjunctive normal form (CNF) formula $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is a conjunction of clauses. For simplicity we assume non-tautological clauses, that is, there is no variable x which occurs positively ($x \in C$) and negative ($\neg x \in C$) in the same clause.

A (partial) mapping $\alpha : V \to \{0, 1\}$ is called an *assignment*. If α maps all variables to a Boolean value, it is *complete*; otherwise, it is a *partial*. The size of an assignment α, denoted as $|\alpha|$, is the number of assigned variables in it. The value of a variable x under an assignment α is denoted as $\alpha[x]$. An assignment α satisfies a clause iff at least one literal evaluates to true under α, and satisfies a CNF formula iff it satisfies all its clauses. A CNF formula F is satisfiable iff there is at least one satisfying assignment. Such a satisfying assignment is also called a *model*. The empty clause $\perp$ is always unsatisfiable, and represents a conflict. SAT is the problem of deciding whether a given CNF formula is satisfiable.

A key procedure in CDCL solvers is *unit propagation*. Whenever a clause has one unset literal and all others false, the unset variable is assigned to satisfy this clause. This process is run until fixpoint or until an *empty clause* (a clause false under the current assignment) is produced, also called *conflict*.

2.2 CDCL Solvers

For the sake of the presentation in this article, see (Marques Silva, Lynce, & Malik, 2021) for a more generic overview on CDCL, we consider CDCL SAT solvers to be composed of a *propagate-and-learn* and a *guessing* part. CDCL solvers do propagate-and-learn eagerly in practice (Section 2.2.1) and implementations do not differ much. However, the guessing policy and also the *search-restart* policy (Section 2.2.2) are both considered to be important for performance and differ across implementations.

Algorithm 1: Typical CDCL algorithm: CDCL(F, α)

```
1  dl ← 0;                    //decision level
2  if UnitPropagation(F, α)==CONFLICT then return UNSAT
3  while ∃ unassigned variables do
       /* PickBranchVar picks a variable to assign not in α          */
4      x ← PickBranchVar(F, α);
       /* PickBranchDirection picks the respective value             */
5      v ← PickBranchDirection(F, x, α);
6      dl ← dl + 1;
7      α ← α ∪ {(x, v)};
8      if UnitPropagation(F, α)=CONFLICT then
9          bl ← ConflictAnalysisAndLearning(F, α);
10         if bl < 0 then
11             return UNSAT;
12         else
13             Backtrack(F, α, bl);
14             dl ← bl;
15 return SAT;
```

2.2.1 Overall Organization

Algorithm 1 shows the standard procedure of a CDCL solver, where α is the current assignment, dl is the current decision level and bl denotes the backtrack level. Arguments to the functions are assumed to be passed by reference.

The *UnitPropagation* procedure performs Boolean constraint propagation on the formula and identifies potential conflicts. Once a conflict is derived, it is analyzed and a clause is derived by the *ConflictAnalysisAndLearning* function. This *learned clause* is then added to the clause database. Finally, *Backtrack* adapts the search to the newly learned clause. The branching heuristics consists of two procedures, where *PickBranchVar* selects a variable to assign and *PickBranchDirection* the respective phase.

Note that Algorithm 1 shows a simplified skeleton of a typical CDCL algorithm. It is still missing several important techniques, including restarts, clause deletion policies, and learned clause simplification, among others, as explained in (Marques Silva et al., 2021).

2.2.2 Decision Heuristics and Backtracking

There is a long history of research on branching heuristics in SAT. The choice of the branching heuristics is still considered today to have a large impact on the performance of SAT solvers. See for instance (Biere & Fröhlich, 2015) for a survey on the effect of branching heuristics. Here we briefly discuss three branching heuristics, that is, variants of function `PickBranchVar` of Algorithm 1, which are relevant to this article.

Variable State Independent Decaying Sum (VSIDS) is known to be the first heuristics to use information from recent conflicts instead of all present clauses (Moskewicz,

Madigan, Zhao, Zhang, & Malik, 2001). It relies on the concept of "variable activity". We describe the version used in MiniSAT (Eén & Sörensson, 2003) and most modern implementations of CDCL. Each variable has an *activity* attached to it. Each time a variable occurs in resolved clauses during conflict analysis, the solver increases its activity. This is referred to as *bumping*, effectively decaying the activity of all other variables. When selecting a branching variable, VSIDS picks the variable with the maximum activity score.

Learning-Rate Branching (LRB) (Liang, Ganesh, Poupart, & Czarnecki, 2016) frames branching as an optimization problem that picks a variable to maximize a metric called *learning rate*. The learning rate of a variable x at interval I is $\frac{P(x,I)}{|I|}$, where I is the sequence of conflicts that occurred between the assignment of x until it transitioned back to unassigned, $P(x, I)$ measures the number of conflicts in I, for which x occurred in at least one clause during resolving the corresponding learned clause, and $|I|$ matches the number of learned clauses generated in interval I. Furthermore, the authors of LRB interpret variable selection as optimization problem which is solved via a multi-armed bandit algorithm.

Move to front (VMTF) (Ryan, 2004; Biere & Fröhlich, 2015) is a heuristic to focus aggressively on literals involved in the most recent conflicts. It provides a simpler and more efficient implementation that focuses on the literals involved in the last conflicts. The idea is to mark the last learned literals as the most important – they are moved to the front of the queue and will be selected next (last in first out).

Phase Saving. Most modern decision heuristics, particularly all presented above, pick a variable first and then use another heuristic to determine its value, or *phase*, that it should be set to (i.e., whether the variable should be decided positively or negatively). This is function `PickBranchDirection` of Algorithm 1. Picking the right phase is actually the most important heuristic for satisfiable instances, because the solver can pick variables in arbitrary order if the phases form already a model (satisfying assignment).

Formerly, some state-of-the-art SAT solvers like Chaff (Moskewicz et al., 2001) used information based on the number of occurrences (Moskewicz et al., 2001) with the aim of increasing the number of satisfied clauses under the current assignment, but this is expensive. 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 (Pipatsrisawat & Darwiche, 2007) 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 quite remarkable 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. For instance, if the formula is composed of independent components, phase saving cheaply allows the solver to focus on one component instead of working on multiple components at the same time. In particular, if the problem includes *disjoint* satisfiable components, the interplay between the decision and phase saving heuristics achieves that each component is solved independently and satisfying assignments of previously solved components are maintained.

Rephasing. Clearly, phase saving should be considered an *intensification* strategy and by applying general heuristic search principles should benefit from complementing it with a corresponding *diversification* strategy. Accordingly, the idea of rephasing is to regularly reset saved phases. In principle, saved phases can be set arbitrarily, as phase selection does not influence correctness nor termination of CDCL.

The SAT solvers PrecoSAT (Biere, 2010) and PicoSAT (Biere, 2010) use a Jeroslow-Wang score (Jeroslow & Wang, 1990) to change the saved phases either on all or on irredundant only) clauses in regular intervals, following a Luby sequence. The motivation is to adapt the saved phases to the current formula. The SAT solver StrangeNight (Soos, 2013) 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 (Manthey, 2010, Section 3.1) for the SAT solver Riss (Balint, Belov, Järvisalo, & Sinz, 2015) 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 efficiency of SAT solvers on practical instances, restarts turn out to be important. In particular, fast restarts (Ramos, van der Tak, & Heule, 2011) are now common. Originally "Luby restarts" were heavily used because they are a priori optimal strategy (Luby, Sinclair, & Zuckerman, 1993). However, in recent years, most implementations switched to Glucose-style restarts (Audemard & Simon, 2012a), basically, a requirement for prevailing in the SAT Competition. Biere and Fröhlich's presentation acts as a survey on various restart heuristics (Biere & Fröhlich, 2015). To keep completeness of SAT solvers, either restarts must be delayed more and more (for instance following a Luby sequence), or alternatively the number of clauses kept during database reduction needs to be increased (as it is usually done for Glucose-style restarts).

The potential overhead generated by frequent restarts in performing the same decisions and propagations over and over again can be lessened by cheaply reusing parts of the trail (Ramos et al., 2011). 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, the 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 (Audemard & Simon, 2012b), restarts can be *blocked* and delayed whenever the current assignment looks *promising*, for example, if the trail length has increased by a predefined factor since the latest restart (Audemard & Simon, 2012b). Delaying restarts is mostly useful for satisfiable problems.

Another option is to alternate restart policies (Oh, 2015) and restart less or even suppress restarts for some time in regular intervals during the search process. The latter was shown to be beneficial for satisfiable instances in particular and can be considered as one corner stone to the large improvement of SAT solvers witnessed in the SAT Competition 2016.

2.3 Local Search Solvers

Local search algorithms (Hoos & Stützle, 2004) explore the search space using a neighborhood relation. They start somewhere in the search space and the space is explored following a neighboring relation until some criterion is met. In the context of SAT, the search space

is the set of complete assignments which is characterized as the set of strings $\{0,1\}^n$, where n is the number of variables in the formula.

For SAT, the most natural neighborhood maps candidate solutions to their set of Hamming neighbors, that is, candidate solutions that differ in exactly one variable, until a model is found. In this view, a step in SAT local search consists of flipping the value assigned to a single variable. A survey by the first author (Cai, 2015) provides more details on local search SAT solvers.

2.4 Experiment Preliminaries

This sections describes the set up to evaluate our proposed methods, including a description of the SAT solvers, benchmarks, running environment and experimental methodology.

Base Solvers. We choose several state-of-the-art CDCL solvers as the base solvers for our studies, namely Glucose v4.0 (Audemard & Simon, 2009), CaDiCaL 0v9 for evaluating target phases, which in essence is the version submitted to the SAT Competition 2020 (Biere et al., 2020), MapleLcmDistChronoBT-DL v2.1 (Kochemazov et al., 2019), and Kissat_sat (Biere et al., 2020). Glucose is a milestone of modern CDCL solvers and has won several gold medals in SAT Competitions. MapleLcmDistChronoBT-DL won the SAT Race 2019 and Kissat_sat won the Main Track of SAT Competition 2020.

We choose CCAnr (Cai et al., 2015) as the local search solver to integrate into the CDCL solvers Glucose and MapleLcmDistChronoBT-DL, while Kissat_sat already includes a simple local search procedure inspired by ProbSAT (Balint & Schöning, 2012) and particularly YalSAT (Biere, 2014). CCAnr is a local search solver with the aim for solving structured SAT instances and has shown competitive results on various structured instances from SAT competitions and applications.

Benchmarks. The experiments are carried out on the main track benchmarks of the SAT Competitions and one SAT Race of the last three years (2019 – 2021). Additionally, we evaluate the solvers on an important application benchmark suite consisting of 10 000 instances[1] from the spectrum repacking in the context of bandwidth auction which resulted in about 7 billion dollar revenue (Newman et al., 2018).

Experiment Setup. We conducted all experiments on a cluster of computers with Intel Xeon Platinum 8153 @2.00 GHz CPUs and 1 024 GB RAM under the operating system CentOS 7.7.1908. For each instance, each solver run with a cutoff time of 5 000 s. For each solver and benchmark year, we report the number of solved SAT/UNSAT instances and the total solved instances, denoted as '#SAT', '#UNSAT', and '#Solved', and the penalized run time average 'Avg' PAR2 score (as used in SAT Competitions), where the run time of a failed run is penalized by twice the cutoff time.

We show the results as tables and a cumulative distribution function (CDF, and not as a cactus plot), that is, as a graph showing the number of solved instances depending on the time. The higher the curve, the better the solver. The source codes[2] and detailed experiment results[3] are available online.

1. https://www.cs.ubc.ca/labs/beta/www-projects/SATFC/cacm_cnfs.tar.gz
2. http://lcs.ios.ac.cn/~caisw/Code/JAIR-SATcodes.zip
3. http://lcs.ios.ac.cn/~caisw/Code/JAIR-SATtables.zip

3. Exploring Promising Branches

CDCL attempts to produce short proofs and hence often restarts. To focus the search towards models, we force the CDCL part to improve models by using target phasing following ideas from local search. This forces CDCL to stay close to the target assignment (Section 3.1). Another way to improve models is to use a local solver directly to explore promising directions (Section 3.2) during the CDCL search.

3.1 Exploring Promising Branches By Directing CDCL

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, which prohibits restarts for the next 50 conflicts) (Audemard & Simon, 2012b).

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 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. It follows an idea from local search: the target is the assignment the solver tries to fulfill and one mutation corresponds to finding a better assignment. Unlike most local search methods, we still use and prioritize unit propagations over the target: propagations ignore the saved target assignment. Only decisions follow the previously saved target phases. More precisely, target phasing consists of the following three parts.

1. First, an implicit target assignment is saved. Whenever the current assignment becomes more promising (better) than the saved one, the latter is replaced. The current assignment, as represented by the "trail" of the solver is more promising if it assigns more variables (in terms of the size of the "trail") without leading to a conflict after propagation. Then the entire current assignment, that is, the trail, becomes the new target assignment. The replacement is done before each decision if there is no conflict.

2. 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 of the selected variable is unassigned, the solver defaults to the value provided by phase saving or even to the default phase if the variable was never assigned yet.

3. Third, the target assignment is reset after each rephasing to the initial all-unassigned state. This diversification strategy encourages the solver to find larger and larger target assignments until the next rephasing and has proved to be useful empirically.

Example 1. *To better understand the technique, we give a sketchy example. Assume we start from the empty trail ε, the target phase $\neg B \neg E$, and the saved phase $ABCDEF$. Then*

we decide the variable A. It has no value in the target phases, so we go for the default true *phase. Then we propagate to get the trail $A^+\neg BC$ and the saved phases $A\neg BCDEF$. We have found no conflict and a more promising model, so the new target phase is $A\neg BC$.*

Then we decide another variable D which is set to the default positive value as for A. We get the trail $A^+\neg BCD^+\neg E$ and the saved phases $A\neg BCD\neg EF$. We find a conflict and the trail becomes $\neg E$. We now decide another variable B which is set according to its target phase thus to false *(as $\neg B$ was saved as target phase).*

Restart Policy. To increase the chance of finding a model, the solver must work on relatively long assignments. However, this increases the risk of encountering heavy-tailed behavior (Gomes, Selman, Crato, & Kautz, 2000) 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)[4] in the spirit of the work by Chanseok Oh (Oh, 2015). Maple is based on Glucose, but the restart mechanism does not include its blocked literals and it has a mechanism to avoid some of the restarts when it is using LRB as decision heuristic.

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). 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. Instead of the Glucose default of 50, 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 mode switches though instead of $\mathcal{O}(n^2)$ (Biere et al., 2020).

3.2 Exploring Promising Branches By Local Search During CDCL

The previous section used CDCL to improve promising models, but a local search solver can achieve the same effect.

First, we provide the motivation of our method. By using reasoning techniques, CDCL solvers are able to prune most of the branches of the search tree. This is useful for solving unsatisfiable instances — to prove a formula is unsatisfiable, a CDCL solver needs to examine the whole search space, and therefore the more of the search tree is pruned, the more efficient the solver is. However, when solving satisfiable formulas, some promising branches are not immediately explored. This makes CDCL solvers miss opportunities for finding a solution. The exploration of promising branches can improve CDCL solvers on satisfiable formulas, and a natural way to do so is to employ local search at such branches.

Now, we present a method to explore promising branches by plugging a local search solver into the CDCL solver, which can improve the ability to find solutions while keeping the completeness of the CDCL solver. The framework of our method is described as follows (Figure 1).

During CDCL, whenever a node is reached with a promising assignment, the search is paused. The algorithm enters a non-backtracking mode, which uses unit propagation and

4. The stable mode was initially called "stable phase" (Biere, 2018), 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 the SAT Competition 2020 (Biere et al., 2020)

a decision heuristic to assign the remaining variables without backtracking, *relaxing* the condition to stop on the first identified conflict clauses. At the end, this leads to a complete assignment β, which the local search solver uses to search for a model nearby. If the local search fails to find a model within a certain time budget, then the algorithm goes back to the normal CDCL search from the node where it was paused (we call this a breakpoint).

We need to identify which branches (i.e., partial assignments) deserve exploration. We propose two conditions below, and any assignment α satisfying at least one of them is considered as promising and will be explored:

- A certain ratio of variables is assigned, that is, $\frac{|\alpha|}{|V|} > p$ and there is no conflict under α, where p is a parameter and is set to 0.4 according to preliminary experiments on a random sample of instances from recent SAT Competitions.
- A length ratio similar to the best saved assignment, that is, $\frac{|\alpha|}{|\alpha\_longest|} > q$ and there is no conflict under α, where q is set to 0.9 for the same reason.

In order to ensure that the search space of adjacent local search calls is sufficiently different, we disallow local search for a certain number of k restarts, where k is set to 500 for Glucose, and 400 for Maple.

As a starting point for local search we first have to create a full assignment. The simplest solution would be to use some saved information. However, it is difficult for local search to achieve unit propagation (as it would require to flip the right literals, making it very unlikely to find propagation chains). Hence we relax CDCL and complete the current partial assignment by alternating decisions and propagations while ignoring all conflicts. Notably, our implementation uses the same Boolean constraint propagation procedure[5] and therefore, also updates the watched literals and the blocking literals of the clause. The current implementation performs unit propagation whenever possible, and decides variables by randomly picking an unassigned variable and assigning a value to it using phase saving (as CDCL does) when propagation cannot continue. Note that the conflicting variables in the relaxed propagation keep their value and are not changed. This approach reuses the propagation loop and phase saving heuristics (although ignores conflicts). We call this approach *relaxed* CDCL because it allows some branches to be extended to a leaf even meeting conflicts, but does not change the data structures and the completeness. We could even reuse the decision heuristic to select variables. Besides the watched literals, the non-backtracking phase does not change the data structures used for CDCL search process.

After obtaining a complete assignment through this relaxed CDCL procedure, the local search solver is called on all problem clauses and all the permanently added learned clauses (i.e., of low LBD and thus heuristically important). On the contrary, Kissat uses only the irredundant ones. However, inprocessing is heavily used, hence it removes subsumed clauses and replaces them by smaller ones. Therefore, the short learnt clauses kept forever in Glucose have a high chance to be in the irredundant clauses, achieving a similar effect.

In general, the time spent in local search has to be limited. To keep the solving process deterministic, we count memory-accesses to estimate time, instead of relying on explicit time limit. Kissat schedules based on the number of memory accesses, but instead of counting each access, the number of cache line accesses is estimated (e.g., accessing a clause

5. Technically we duplicated the code to ignore conflicts, but otherwise there are no differences.

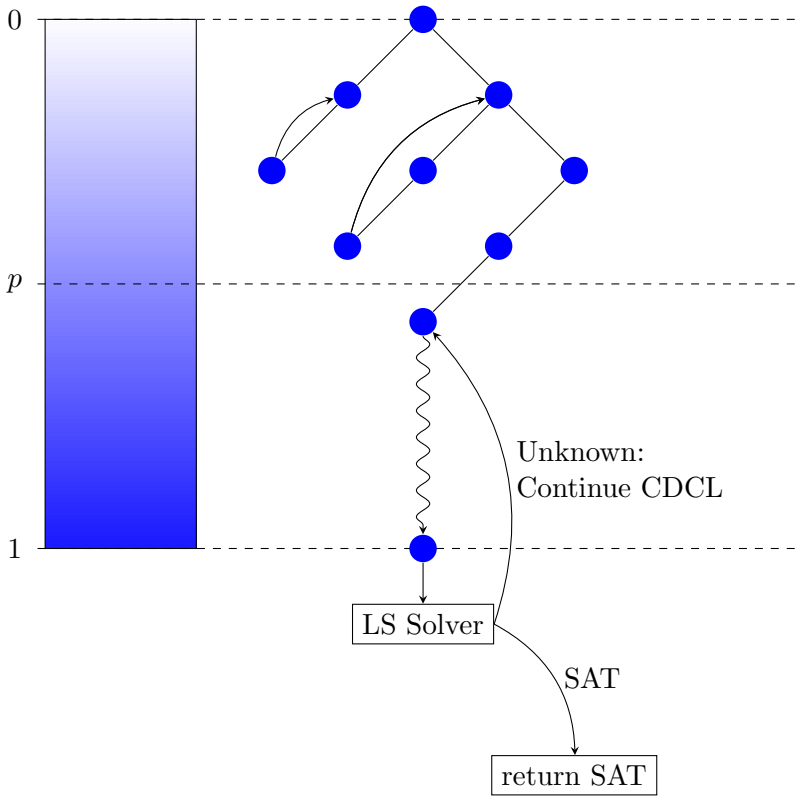


Figure 1: Overall procedure of relaxed CDCL

counts as one, whether one or all elements are evaluated), and the time spent on random walk is increasing in increasing intervals. For our relaxed CDCL implementation, we simply count the accesses to the vector saving the candidate variables. The limit is set to 5×10^7 and fixed during search.

4. Rephasing Heuristics

Exploring promising branches is already a useful addition to CDCL, but it is too stubborn in particular in combination with target phases. In this section, we propose two rephasing heuristics to provide more diversification. By *rephasing* we mean to globally reset or change saved values, and we call a variant of such transformations a *rephasing heuristic*. The first variant uses the improved assignment produced by the local seach solver (Section 4.1). To introduce more scrambling, we also make use of a structured rephasing, for example, setting all phases to true/false (Section. 4.2).

4.1 Local-search Rephasing

In Section 3.2 we proposed a method to plug a local search solver into CDCL solvers, where the CDCL solver helps the local search solver by providing a sensible starting point, from which local search is hoped to find a satisfying assignment in small number of steps. Now,

we propose a rephasing heuristic to import back an improved assignment obtained by the local search process, which is referred to as local-search rephasing (LS rephasing for short).

Algorithm 2: Relaxed CDCL Algorithm with Local-search Rephasing

```
1  dl ← 0, α ← ∅, α_longest ← ∅ ;
2  if UnitPropagation(F, α)=CONFLICT then
3      return UNSAT
4  while ∃ unassigned variables do
5      x ← PickBranchVar(F, α);
6      v ← PickBranchDirection(F, α);
7      dl ← dl + 1;
8      α ← α ∪ {(x, v)};
9      if UnitPropagation(F, α)=CONFLICT then
10         bl ← ConflictAnalysis(F, α);
11         if bl < 0 then
12             return UNSAT
13         else
14             α_longest ← max(α_longest, α);
15             Backtrack(F, α, bl), dl ← bl;
       /* lines 16-22 corresponds to the technique in Section 3.2        */
16     else if (|α|/|V| > p OR |α|/|α_longest| > q) then
17         β ← α;
18         while β is not complete do
19             x ← PickBranchVar(F, β);
20             v ← PickBranchDirection(F, β);
21             β ← β ∪ {(x, v)};
22             UnitPropagation(F, β);
23         if LocalSearch(β, terminate_condition) then
24             return SAT
25     if Meet Restart Conditions then
26         Backtrack(F, α, 0) ;
27         dl ← 0;
28         RephaseFromLocalSearch();          //corresponds to Section 4.1
29 return SAT;
```

Algorithm 2 describes a CDCL solver that implements the idea of exploring promising branches via local search, as well as the LS rephasing heuristic. Every time the CDCL solver restarts (which is forced by a certain schedule and simply backtracks to decision level zero), the LS rephasing heuristic overwrites the saved phases of all variables with assignments produced by local search. To this end, we record the best assignment (with the fewest unsatisfied clauses) in each run of the local search solver, and when we say the assignment of a local search procedure (run), we refer to the best assignment in this procedure.

Phase Name	α_longest_LS	α_latest_LS	α_best_LS	no change
Probability	20%	65%	5%	10%

Table 1: Probability of different phases in our local-search rephasing mechanism.

For our LS rephasing technique, we consider the following assignments, all of which come from the assignments of the local search procedures.

- α_longest_LS: This refers to the assignment of the local search procedure in which the initial solution is extended based on $\alpha\_longest$, where $\alpha\_longest$ is the longest assignment met during past CDCL search. Thus, whenever α_longest is updated, the algorithm calls the local search solver and updates α_longest_LS.
- α_latest_LS: This is the assignment of the latest local search procedure.
- α_best_LS: Among all local search assignments so far, we denote the best one (with the fewest unsatisfied clauses) as α_best_LS.

Local-search Rephasing: Whenever CDCL is restarted, we overwrite the saved phases. We reset all variables with one complete assignment which is selected according to the rephasing probabilities given in Table 1. Such changes are always allowed, because they do not impact the underlying CDCL calculus, its correctness, nor termination.

As can be seen, the LS rephasing considers both intensification and diversification — α_longest_LS and α_best_LS serve for the aim to derive longer models, while α_latest_LS adds diversification, as different local search procedures start with initial assignments built upon different branches. Given how fast restarts are scheduled in modern SAT solvers, the rephasing is done quite often, and with a certain probability (25%), it goes in the directions given by either α_longest_LS or α_best_LS, making it rather aggressive. Overall, it is expected this LS rephasing technique would work well particularly for satisfiable instances, and our experiment results confirm this. To determine the precise percentage we tried every combination (with a 5% increment).

One implementation detail worth mentioning is the restarting policy in Glucose. Its default configuration adapts the strategy according to statistics gathered during the first 100 000 conflicts. We do not change that. However, it blocks restarts (Audemard & Simon, 2012b) as mentioned before. In our first implementation, the restart frequency was so slow that the effect of rephasing was not good. So we remove the blocking restart method from Glucose and only use the LBD quality-based restart policy as Maple to increase the restart frequency and instead rely on our methods to derive longer models. In particular, we removed the restarts from the strategy adaption used by Glucose.

4.2 Fixed Rephasing

The previous section introduced some diversification. However, it is still very search related. In this section, we introduce a more structured version of rephasing that not only considers the past search behavior, but also changes the phases independently to cover more parts of the search space.

4.2.1 Rephasing Options

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 learned clauses. We conjecture that this, in turn, improves the efficiency of inprocessing (e.g., learning additional important clauses, like small 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 CaDiCaL and Kissat 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 (Audemard & Simon, 2009) to apply those rephasings does not make solving certain adversarial factorization problems (Biere, 2017b) 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, for example, simulating hyper binary resolution (Heule, Järvisalo, & Biere, 2013), 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 or a saved assignment. Our local search implements the ProbSAT strategy (Balint, 2014). 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 (Soos, Nohl, & Castelluccia, 2009) uses local search in a similar way (Soos & Biere, 2019). The search is in essence similar to the one described in the previous section, but it works on a different model: Instead of exploring the current promising assignment while executing CDCL, it alternates between the best model found so far and the current model formed by the saved phases.

This method is very similar to the one described in Section 4.1 but scheduled differently, less frequently, alternating with the other rephasing procedures and originally (and inde-

pendently) employed in the SAT solvers CaDiCaL and Kissat of the last author, Biere, who developed it first.

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 ever in contrast to target phases which are reset during rephasing. For unsatisfiable instances, this corresponds to focusing on the unsatisfiable region of the search space. This differs from the rephasing heuristic used in the previous section where the model is never reset. The intuition behind that choice is that the best model can get stuck to a local optimum, hence resetting it can help changing the search direction.

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) (Han & Somenzi, 2009; Hamadi, Jabbour, & Sais, 2009) or in combination with chronological backtracking (Nadel & Ryvchin, 2018; Möhle & Biere, 2019). 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).

4.2.2 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 potentially harming satisfiable instances that can be split into components. To avoid that, we detect such cases, called *autarkies*, in Kissat, enabling the removal of satisfied components (See Section 6).

Rephasing Strategies. We schedule the different rephasing heuristics in geometrically increasing intervals, unlike the heuristics described in the previous section that are applied after each and every restart. 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 7, but the idea is to apply them in geometrically increasing intervals.

5. Directing the Branching Heuristic with Local Search

In the current presentation, CDCL and the local search solver only exchange assignments, but no information on the search process. In particular, there is no exchange on the variables that are usually involved in conflict clauses, while both solvers use this information: the branching heuristic of CDCL focuses on such variables (the more often it is in a conflict,

the more you should focus on it) and the local search solver prefers to flip those variables. To transmit information to CDCL, we use the conflict frequency of variables in the *latest* local search procedure.

Definition 2 (Conflict frequency)**.** *In a local search process, the conflict frequency of a variable x, denoted as* ls_confl_freq(x)*, is the ratio of flips in which x appears in at least one unsatisfied clause.*

The intuition behind the definition is that the unsatisfied clauses have a similar role to the conflicts during CDCL, so we use that information to adapt the scores in the branching heuristics of CDCL. To update the scores in the branching heuristics, we first multiply ls_confl_freq(x) with a constant integer (100 in this work), and the resulting number is denoted as ls_conflict_num(x). After each restart of the CDCL solver, ls_conflict_num(x) is used to modify the activity score of the variable x for VSIDS and learning rate for LRB.

VSIDS: for each variable x, its activity score is increased by ls_conflict_num(x).

LRB: for each variable x, the number of learned clause during its period I is increased by the number of conflicts ls_conflict_num(x). That is, both $P(x, I)$ and $L(I)$ are increased by ls_conflict_num(x).

The bumping is not done immediately, but only be executed after the next restart, that is, while doing the local-search rephasing described in Section 4.1. We delay rephasing to avoid changing the exploration direction that is done by the CDCL solver currently: As important variables might have been already set, they cannot be decided again, limiting the effect of the search redirecting.

6. Autarky Detection

When the initial problem is composed of several independent subproblems and one such subprolbem is satisfied, the SAT solver will focus on the other parts. With phase saving, the component will remain satisfied. One limitation of rephasing is that the satisfying values for these components are lost. To overcome the issue, it is possible to rephase only some variables without changing the phase of satisfied components or to explicitly identify components, called *autarkies*, that are satisfied by the current assignment and remove them from the overall formula (Section 6.1). If the overall problem is deemed satisfiable, the full model is reconstructed at the end (Section 6.2).

6.1 Algorithm

An autarky is a assignment that fulfills parts of the formula without touching other parts of the formula. This allows for the fulfilled part to be removed from the overall formula without changing the status (SAT or UNSAT). More formally:

Definition 3 (Autarky)**.** *An autarky is an assignment α such that every clause C of F is either entailed ($\alpha \vDash C$) or disjoint ($\alpha \cap \neg C = \emptyset$).*

We use an algorithm originally proposed by Kullmann and described in a publication by Kiesl et al. (Kiesl, Heule, & Biere, 2019). Instead of forcing the autark assignment we simply eliminate the clauses touched by the autarky as well as the variables assigned by it.

Algorithm 3: Algorithm to identify and adapt an autarky from a model

Data: An assignment α and a formula F
Result: An autarky and the adapted formula

```
while there is a clause C ∈ F such that α ⊭ C and α ∩ ¬C ≠ ∅ do
    α := α \ ¬C
foreach literal ℓ ∈ α do
    remove all clauses C from F with ℓ ∈ C;
    add the unit clause ℓ to the reconstruction stack with ℓ as witness;
return (α, F)
```

Algorithm 4: Implementation of the algorithm to identify an autarky from a model

Data: An assignment α and a formula F
Result: An autarky

```
α_S := α;
while α_S ≠ ∅ do
    ℓ := pop α_S;
    foreach clause C in F containing ℓ such that α ⊭ C and ¬C ∩ α ≠ ∅ do
        α_S := α_S ∪ (¬C ∩ α);
        α := α \ ¬C
return α
```

The algorithm is given in Algorithm 3. It is composed of two loops. The first one reduces the current assignment by removing all literals that falsify any clause. If the remaining assignment is not empty, then an autarky was found and the formula can be trimmed by removing all entailed clauses from the set of clauses.

Theorem 4 (Correctness)**.** *Given an assignment α and $\alpha_0 \subseteq \alpha$ be any autarky. After running Algorithm 3, the resulting assignment α' contains α_0 and is an autarky.*

Proof. Let β_i be the updated α after going through the first loop i times. Our first goal is to prove $\alpha_0 \subseteq \beta_i$. Note that $\alpha_0 \subseteq \beta_0$ by assumption, as β_0 is the original α. For the induction step assume $\alpha_0 \subseteq \beta_i$ and that there is still a clause $C \in F$ with $\beta_i \not\models C$ and $\beta_i \cap \neg C \neq \emptyset$. Since $\alpha_0 \subseteq \beta_i$ the first condition shows $\alpha_0 \not\models C$ which implies $\alpha_0 \cap \neg C = \emptyset$ as α_0 is an autarky. Therefore $\alpha_0 = (\alpha_0 \backslash \neg C) \subseteq (\beta_i \backslash \neg C) = \beta_{i+1}$. To prove that α' is an autarky if the loop terminates after n iterations with $\beta_n = \alpha'$ follows immediately as the negation of the loop condition matches the definition of autarky. □

Now, we can prove that Algorithm 3 not only derives an autarky, but the maximal autarky contained in the initial assignment, possibly the empty assignment.

Corollary 5 (Maximal Autarky)**.** *Each assignment α has a unique maximum autarky with $\alpha' \subseteq \alpha$ among all autarkies $\alpha'' \subseteq \alpha$. This maximum α' is computed by Algorithm 3.*

Proof. Assume $\alpha_0, \alpha_1 \subseteq \alpha$ are autarkies. Theorem 4 shows that $\alpha' \supseteq (\alpha_0 \cup \alpha_1)$. Therefore there can not be two different maximal autarkies. □

Algorithm 4 is a refined version with more implementation details of the first loop of Algorithm 3. The solver relies on the occurrence lists to efficiently find all clauses containing a given literal. This is not too costly as redundant learned clauses can be ignored. The algorithm terminates because α_S can contain each literal at most once during execution.

Theorem 6 (Complexity)**.** *The complexity of Algorithm 3 is* $\mathcal{O}(\sum_{C \in F} |C|^2)$.

Proof. The algorithm iterates over every occurrence of a literal in the clauses at most twice during the execution: once if coming from the initial α and potentially a second time if the literal was removed from the assignment α. Therefore, every clause will be read at most twice for each of its literals, and each clause check requires iterating over all the literals. □

The implementation in our SAT solver Kissat is made more efficient by first running the loop of Algorithm 3 *without* adding literals to α_S (called `work` in the actual code[6]). This reduces the number of clauses to visit. Our actual implementation in our SAT solver Kissat is slightly more complicated, because binary clauses are only represented implicitly. Additionally, whenever α becomes empty the execution is stopped immediately as the inner loop condition in Algorithm 4 will be false for all literals.

Performance is further improved by the fact that all irredundant clauses follow each other in memory (without interleaved redundant clauses), improving processor cache efficiency. In our experiments with the SAT solver Kissat, the time spent to identify autarkies is small enough to not be a problem in general and the algorithm can be run until completion, even for very large instances.

6.2 Model Reconstruction

Whenever a non-trivial (non-empty) autarky is found the formula is simplified by removing clauses satisfied by the autarky. Then the solver continues searching for a model. However, if this is successful and a model for the simplified formula is found later, that model does not necessarily satisfy the original formula before applying the autarky assignment and removing the touched clauses.

A similar situation occurs when lifting models to the original formula after variable elimination or removing blocked clauses. The standard solution is to use a *reconstruction stack* on which removed clauses paired with witnesses (in the form of cubes resp. partial assignments) are pushed. During model reconstruction these clauses are consulted and the model is fixed by applying the witness assignment in case such a clause turns out not to be satisfied. This technique was first described in (Järvisalo & Biere, 2010) but goes back to to Niklas Sörensson who proposed it in the context of MiniSAT. For more details please refer to the recent Handbook chapter on preprocessing (Biere, Järvisalo, & Kiesl, 2021).

Thus the implementation of our autarky algorithms also has to properly fill the reconstruction stack. The main question is which witness should be used and whether a single witness clause pair is sufficient. For Algorithm 3 we decided to simply add the literals $\ell \in \alpha$

6. see file `autarky.c` available at `fmv.jku.at/kissat` and in particular the function `propagate_clause` that trims the model for non-binary clauses.

satisfied by the autarky α as units to the reconstruction stack with itself as witness. After removing all the clauses satisfied by the autarky these literals do not occur in the formula anymore and thus we can simply force them to be true during reconstruction.

In principle this has the same effect as learning these units as redundant (PR) clauses as proposed in (Heule, Kiesl, & Biere, 2020) and then removing the same clauses but now because they are satisfied by these units. Unfortunately, this approach would require more sophisticated proof checking, as adding those units is not model preserving, while adding the units only on the reconstruction stack does not influence proof generation and checking.

However, Algorithm 3 is not compatible with incremental SAT solving (Fazekas, Biere, & Scholl, 2019) as those unit witnesses are clearly dependent on each other. It is possible to simply use the full autarky as witness instead, but that blows up the reconstruction stack (quadratically in the worst case) particularly if the autarky is large.

This potentially exploding reconstruction stack is a problem we also saw in the context of globally blocked clauses (Kiesl et al., 2019), also in practice, as well as for covered clause elimination (Barnett, Cerna, & Biere, 2020). We leave it to future work to come up with a space efficient method for autarky reconstruction in incremental SAT Solving. This probably requires a non-clausal reconstruction stack (Barnett & Biere, 2021).

7. Experiments

We have implemented the techniques described in the previous sections in several solvers. Even though technically different, they have the same motivation and share ideas, and all have the same goal to enhance the decision heuristics of CDCL. The results presented in this section show that these improved heuristics yield better performance, particularly for satisfiable instances.

We implemented target phasing and fixed rephasing in CaDiCaL, Glucose, and Kissat (Section 7.1), while the deep combination between CDCL and local search is implemented in Glucose, Maple, and Kissat (Section 7.2). We further combine the techniques of both lines of research in Glucose and Kissat (Section 7.3).

7.1 Directed CDCL

Our heuristics for directed CDCL have been implemented in the SAT solvers CaDiCaL and Kissat and ported to the SAT solver Glucose.

7.1.1 Implementation

Default Policies. We assume that assignments saved either as best or as target phases are good candidates for expansion, and thus finding models faster. Hence, we spend most time on '`B`' phases.

In CaDiCaL the search mode is based on the number of conflicts found so far. In focused mode, the default rephasing policy is '`OI(BWOBWI)`$^\omega$' (`O`riginal, `I`nverted; then `B`est, `W`alk, `O`riginal 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 '`(IBWFBW#BWOBW)`$^\omega$' (`F`lipped 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 '`(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" (Knuth, 2006). Unlike CaDiCaL, in the satisfiable configuration ('`--sat`') submitted to the SAT competition, Kissat alternates between focused and stable mode and always uses target phasing. We have experimented with various parameters to decide how to schedule the rephasing, but the results seems rather robust (even starting with rephasing every 500 conflicts does not lead to worse performance).

All our implementations use the alternation of stable mode with Luby-style restarts and focused mode with Glucose-style restarts. The idea of stable mode is to change less overall. Hence, Kissat and CaDiCaL use chronological backtracking (Nadel & Ryvchin, 2018; Möhle & Biere, 2019). Both solvers also use two separate decision queues as suggested by Oh (Oh, 2015). They use VMTF (Biere & Fröhlich, 2015) in focused and VSIDS in stable mode. VMTF during focused mode makes the solver more agile whereas VSIDS with a low bumping is more stable.

Kissat is the only solver to include autarky detection and elimination. We experimented and found no obvious performance gain or loss. It turns out that autarky detection is fast enough to be executed until completion.

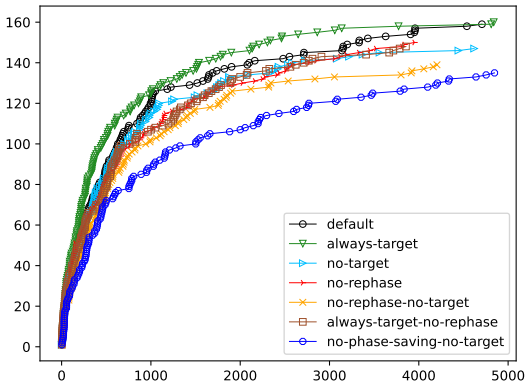
Implementation in Glucose. To have a common platform for comparison between both lines of research, we also implemented our heuristics in the SAT solver Glucose. However, to avoid too many changes to the base solver we did not implement a separate different decision queue for stable mode. Instead, to increase stability, we decrease the bonus that bumped variables get by setting the variable decay to a smaller value. In focused mode we do the opposite and increase the decay compared to the default value in order to follow more closely the search process. 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 (Liang et al., 2016), 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*).

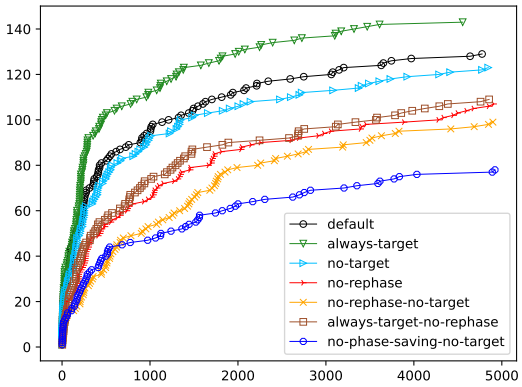
7.1.2 Rephasing and Target Phases

For CaDiCaL and Kissat we have tested 7 configurations.

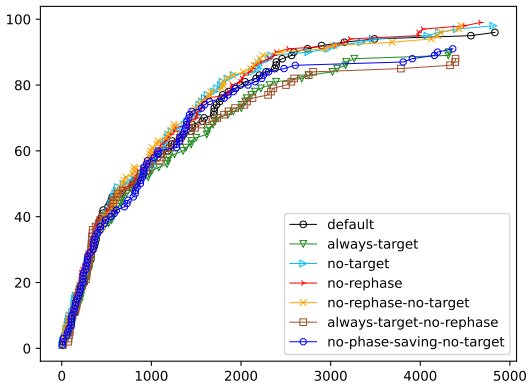
always-target (resp. no-target) always (resp. never) uses target phases to set the value of decision variables. By default, target phasing is only activated in stable mode.



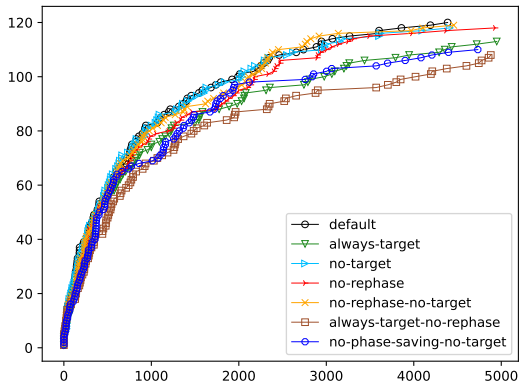
(a) Satisfiable instances only



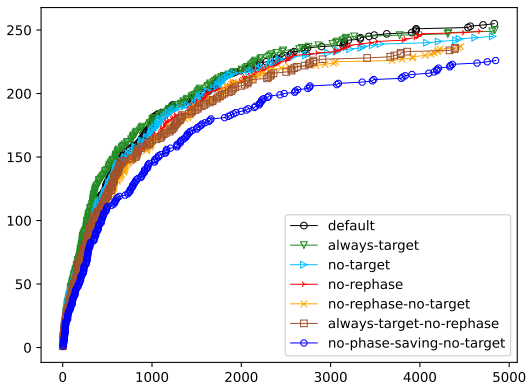
(b) Satisfiable instances only



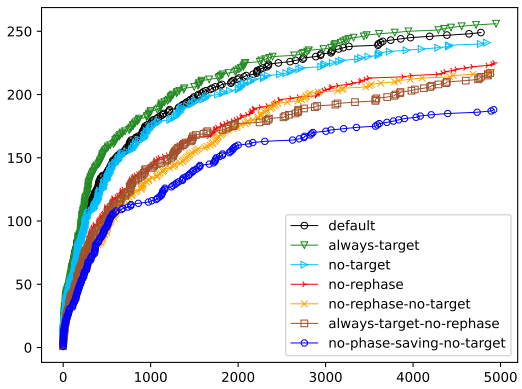
(c) Unsatisfiable instances only



(d) Unsatisfiable instances only

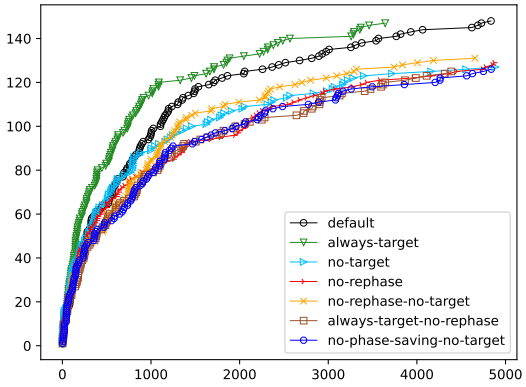


(e) All instances

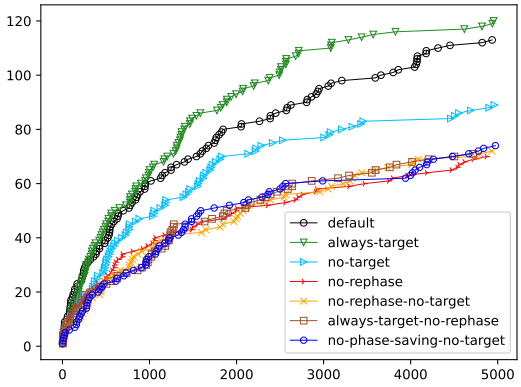


(f) All instances

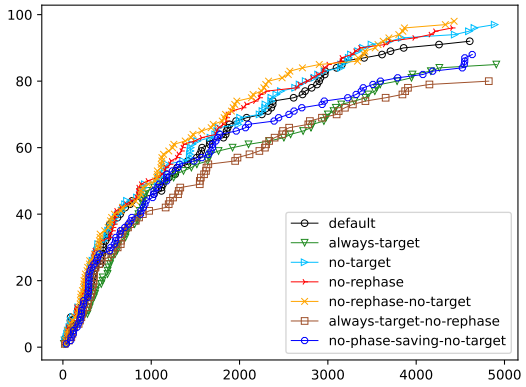
Figure 2: CDF for the solver Kissat on benchmarks from the SAT Race 2019 (left) and SAT Competition 2020 (right)



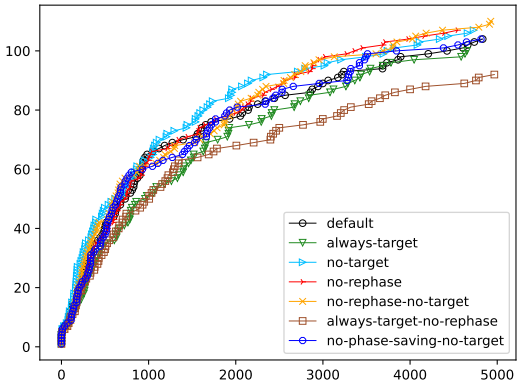
(a) Satisfiable instances only



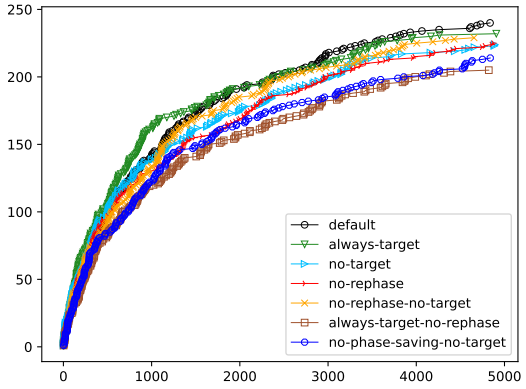
(b) Satisfiable instances only



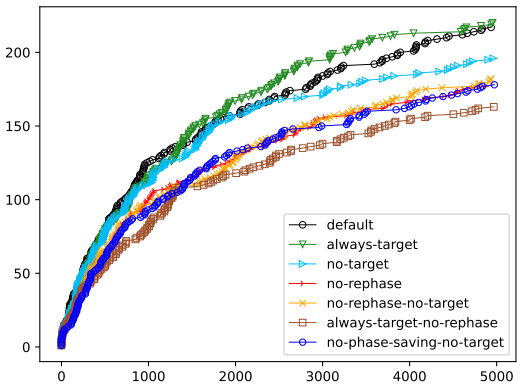
(c) Unsatisfiable instances only



(d) Unsatisfiable instances only

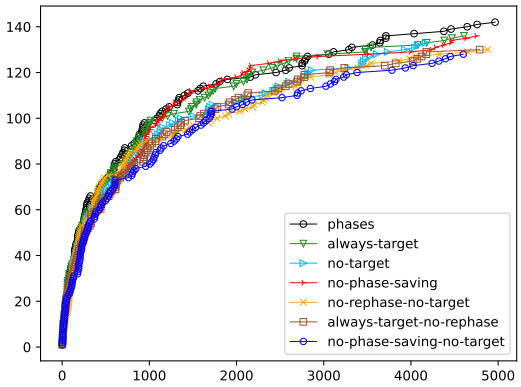


(e) All instances

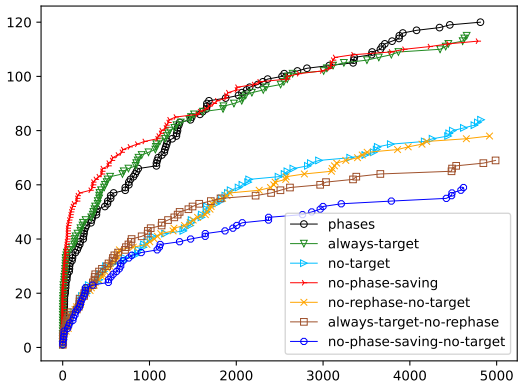


(f) All instances

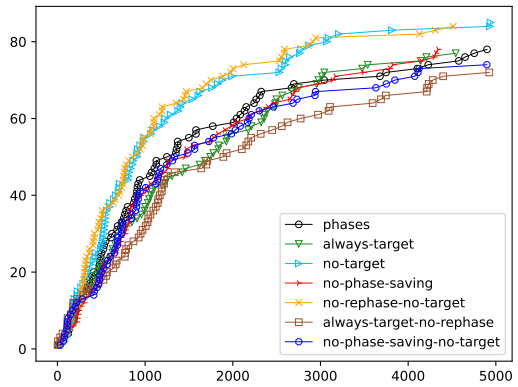
Figure 3: CDF for the solver CaDiCaL on benchmarks from the SAT Race 2019 (left) and SAT Competition 2020 (right)



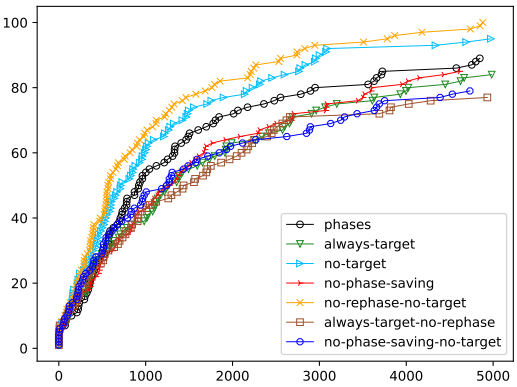
(a) Satisfiable instances only



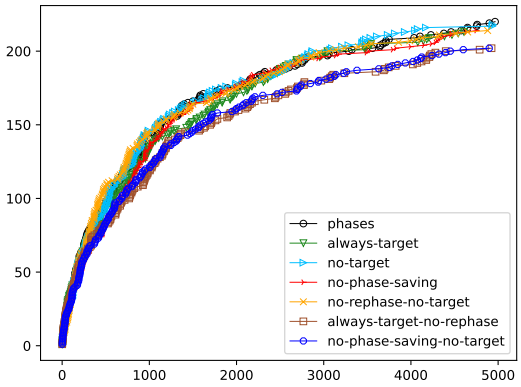
(b) Satisfiable instances only



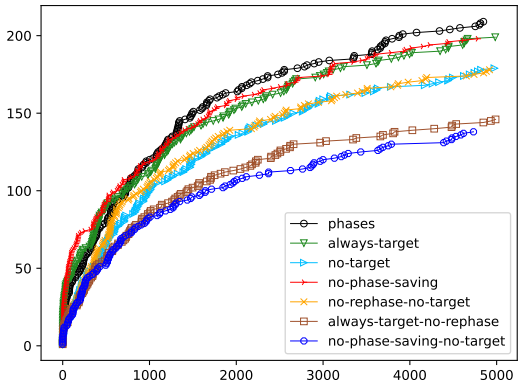
(c) Unsatisfiable instances only



(d) Unsatisfiable instances only



(e) All instances



(f) All instances

Figure 4: CDF for the solver Glucose on benchmarks from the SAT Race 2019 (left) and SAT Competition 2020 (right)

no-rephasing never uses rephasing.

default (called **phases**) is an 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*.

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 focused mode, whereas 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.

Results for Kissat and CaDiCaL are presented in Figures 2 and 3 and in Tables 2 and 4. The effect in the SAT Competition 2021 and SAT Competition 2020 are very similar. Thus, for conciseness, we do not include the CDF for the SAT Competition 2021.[7]

First 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.

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.

Glucose. The results of Glucose give a slightly different picture, see Figure 4 and Table 3, with some interesting results.[8] First, it seems that target phasing degrades performance on unsatisfiable benchmarks, where target phasing is detrimental and our alternating approach is not able to compensate.[9]

Also the impact of the new 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 or a dedicated separated decision heuristic is required to get more out of our heuristics. Third, rephasing is less helpful than for Kissat and CaDiCaL. Fourth, the phases configuration (the same alternating scheduling as for Kissat) solves most SAT problems, albeit by a very small margin. The performance of no-phase-saving is surprising and seems to be due to stable mode: Deactivating it significantly reduces performance on unsatisfiable problems. Remember that *no-phase-saving* uses the alternation between stable and focused mode unlike *original*.

Interpretation. Overall, the performance increases on satisfiable instances. Generally, rephasing with target phasing improves the performance of the solver and makes it more

7. Note to the reviewers, Figures 5 and 6 are part of the appendix.
8. As for the previous case, the CDF from the SAT Competition 2021 is part of the appendix in Figure 7.
9. In our implementation in Glucose 3, this was less detrimental.

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Race 2019(400)				
CaDiCaL-default	148	92	240	4 688.13
CaDiCaL-always-target	147	85	232	4 774.77
CaDiCaL-no-target	127	97	224	5 028.43
CaDiCaL-no-rephase	129	96	225	5 052.92
CaDiCaL-no-rephase-no-target	131	98	229	4 913.18
CaDiCaL-always-target-no-rephase	126	80	206	5 478.14
CaDiCaL-no-phase-saving-no-target	126	88	214	5 313.91
SAT Competition 2020(400)				
CaDiCaL-default	113	104	217	5 316
CaDiCaL-always-target	120	100	220	5 217
CaDiCaL-no-target	90	107	197	5 687
CaDiCaL-no-rephase	70	107	177	6 165
CaDiCaL-no-rephase-no-target	72	110	182	6 094
CaDiCaL-always-target-no-rephase	71	92	163	6 489
CaDiCaL-no-phase-saving-no-target	74	104	178	6 174
SAT Competition 2021(400)				
CaDiCaL-default	115	139	254	4 310
CaDiCaL-always-target	126	134	260	4 243
CaDiCaL-no-target	116	141	257	4 260
CaDiCaL-no-rephase	104	139	243	4 606
CaDiCaL-no-rephase-no-target	107	145	252	4 387
CaDiCaL-always-target-no-rephase	99	122	221	5 151
CaDiCaL-no-phase-saving-no-target	90	137	227	4 966

Table 2: Summary of the performance of the SAT solvers CaDiCaL

robust to solve problems where a model with only *true* variables exists.[10] Performance on unsatisfiable instances degrades slightly except for Glucose.

On the other hand 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.

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, which inprocessing approaches are used, etc.). Our experience is however, that using best rephasing every second or third time rephasing is scheduled gives better results.

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

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Race 2019(400)				
Glucose-phases	142	78	220	5 107
Glucose-always-target	136	77	213	5 247
Glucose-no-target	133	85	218	5 116
Glucose-no-phase-saving	136	78	214	5 234
Glucose-no-rephase-no-target	130	84	214	5 184
Glucose-always-target-no-rephase	130	72	202	5 535
Glucose-no-phase-saving-no-rephase	128	74	202	5 525
SAT Competition 2020(400)				
Glucose-phases	120	89	209	5 404
Glucose-always-target	115	84	199	5 624
Glucose-no-target	84	95	179	6 112
Glucose-no-phase-saving	113	85	198	5 585
Glucose-no-rephase-no-target	78	100	178	6 089
Glucose-always-target-no-rephase	69	77	146	6 791
Glucose-no-phase-saving-no-rephase	59	79	138	6 980
SAT Competition 2021(400)				
Glucose-phases	107	117	224	4 963
Glucose-always-target	111	113	224	4 991
Glucose-no-target	103	128	231	4 799
Glucose-no-phase-saving	105	116	221	5 101
Glucose-no-rephase-no-target	106	128	234	4 716
Glucose-always-target-no-rephase	96	99	195	5 655
Glucose-no-phase-saving-no-rephase	83	119	202	5 605

Table 3: Summary of the performance of the SAT solvers Glucose

7.2 Techniques With Local Search

The techniques of deep combination of CDCL and local search include (1) exploring promising branches by local search (denoted as `rx`, Section 3.2); (2) local-search rephasing (denoted as `rp`, Section 4.1) and (3) directing the branching heuristics with local search conflict frequency (denoted as `cf`, Section 5). The experiment setup is described in Section 2.4.

For Glucose and MapleLcmDistChronoBT-DL-v2.1, we implement all the three techniques in this work. For Kissat, we only implement the `cf` technique because it already has a local search solver. We focus on Kissat_sat, the version of Kissat that focuses on satisfiable instances. Nevertheless, it is easy to apply the `cf` technique to Kissat, which is what we do in this work.

Evaluations on Benchmarks of SAT Competitions. The results of evaluations of all the base solvers and the different versions with our techniques are reported in Table 5. The CDFs of these experiments are included in the appendix. According to the results, we have some observations.

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Race 2019(400)				
Kissat-default	159	96	255	4 213
Kissat-always-target	160	90	250	4 255
Kissat-no-target	147	98	245	4 404
Kissat-no-rephase	150	99	249	4 352
Kissat-no-rephase-no-target	139	98	237	4 609
Kissat-always-target-no-rephase	148	88	236	4 607
Kissat-no-phase-saving-no-target	135	91	226	4 939
SAT Competition 2020(400)				
Kissat-default	129	120	249	4 304
Kissat-always-target	143	113	256	4 123
Kissat-no-target	123	118	241	4 497
Kissat-no-rephase	107	118	225	4 974
Kissat-no-rephase-no-target	99	119	218	5 142
Kissat-always-target-no-rephase	109	108	217	5 171
Kissat-no-phase-saving-no-target	78	110	188	5 785
SAT Competition 2021(400)				
Kissat-default	125	151	276	3 638
Kissat-always-target	135	141	276	3 681
Kissat-no-target	120	149	269	3 896
Kissat-no-rephase	114	151	265	3 985
Kissat-no-rephase-no-target	108	149	257	4 164
Kissat-always-target-no-rephase	118	130	248	4 293
Kissat-no-phase-saving-no-target	94	137	231	4 780

Table 4: Summary of the performance of the SAT solvers Kissat

- The `rx` technique improves Glucose and MapleLcmDistChronoBT-DL-v2.1 on solving satisfiable instances, particularly for the benchmarks of 2020 (increased by 17 and 35 for #SAT). On the other hand, the Glucose+rx and Maple-DL+rx have slightly worse performance than the original versions on unsatisfiable instances, and the decrease on #UNSAT is only 2 on average, considering both solvers on all benchmarks.

- By adding the `rp` technique, Glucose+rx+rp and Maple-DL+rx+rp gain further improvement on #SAT, which is significant for all benchmarks. The increase on satisfiable instances is between 6 and 38 problems. Howover, some unsatisfiable instances (less than 4) are lost.

- The impact of the `cf` technique can be seen from the comparisons of Glucose+rx+rp vs. Glucose+rx+rp+cf, Maple-DL+rx+rp vs. Maple-DL+rx+rp+cf, and Kissat_sat vs. Kissat_sat +cf. The results are mixed: On the 2020 benchmarks for Maple the increase is significant for satisfiable instaces. Similar results appear for the 2019 benchmarks with Glucose. Interestingly, the performance for unsatisfiable

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Competition 2019(400)				
Glucose _4.0	115	86	201	5 531
Glucose+rx	120	85	205	5 430
Glucose+rx+rp	131	86	217	5 191
Glucose+rx+rp+cf	143	87	230	4 915
Maple-DL-v2.1	143	97	240	4 602
Maple-DL+rx	146	93	239	4 602
Maple-DL+rx+rp	152	91	243	4 535
Maple-DL+rx+rp+cf	154	95	249	4 377
Kissat_SAT	160	90	250	4 255
Kissat_SAT +cf	163	91	254	4 189
CCAnr1.0	13	0	13	9 678
SAT Competition 2020(400)				
Glucose _4.0	77	93	170	6 325
Glucose+rx	94	90	184	5 939
Glucose+rx+rp	132	91	223	4 942
Glucose+rx+rp+cf	126	98	224	4 978
Maple-DL-v2.1	86	104	190	5 837
Maple-DL+rx	121	105	226	4 978
Maple-DL+rx+rp	141	101	242	4 512
Maple-DL+rx+rp+cf	151	106	257	4 171
Kissat_SAT	143	113	256	4 123
Kissat_SAT+cf	146	113	259	4 055
CCAnr1.0	45	0	45	8 979
SAT Competition 2021(400)				
Glucose _4.0	96	126	222	5 094
Glucose+rx	103	125	228	4 966
Glucose+rx+rp	120	121	241	4 631
Glucose+rx+rp+cf	125	126	251	4 312
Maple-DL-v2.1	104	133	237	4 703
Maple-DL+rx	108	125	233	4 724
Maple-DL+rx+rp	129	121	250	4 364
Maple-DL+rx+rp+cf	130	123	253	4 293
Kissat_SAT	135	141	276	3 681
Kissat_SAT+cf	138	142	280	3 594
CCAnr1.0	24	0	24	9 409

Table 5: Experiment results on benchmarks from SAT Competitions 2019-2021, where Maple-DL-v2.1 is short for MapleLcmDistChronoBT-DL-v2.1

insteances increases back to the original level or is even slightly better. Kissat_sat sees an increase of performance too.

	Glucose-4.2.1		Maple		Kissat_sat		CCAnr
	.	+	.	+	.	+cf	.
#SAT	7 330	8 075	8 084	8 759	8 192	8 214	7 853
#UNSAT	187	197	215	218	207	211	0
#Solved	7 517	8 272	8 299	8 977	8 399	8 425	7 853
Avg (s)	2 555.85	1 850.58	1 867.13	1 243.66	1 760.55	1 734.61	2 215.35

Table 6: Compared with state-of-the-art solvers on FCC. The default version is marked as ".", whereas "+" stands for `+rx+rp+cf`

- By implementing all the three techniques, very large improvements are obtained for Glucose and MapleLcmDistChronoBT-DL-v2.1 for all the benchmarks. Particularly, Glucose+rx+rp+cf solves 54 additional instances than the original solver, and Maple-DL+rx+rp+cf solves 67 additional instances than its original solver for the SAT Competition 2020 benchmark (which has 400 instances). We note that Maple-DL+rx+rp+cf is a simplified and optimized version of our solver Relaxed_-LcmdCbDl_NewTech which won the gold medal of Main Track SAT category and the silver medal of the Main Track ALL category in SAT Competition 2020.

Evaluations on Benchmarks of Spectrum Repacking. We also carry out experiments on a suite of instances arising from an important real world project — the spectrum repacking project in US Federal Communication Commission (FCC). The instances of this project was available on-line.[11] (Newman et al., 2018). This benchmark contains 10 000 instances, including both satisfiable and unsatisfiable instances. We compare each base CDCL solver with its final version using our techniques, as well as the underlying local search solver CCAnr.

The results on this benchmark suite are reported in Table 6. According to the results, for each of the base CDCL solvers, the improved version with our techniques has better performance than the base solver. Particularly, the Maple-DL+rx+rp+cf solver solves the most instances (8759+218=8977), significantly better than all the other solvers.

Further Analyses on the Cooperation. We perform more analyses to study the role of local search in the hybrid solvers based on Glucose and MapleLcmDistChronoBT-DL. This experiment does not include Kissat_sat as we do not apply the relaxed CDCL framework to it and the statistics in this experiment are not applicable to Kissat_sat +cf. Some important information is provided in Table 7.

We can see that the local search solver returns a solution for some instances, and this number varies considerably with the benchmarks. A natural question is *whether the improvements come mainly from the complementation of CDCL and local search solvers that they solve different instances?* If this were true, then a simple portfolio that runs both CDCL and local search solvers would work similarly to the hybrid solvers in this work. To answer this question, we compare the instances solved by the hybrid solvers with those by the base CDCL solver and the local search solver (both the CDCL and the local search

11. `https://www.cs.ubc.ca/labs/beta/www-projects/SATFC/cacm_cnfs.tar.gz`

	Analysis for SAT				Analysis for UNSAT	
solver	#byLS	#SAT_bonus	#LS_call	LS_time(%)	#LS_call	LS_time(%)
SAT Competition 2019(400)						
Glucose+rx	9	10	33.94	11.7	22.99	6.95
Glucose+rx+rp	6	19	25.83	10.87	19.78	5.99
Glucose+rx+rp+cf	5	31	26.24	11.75	22.52	6.29
Maple+rx	14	7	12.66	2.67	12.94	1.98
Maple+rx+rp	12	16	12.73	2.91	16.79	2.13
Maple+rx+rp+cf	12	15	11.21	3.05	17.23	2.22
SAT Competition 2020(400)						
Glucose+rx	30	6	15.55	12.2	22.18	11.35
Glucose+rx+rp	21	32	13.67	11.36	12.14	10.57
Glucose+rx+rp+cf	20	31	13.26	11.37	12.65	10.32
Maple+rx	19	13	14.21	6.69	10.24	5.25
Maple+rx+rp	21	30	10.89	6.32	13.09	5.67
Maple+rx+rp+cf	23	36	10.95	6.05	14.17	5.42
SAT Competition 2021(400)						
Glucose+rx	23	7	24.32	13.8	24.9	6.13
Glucose+rx+rp	21	25	16.43	14.07	19.56	5.37
Glucose+rx+rp+cf	17	27	20.1	14.1	14.66	5.53
Maple+rx	17	8	7.47	6.09	5.62	1.69
Maple+rx+rp	17	23	12.84	5.84	6.35	1.71
Maple+rx+rp+cf	14	26	12.73	6.26	5.76	1.69

Table 7: Analyses on the impact of Local Search on the CDCL solvers. Maple is short for Maple-DL to save space, #byLS is the number of instances for which the solution is given by the local search solver, #SAT_bonus is the number of instances for which both base CDCL solver and Local Search solver fail to solve but the hybrid solver finds a satisfiable solution. #LS_call is the average number of calls on Local Search, while LS_time is the average value of the proportion of time (in percentage %) spent on local search in the whole run, and these two figures are calculated for satisfiable and unsatisfiable instances respectively.

solver are given 5000 seconds for each instance). We observe that, there is a large number of instances (denoted by #SAT_bonus) that both CDCL and local search solvers fail to solve but can be solved by the hybrid solvers. For these instances, even a virtual best solver that picks the solver with the best result for each instance would fail. For Glucose, this number reaches 31, 31, and 27 for the three benchmarks respectively, while for MapleLcmDist ChronoBT-DL, this number reaches 15, and 36, and 26 respectively. This clearly indicates that our new cooperation techniques have essential contributions to the good performance of the hybrid solvers.

We have also calculated the number of calls of the local search solver in each run. This figure usually ranges from 10 to 25 calls per run for these benchmarks. As for the run time of local search, which can be seen as the price paid for the benefit of using local search, we calculate the portion of the time spent on local search. This figure is between 6% and 20% for the satisfiable instances, and it drops significantly on unsatisfiable instances, which is usually less than 7%. This is consistent with the observations that the number of local search calls is not necessarily fewer on unsatisfiable instances, because the portion of the time on local search also depends on the total time of the hybrid solver.

On average the time for solving unsatisfiable instances is about 1.5× to 2× the time it takes to solve satisfiable instances for both Glucose+rx+rp+cf and Maple-DL+rx+rp+cf. In a nutshell, the price is acceptable and usually small for the unsatisfiable instances, which also partly explains that our techniques do not have an obvious negative impact on solving unsatisfiable instances although they incline to the satisfiable side.

7.3 Combination of Our Techniques

Finally, we also combined our techniques in a single SAT solver to be able to compare them. Even though they were developed independently, but with the same overall motivation in mind, and also differ on the technical level, the similar positive effects can be observed.

The results are given in Table 8. Overall we can see that both approaches improve the performance of Glucose and in particular on SAT problems. In more details, the (slightly simplified[12]) version of target phasing and rephasing is outperformed by the `+rx+rp+cf`, especially on the SAT 2020 and 2021 benchmarks. Interestingly, the combination of all techniques outperforms both versions.

In an attempt to better understand the phenomenon, we deactivate `+cf` from the combination for Kissat, introducing a performance regression. Understanding the difference better is left to future work.

8. Related Work

There has been interest in combining systemic search and local search for solving SAT. Indeed, it was even included as one of the challenges in Selman et al (Selman, Kautz, & McAllester, 1997). Previous attempts can be categorized into two families according to the type (DPLL/CDCL or local search) of the main body solver.

A family of hybrid solvers use a local search solver as the main body solver. UnitWalk (Hirsch & Kojevnikov, 2005) is among one the first local-search algorithms that try to include a technique mimicing propagation from CDCL, called unit clause elimination. An incomplete hybrid solver hybridGM (Balint et al., 2009) calls CDCL search around local minima with only one unsatisfied clause. Audemard et al. proposed a hybrid solver named Sathys (Audemard, Lagniez, Mazure, & Sais, 2009; Audemard et al., 2010). Each time the local search solver reaches a local minimum, a CDCL solver is launched. Some reasoning techniques or information from CDCL solvers have been used to improve local

12. Missing is in particular the scaling of the random walk at the beginning of walking phases and the better scheduling of which model is extended when walking.

solver	#SAT	#UNSAT	#Solved	PAR2
SAT Competition 2019(400)				
Glucose _4.0	115	86	201	5531.29
Glucose+rx+rp+cf	143	87	230	4915.2
Glucose _phases	142	78	220	5107.06
Glucose+all	151	71	222	5045.35
Kissat	159	96	255	4212.63
Kissat +cf	159	99	258	4157.38
Kissat_SAT	160	90	250	4255.0
Kissat_SAT +cf	163	91	254	4189.04
SAT Competition 2020(400)				
Glucose _4.0	77	93	170	6325.36
Glucose+rx+rp+cf	126	98	224	4977.58
Glucose _phases	120	89	209	5404.23
Glucose+all	142	97	239	4616.08
Kissat	129	120	249	4304.49
Kissat +cf	140	124	264	4042.65
Kissat_SAT	143	113	256	4122.68
Kissat_SAT+cf	146	113	259	4055.31
SAT Competition 2021(400)				
Glucose _4.0	96	126	222	5094.43
Glucose+rx+rp+cf	125	126	251	4312.46
Glucose _phases	107	117	224	4962.6
Glucose+all	130	124	254	4230.77
Kissat	125	151	276	3637.79
Kissat+cf	130	152	282	3562.96
Kissat_SAT	135	141	276	3681.33
Kissat_SAT+cf	138	142	280	3594.41

Table 8: Experiment results of combination techniques on benchmarks from SAT Competitions 2019-2021

search solvers. Resolution techniques were integrated to local search solvers (Cha & Iwama, 1996; Anbulagan, Pham, Slaney, & Sattar, 2005).

Recently, Lorenz and Wörz developed a hybrid solver GapSAT (Lorenz & Wörz, 2020), which used a CDCL solver as a preprocessor before running the local search solver ProbSat. The experiments showed that the learned clauses produced by the CDCL solver were useful to improve the local search solver on random instances.

The other family of hybrid solvers focuses on boosting CDCL solvers by local search, and this work belongs to this line. One simple way of hybridizing is to call local search before CDCL is run, trying to solve the instance by the local search solver alone. This gives the same benefits as a portfolio approach. Additionally, information derived during the local search, such as variable ordering, can be used in the following CDCL solver call. The

hybrid solvers Sparrow2Riss (Balint & Manthey, 2018), CCAnr+Glucose (Cai, Luo, & Su, 2014) and SGSeq (Li & Habet, 2014) belong to this family. In contrast to our approach, there is no information flow back from CDCL to the local search solver. We actually switch between local search and CDCL in regular intervals and further exchange information in both directions in an "inprocessing" fashion (Järvisalo, Heule, & Biere, 2012).

Some works use local search to find a subformula for CDCL to solve. The local search solver (Mazure et al., 1998) finds a part of the formula which is satisfiable, which helps to divide the formula into two parts for the DPLL solver to allow the SAT solver to focus on the unsatisfiable part. In Hinotos (Letombe & Marques-Silva, 2008), a local search identifies a subset of clauses to be passed to a CDCL solver in an incremental way.

Although these previous attempts have been made to combine the strength of CDCL and local search, they did not lead to hybrid solvers essentially better than CDCL solvers on application instances. This work, for the first time, meets the standard of the challenge "create a new algorithm that outperforms the best previous examples of both approaches" (Selman et al., 1997) on standard application benchmarks from SAT Competitions.

9. Conclusion

This work takes a large step towards deep cooperation of CDCL and local search by presenting four techniques for effectively using local search to improve CDCL solvers. The first idea extends promising branches from being pruned by targeting phases of large consistent assignments. The second idea relaxes CDCL by extending such promising branches in order to let local search find a satisfying assignment nearby. The third idea is to utilize assignments minimizing the number of unsatisfiable clauses found during local search and use them as saved phases in the phase selection heuristic. Finally, we proposed to enhance the branching strategy of CDCL solvers by considering the conflict frequency of variables in the local search process. These techniques significantly improve the performance of state-of-the-art CDCL solvers on real-world application benchmarks. As generic techniques they are expected to improve other CDCL solvers too.

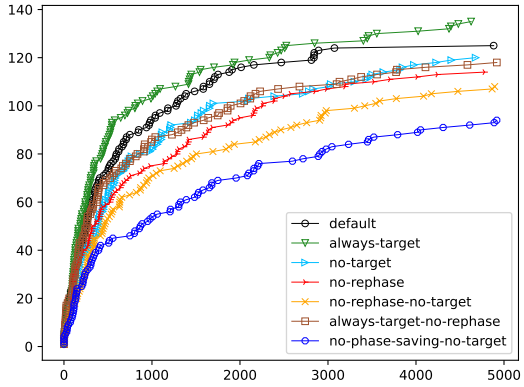
This is the first time that the combination of stochastic search and systematic search techniques leads to substantial improvement of the state of the art on application benchmarks, compared to using only one technique alone, thus positively resolving Challenge 7 of the "*Ten Challenges in Propositional Reasoning and Search*" (Selman et al., 1997).

Acknowledgement

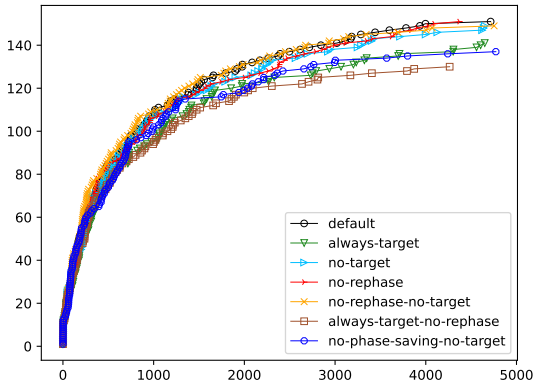
This work is supported by NSFC Grant 62122078, Beijing Academy of Artificial Intelligence (BAAI), the Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. Sibylle Möhle and the anonymous reviewers suggested many textual improvements on previous versions of this work.

Appendix

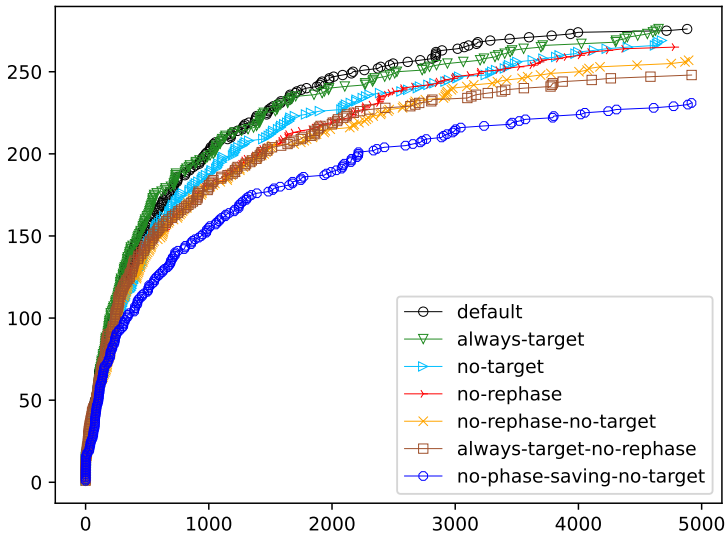
In this section, two classes of CDF plots are listed for component effectiveness analysis. The first three figures evaluate the directly CDCL guided exploring methods. Figures 5-7 show the results of different strategy combinations on the top of Kissat, CaDiCaL and Glucose respectively. The last three figures compare the effectiveness of the local search related strategies, which are implemented based on Glucose and Maple. Figures 8-10 show the results of the different benchmarks form SAT RACE 2019, SAT Competition 2020 and SAT Competition 2021.



(a) Satisfiable instances only

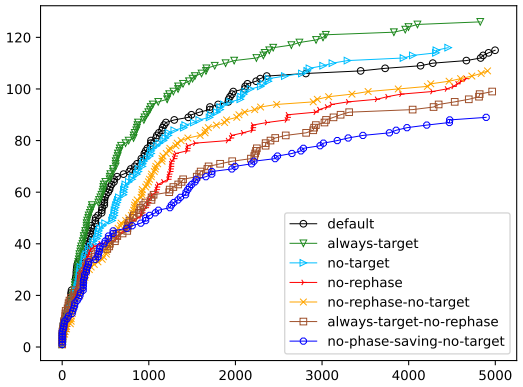


(b) Unsatisfiable instances only

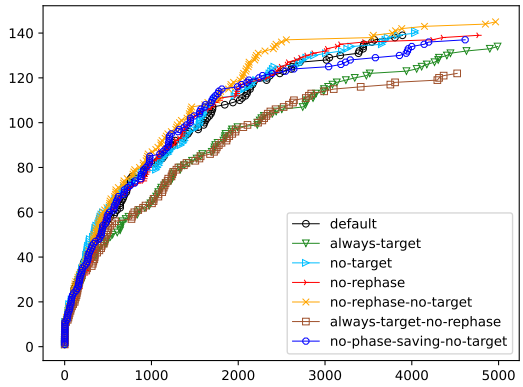


(c) All instances

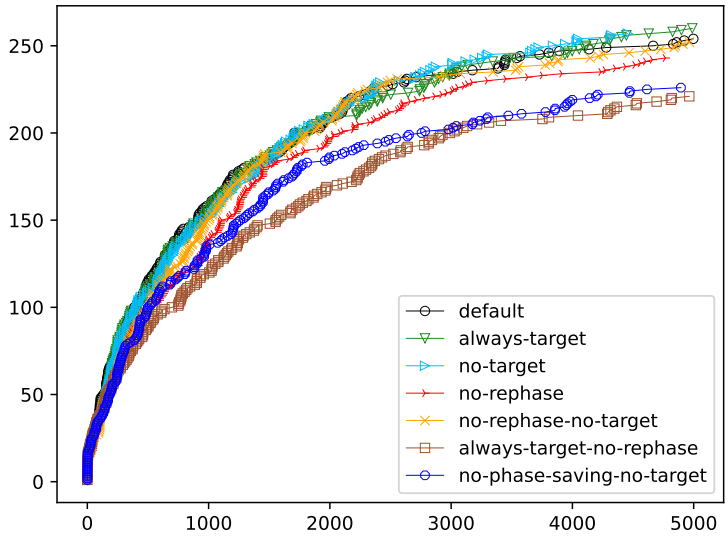
Figure 5: CDF for the solver Kissat on benchmarks from the SAT Competition 2021



(a) Satisfiable instances only

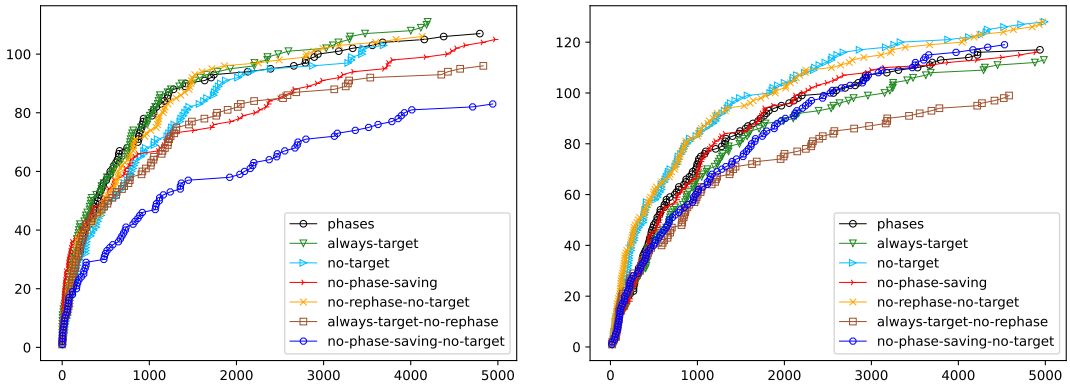


(b) Unsatisfiable instances only



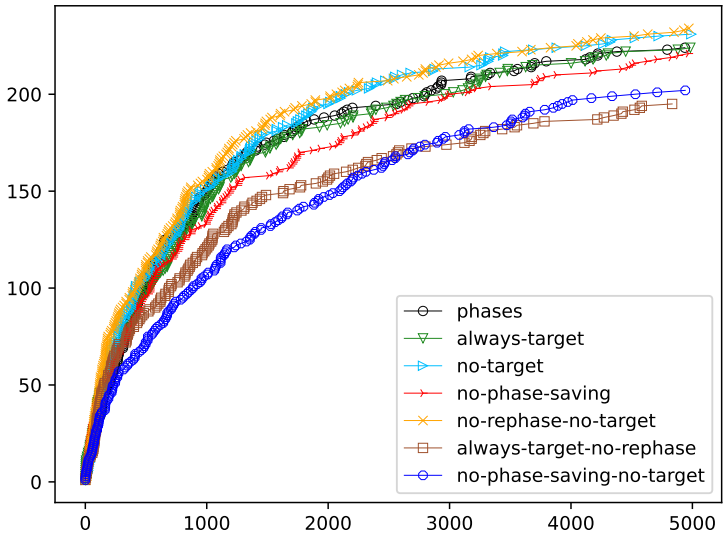
(c) All instances

Figure 6: CDF for the solver CaDiCaL on benchmarks from the SAT Competition 2021



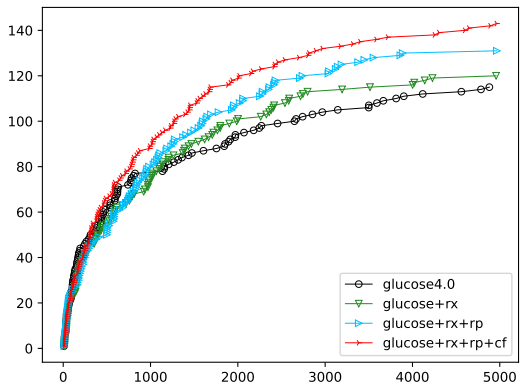
(a) Satisfiable instances only

(b) Unsatisfiable instances only

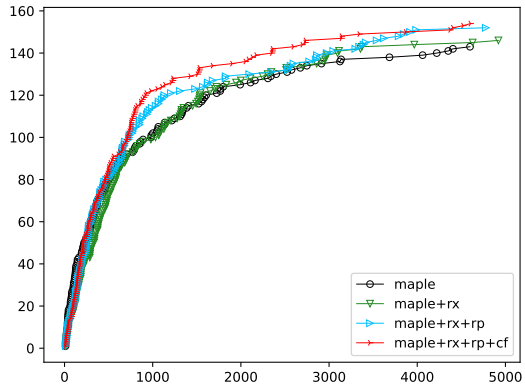


(c) All instances

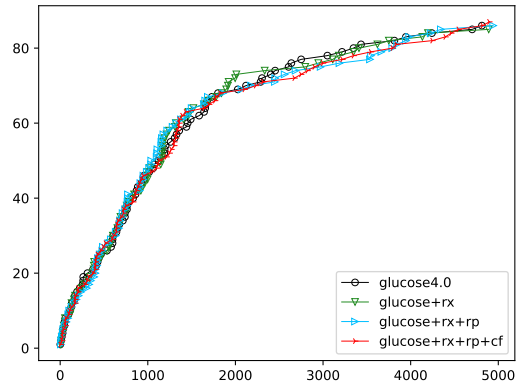
Figure 7: CDF for the solver Glucose on benchmarks from the SAT Competition 2021



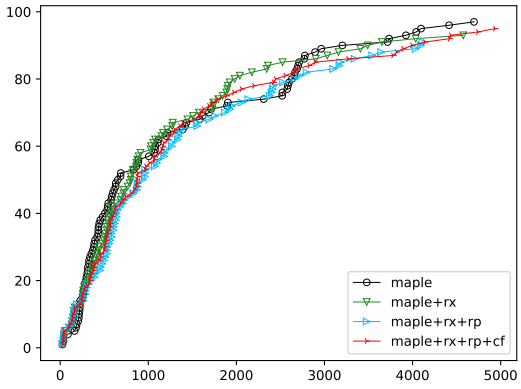
(a) Satisfiable instances only



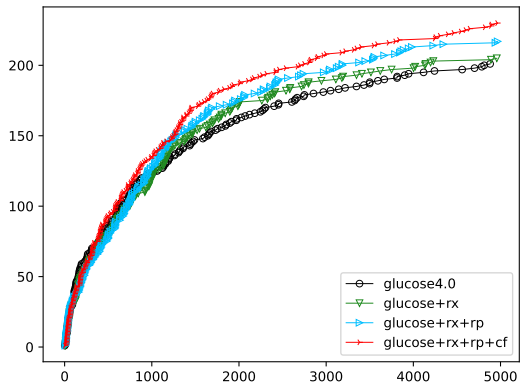
(b) Satisfiable instances only



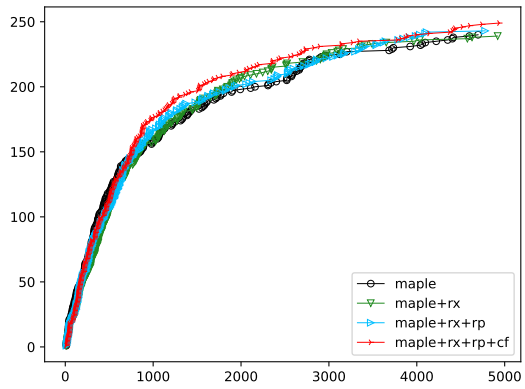
(c) Unsatisfiable instances only



(d) Unsatisfiable instances only

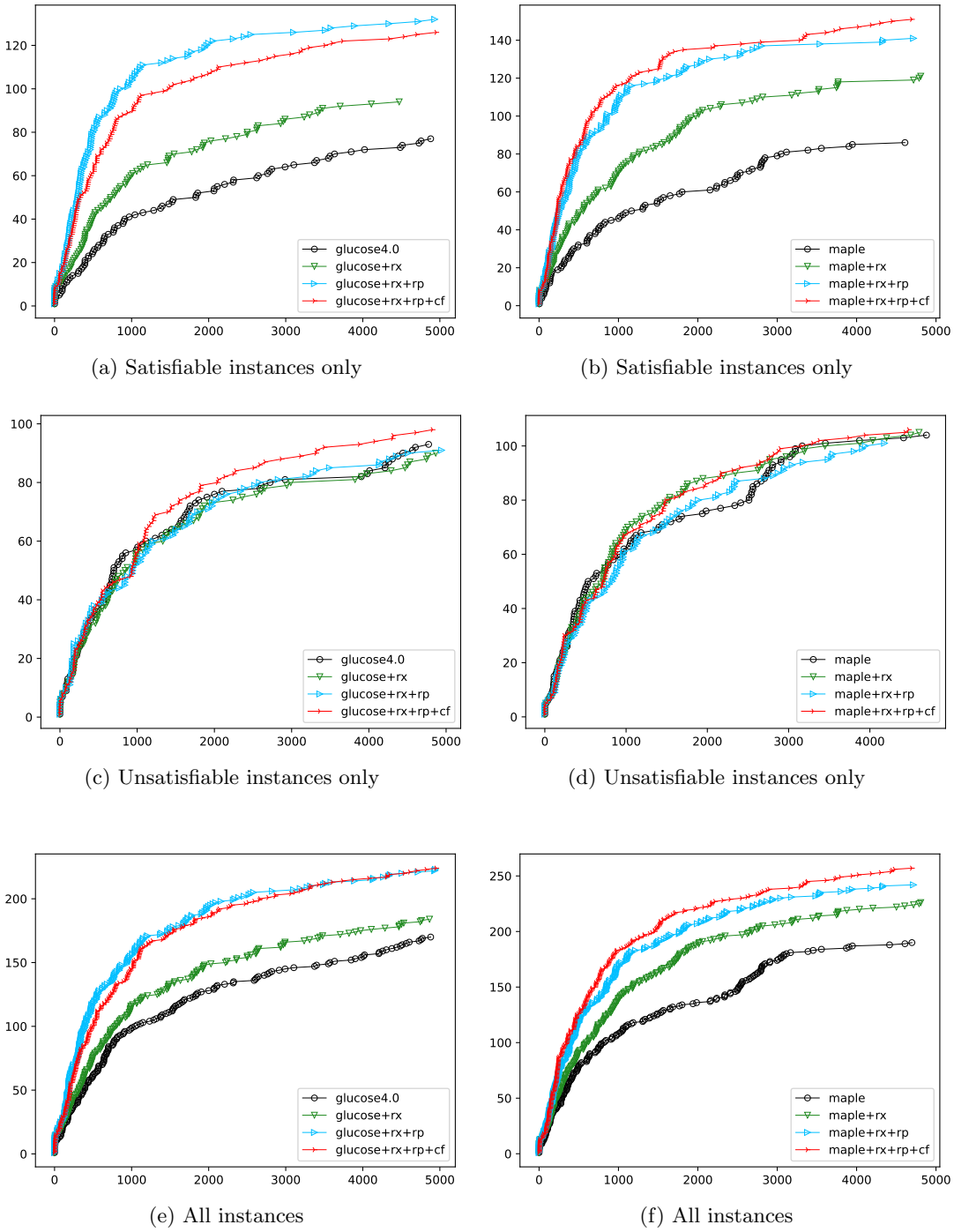


(e) All instances



(f) All instances

Figure 8: CDF for the solvers Glucose (left) and Maple (right) about the relaxed CDCL, local search rephasing and conflict frequency on benchmarks from the SAT Race 2019



(a) Satisfiable instances only

(b) Satisfiable instances only

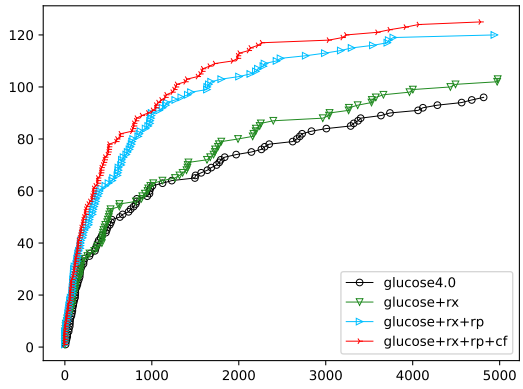
(c) Unsatisfiable instances only

(d) Unsatisfiable instances only

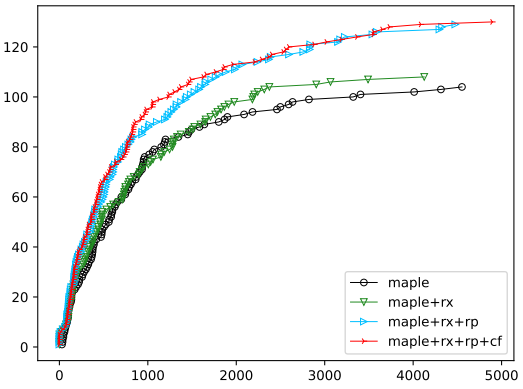
(e) All instances

(f) All instances

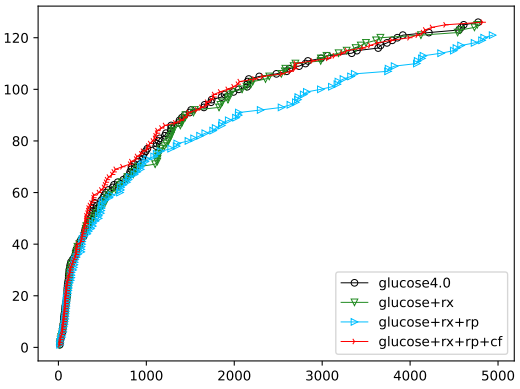
Figure 9: CDF for the solvers Glucose (left) and Maple (right) about the relaxed CDCL, local search rephasing and conflict frequency on benchmarks from the SAT Competition 2020



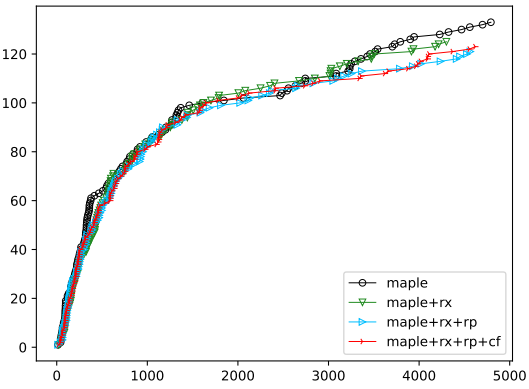
(a) Satisfiable instances only



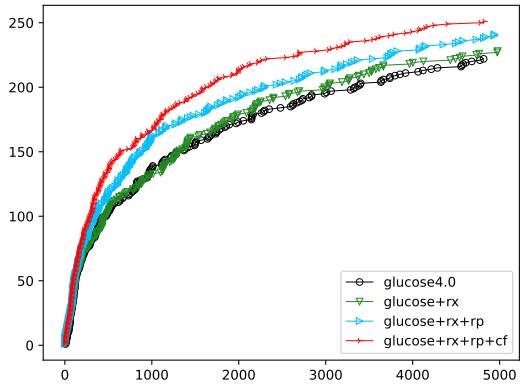
(b) Satisfiable instances only



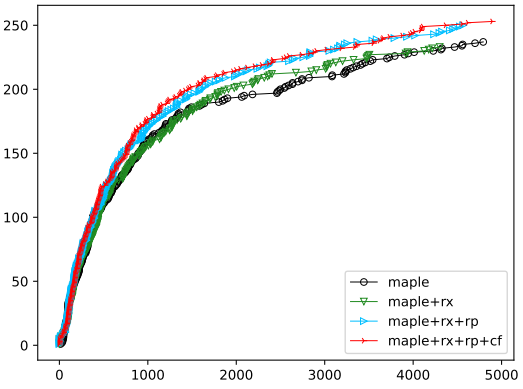
(c) Unsatisfiable instances only



(d) Unsatisfiable instances only



(e) All instances



(f) All instances

Figure 10: CDF for the solvers Glucose (left) and Maple (right) about the relaxed CDCL, local search rephasing and conflict frequency on benchmarks from the SAT Competition 2021

References

Anbulagan, Pham, D. N., Slaney, J. K., & Sattar, A. (2005). Old resolution meets modern SLS. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pp. 354–359. AAAI Press / The MIT Press.

Audemard, G., Lagniez, J., Mazure, B., & Sais, L. (2009). Integrating conflict-driven clause learning to local search. In Deville, Y., & Solnon, C. (Eds.), *Proceedings 6th International Workshop on Local Search Techniques in Constraint Satisfaction, LSCS 2009, Lisbon, Portugal, 20 September 2009*, Vol. 5 of *EPTCS*, pp. 55–68.

Audemard, G., Lagniez, J., Mazure, B., & Sais, L. (2010). Boosting local search thanks to CDCL. In Fermüller, C. G., & Voronkov, A. (Eds.), *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, Vol. 6397 of *Lecture Notes in Computer Science*, pp. 474–488. Springer.

Audemard, G., & Simon, L. (2009). 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.

Audemard, G., & Simon, L. (2012a). Glucose 2.1: Aggressive—but reactive—clause database management, dynamic restarts. In *Workshop on the Pragmatics of SAT 2012*.

Audemard, G., & Simon, L. (2012b). 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*, Vol. 7514 of *Lecture Notes in Computer Science*, pp. 118–126. Springer.

Balint, A. (2014). *Engineering stochastic local search for the satisfiability problem.* Ph.D. thesis, University of Ulm.

Balint, A., Belov, A., Järvisalo, M., & Sinz, C. (2015). Overview and analysis of the SAT Challenge 2012 solver competition. *Artificial Intelligence*, *223*, 120–155.

Balint, A., Henn, M., & Gableske, O. (2009). A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem. 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*, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 284–297. Springer.

Balint, A., & Manthey, N. (2018). SparrowToRiss 2018. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, Vol. B-2018-1 of *Department of Computer Science Report Series B*, pp. 38–39, Finland. Department of Computer Science, University of Helsinki.

Balint, A., & Schöning, U. (2012). Choosing probability distributions for stochastic local search and the role of make versus break. In *Proceedings of SAT 2012*, pp. 16–29.

Barnett, L. A., & Biere, A. (2021). Non-clausal redundancy properties. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, Vol. 12699, pp. 252–272.

Barnett, L. A., Cerna, D. M., & Biere, A. (2020). Covered clauses are not propagation redundant. In Peltier, N., & Sofronie-Stokkermans, V. (Eds.), *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, Vol. 12166 of *Lecture Notes in Computer Science*, pp. 32–47. Springer.

Barrett, C., Sebastiani, R., Seshia, S. A., & Tinelli, C. (2021). Satisfiability modulo theories. In Biere, A., Heule, M. J. H., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability* (Second edition)., Vol. 336, pp. 1267–1369. IOS Press.

Biere, A. (2010). Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. In *FMV Report Series Technical Report*, Vol. 10.

Biere, A. (2014). Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. 2014, p. 65.

Biere, A. (2017a). CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Balyo, T., Heule, M. J. H., & Järvisalo, M. (Eds.), *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, Vol. B-2017-1 of *Department of Computer Science Series of Publications B*, pp. 14–15. University of Helsinki.

Biere, A. (2017b). Deep Bound Hardware Model Checking Instances, Quadratic Propagation Benchmarks and Reencoded Factorization Problems Submitted to the SAT Competition 2017. In Balyo, T., Heule, M. J. H., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2017 – Solver and Benchmark Descriptions*, Vol. B-2017-1 of *Department of Computer Science Series of Publications B*, pp. 40–41. University of Helsinki.

Biere, A. (2018). CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, Vol. B-2018-1, pp. 13–14.

Biere, A. (2019). CaDiCaL at the SAT Race 2019. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, Vol. B-2019-1 of *Department of Computer Science Series of Publications B*, pp. 8–9. University of Helsinki.

Biere, A., Fazekas, K., Fleury, M., & Heisinger, M. (2020). CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*, pp. 51–53.

Biere, A., & Fleury, M. (2020). Chasing target phases. In *Workshop on the Pragmatics of SAT 2020*.

Biere, A., & Fröhlich, A. (2015). 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, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 405–422. Springer.

Biere, A., & Fröhlich, A. (2015). 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*, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 405–422. EasyChair.

Biere, A., Järvisalo, M., & Kiesl, B. (2021). Preprocessing SAT solving (second edition). In Biere, A., Heule, M. J. H., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 391–435. IOS Press.

Biere, A., Järvisalo, M., Le Berre, D., Meel, K. S., & Mengel, S. (2020). *The SAT Practitioner's Manifesto*.

Biere, A., & Kröning, D. (2018). SAT-based model checking. In *Handbook of Model Checking*, pp. 277–303. Springer.

Cai, S. (2015). *Novel Local Search Methods for Satisfiability*. Ph.D. thesis, Griffith University.

Cai, S., Luo, C., & Su, K. (2014). CCAnr+glucose in SAT Competition 2014. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, No. 2 in Department of Computer Science Series of Publications B, p. 17.

Cai, S., Luo, C., & Su, K. (2015). CCAnr: A configuration checking based local search solver for non-random satisfiability. In Heule, M., & Weaver, S. A. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 1–8. Springer.

Cai, S., Luo, C., & Zhang, H. (2017). From decimation to local search and back: A new approach to MaxSAT. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 571–577. ijcai.org.

Cai, S., Luo, C., Zhang, X., & Zhang, J. (2021). Improving local search for structured SAT formulas via unit propagation based construct and cut initialization (short paper). In Michel, L. D. (Ed.), *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, Vol. 210 of *LIPIcs*, pp. 5:1–5:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Cai, S., & Zhang, X. (2018). ReasonLS. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, Vol. B-2018-1 of *Department of Computer Science Series of Publications B*, pp. 52–53, Finland. Department of Computer Science, University of Helsinki.

Cai, S., & Zhang, X. (2019). Four relaxed CDCL solvers. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*,

Vol. B-2019-1 of *Department of Computer Science Report Series B*, p. 35, Finland. Department of Computer Science, University of Helsinki.

Cai, S., & Zhang, X. (2021). Deep cooperation of CDCL and local search for SAT. In Li, C.-M., & Manyà, F. (Eds.), *Theory and Applications of Satisfiability Testing – SAT 2021*, pp. 64–81, Cham. Springer International Publishing.

Cha, B., & Iwama, K. (1996). Adding new clauses for faster local search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1*, pp. 332–337. AAAI Press / The MIT Press.

Eén, N., & Sörensson, N. (2003). 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*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer.

Fang, H., & Ruml, W. (2004). Complete local search for propositional satisfiability. In McGuinness, D. L., & Ferguson, G. (Eds.), *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pp. 161–166. AAAI Press / The MIT Press.

Fazekas, K., Biere, A., & Scholl, C. (2019). Incremental inprocessing in SAT solving. 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*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 136–154. Springer.

Gomes, C. P., Selman, B., Crato, N., & Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reasoning*, *24*(1/2), 67–100.

Habet, D., Li, C. M., Devendeville, L., & Vasquez, M. (2002). A hybrid approach for SAT. In Hentenryck, P. V. (Ed.), *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, Vol. 2470 of *Lecture Notes in Computer Science*, pp. 172–184. Springer.

Hamadi, Y., Jabbour, S., & Sais, L. (2009). 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.

Han, H., & Somenzi, F. (2009). 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*, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 209–222. Springer.

Heule, M. J. H., Järvisalo, M., & Biere, A. (2013). 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, Vol. 7874 of *Lecture Notes in Computer Science*, pp. 77–93. Springer.

Heule, M. J. H., Kiesl, B., & Biere, A. (2020). Strong extension-free proof systems. *J. Autom. Reasoning*, *64*, 533–554.

Heule, M. J. H., Kullmann, O., & Marek, V. W. (2016). Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. 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*, Vol. 9710 of *Lecture Notes in Computer Science*, pp. 228–245. Springer.

Hirsch, E. A., & Kojevnikov, A. (2005). UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Ann. Math. Artif. Intell.*, *43*(1), 91–111.

Hoos, H. H., & Stützle, T. (2004). *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann.

Järvisalo, M., & Biere, A. (2010). Reconstructing solutions after blocked clause elimination. In Strichman, O., & Szeider, S. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, Vol. 6175 of *Lecture Notes in Computer Science*, pp. 340–345. Springer.

Järvisalo, M., Heule, M., & Biere, A. (2012). Inprocessing rules. In Gramlich, B., Miller, D., & Sattler, U. (Eds.), *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, Vol. 7364 of *Lecture Notes in Computer Science*, pp. 355–370. Springer.

Jeroslow, R. G., & Wang, J. (1990). Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, *1*, 167–187.

Kautz, H. A., & Selman, B. (1992). Planning as satisfiability. In Neumann, B. (Ed.), *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pp. 359–363. John Wiley and Sons.

Kiesl, B., Heule, M. J. H., & Biere, A. (2019). 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*, Vol. 11781 of *Lecture Notes in Computer Science*, pp. 48–64. Springer.

Knuth, D. E. (2006). *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional.

Kochemazov, S., Zaikin, O., Kondratiev, V., & Semenov, A. (2019). MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, Vol. B-2019-1 of *Department of Computer Science Report Series B*, pp. 24–24, Finland. Department of Computer Science, University of Helsinki.

Letombe, F., & Marques-Silva, J. (2008). Improvements to hybrid incremental SAT algorithms. In Büning, H. K., & Zhao, X. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, Vol. 4996 of *Lecture Notes in Computer Science*, pp. 168–181. Springer.

Li, C. M., & Habet, D. (2014). Description of RSeq2014. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. 2014, p. 72.

Li, C. M., & Li, Y. (2012). Satisfying versus falsifying in local search for satisfiability - (poster presentation). In Cimatti, A., & Sebastiani, R. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Vol. 7317 of *Lecture Notes in Computer Science*, pp. 477–478. Springer.

Liang, J. H., Ganesh, V., Poupart, P., & Czarnecki, K. (2016). 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*, Vol. 9710 of *Lecture Notes in Computer Science*, pp. 123–140. Springer.

Lorenz, J., & Wörz, F. (2020). On the effect of learned clauses on stochastic local search. In Pulina, L., & Seidl, M. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, Vol. 12178 of *Lecture Notes in Computer Science*, pp. 89–106. Springer.

Luby, M., Sinclair, A., & Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, *47*(4), 173–180.

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

Marques Silva, J. P., Lynce, I., & Malik, S. (2021). Conflict-drive clause learning SAT solvers. In *Handbook of Satisfiability* (Second edition)., Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 133–182. IOS Press.

Mazure, B., Sais, L., & Grégoire, É. (1998). Boosting complete techniques thanks to local search methods. *Ann. Math. Artif. Intell.*, *22*(3-4), 319–331.

Möhle, S., & Biere, A. (2019). Backing backtracking. In Janota, M., & Lynce, I. (Eds.), *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference,*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 250–266. Springer.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001*, pp. 530–535.

Nadel, A., & Ryvchin, V. (2018). 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*, Vol. 10929 of *Lecture Notes in Computer Science*, pp. 111–121. Springer.

Newman, N., Fréchette, A., & Leyton-Brown, K. (2018). Deep optimization for spectrum repacking. *Commun. ACM*, *61*(1), 97–104.

Oh, C. (2015). Between SAT and UNSAT: the fundamental difference in CDCL SAT. In Heule, M., & Weaver, S. A. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, Vol. 9340 of *Lecture Notes in Computer Science*, pp. 307–323. Springer.

Pipatsrisawat, K., & Darwiche, A. (2007). 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*, Vol. 4501 of *Lecture Notes in Computer Science*, pp. 294–299. Springer.

Prasad, M. R., Biere, A., & Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *Int. J. Softw. Tools Technol. Transf.*, *7*(2), 156–173.

Ramos, A., van der Tak, P., & Heule, M. J. H. (2011). 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*, Vol. 6695 of *Lecture Notes in Computer Science*, pp. 216–229. Springer.

Ryan, L. (2004). Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University.

Ryvchin, V., & Strichman, O. (2008). Local restarts. In Büning, H. K., & Zhao, X. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, Vol. 4996 of *Lecture Notes in Computer Science*, pp. 271–276. Springer.

Selman, B., Kautz, H. A., & McAllester, D. A. (1997). Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pp. 50–54. Morgan Kaufmann.

Silva, J. P. M., & Sakallah, K. A. (2000). Boolean satisfiability in electronic design automation. In Micheli, G. D. (Ed.), *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000*, pp. 675–680. ACM.

Soos, M. (2013). Strangenight. In Balint, Adrian Belov, A., Heule, M. J. H., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, Vol. B-2013-1 of *Department of Computer Science Series of Publications B*, p. 1. University of Helsinki.

Soos, M., & Biere, A. (2019). CryptoMiniSat 5.6 with YalSAT at the SAT Race 2019. In Heule, M. J. H., Järvisalo, M., & Suda, M. (Eds.), *Proceedings of SAT Race 2019 – Solver and Benchmark Descriptions*, Vol. B-2019-1 of *Department of Computer Science Series of Publications B*, pp. 14–15. University of Helsinki.

Soos, M., Nohl, K., & Castelluccia, C. (2009). 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, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 244–257. Springer.

Vizel, Y., Weissenbacher, G., & Malik, S. (2015). Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE*, *103*(11), 2021–2035.

Zhang, W., Sun, Z., Zhu, Q., Li, G., Cai, S., Xiong, Y., & Zhang, L. (2020). NLocalSAT: Boosting local search with solution prediction. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 1177–1183. ijcai.org.