Prioritised Unit Propagation by Partitioning the Watch Lists

Benjamin Kaiser*, Robert Clausecker and Michael Mavroskoufis

Zuse Institute Berlin, Germany

Abstract

Conflict Driven Clause Learning (CDCL) SAT solvers spend most of their execution time iterating through clauses in unit propagation. Efficient implementations of unit propagation mostly rely on the *two watched literals (TWL) scheme*, a specialised data structure watching two literals per clause to prevent as many clause lookups as possible. In this paper, we present *Priority Propagation (PriPro)*—a minimally invasive method to adapt unit propagation in existing SAT solvers. With PriPro the traditional TWL scheme is partitioned to allow for rearranging the order in which clauses are examined. This is used to achieve a prioritisation of certain clauses. Using PriPro in combination with a dynamic heuristic to prioritise resolvents from recent conflicts, the effectiveness of unit propagation can be increased. In the state-of-the-art CDCL SAT solver CaDiCaL modified to use PriPro, we obtained a 5–10 % speedup on the SAT Competition 2021 benchmark set in a fair comparison with the unmodified CaDiCaL.

Keywords

SAT, Conflict Driven Clause Learning, Boolean Constraint Propagation, Unit Propagation, Two Watched Literals

1. Introduction

SAT solvers, solvers for the Boolean satisfiability problem, spend a large portion of their runtime on propagation. Previous approaches to accelerate propagation include code optimisations with specialised data structures as well as introducing deletion policies of clauses suspected to be useless.

Following its debut in the special innovation award winning solver CaDiCaL_PriPro at SAT Competition 2021, this paper presents motivations for introducing PriPro—a simple adaption of the traditional implementation of unit propagation using *two* TWL data structures instead of one. In addition to previous approaches to improve unit propagation, PriPro aims at avoiding less useful clauses without deleting them, but deliberately changing the solvers path to focus on prioritised clauses. We propagate prioritised clauses first, only propagating other clauses once the prioritised clauses are exhausted. We describe the background needed in § 2 and discuss related attempts to improve unit propagation in § 4.

14th International Workshop on Pragmatics of SAT (PoS 2023)

*Corresponding author.

kaiser@zib.de (B. Kaiser); clausecker@zib.de (R. Clausecker); mavroskoufis@zib.de (M. Mavroskoufis)

https://www.zib.de/members/kaiser (B. Kaiser); https://www.zib.de/members/clausecker (R. Clausecker); https://www.zib.de/members/mavroskoufis (M. Mavroskoufis)

0000-0003-2929-4439 (B. Kaiser)

Our main contributions are as follows:

- We introduce a novel scheme to prioritise a part of the clause database during propagation. Prioritisation of clauses is chosen/adjusted dynamically during runtime of the solver. We name this approach PriPro and describe it in § 3.
- The approach mainly introduces a second TWL scheme for prioritised clauses and dynamically up-/ and downgrades clauses to/from the additional, prioritised TWL scheme under certain conditions. We use a simple parameterised heuristic based on recent resolvents to decide which clauses should be prioritised (see § 3.3).
- We study the performance of PriPro in comparison to the baseline solver CaDiCaL and illustrate its impact on the solver's performance on the problem set of the SAT Competition 2021 containing 400 problems in terms of number of solved instances and runtime (see § 5.2, 5.3, and Tables 1, 2).
- The evaluation shows that PriPro is able to solve up to 11 more SAT and up to 6 more UNSAT instances compared to the baseline solver CaDiCaL, despite the simplicity of the used prioritisation heuristics. PriPro consistently solves UNSAT instances significantly faster than the baseline solver (see § 5).
- We confirm Jingchao Chen's hypothesis [1] that learning shorter conflict clauses by rearranging the order of propagation is possible (see § 4.3, 5.2.1).
- We observe particularly good results with a *Literals Blocks Distance (LBD)* ≤ 6 as an upgrade condition and a scheduled downgrade interval $\geq 15\,000$ conflicts leading to solving ≈ 12 instances more than the baseline on average (see § 5).

2. SAT Solving Background

A *Boolean satisfiability (SAT) problem* [2] is given by a propositional formula in *conjunctive normal form (CNF)* [3]

$$F = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{k_i} l_{ij} \right) = \big((l_{11} \vee \ldots \vee l_{1k_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mk_m}) \big),$$

whose disjunctions $\bigvee_{j=1}^{k_i} l_{ij}$ are called *clauses* consisting of *literals*

$$l_{ij} \in \{x_1, \neg x_1,\ x_2, \neg x_2,\ \ldots,\ x_n, \neg x_n\}$$

for some Boolean variables $x_1,\ x_2,\ \ldots,\ x_n$.

To solve a SAT problem, an assignment of all variables with truth values from $\{\textbf{true}, \textbf{false}\}$ needs to be found such that one literal of each clause is assigned **true**. This is called a *satisfying assignment*. The problem of finding such a solution is NP-hard [4]. Similarly, constructing a proof that no such solution exists is co-NP-hard. A SAT solver either finds such a satisfying assignment or produces a proof that no such assignment exists. We call instances of the SAT problem that have a satisfying assignment SAT instances, the others, UNSAT instances.

Many problems of high interest can be translated to SAT and effectively solved using modern SAT solvers such as problems arising in circuit design [5], automated planning and scheduling [6], automotive software verification [7, 8] or automated theorem proving [9, 10]. This makes SAT solving an important topic of current research.

2.1. State-of-the-Art Approaches to SAT

SAT instances derived from real world problems have a structure distinct from randomly generated instances [11]. Nevertheless, the immense size of the problems makes it very difficult or even seemingly impossible to see through and exploit this structure [11, 12]. Modern SAT solvers accomplish this using *Conflict Driven Clause Learning (CDCL)* [13, 14], an algorithm relying on *unit propagation*, a special case of *Boolean Constraint Propagation (BCP)* [15, 16].

A CDCL solver successively chooses variables and assigns them a truth value. Typically, such a *decision* is made by first choosing a *decision variable* according to some *decision heuristic* [17, 18] and afterwards choosing the truth value according to the variable's current *polarity*. The variables' polarities are stored in *phases*, which resemble known and constantly updated information about promising assignments of the variables [19]. Switching between different phases is called *rephasing* [20, 21, 22]. To prevent CDCL solvers from getting stuck in futile parts of the search space, often *restarts* are performed [23], i.e. all previous decisions are discarded.

After each decision, BCP checks for clauses that have all of their literals assigned **false** (a *conflicting clause*). In such case, the current partial assignment of the variables is said to be *conflicting*. If all but one literal of a clause have been assigned **false**, the final literal has to be assigned **true** to prevent a conflict from occurring. Such assignments are called *unit implications* and may in turn result in further unit implications or conflicting clauses. When a conflict arises, CDCL analyses it and learns a new *conflict clause* [13] that is added to the set of clauses representing the problem, i.e. the *clause database*. Conflict clauses prevent conflicts from reappearing. The clauses used to form the new conflict clause are called *resolvents* as conflict analysis is a specialised application of propositional logic resolution. *Learned Clause Minimisation (LCM)* [24] may be used to remove redundant literals from the conflict clause, prior to adding it to the problem.

In order to restore a non-conflicting assignment during the solving process, some decisions are reversed in a process called *backtracking* [25]. During backtracking, some recent decision that lead to the conflict can be (and typically is) replaced by a unit implication resulting from the newly learned conflict clause. Traditionally, backtracking is performed by *backjumping* resp. *non-chronological backtracking*, which reverses (possibly) several decisions at once. More recently, only reversing the most recent decision, i.e. *chronological backtracking* [26], became widely used, but its implementation and theory is more involved.

2.2. Deleting Useless Clauses

As the CDCL algorithm progresses, growth of the clause database adversely affects propagation speed (and may exhaust available memory). Therefore, SAT solvers commonly implement *clause deletion policies*, reducing the size of the clause database every once in a while [27]. This concept was already implemented in various ways in the first CDCL solvers [25]. But many clause deletion heuristics evolved and were discussed later [28]. Deciding which clauses aid the solving process best remains challenging as the wide variety of different clause deletion heuristics indicates. Such approaches use heuristics based on *clause activity* and *Literals Blocks Distance (LBD)* [29] and include clause grouping mechanisms such as the *core tier* of almost never deleted clauses and the *mid tier* of clauses that are spared from deletion for some time [30, 31].

Later, it was discovered that deleting clauses not only speeds up unit propagation, but may be vital for the CDCL algorithm to find shorter proofs. In UNSAT instances, up to 50 % of the learned clauses appear to be not required to complete the found proof [32]. Moreover, a more aggressive clause deletion heuristic based on the LBD of clauses leads to a solver whose proofs require even fewer clauses [32].

2.3. Avoiding Clause Lookups

The speed of unit propagation can be improved by ignoring clauses that can neither lead to unit propagation nor conflicts. When searching for conflicts and implications in the current state of the problem, it is a simple optimisation to only iterate through clauses containing newly falsified literals. With *occurrence lists* one can keep track of the clauses in which a literal occurs. When searching for conflicts, this mechanism can be optimised using *watch lists* instead of full occurrence lists. Each clause is *watched* through one of its literals by saving a *watch* referencing the clause in the literal's watch list. Only if this literal is assigned **false**, we need to check for conflicts in the clause. If the clause is found not to conflict, it is moved to the watch list of one of its literals currently assigned **true** or unassigned. However, by iterating over watch lists in this *one watched literal (OWL) scheme*, it is not possible to detect all unit implications.

To additionally support searching for unit implications, one can adapt OWL to watching two distinct literals per clause. Only if one of those two literals is assigned **false**, we need to check for a unit implication or conflict in this clause. Conceptually, such a *two watched literals (TWL) scheme* [33, 34] resembles OWL, but its implementation is more complex. When chronological backtracking is used, the situation is further complicated by *out-of-order literals* and *missed lower implications* [26, 35]. However, the implementation of PriPro is orthogonal to chronological backtracking and a discussion of chronological backtracking is beyond the scope of this paper.

TWL can be improved by augmenting each watch with another literal of the referenced clause, called the *blocking literal* [36, 37, 38]. If the blocking literal is assigned **true**, the referenced clause neither can lead to an implication nor a conflict. This way, unnecessary clause dereferences can potentially be avoided and—in such case—the watch does not need to be moved to another literal's watch list. The majority of state-of-the-art CDCL SAT solvers use such a TWL scheme with blocking literals [39, 40, 28, 41]. Even further optimisations of the TWL scheme are possible by moving the watches in a circular manner if no conflict or unit implication is found [42]. Recently, new research towards watcher placement emerged, e. g. so-called *stable watches* [43].

The use of existing techniques to avoid clause lookups is largely based on highly-specialised data structures. While not explicitly intended, all these techniques change the order in which implications are found, leading to different conflicts being found and implication graphs constructed. So, the solver takes a different search path and learns different clauses during conflict analysis. Some related techniques with similarly small changes to the propagation order, in contrast, deliberately cause changes on the solver's path through the search space. We discuss them separately in § 4, where we also discuss their role as a predecessor to the algorithmic idea of PriPro presented in § 3.

3. Priority Propagation

Priority Propagation (PriPro) is an adaptation of unit propagation that reduces the number of clauses considered during propagation without directly deleting the remaining clauses. Other techniques for faster propagation typically avoid looking at clauses either by clause deletion or by augmenting the solver's data structures without the intent to significantly impact the search path.

In contrast, PriPro aims at preventing direct clause deletion and instead focusing the propagation to a subset of prioritised clauses. Thereby, it deliberately and significantly impacts the search path. We thus would like to determine by some heuristic which clauses are likely to be relevant and only consider remaining clauses once further propagations from the prioritised clauses cannot be obtained. Potentially, this permits the solver to focus on a set of clauses that appear to be useful in the current situation without disregarding other clauses that our heuristic has missed. We discuss the algorithm of unit propagation with PriPro and its implementation to typical state-of-the-art SAT solvers in the following.

3.1. Prioritised Clauses and Prioritised Watches

With PriPro, we partition the clauses into two distinct groups. One consists of *prioritised* clauses, which are to be propagated with high priority, and the other of *regular* clauses to be propagated with regular priority.

As in the traditional approach to the TWL scheme for fast unit propagation, with PriPro, each clause is watched through exactly two of its literals and watch lists store which clauses are watched. For each literal `lit`, its watch list is a dynamically sized vector `wtab[lit]` that holds all watches corresponding to clauses watched with respect to `lit`. However, to hold watches of prioritised clauses, for each literal `lit` a second watch list `pripro_wtab[lit]` is introduced. We call this additional watch list the *prioritised watch list* and the prioritised watch lists form the *prioritised TWL scheme*. Effectively, with PriPro the original TWL scheme is partitioned into two by moving the watches of prioritised clauses from the original resp. *regular TWL scheme* containing *regular watches* to the prioritised TWL scheme containing *prioritised watches*.

Whenever we want to prioritise a regular clause `c`, we move both watches corresponding to `c` from the regular TWL scheme to the prioritised TWL scheme. I. e., if `w` is one of those two watches of clause `c`, and `w` corresponds to literal `lit`, we move `w` from the regular watch list `wtab[lit]` of `lit` to the end of its prioritised watch list `pripro_wtab[lit]`. We refer to this as *upgrading* the clause `c`. We *downgrade* clauses analogously with the two TWL schemes swapped. When upgrading or downgrading a clause, one might preserve the blocking literal (or any other information stored in the watches such as the clause's size). In CaDiCaL_PriPro a new blocking literal is chosen as if the clause was newly learned.

To implement PriPro in a solver that uses preprocessing and/or inprocessing techniques that depend on the original TWL scheme, it is often sufficient to downgrade all clauses before such a technique is applied, without any changes to its implementation. Downgrading clauses may also be necessary before clause database reductions. In PriPro, one bit in the clause header may be used to signal whether a clause is currently prioritised or not. This allows for a temporary removal and later attachment of the clause from/to the correct TWL scheme. In CaDiCaL,

such a temporary detachment of a clause from the TWL scheme is used during analysis when chronological backtracking is utilised. No other adjustment is necessary to combine PriPro with the implementation of chronological backtracking in CaDiCaL.

Due to implementation difficulties, we were not able to combine PriPro with CaDiCaL's *compacting* pre- and inprocessing technique [44]. Compacting removes variables from the solver that are no longer used, restoring a contiguous interval of variables. For example, variables whose assignment is fixed can be removed and the others renumbered appropriately. In CaDiCaL_PriPro, compacting is completely disabled. Similar to techniques described in § 2.3, compacting is an optimisation of the data structures the solver uses. Changes to the search path due to compacting are most likely negligible and not intended (we include a comparison of CaDiCaL with and without compacting in the results of § 5.2 without further discussion).

3.2. Changes to Unit Propagation

Traditionally, unit propagation is realised by iterating once through all assigned literals, stored in order of their assignment in a dynamically sized array called the *trail* [3]. At each literal, all its watches (and if necessary the corresponding clauses) are examined until either a conflict or all possible implications have been found. We refer to this as *propagating the literal* resp. *propagating the literal's watches* (see § 2.3). An integer `propagated` is used to indicate the position on the trail up to which the literals have already been propagated, i. e., `propagated` indicates the *current literal* to be propagated.

With PriPro, we distinguish between propagating the literal's regular watches and the literal's prioritised watches. As depicted in Alg. 1, unit propagation is adapted to propagate the prioritised watches of all newly assigned literals at each iteration step. Only then the regular watches of the current literal are propagated. One might say that with PriPro, propagating a literal involves propagating the prioritised watches of all literals on the trail before propagating its regular watches.

Algorithm 1 Unit propagation with PriPro

```
while ¬conflict ∧ propagated ≠ trail_size do
  lit ← −trail[propagated]
  propagated ← propagated + 1
  conflict ← propagate_prioritised_watches()
  if ¬conflict then
    conflict ← propagate_regular_watches_of (lit)
  end if
end while
return conflict
```

The propagation of the prioritised watches of all literals on the trail is analogous to the traditional unit propagation and depicted in Alg. 2. Apart from using prioritised watches we only use a second integer `pripro_propagated` to indicate the position on the trail up to which the literals have already been propagated with respect to their prioritised watches.

Algorithm 2 The *propagate_prioritised_watches* function

while $\neg$*conflict* $\wedge$ *pripro_propagated* $\neq$ *trail_size* **do**
  $lit \leftarrow -trail[pripro\_propagated]$
  $pripro\_propagated \leftarrow pripro\_propagated + 1$
  $conflict \leftarrow propagate\_prioritised\_watches\_of\ (lit)$
end while
return *conflict*

3.3. Selecting Clauses of Higher Priority

PriPro leaves room to experiment with various clause selection heuristics to decide which and when clauses are up- or downgraded. Inspired by clause deletion heuristics, we assumed that resolvents are likely to become relevant soon again. Such clauses are thus considered for an immediate upgrade. However, only clauses of up to some fixed LBD/size threshold are upgraded, avoiding to propagate those considered to be less useful (see § 2.2). In the default configuration, the threshold is set to an LBD of ≤ 6. We discuss the effect of varying this parameter in § 5.2. Moreover, newly learned clauses are considered useful, regardless of their LBD and size, and are initially prioritised. As newly learned conflict clauses are assumed to prevent the same conflict from reappearing, the corresponding conflicting clauses are never upgraded, despite being a resolvent. At the beginning of the solving process, clauses from the original problem did not yet participate in any conflicts, and accordingly are initially not prioritised.

Prioritising more and more clauses eventually degrades the algorithm to behave like unit propagation without PriPro. Therefore, it seems necessary to downgrade clauses at some point. As discussed in § 3.1, combining PriPro with some pre- and inprocessing techniques can be done by downgrading all prioritised clauses at once before applying that pre- or inprocessing technique. Inspired by such *forced* downgrades, we only consider heuristics that downgrade all prioritised clauses at once, instead of downgrading clauses individually.

We downgrade under the following five conditions. We perform downgrades (1) prior to any pre- or inprocessing technique, (2) prior to clause database reductions and (3) at rephasing. Optionally, we may perform downgrades (4) at *restarts* (but not in our default configuration). For simplicity, we consider downgrades performed at the events (2)–(4) as *forced*, too, despite not being necessarily required. In addition to forced downgrades, we perform (5) *scheduled downgrades*, which are performed at a constant interval. The interval between scheduled downgrades is chosen to be a constant number of conflicts, and scheduled downgrades are performed even if forced downgrades occurred in the meantime. If the downgrade interval is chosen to be large enough, forced downgrades dominate. Scheduled downgrades then play the role of periodically cutting apart the interval between two forced downgrades.

4. Related Work

Similarly to PriPro, and in contrast to fundamental propagation techniques described in § 2, some propagation techniques deliberately rearrange the order of propagation. Most importantly, the propagation of binary clauses is often favoured over the propagation of other clauses (see § 4.1). In proof validation, fewer clauses need to be checked if during (reverse) unit propagation yet-unused clauses are considered last (see § 4.2). Also, it was tried to sort the watches within a literal's watch list to propagate low LBD clauses first (see § 4.3). Moreover, a technique to temporarily detach inactive clauses from the TWL scheme was proposed (see § 4.4).

4.1. Propagating Binary Clauses First

The cost of propagating watches of binary clauses can be lowered by storing the other literal in the watcher and avoiding the dereference of the clause [45]. For example, it can be stored as (resp. instead of) the blocking literal. This can be achieved with little to no effect on the order of propagation, such as in the solver Riss 4.27 [46]. Highly efficient implementations with respect to memory and cache usage are possible with a single TWL scheme, but are non-trivial and come with other disadvantages [47]. Most importantly, storing the binary clause may be omitted as the clause is implicitly stored through its watches. For example, Kissat [40] makes use of this technique.

Many solvers such as Glucose [39] and many of its descendants such as SWDiA5BY [48], COMiniSatPS [49], MapleCoMSPS [50] and MapleLCMDistChronoBT [41] introduce a second TWL scheme to handle these binary clauses. For every literal, a second watch list for binary clauses is added. Such a watch list for binary clauses effectively becomes a simple implication list and may be implemented as such. When propagating a literal, first the watches of binary clauses are propagated, and only then are the literal's remaining watches propagated. This approach alters the order of propagation in favour of binary clauses. It served as an inspiration for PriPro to introduce a second TWL scheme for prioritised clauses. In other solvers such as CaDiCaL [40], a comparable effect on the propagation order is achieved without separate watch lists for binary clauses by moving watches of binary clauses to the beginning of their literal's watch list.

Other solvers such as PicoSAT [51] and PrecoSAT [52] propagate all binary clauses before any larger clauses are propagated. In a similar manner, PicoSAT also ensures that *all-different constraint (ADC)* clauses [53] are checked after all other clauses have been propagated. CryptoMiniSAT 2.5.0 treats *xor clauses* [54, 55] similarly. This could be seen as a special case of PriPro with static up-/downgrade heuristics. PriPro differs from these ideas in that we decide based on dynamic factors which clauses to propagate first.

PicoSAT 193 [52] and CryptoMiniSAT 2.5.0 [54] furthermore continue to propagate all binary clauses within a watch list, even if a conflict occurred and use the last binary conflict encountered instead of the first. This behaviour is inspired by effects observed when minimising learned clauses [54, 24]. It may be regarded as reordering the binary watches such that some are a posteriori ignored. No such ideas were considered in the design of PriPro.

For ternary clauses, there exist some optimised propagation mechanisms [56, 40], too. But these are not as widely used as special propagation mechanisms for binary clauses.

4.2. Reducing Proof Checking Effort

The result that a problem is unsatifisable can be verified by validating proofs emitted by the SAT solver with external, possibly mechanically verified tools [57, 58, 59]. While other proof formats exist [60], *clausal proofs* [61] consisting of a sequence of clauses learned during solving process are commonly used, e.g. in SAT Competitions [62]. Checking a clausal proof involves checking that each clause in this sequence is derivable from clauses in the original formula and clauses appearing earlier in the sequence. Additionally provided clause deletion information may be necessary or helpful to be taken into account during the validation process [63, 64].

A common optimisation of the proof validation process is to check the clauses in the proof in reverse order. Such *backwards checking* [61] starts with the last clause (which must be the empty clause). During the validation of each clause in the proof, all clauses that were used by the proof checker in the clause's derivation are marked as *core clauses* [65]. The validation of unmarked clauses in the sequence can then be skipped as they have not been used by the proof checker in the derivation of the empty clause.

Validation of clauses learned during 'pure' CDCL is possible with *reverse unit propagation* [66], i.e., by performing unit propagation on the negation of the literals in the to-be-checked clause using only clauses in the original formula and clauses appearing earlier in the proof sequence. To minimise the number of clauses that need to be added to the core, the propagation of non-core clauses can be postponed until propagation of core clauses is complete [65] (which is an adaption of a technique used in *Minimal Unsatisfiable Core Extraction* to increase the chances of learning *remainder clauses* [67]). Such *CoreFirstBCP* then proceeds to propagate non-core clauses until a single implication is derived (or a conflict is detected) and immediately returns to propagate core clauses with the newly derived implication.

The prioritisation of core clauses in CoreFirstBCP resembles PriPro, but its aim, application context and the necessary means for an implementation differ. In backwards checking, the goal of propagating core clauses first is to reduce the number of clauses that need to be validated in the proof. Accordingly, returning to propagate core clauses after each new implication is likely essential. To implement this, the position of the last propagated watcher to non-core clauses needs to be preserved. Later, the propagation of non-core clauses needs to be resumed at this position.

In the context of regular SAT solving, no such considerations are necessary. Hence, to realise PriPro, a simpler propagation scheme can be used. Moreover, a dynamic prioritisation of useful clauses is possible. While an implementation of CoreFirstBCP using two TWL schemes would be possible, it was implemented using a single TWL scheme in the proof checker *DRUP-trim* [65] and its successor *DRAT-trim* [64], adding further complexity.

4.3. Core First Unit Propagation

Inspired by CoreFirstBCP from proof validation [68] and other more recent techniques from (verified) proof validation [69, 70], Jingchao Chen studied reordering the watch lists during SAT solving to potentially propagate more useful clauses first. His approach, *Core First Unit Propagation* (CFUP) [1], as implemented in the solvers Smallsat, Optsat and MapleLCMdistCBT-coreFirst [71], was to move watches of clauses with an LBD of ≤ 7 within the watch list of a

literal to its beginning. His hypothesis was that prioritising these clauses during propagation leads to more useful conflicts, resulting in shorter conflict clauses learned.

Jingchao Chen's experiments suggest that a small performance improvement over the reference solver MapleLCMDistChronoBT is possible, if CFUP is used only for some million conflicts after which the solver switches back to propagate normally using BCP. When using CFUP during an initial phase of 2 million conflicts, the solver was able to solve a couple of SAT instances more and for most UNSAT instances the runtime of the solver was comparable to the runtime without CFUP. He suspects that as the solver learns more and more clauses of low LBD,[1] CFUP deteriorates to behave similarly to BCP without CFUP, but with a higher runtime cost. However, he did not investigate the effects of CFUP on the solving process, such as whether there is a reduction in learned clause size.

PriPro was inspired by CFUP to focus on clauses of low LBD in its default configuration. However, unit propagation and the order of propagation in PriPro are altered beyond the scope of individual literal's watch lists—to an extent that is only comparable to how all binary clauses are propagated first in PicoSAT, PrecoSAT 193 and CryptoMiniSAT (see § 4.1), and how the propagation of not-yet-used clauses is delayed in backwards proof checking (see § 4.2).

4.4. Freezing Clauses

Audemard et al. introduced the idea to *freeze* clauses by detaching them from the TWL scheme [72]. In contrast to deleted clauses, frozen clauses have the chance to be reactivated. By freezing clauses during clause database cleanings instead of deleting them, the deletion of only temporarily inactive clauses may be prevented. Simultaneously, propagation speed is retained as if the clauses had been deleted.

In addition to freezing inactive clauses during clause database cleanings, clauses which had been sufficiently long frozen were deleted and others reactivated. Audemard et al. based the decisions which clauses would be frozen or reactivated on the *progress saving based quality measure (psm)*. This highly dynamic heuristic is based on the current *phase* and the clause's literals, but independent of whether the variables are currently assigned. It roughly captures how many variables would need to switch their polarity before the clause may lead to a unit implication or conflict.

In experiments, Audemard et al. verified that psm is indeed a good indicator for how often clauses are used during unit propagation and that freezing clauses aids the solving process. The idea of freezing clauses may be regarded as a predecessor of PriPro in which only prioritised clauses are propagated, and the up- and downgrade heuristic is based on psm. However, the ideas of freezing clauses and PriPro are somewhat complementary, and combining them in future research seems interesting.

[1] In MapleLCMDistChronoBT, clauses of LBD ≤ 3 in particular accumulate in the core tier (see § 2.2) and are mostly spared from deletion.

5. Experimental Results

To empirically evaluate the effect of PriPro, we have implemented this approach in the solver CaDiCaL [40] (the *baseline solver*), resulting in the solver CaDiCaL_PriPro. We have not further analysed our previous PriPro implementation in CleanMaple [73, 74]. The version of CaDiCaL_PriPro discussed here differs from the one submitted to SAT Competition 2021 in some minor bug fixes discovered during the review of a previous draft of this paper.[2] As discussed in § 3.1, CaDiCaL_PriPro cannot make use of *compacting*. To obtain more meaningful results, we also disable compacting in the baseline solver for our measurements. However, we include a comparison of CaDiCaL with and without compacting in the results of § 5.2 without further discussion.

To evaluate the performance of CaDiCaL_PriPro for various parameter combinations, we tested the solvers on the SAT Competition 2021 problem set [75]. The set was obtained from the *Global Benchmark Database (GBD)* [76], following established conventions in the SAT community. This problem set contains 400 problem instances both from theoretical and practical domains, and a time limit of $5\,000$ seconds per instance was used. The results presented in the following sections were computed on a 48 node cluster. Each node is a Dell PowerEdge C6520 server, fitted with 1024 GiB of memory and two Intel Xeon Gold 6338 processors with 32 cores each. Solvers were run in parallel with one solver process per core and hyper threading disabled, providing 16 GiB of memory for each solver process. To avoid variations in runtime, no other jobs were run during our measurements.

The results for the number of solved instances and the PAR2 score (see § 5.2) were collected in a run separate from the other statistics, such that their collection does not affect the solver's performance. Furthermore, the results of the up- and downgrade tests were collected in another separate run. In the end, we added some further downgrade variants, leading to five runs in total. All of these were performed under the exact same conditions. For each run, the baseline solver was re-run and additional solvers were run as needed, always using the whole benchmark set. The results of individual runs of the baseline varied by up to 2 instances more or less solved, but were overall very consistent with the PAR2 score (see § 5.2) having a standard deviation of just $\sigma = 12.3$ ($n = 5, \bar{x} = 3303.02$).

5.1. Increased Variance with PriPro

During evaluation of the runs, we observed an anomaly: the PriPro variant with a downgrade interval of $10\,000$ performed much worse than other comparable PriPro variants. We decided to rerun the measurements of this variant to exclude measurement error. As the parameters of the PriPro variant "no scheduled downgrade" of § 5.2 are identical to the parameters of the PriPro variant "LBD ≤ 6" of § 5.3, we obtained another pair of measurements that can be compared.

When comparing the two runs of the interval $10\,000$ downgrade-variant, we noticed that the new run of the interval $10\,000$ variant solved 8 instances more (5 on UNSAT, 3 on SAT), and

[2] Thanks to a reviewer, we noticed that in the version of CaDiCaL_PriPro submitted to SAT Competition 2021, a conflict could in some cases be overwritten by a subsequent binary conflict. Despite this bug, the solver's correctness was preserved. In preliminary results, we observed that this bug harmed the performance of the solver, but did not investigate further.

the nPAR2 score decreased by $4.18\,\%$ from $3\,191.18$ to $3\,057.75$. Similarly, when comparing the two runs of the variant with no scheduled downgrades, we noticed that one of the runs solved 4 instances more (-2 on UNSAT, $+6$ on SAT); the nPAR2 score decreased by $1.43\,\%$ from $2\,997.12$ to $2\,954.13$.

Despite having been run under the same circumstances, no runs of the baseline solver or variants with PriPro disabled showed such a variability. We suspect that this difference may be due to cache effects; as PriPro focuses the solver on a smaller working set of clauses, its cache usage pattern should differ from the baseline solver and may compete with other solvers using the same shared L3 cache more severely than the baseline solver. We believe these variances do not significantly impact the observations and conclusion given in this paper.

5.2. Varying the Downgrade Interval

In this subsection, we discuss our experiments with varying downgrade heuristics. In all tested variants the upgrade condition is chosen to be CaDiCaL_PriPro's default upgrade heuristic, i. e., we upgrade conflict clauses with an LBD of 6 or less and newly learned clauses immediately. As described in § 3.3 we only consider downgrade heuristics in which all clauses are downgraded at once. All variants downgrade when forced. Most variants additionally perform scheduled downgrades at a constant downgrade interval. We tested with downgrade intervals ranging from a very low value of 2 conflicts up to a very high value of $1\,000\,000$ conflicts. In this subsection, we refer to the variant with the constant downgrade interval of $10\,000$ conflicts as the default CaDiCaL_PriPro variant.

Table 1 shows that most variants solve more instances than the baseline solver. The poor performance of variants with a downgrade interval of less than 50 is due to UNSAT instances. Solving SAT instances seems to be less affected by too small downgrade intervals, but in general large downgrade intervals appear to be beneficial. From the absolute numbers of solved instances, PriPro seems to improve the performance on SAT instances in particular. However, measurements on SAT instances are particularly noisy and are known to fluctuate with even slight variations in any solver parameter.

We have studied scatter plots of the PriPro variants against the baseline solver (cf. Fig. 1) which paint a different picture. Scatter plots depict the solving times of two solvers in relation to each other for each instance individually, i. e., the speedup or slowdown on each instance can be seen. For all variants with sufficiently large downgrade intervals (including no scheduled downgrades) we observe a clear tendency of PriPro to reduce runtime on UNSAT instances. No such tendency is apparent for SAT instances. We have included plots comparing against the baseline solver with compacting (Fig. 1b), and corresponding cactus plots (Fig. 2).

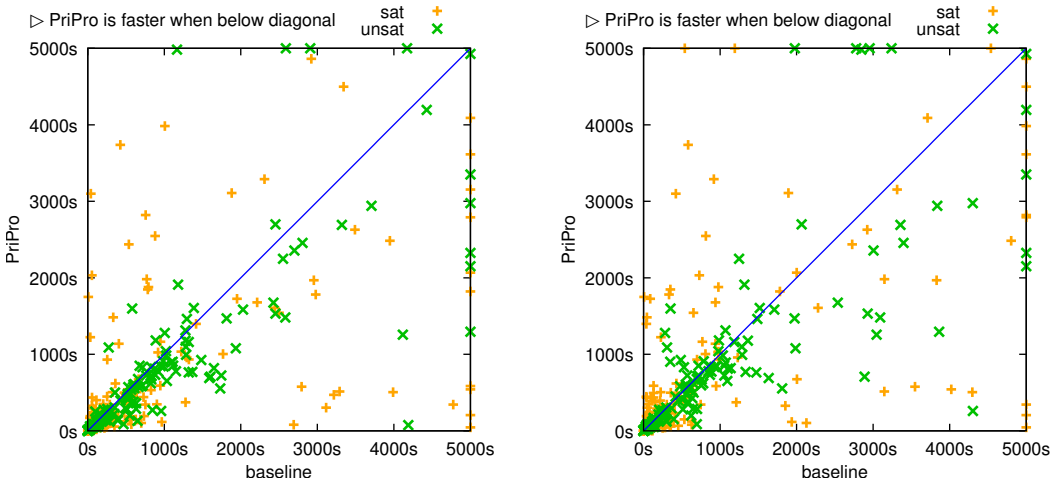
Due to using a timeout, in SAT solving statistical analysis of solver runtimes/speedups is difficult. Comparing the speed of two solvers is only possible on instances that have been solved by both. To overcome difficulties with instances that have been solved by only one of the two solvers, the *PAR2 score* is frequently used. PAR2 treats unsolved instances as having been solved in twice the timeout. We normalise the PAR2 score by dividing by the number of instances in the benchmark set, giving the *nPAR2 score* [43]. The nPAR2 score is superficially comparable to an average runtime and useful to measure the difficulty of two problem sets with respect to a given solver (even if these two sets contain a different amount of instances).

Table 1
Downgrade Interval Variation Experiments

downgrade interval		*solved instances of* 400 *in* $\leq 5000s$ ALL	UNSAT	SAT	*nPAR2 score* ALL	UNSAT	SAT
			baseline solvers				
comp. on		297 (+6)	158 (+2)	139 (+4)	−4.6 %	−5.1 %	−6.8 %
comp. on *		297 (+8)	158 (+3)	139 (+5)	−5.4 %	−6.2 %	−7.8 %
comp. off		**291**	**156**	**135**	**3 282.93**	**2 010.66**	**3 329.26**
comp. off *		**289**	**155**	**134**	**3 316.44**	**2 043.81**	**3 369.07**
PriPro off		291 (±0)	156 (±0)	135 (±0)	1.1 %	2.0 %	1.2 %
PriPro off *		289 (±0)	155 (±0)	134 (±0)	1.0 %	2.1 %	0.9 %
			downgrade at restarts				
10 000		298 (+7)	156 (±0)	142 (+7)	−5.4 %	−3.0 %	−9.6 %
10 000 *		299 (+10)	156 (+1)	143 (+9)	−7.4 %	−3.9 %	−13.3 %
50 000 *		300 (+11)	158 (+3)	142 (+8)	−8.3 %	−8.5 %	−12.6 %
no scheduled *		291 (+2)	156 (+1)	135 (+1)	−2.0 %	−5.9 %	−0.8 %
			no downgrade at restarts				
2		287 (−4)	151 (−5)	136 (+1)	10.7 %	31.2 %	4.4 %
5		285 (−6)	149 (−7)	136 (+1)	8.8 %	27.8 %	2.3 %
10		288 (−3)	152 (−4)	136 (+1)	4.8 %	13.8 %	2.0 %
25		293 (+2)	155 (−1)	138 (+3)	0.5 %	5.7 %	−2.3 %
50		298 (+7)	159 (+3)	139 (+4)	−3.1 %	−3.1 %	−4.8 %
100		296 (+5)	157 (+1)	139 (+4)	−2.7 %	−2.9 %	−4.0 %
250		296 (+5)	160 (+4)	136 (+1)	−4.5 %	−13.7 %	−1.4 %
500		298 (+7)	159 (+3)	139 (+4)	−4.8 %	−10.3 %	−4.3 %
750		291 (±0)	158 (+2)	133 (−2)	−1.7 %	−9.4 %	1.8 %
1 000		299 (+8)	160 (+4)	139 (+4)	−7.9 %	−13.9 %	−8.6 %
2 500		300 (+9)	158 (+2)	142 (+7)	−7.6 %	−10.0 %	−10.2 %
5 000		296 (+5)	158 (+2)	138 (+3)	−6.1 %	−10.2 %	−6.9 %
7 500		297 (+6)	158 (+2)	139 (+4)	−5.9 %	−11.6 %	−5.7 %
10 000		291 (±0)	156 (±0)	135 (±0)	−2.8 %	−6.2 %	−2.3 %
10 000 *		299 (+10)	161 (+6)	138 (+4)	−7.8 %	−18.4 %	−5.7 %
15 000 *		304 (+15)	160 (+5)	144 (+10)	−9.9 %	−14.9 %	−12.2 %
20 000		299 (+8)	159 (+3)	140 (+5)	−7.8 %	−13.5 %	−8.7 %
50 000		304 (+13)	158 (+2)	146 (+11)	−11.5 %	−11.6 %	−16.2 %
100 000 *		305 (+16)	161 (+6)	144 (+10)	−13.2 %	−19.8 %	−16.3 %
1 000 000		300 (+9)	158 (+2)	142 (+7)	−9.2 %	−11.7 %	−12.6 %
no scheduled	cf. Fig. 1, 2	304 (+13)	159 (+3)	145 (+10)	−10.0 %	−12.3 %	−14.0 %

* *Results from the revised run, see remark in § 5.1.*
*Relative values refer to the baseline solver (**comp. off**) of the same run.*

The nPAR2 score of the baseline solver is 3282.93 and the nPAR2 score of the default PriPro variant is 10.0 % lower. Across a wide range of downgrade intervals, PriPro shows a tendency of speeding up the solving process on UNSAT instances more than on SAT instances. This is despite the effect on the number of solved instances being more pronounced for SAT instances.



(a) baseline is without compacting

(b) baseline is with compacting

Figure 1: Scatter Plots of CaDiCaL_PriPro without scheduled downgrades ('PriPro') against CaDiCaL ('baseline')

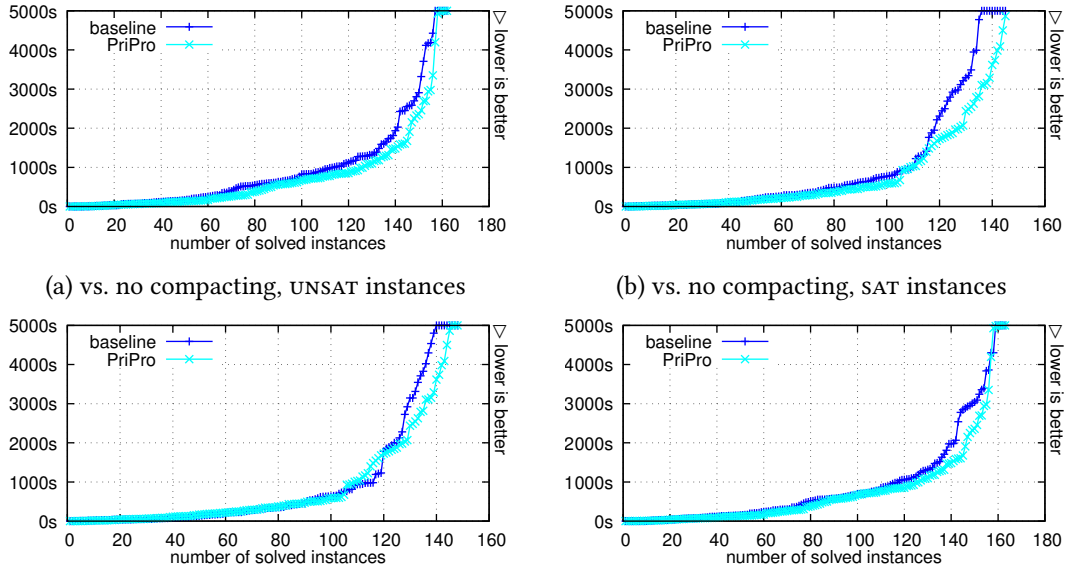
To evaluate the actual speedup, we are forced to ignore between 5.70 % and 9.24 % of instances that were only solved by one of two solvers (6.43–12.33 % on SAT, 3.16–7.41 % on UNSAT). While this introduces some uncertainty, it is worth noting that CaDiCaL_PriPro's default variant solves 80 % solved UNSAT instances faster than the baseline solver (120 of 150 mutually solved instances). In fact, each variant with a downgrade interval of 750 conflicts or more leads to a speed up on at least 66 % of all UNSAT instances that where solved by both, that variant and the baseline solver.

5.2.1. Further analysis

For the variants with a downgrade interval of 2, 5, 50, 500, 5 000, 50 000, as well as for the variant with no scheduled downgrades and the variant with downgrades at restarts, we collected further metrics.

We noticed that with PriPro the solver learns shorter clauses. We compared the average learned clause size before (resp. after) LCM of the solvers with PriPro to the average learned clause size before (resp. after) LCM of the baseline solver. For the length of learned clauses with downgrade intervals of 5 000 or longer (including at restarts), we observed that the pre-LCM length was reduced by about 7 % for SAT instances and about 11 % for UNSAT instances. After LCM, the reduction grows to 21 % for UNSAT instances, but drops to about 6 % for SAT instances. Therefore, the implementation of PriPro in CaDiCaL supports Jingchao Chen's hypothesis that a different propagation order may reduce the learned clauses' size.

The average *effective downgrade interval*, i. e., the actual number of conflicts between two downgrades regardless of whether these are scheduled or forced, is close to the scheduled downgrade interval as long as the scheduled downgrade interval is less than about 500. As the downgrade interval grows, forced downgrades increasingly preempt scheduled downgrades,



(a) vs. no compacting, UNSAT instances

(b) vs. no compacting, SAT instances

(c) vs. with compacting, UNSAT instances

(d) vs. with compacting, SAT instances

Figure 2: Cactus Plots comparing CaDiCaL_PriPro without scheduled downgrades ('PriPro') against CaDiCaL ('baseline') with and without compacting

capping the average effective downgrade interval at about 15 000 even if there are no scheduled downgrades. We expect these numbers to depend strongly on the number and frequency of inprocessing techniques and whether these force downgrades or not. In CaDiCaL, the variant with downgrades at restarts has an average effective downgrade interval of about 40.

5.3. Varying the Upgrade Condition

In this subsection, we discuss our experiments of varying the upgrade conditions as depicted in Table 2. As discussed in § 3.3, we upgrade conflict clauses if a further condition is met. As this condition, we chose to only upgrade clauses of small LBD, smaller than some constant threshold, or similarly, to only upgrade clauses of small size with respect to some constant threshold. Furthermore, we tested a variant in which every conflict clause is upgraded regardless of its size or LBD (*always*).

For the number of overall solved instances, we noticed that PriPro performs well for a wide range of choices of LBD/size limits. Mainly, an unreasonably small LBD limit leads to poor performance. Variants with a large or no size limit solve the most SAT instances. In general, variants with a size limit solve more additional SAT instances than additional UNSAT instances. In contrast to that, variants with an LBD limit solve the most UNSAT instances, but perform well on SAT instances, too. Looking at the NPAR2 score confirms these observations. We thus recommend an LBD limit for general instances and a large size limit or no limit when it is known or suspected that the instance is satisfiable.

The results of this subsection give the impression that the improvement on UNSAT instances observed in § 5.2 is strongly influenced by the choice of using an LBD limit in these measurements.

Table 2
Upgrade Condition Variation Experiments

upgrade condition	*solved instances of* 400 *in* $\leq 5000s$ ALL	UNSAT	SAT	*nPAR2 score* ALL	UNSAT	SAT
			baseline solvers			
comp. on	298 (+8)	158 (+2)	140 (+6)	−5.45 %	−5.00 %	−8.57 %
comp. off	**290**	**156**	**134**	**3 372.20**	**2 019.65**	**3 372.70**
PriPro off	289 (−1)	155 (−1)	134 (±0)	1.29 %	3.12 %	0.89 %
			upgrade with LBD limit			
LBD ≤ 0	286 (−4)	154 (−2)	132 (−2)	2.27 %	3.41 %	2.80 %
LBD ≤ 1	283 (−7)	155 (−1)	128 (−6)	5.13 %	0.41 %	10.57 %
LBD ≤ 2	299 (+9)	159 (+3)	140 (+6)	−8.49 %	−12.54 %	−10.60 %
LBD ≤ 3	293 (+3)	158 (+2)	135 (+1)	−5.92 %	−11.02 %	−6.06 %
LBD ≤ 4	299 (+9)	160 (+4)	139 (+5)	−8.55 %	−14.96 %	−9.31 %
LBD ≤ 5	300 (+10)	161 (+5)	139 (+5)	−9.18 %	−17.56 %	−9.12 %
LBD ≤ 6	300 (+10)	161 (+5)	139 (+5)	−9.38 %	−18.80 %	−8.82 %
LBD ≤ 7	292 (+2)	158 (+2)	134 (±0)	−4.42 %	−11.30 %	−2.74 %
LBD ≤ 8	300 (+10)	162 (+6)	138 (+4)	−9.32 %	−21.42 %	−7.18 %
			upgrade with size limit			
size ≤ 2	294 (+4)	157 (+1)	137 (+3)	−3.08 %	−4.43 %	−3.92 %
size ≤ 4	289 (−1)	154 (−2)	135 (+1)	0.05 %	2.25 %	−1.20 %
size ≤ 6	298 (+8)	159 (+3)	139 (+5)	−7.03 %	−8.90 %	−9.63 %
size ≤ 8	296 (+6)	157 (+1)	139 (+5)	−5.16 %	−5.80 %	−7.51 %
size ≤ 10	296 (+6)	158 (+2)	138 (+4)	−6.14 %	−7.56 %	−8.55 %
size ≤ 12	301 (+11)	160 (+4)	141 (+7)	−9.43 %	−14.70 %	−11.32 %
always	299 (+9)	157 (+1)	142 (+8)	−9.90 %	−10.07 %	−15.01 %

6. Conclusion

PriPro is a technique to propagate clauses that are deemed more useful first. We have shown how PriPro is used with simple and yet effective dynamic heuristics to improve CaDiCaL's performance. Moreover, we observe a significant reduction of learned clause size of about 20 % on UNSAT instances and 6 % on SAT instances. As seen in Tables 1 and 2, this improvement is robust for different parameters of our up- and downgrade heuristics. Most variations solve more instances than the baseline. Even in the worst case, we only observe a slight decrease in the number of instances solved. Based on the observed performance of LBD and size limits, we recommend upgrading resolvents based on an LBD limit. A size limit, or no limit may be used to solve SAT instances in particular. Using an LBD limit of ≤ 6 and a downgrade interval of 15 000 seems to be an overall good choice. However, a wide range of parameters for both the upgrade conditions and the downgrade intervals appear reasonable and yield comparable results.

To solve UNSAT problems, the downgrade interval should be chosen to be at least 250. For SAT problems a larger downgrade interval of at least 15 000 appears sensible. If a solver forces downgrades for pre-/inprocessing techniques, rephasing, clause database reductions or similar, experimenting with no scheduled downgrades may be interesting for solving SAT instances.

Acknowledgments

We would like to thank the IT and data services members of Zuse Institute Berlin for providing the infrastructure, and Florian Schintke for his work in reviewing the paper and for his many useful suggestions and improvements. Benjamin Kaiser would like to thank Marc Hartung, his former advisor, for introducing him to the SAT problem. The authors are grateful for the services provided by the Global Benchmark Database.

References

[1] J. Chen, Core First Unit Propagation, arXiv preprint arXiv:1907.01192 (2019).
[2] D. E. Knuth, The Art of Computer Programming, volume 4B, Combinatorial Algorithms, Part 2, Addison-Wesley, 2022.
[3] A. Biere, M. Heule, H. van Maaren, Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, second ed., IOS Press, 2021.
[4] S. A. Cook, The complexity of theorem-proving procedures, in: Proceedings of the third annual ACM symposium on Theory of computing, 1971, pp. 151–158.
[5] J. Marques-Silva, Practical applications of boolean satisfiability, in: 2008 9th International Workshop on Discrete Event Systems, IEEE, 2008, pp. 74–80.
[6] H. A. Kautz, B. Selman, et al., Planning as Satisfiability., in: ECAI, volume 92, Citeseer, 1992, pp. 359–363.
[7] E. M. Clarke, D. Kroening, F. Lerda, A Tool for Checking ANSI-C Programs, in: International Conference on Tools and Algorithms for Construction and Analysis of Systems, 2004.
[8] S. Khurshid, D. Marinov, TestEra: Specification-Based Testing of Java Programs Using SAT, Autom. Softw. Eng. 11 (2004) 403–434. doi:{10.1023/B:AUSE.0000038938.10589.b9}.
[9] R. E. Mcgregor, Automated theorem proving using SAT, Clarkson University, 2011.
[10] C. E. Brown, Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems, Journal of Automated Reasoning 51 (2013) 57–77.
[11] T. N. Alyahya, M. E. B. Menai, H. Mathkour, On the Structure of the Boolean Satisfiability Problem: A Survey, ACM Comput. Surv. 55 (2022). URL: https://doi.org/10.1145/3491210. doi:10.1145/3491210.
[12] C. Li, J. Chung, S. Mukherjee, M. Vinyals, N. Fleming, A. Kolokolova, A. Mu, V. Ganesh, On the Hierarchical Community Structure of Practical Boolean Formulas, in: C.-M. Li, F. Manyà (Eds.), Theory and Applications of Satisfiability Testing – SAT 2021, Springer International Publishing, Cham, 2021, pp. 359–376.
[13] J. P. Marques-Silva, K. A. Sakallah, Grasp: A search algorithm for propositional satisfiability, IEEE Transactions on Computers 48 (1999) 506–521.
[14] V. Ganesh, M. Y. Vardi, On the Unreasonable Effectiveness of SAT Solvers., 2020.
[15] M. Davis, H. Putnam, A Computing Procedure for Quantification Theory, J. ACM 7 (1960) 201–215. URL: https://doi.org/10.1145/321033.321034. doi:10.1145/321033.321034.
[16] R. Zabih, D. McAllester, A Rearrangement Search Strategy for Determining Propositional

Satisfiability, in: Proceedings of the Seventh AAAI National Conference on Artificial Intelligence, AAAI'88, AAAI Press, 1988, p. 155–160.

[17] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, K. Czarnecki, Understanding VSIDS Branching heuristics in Conflict-Driven Clause-Learning SAT Solvers, in: N. Piterman (Ed.), Hardware and Software: Verification and Testing, Springer International Publishing, Cham, 2015, pp. 225–241.

[18] J. H. Liang, V. Ganesh, P. Poupart, K. Czarnecki, Learning Rate Based Branching Heuristic for SAT Solvers, in: N. Creignou, D. Le Berre (Eds.), Theory and Applications of Satisfiability Testing – SAT 2016, Springer International Publishing, Cham, 2016, pp. 123–140.

[19] K. Pipatsrisawat, A. Darwiche, A Lightweight Component Caching Scheme for Satisfiability Solvers, in: J. Marques-Silva, K. A. Sakallah (Eds.), Theory and Applications of Satisfiability Testing – SAT 2007, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 294–299.

[20] A. Biere, CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2018, in: M. J. H. Heule, M. J. Järvisalo, M. Suda (Eds.), Proceedings of SAT Competition 2018 : Solver and Benchmark Descriptions, volume B-2017-1, 2018.

[21] Biere, Armin, CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2017, in: T. Balyo, M. J. H. Heule, M. J. Järvisalo (Eds.), Proceedings of SAT Competition 2017 : Solver and Benchmark Descriptions, Department of Computer Science Report Series B, Department of Computer Science, University of Helsinki, 2017, pp. 14–15.

[22] A. Biere, CaDiCaL at the SAT Race 2019, in: M. J. H. Heule, M. J. Järvisalo, M. Suda (Eds.), Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions, volume B-2019-1, 2019.

[23] G. Audemard, L. Simon, Refining restarts strategies for sat and unsat, in: M. Milano (Ed.), Principles and Practice of Constraint Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 118–126.

[24] N. Sörensson, A. Biere, Minimizing learned clauses, in: Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings 12, Springer, 2009, pp. 237–243.

[25] J. Marques Silva, K. Sakallah, GRASP-A new search algorithm for satisfiability, in: Proceedings of International Conference on Computer Aided Design, 1996, pp. 220–227. doi:10.1109/ICCAD.1996.569607.

[26] A. Nadel, V. Ryvchin, Chronological backtracking, in: 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 21, 2018, pp. 111–121.

[27] T. Krüger, J. Lorenz, F. Wörz, Too much information: CDCL solvers need to forget and perform restarts, CoRR abs/2202.01030 (2022). URL: https://arxiv.org/abs/2202.01030. arXiv:2202.01030.

[28] N. Eén, N. Sörensson, An Extensible SAT-solver, in: International Conference on Theory and Applications of Satisfiability Testing, 2003, pp. 502–518.

[29] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: Twenty-first international joint conference on artificial intelligence, Citeseer, 2009.

[30] C. Oh, Between SAT and UNSAT: The Fundamental Difference in CDCL SAT, in: M. Heule, S. Weaver (Eds.), Theory and Applications of Satisfiability Testing – SAT 2015, Springer

International Publishing, Cham, 2015, pp. 307–323.
[31] C. Oh, Improving SAT solvers by exploiting empirical characteristics of CDCL, Ph.D. thesis, New York University, 2016.
[32] L. Simon, Post Mortem analysis of SAT Solver Proofs, in: D. L. Berre (Ed.), POS-14. Fifth Pragmatics of SAT workshop, volume 27 of *EPiC Series in Computing*, EasyChair, 2014, pp. 26–40. URL: https://easychair.org/publications/paper/N3GD. doi:10.29007/gpp8.
[33] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232), 2001, pp. 530–535. doi:10.1145/378239.379017.
[34] H. Zhang, SATO: An Efficient Propositional Prover, in: Proceedings of the 14th International Conference on Automated Deduction, CADE-14, Springer-Verlag, Berlin, Heidelberg, 1997, p. 272–275.
[35] A. Nadel, Introducing Intel(R) SAT Solver, in: K. S. Meel, O. Strichman (Eds.), 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022), volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, pp. 8:1–8:23. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16682. doi:10.4230/LIPIcs.SAT.2022.8.
[36] S. Hölldobler, N. Manthey, A. Saptawijaya, Improving resource-unaware SAT solvers, in: Logic for Programming, Artificial Intelligence, and Reasoning: 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings 17, Springer, 2010, pp. 519–534.
[37] N. Sörensson, N. Eén, MiniSat 2.1 and MiniSat++ 1.0 - SAT Race 2008 Editions, SAT (2009) 31.
[38] G. Chu, A. Harwood, P. Stuckey, Cache Conscious Data Structures for Boolean Satisfiability Solvers, JSAT 6 (2009) 99–120. doi:10.3233/SAT190064.
[39] G. Audemard, L. Simon, Glucose: a solver that predicts learnt clauses quality, SAT Competition (2009) 7–8.
[40] A. Biere, K. Fazekas, M. Fleury, M. Heisinger, CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020, in: T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions, volume B-2020-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2020, pp. 51–53.
[41] V. Ryvchin, A. Nadel, Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking, in: M. J. H. Heule, M. J. Järvisalo, M. Suda (Eds.), Proceedings of SAT Competition 2018 : Solver and Benchmark Descriptions, volume B-2018-1 of *Department of Computer Science Report Series B*, Department of Computer Science, University of Helsinki, 2018, p. 29.
[42] I. Gent, Optimal implementation of watched literals and more general techniques, Journal of Artificial Intelligence Research 48 (2013) 231–252. Includes 2 appendixes: one with additional proofs and one with code, scripts and data.
[43] M. Iser, T. Balyo, Unit Propagation with Stable Watches, in: L. D. Michel (Ed.), 27th International Conference on Principles and Practice of Constraint Programming (CP 2021), volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 6:1–6:8. URL: https://drops.dagstuhl.de/opus/volltexte/2021/15297. doi:10.4230/LIPIcs.CP.2021.6.

[44] B. Dutertre, An Empirical Evaluation of SAT Solvers on Bit-vector Problems., in: SMT, 2020, pp. 15–25.
[45] S. Pilarski, G. Hu, Speeding up SAT for EDA, in: Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition, IEEE, 2002, p. 1081.
[46] N. Manthey, Riss 4.27, in: A. Belov, D. Diepold, M. J. Heule, M. Järvisalo (Eds.), Proceedings of SAT Competition 2014, volume B-2014-2 of *Department of Computer Science Series of Publications B*, University of Helsinki, Helsinki, Finland, 2014, pp. 65–67.
[47] M. Soos, On using less memory for binary clauses in lingeling's watchlists – Wonderings of a SAT geek, 2014. URL: https://www.msoos.org/2014/08/on-using-less-memory-for-binary-clauses-in-lingeling, [Online; accessed 11. Mar. 2023].
[48] C. Oh, Minisat hack 999ed, minisat hack 1430ed and swdia5by, SAT Competition (2014).
[49] C. Oh, Patching MiniSat to Deliver Performance of Modern SAT Solvers, 2015.
[50] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, P. Poupart, Maple-comsps, maplecomsps lrb, maplecomsps chb, Proceedings of SAT Competition 2016 (2016).
[51] A. Biere, PicoSAT essentials, Journal on Satisfiability, Boolean Modeling and Computation 4 (2008) 75–97.
[52] A. Biere, P{re,i}coSAT@ SC'09, SAT 4 (2009) 41–43.
[53] A. Biere, R. Brummayer, Consistency checking of all different constraints over bit-vectors within a SAT solver, in: 2008 Formal Methods in Computer-Aided Design, IEEE, 2008, pp. 1–4.
[54] M. Soos, CryptoMiniSat 2.5.0, http://baldur. iti. uka. de/sat-race-2010/descriptions/solver_13. pdf (2010).
[55] M. Soos, K. Nohl, C. Castelluccia, Extending SAT solvers to cryptographic problems, in: Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings 12, Springer, 2009, pp. 244–257.
[56] L. Ryan, Efficient Algorithms for Clause-Learning SAT Solvers, Master's thesis, 2004.
[57] T. Weber, Efficiently Checking Propositional Resolution Proofs in Isabelle/HOL, in: International Workshop on the Implementation of Logics (IWIL), volume 212, 2006, pp. 44–62.
[58] Weber, Tjark and Amjad, Hasan, Efficiently Checking Propositional Refutations in HOL Theorem Provers, Journal of Applied Logic 7 (2009) 26–40.
[59] Darbari, Ashish and Fischer, Bernd and Marques-Silva, Joao, Industrial-Strength Formally Certified SAT Solving, arXiv preprint arXiv:0911.1678 (2009).
[60] L. Zhang, S. Malik, Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications, in: 2003 Design, Automation and Test in Europe Conference and Exhibition, IEEE, 2003, pp. 880–885.
[61] E. Goldberg, Y. Novikov, Verification of Proofs of Unsatisfiability for CNF Formulas, in: 2003 Design, Automation and Test in Europe Conference and Exhibition, IEEE, 2003, pp. 886–891.
[62] SAT Competition, 2023. URL: https://satcompetition.github.io/2023/certificates.html, [Online; accessed 20. Jul. 2023].
[63] Heule, Marijn JH and Hunt Jr, Warren A and Wetzler, Nathan, Bridging the Gap Between Easy Generation and Efficient Verification of Unsatisfiability Proofs, Software Testing,

Verification and Reliability 24 (2014) 593–607.
[64] N. Wetzler, M. J. H. Heule, W. A. Hunt, DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs, in: C. Sinz, U. Egly (Eds.), Theory and Applications of Satisfiability Testing – SAT 2014, Springer International Publishing, Cham, 2014, pp. 422–429.
[65] Heule, Marijn J.H. and Hunt, Warren A. and Wetzler, Nathan, Trimming while Checking Clausal Proofs, in: 2013 Formal Methods in Computer-Aided Design, 2013, pp. 181–188. doi:10.1109/FMCAD.2013.6679408.
[66] Van Gelder, Allen, Verifying RUP Proofs of Propositional Unsatisfiability., in: ISAIM, 2008.
[67] Ryvchin, Vadim and Strichman, Ofer, Faster Extraction of High-Level Minimal Unsatisfiable Cores, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2011, pp. 174–187.
[68] Marijn J.H. Heule, 4th July 2023 at 14th Pragmatics of SAT international workshop, 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), personal communication, 2023.
[69] J. Chen, Fast Verifying Proofs of Propositional Unsatisfiability via Window Shifting, 2018. arXiv:1611.04838.
[70] P. Lammich, Efficient verified (UN) SAT certificate checking, in: Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings, Springer, 2017, pp. 237–254.
[71] J. Chen, Smallsat, Optsat and MapleLCMdistCBTcoreFirst: Containing Core First Unit Propagation, SAT RACE 2019 (2019) 31.
[72] G. Audemard, J.-M. Lagniez, B. Mazure, L. Sais, On freezing and reactivating learnt clauses, in: Theory and Applications of Satisfiability Testing-SAT 2011: 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings 14, 2011, pp. 188–200. doi:10.1007/978-3-642-21581-0_16.
[73] B. Kaiser, R. Clausecker, CleanMaple, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. J. Järvisalo, M. Suda (Eds.), Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions, volume B-2021-1 of *Department of Computer Science Report Series B*, Department of Computer Science, University of Helsinki, 2021, p. 24.
[74] B. Kaiser, R. Clausecker, CleanMaple_PriPro, CaDiCaL_PriPro and CaDiCaL_PriPro_no_bin, in: T. Balyo, N. Froleyks, M. J. H. Heule, M. J. Järvisalo, M. Suda (Eds.), Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions, volume B-2021-1 of *Department of Computer Science Report Series B*, Department of Computer Science, University of Helsinki, 2021, p. 25.
[75] SAT Competition 2021 Problems, 2021. [Online; accessed 19. Mar. 2023].
[76] M. Iser, C. Sinz, A Problem Meta-Data Library for Research in SAT, in: D. L. Berre, M. Järvisalo (Eds.), Proceedings of Pragmatics of SAT 2015 and 2018, volume 59 of *EPiC Series in Computing*, 2019, pp. 144–152. doi:10.29007/gdbb.