Trail Saving on Backtrack

Randy Hickey(✉) and Fahiem Bacchus(✉)

Department of Computer Science, University of Toronto, Toronto, Canada
{rhickey,fbacchus}@cs.toronto.edu

Abstract. A CDCL SAT solver can backtrack a large distance when it learns a new clause, e.g, when the new learnt clause is a unit clause the solver has to backtrack to level zero. When the length of the backtrack is large, the solver can end up reproducing many of the same decisions and propagations when it redescends the search tree. Different techniques have been proposed to reduce this potential redundancy, e.g., partial/chronological backtracking and trail saving on restarts. In this paper we present a new trail saving technique that is not restricted to restarts, unlike prior trail saving methods. Our technique makes a copy of the part of the trail that is backtracked over. This saved copy can then be used to improve the efficiency of the solver's subsequent redescent. Furthermore, the saved trail also provides the solver with the ability to look ahead along the previous trail which can be exploited to improve its efficiency. Our new trail saving technique offers different tradeoffs in comparison with chronological backtracking and often yields superior performance. We also show that our technique is able to improve the performance of state-of-the-art solvers.

1 Introduction

The vast majority of modern SAT solvers that are used to solve real-world problems are based on the conflict-driven clause learning (CDCL) algorithm. In a CDCL SAT solver, backtracking occurs after every conflict, where all literals from one or more decision levels become unassigned before the solver resumes making decisions and performing unit propagations. Traditionally, CDCL solvers would backtrack to the conflict level, which is the second highest decision level remaining in the conflict clause after conflict analysis has resolved away all but one literal from the current decision level [9]. Recently, however, it has been shown that partial backtracking [6] or chronological backtracking, C-bt, (i.e., backtracking only to the previous level after conflict analysis) [8,11] can be effective on many instances. Partial backtracking has been used in the solvers that won the last two SAT competitions. Although chronological backtracking breaks some of the conventional invariants of CDCL solvers, it has been formalized and proven correct [8] (also see related formalizations [10,12]).

The motivation for using C-bt is the observation that when a solver backtracks across many levels, many of the literals that are unassigned during the backtrack might be re-assigned again in roughly the same order when the solver

© Springer Nature Switzerland AG 2020
L. Pulina and M. Seidl (Eds.): SAT 2020, LNCS 12178, pp. 46–61, 2020.
https://doi.org/10.1007/978-3-030-51825-7_4

redescends. This observation was first made in the context of restarts by van der Tak et al. [14]. Their technique backtracks to the minimum change level, i.e., the first level at which the solver's trail can change on redescent. However, their technique cannot be used when backtracking from a conflict: the solver's trail is going to be changed at the backtrack level so the minimum change level is the same as the backtrack level.

Chronological backtracking or partial backtracking instead allows a reduction in the length of the backtrack by placing literals on the trail out of decision level order. By reducing the length of the backtrack the solver can keep more of its assignment trail intact. This can save it from the work involved in reconstructing a lot of its trail. Using C-bt is not a panacea however. Its application must be limited for peak effectiveness. This indicates that it is sometimes beneficial for the solver to backtrack fully and redo its trail, even if this takes more work. We will expand on why this might be the case below.

In this paper we present a new trail saving method whereby we save the backtracked part of the solver's trail and attempt to use that information to make the solver's redescent more efficient. Unlike C-bt, our trail saving method preserves the traditional invariants of the SAT solver and its basic version is very simple to implement. It allows the search to retain complete control over the order of decisions, but helps make propagation faster. We develop some enhancements to make the idea more effective, and demonstrate experimentally that it performs as well as and often better than chronological backtracking. We also show that with our enhancements we are able to improve the performance of state-of-the-art solvers.

2 Background

SAT solvers determine the satisfiability of a propositional formula $\mathcal{F}$ expressed in Conjunctive Normal Form (CNF). $\mathcal{F}$ contains a set of variables V. A literal is a variable $v \in V$ or its negation $\neg v$, and for a literal l we let $\text{var}(l)$ denote its underlying variable. A CNF consists of a conjunction of clauses, each of which is a disjunction of literals. We often view a clause as being a set of literals and employ set notation, e.g., $\ell \in C$ and $C' \subset C$. We will assume that the reader is familiar with the basic operations of CDCL SAT solvers. A good source for this background is [13].

Trails. CDCL SAT solvers maintain a trail which is the sequence of literals that have currently been assigned TRUE by the solver. During its operation a SAT solver will add newly assigned literals to the end of the trail, and on backtrack remove literals from the end of the trail. For convenience, we will regard *literals as having been assigned* TRUE *if and only if they are on the trail.* So removing/adding a literal to the trail is equivalent to unassigning/assigning the literal TRUE.

A SAT solver's trail satisfies a number of conditions. However, in this work we will need some additional flexibility in our definitions, as we will sometimes

be working with trails that would never be constructed by a SAT solver. Hence, we define a *trail* to be a sequence of literals each of which is either a *decision* literal or an *implied* literal, and each of which has a *reason*. These two types of literals are distinguished by their *reasons*. Decision literals d have a null reason, $reason(d) = \varnothing$. Implied literals l have as a reason a clause of the formula $\mathcal{F}$, $reason(l) = C \in \mathcal{F}$. (The clause $reason(l)$ can be a learnt clause that has been added to $\mathcal{F}$).

If literal ℓ is on the trail $\mathcal{T}$ let $\iota_{\mathcal{T}}(\ell)$ denote its index on the trail, i.e, $\mathcal{T}[\iota_{\mathcal{T}}(\ell)] = \ell$. If x and y are both on the trail and $\iota_{\mathcal{T}}(x) < \iota_{\mathcal{T}}(y)$ we say that x *appears before* y *on the trail*. For convenience, when the trail being discussed is clear from context we simply write ι instead of $\iota_{\mathcal{T}}$.

Each literal $\ell \in \mathcal{T}$ has a decision level $decLvl(\ell)$ which is equal to the number of decision literals appearing on the trail up to and including ℓ ; hence, $decLvl(d) = 1$ for the first decision literal $d \in \mathcal{T}$. The set of literals on $\mathcal{T}$ that have the same decision level forms a *contiguous* subsequence[1] that starts with a decision literal d_i and ends just before the next decision literal d_{i+1}. We will often need to refer to different decision level subsequences of $\mathcal{T}$. Hence, we let $\mathcal{T}[[i]]$ denote the subsequence of literals at decision level i; and let $\mathcal{T}[[i \dots j]]$ denote the subsequence of literals at decision levels k for $i \leq k \leq j$.

Definition 1. *A clause C has* ***been made unit by $\mathcal{T}$ implying*** *l when $l \in C \wedge \big(\forall x \in C. x \neq l \rightarrow \neg x \in \mathcal{T}\big)$. That is, all literals in C except l must have been falsified by $\mathcal{T}$*

Now we define the following properties that a trail $\mathcal{T}$ can have.

non-contradictory: A variable cannot appear in both polarities in the trail: $l \in \mathcal{T} \rightarrow \neg l \notin \mathcal{T}$.

non-redundant: A literal can only appear once on $\mathcal{T}$.

reason-sound: For each implied literal $l \in \mathcal{T}$ we have that its reason clause $reason(l) = C$ has been made unit by $\mathcal{T}$ implying l, and for each $x \in C$ with $x \neq l$ we have that $\neg x$ appears before l on $\mathcal{T}$: $\forall l \in \mathcal{T}.\, reason(l) \neq \varnothing \rightarrow l \in reason(l) \wedge \big(\forall x \in reason(l).\, x \neq l \rightarrow \neg x \in \mathcal{T} \wedge \iota(\neg x) < \iota(l)\big)$.

propagation-complete: Unit propagation has been run to completion at all decisions levels of $\mathcal{T}$. This means that literals appear on $\mathcal{T}$ at the first decision level they were unit implied. Formally, this can be captured by the condition: $\forall i \in \{decLvl(l) \,|\, l \in \mathcal{T}\}.\big(\exists C \in \mathcal{F}.C$ is made unit by $\mathcal{T}[[0 \dots i]]$ implying $l\big) \rightarrow l \in \mathcal{T}[[0 \dots i]]$. Note that propagation completeness implies that $reason(l) \neq \varnothing$ must contain at least one other literal $y \neq l$ with $decLvl(y) = decLvl(l)$.

conflict-free: No clause of F is falsified by $\mathcal{T}$. Clauses $C \in F$ falsified by $\mathcal{T}$ are typically called *conflicts*.

[1] Our approach uses standard trails in which the decision levels are contiguous. Chronological backtracking [6,8,11] generates trails with non-contiguous decision levels.

In CDCL solvers using standard conflict directed backtracking all properties hold of the prefix of the solver's trail consisting of all decisions levels but the deepest. The full trail might, however, contain a conflict at its deepest level so is not necessarily conflict-free. The full trail might also not be propagation-complete, as unit propagation at the deepest level is typically terminated early if a conflict is found. It can further be noted that the first four properties imply that if a clause C is falsified at decision level k, then C must contain at least two literals at level k (otherwise C would have become unit at a prior level and then satisfied by making its last unfalsified literal TRUE).

Standard Backtracking. In CDCL SAT solving the solver extends its trail by adding new decision literals followed by finding and adding all unit implied literals arising from that new decision. This continues until it reaches a decision level L_{deep} where a conflict C is found.

In standard backtracking, the solver then constructs a new 1-UIP clause by resolving away all but one literal at level L_{deep} from the conflict C using the reason clauses of these literals. (As noted above C must contain at least two literals at level L_{deep}). Hence, the new clause $C_{1\text{-}UIP}$ will contain one literal ℓ_{deep} at level L_{deep} and have all of its other literals a levels less than L_{deep}. The solver then backtracks to L_{back} the second deepest level in $C_{1\text{-}UIP}$. This involves changing $\mathcal{T}$ to its prefix $\mathcal{T}[[0 \ldots L_{back}]]$ (by our convention all literals removed from $\mathcal{T}$ are now unassigned). The new clause $C_{1\text{-}UIP}$ is made unit by $\mathcal{T}[[0 \ldots L_{back}]]$ implying ℓ_{deep}, so the solver then adds ℓ_{deep} to the trail and executes another round of unit propagation at level L_{back}, after which it continues by once again growing the trail with new decisions and unit implied literals until a new conflict or a satisfying assignment is found.

In standard backtracking, the difference between the backtrack level, L_{back} and the current deepest level L_{deep} can be very large. During its new descent from L_{back} the solver can reproduce a large number of the same decisions and unit propagations, essentially wasting work. This potential inefficiency has been noted in prior work [6,8,11,14].

In [14] a technique for reducing the length of the backtrack during restarts was presented. In restarts, the solver backtracks to level 0, and this technique involves computing a new deeper backtrack level $M > 0$ for which it is known that on redescent the first $M + 1$ levels of the trail will be unchanged (except perhaps for the ordering of the literals). This technique removes the redundant work of reproducing the first M trail levels. When backtracking from a conflict, however, the trail will be changed at level L_{back} (ℓ_{deep} will be newly inserted at this level). Hence this technique cannot reduce the length of the backtrack. In this paper we will show that although we have to backtrack to L_{back} we can make the subsequent redescent much more efficient.

Chronological Backtracking. Chronological backtracking (C-bt) and partial backtracking in the context of clause learning solvers are alternatives to standard backtracking which allow the solver to execute a shorter backtrack. That is,

with these techniques the solver can avoid having to go all the way back to the second deepest level in the learnt clause, as in standard backtracking.

Formalisms for partial backtracking in clause learning solvers have been presented in [10,12]. In [6] practical issues of implementation were addressed, and experiments shown with a CDCL solver using partial backtracking. In [11] improved and more efficient implementation techniques were developed which allowed C-bt to make improvements to state-of-the-art SAT solvers, and [8] presented additional implementation ideas and details along with correctness results for these methods.

The aim of partial backtracking is to reduce the redundant work that might be done by the SAT solver on its redescent from the backtrack level L_{back}. The technique allows the solver to backtrack to any level j in the range $L_{back} \leq j \leq L_{deep}-1$ (where L_{deep} is the level the conflict was discovered). Nadel and Ryvchin [11] proposed to always backtrack chronologically to $L_{deep}-1$ while Möhle and Biere [8] returned to the proposal of [6] of flexibly backtracking to any level in the allowed range. Note that the new learnt 1-UIP clause $C_{\mathit{1\text{-}UIP}}$ is made unit at every level in this range. So after backtracking to level j the newly implied literal ℓ_{deep} is added to the trail with $\mathit{reason}(\ell_{deep}) = C_{\mathit{1\text{-}UIP}}$, and $\mathit{decLvl}(\ell_{deep})$ is set to L_{back} (the second deepest level in $C_{\mathit{1\text{-}UIP}}$).

This means that the decision levels on the trail are no longer contiguous, as ℓ_{deep} has a different level than the other literals at level j (if $j \neq L_{back}$). This change has a number of consequences for the SAT solver's operation, all of which were described in [6]. Möhle and Biere [8] showed that despite these consequences partial backtracking can be made to preserve the soundness of a CDCL solver.

3 Chronological Backtracking Effects on Search

In this paper we present a new technique that allows the SAT solver to use standard backtracking, but also allows saving some redundant work on its redescent. Our method has more overhead than C-bt so the first question that must be addressed is why not just use chronological backtracking.

Although C-bt is able to avoid a lot of redundant work it also has other effects on the SAT solver search. These effects are sometimes detrimental to the solver's performance and so it is not always beneficial to use C-bt. In fact, in both [11] and [8] it was found that fairly limited application of C-bt performed best. In [11] C-bt was applied only when the length of the standard backtrack, $L_{back} - L_{deep}$ was greater than a given threshold T. In their experiments they found that $T = 100$ was the best value, i.e., C-bt is done only on longer backtracks. In practice, this meant that C-bt was relatively infrequent; in our measurements with their solver only about 3% of the solver backtracks were C-bt backtracks. In [8] the value $T = 100$ was also applied. However, they introduced an additional technique to add some applications of C-bt when the length of the backtrack is less than T. This allowed [8] to utilize C-bt in about 15% of the backtracks.

Although it is difficult to know precisely why C-bt is not always beneficial, we can identify some different ways in which C-bt can affect the SAT solver's

search. With standard backtracking the literal ℓ_{deep} is placed on the trail at the end of L_{back} and then unit propagated. This could impact the trail in at least the following ways. First, some literals might become unit at earlier levels. This could include decision literals becoming forced which might compress some decision levels together. Second, different decisions might be made due to changes in the variable scores arising from the newly learnt clause. And third, literals might be unit implied with different reasons. C-bt can change all of these things, each of which could have an impact on the future learnt clauses, and thus on the solver's overall efficiency.

The second impact, changing variable scores, is partially addressed in [8] who utilize the ideas of [14] to backtrack to a level where the decisions would be unchanged. However, if the length of the backtrack is greater than 100 there could still be a divergence between the variable decisions generated in standard backtracking and C-bt. An argument is also given in [14] that the third impact, changing literal reasons, is not significant. However, the experiments in [14] were run before good notions of clause quality were known [1]. Our empirical results indicate that once clause quality is accounted for, changing the literal reasons can have a significant impact.

The first impact is worth discussing since it was mentioned in [6] but not in the subsequent works. This is the issue of changing the decision levels of literals on the trail. C-bt computes the decision level of each implied literal based on the decision levels of the literals in its reason, but it does not go backwards to change the decisions levels of literals earlier on the trail.

Example 1. For example, suppose that $(x, \neg y) \in \mathcal{F}$, the literal x is a decision literal on the trail with $decLvl(x) = 2$, and that the solver is currently at level 150 where it encounters a conflict. If this conflict yields the unit clause (y), standard backtracking would backtrack to level 0, where x would be implied. On redescent, x would no longer form a new decision level and it would not appear in any new clauses (as it is entailed by $\mathcal{F}$). C-bt, on the other hand, would backtrack to level 149. On its trail x would still be at level 2. Until a backtrack past level 2 occurs, learnt clauses might contain $\neg x$, and thus have level 2 added to their set of levels (potentially changing their LBD score). Only when backtrack past level 2 occurs would x be restored to its correct level 0, and it would require inprocessing simplifications to remove x from the learnt clauses.

In sum, although these impacts of C-bt on the SAT solver's search might or might not be harmful to the SAT solver, they do exist. In fact, there are two pieces of evidence that these impacts can sometimes be harmful. First, as mentioned above, previous work found that it is best to only apply C-bt on large backtracks where it has the potential to save the most work. If there were no harmful effects it would always be effective to apply C-bt. And second, in our empirical results below we show that our new trail saving technique, which always uses standard backtracking, can often outperform C-bt. Although our technique reduces the solver's work on redescent it does not completely eliminate it like C-bt does. Hence its superior performance can only occur if C-bt is sometimes harmful.

It is possible to combine C-bt with our trail saving technique to reduce the amount of work required whenever the solver performs non-chronological backtracking. However, C-bt greatly reduces the potential savings that could be achieved by our method since most of its non-chronological backtracks are relatively short (less than threshold T levels). In our preliminary experiments this combination did not seem promising.

Nevertheless, there is good evidence that C-bt can improve SAT solver performance.[2] Hence, it should be that it is better to perform C-bt in some branches. Hence, an interesting direction for future work would be to develop better heuristics about when to use C-bt in a branch and when to use standard backtracking augmented by our trail saving method.

4 Trail Saving

Our approach is to save the trail $\mathcal{T}$ on backtrack, and to use the saved trail $\mathcal{T}_{save}$ when the solver redescends to improve the efficiency of propagations without affecting the decisions the solver wants to make. The saved trail $\mathcal{T}_{save}$ also provides a secondary "lookahead mechanism" that the SAT solver can exploit as it redescends.

Suppose that the solver is at L_{deep} where it has encountered a conflict. From the 1-UIP clause it learns, $C_{1\text{-}UIP}$, it now has to backtrack to L_{back}. This is accomplished by calling BACKTRACK(L_{back}), shown in Fig. 1, which saves the backtracked portion of the trail.

Note that BACKTRACK does not save the deepest level of $\mathcal{T}$. The full $\mathcal{T}$ contains a conflict (at its deepest level). Hence the solver will never reproduce all the same levels, and it would be useless to save all of them. Note also that in addition to saving the literals in $\mathcal{T}_{save}$ we also save the clause reason of the unit implied literals in a separate $reason_{save}$ vector. Finally, we see that after backtrack the first literal on $\mathcal{T}_{save}$ is a decision literal: it is the first literal of $\mathcal{T}$ at decision level $L_{back}+1$. Literals will be removed from $\mathcal{T}_{save}$ during its use, but always in units of complete decision levels. So $\mathcal{T}_{save}[0]$ will always be a (previous) decision literal.

After backtrack the solver will add ℓ_{deep} to the end of the updated $\mathcal{T}$ with $reason(\ell_{deep}) = C_{1\text{-}UIP}$ and then invoke unit propagation. $\mathcal{T}_{save}$ is exploited during propagation by the version of PROPAGATE shown in Fig. 1, which will initially be invoked with the argument $\iota(\ell_{deep})$ (i.e., the trail index of the newly added implicant). The saved trail will be continually consulted during the solver's descent whenever unit propagation is performed. When backtrack occurs $\mathcal{T}_{save}$ will be overwritten to store the new backtracked portion of $\mathcal{T}$.

$\mathcal{T}_{save}$ is consulted in the procedure USESAVEDTRAIL (Fig. 1). This procedure tries to add saved implied literals and their reasons to the solver's trail, when

[2] C-bt can also be extremely useful in contexts where each descent can be very expensive, e.g., when doing theory propagation in SMT solving, or component analysis in #SAT solving. In these cases, C-bt, by avoiding backtracking and subsequent redescent, has considerable potential for improving solver performance.

1: BACKTRACK(L_{back})
2: $\forall \ell \in \mathcal{T}[[L_{back}{+}1 \ldots L_{deep}{-}1]]$ $reason_{save}(\ell) = reason(\ell)$
3: $\mathcal{T}_{save} = \mathcal{T}[[L_{back}{+}1 \ldots L_{deep}{-}1]]$
4: $\mathcal{T} = \mathcal{T}[[0 \ldots L_{back}]]$

1: PROPAGATE(idx)
2: **while** idx $< \mathcal{T}$.size()
3: $c \leftarrow$ USESAVEDTRAIL ()
4: **if** $(c \neq \varnothing)$ **return** c ▷ Found conflict from $\mathcal{T}_{save}$
5: $\ell \leftarrow \mathcal{T}[idx]$
6: **for** each clause $c \in$ watchlist($\neg\ell$)
7: **if** (c is unit implying x)
8: $\mathcal{T}$.addToEnd(x); $reason(x) \leftarrow c$
9: **else if** (c is falsified) **return** c ▷ Found conflict from unit prop.
10: **else** Update c's watches.
11: $idx{+}{+}$

12: USESAVEDTRAIL()
13: $idx \leftarrow 0$; $c \leftarrow \varnothing$
14: **for** (; $idx < \mathcal{T}_{save}$.size(); $idx{+}{+}$)
15: $l_{save} \leftarrow \mathcal{T}_{save}[idx]$
16: **if** $(reason_{save}(l_{save}) = \varnothing)$ ▷ Decision on $\mathcal{T}_{save}$
17: **if** $(l_{save} \in \mathcal{T})$ **continue** ▷ TRUE in solver, we can use its implied lits
18: **else break** ▷ Only solver can set decisions, so we stop here
19: **else** ▷ Implied Literal on $\mathcal{T}_{save}$
20: **if** $(l_{save} \in \mathcal{T})$ **continue** ▷ ignore redundant lits
21: **else if** $(\neg l_{save} \in \mathcal{T})$ ▷ contradiction, return conflict and reset idx
22: $c \leftarrow reason_{save}(l_{save})$; $idx \leftarrow 0$; **break**
23: **else** ▷ unset in solver, add to solver's trail
24: $\mathcal{T}$.addToEnd(l_{save})
25: $reason(l_{save}) \leftarrow reason_{save}(l_{save})$
26: **for** ($i \leftarrow 0$; $i < idx$; $i{+}{+}$) ▷ $\mathcal{T}_{save}$ is unchanged when $idx = 0$
27: $\mathcal{T}_{save}$.removeFront()
28: **return** c

Fig. 1. Using $\mathcal{T}_{save}$ in unit propagation and conflict detection

these implications are valid. We will show below that those implications that are added are in fact valid. We do not interfere with the solver's variable decisions. Instead we opportunistically test to see if literals implied on $\mathcal{T}_{save}$ are valid implications for the solver given the solver's current decisions.

$\mathcal{T}_{save}[0]$ is always a (previous) decision literal d with $reason_{save}(d) = \varnothing$. Note that, since new literals (e.g., ℓ_{deep}) have been added to $\mathcal{T}$, d might now be an implied literal on $\mathcal{T}$ (i.e., $reason(d) \neq \varnothing$) even though before the backtrack it was previously a decision (i.e., $reason_{save}(d) = \varnothing$). If d has not been assigned TRUE by the solver (i.e., $\neg d \in \mathcal{T}$), we cannot add any implied literals below it on $\mathcal{T}_{save}$ to $\mathcal{T}$ as these implied literals depend on d being assigned TRUE. In this case we stop looking for more literals to add to $\mathcal{T}$ (line 18).

$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	l_5^{3*}	l_6^3	l_7^{4*}	l_8^4	l_9^{5*}	l_{10}^5	l_{11}^5	l_{12}^{6*}	l_{13}^6	l_{14}^6
$\mathcal{T}_{save}$														
1-UIP $(\neg l_1, \neg l_3, \neg l_{12})$ and backtrack 2 (L_{back})														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2										
$\mathcal{T}_{save}$					l_5^*	l_6	l_7^*	l_8	l_9^*	l_{10}	l_{11}			
Unit Prop $\neg l_{12}$ (ℓ_{deep}), $\mathcal{T}_{save}$ not yet helpful														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	$\neg l_{12}^2$	l_7^2	l_9^2							
$\mathcal{T}_{save}$					l_5^*	l_6	l_7^*	l_8	l_9^*	l_{10}	l_{11}			
Make decision														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	$\neg l_{12}^2$	l_7^2	l_9^2	l_5^{3*}						
$\mathcal{T}_{save}$					l_5^*	l_6	l_7^*	l_8	l_9^*	l_{10}	l_{11}			
Now $\mathcal{T}_{save}$ can be used to augment trail														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	$\neg l_{12}^2$	l_7^2	l_9^2	l_5^{3*}	l_6^3	l_8^3	l_{10}^3	l_{11}^3		
$\mathcal{T}_{save}$														

Fig. 2. Use of $\mathcal{T}_{save}$ from Example 2. The literal's decision level is indicated in its superscript, and a * superscript indicates that the literal is a decision.

On the other hand if d has been made TRUE by the solver we can continue to add all of the implied literals below it (up to but not including the next decision literal on $\mathcal{T}_{save}$) to $\mathcal{T}$ (line 24), reusing their saved reasons. Any literals that have already been made true by the solver can be skipped (line 20). Finally, if we encounter a literal that has already been falsified by the solver, then its saved reason clause must be falsified by the solver and we can return it as a conflict (line 21). If a conflict is encountered we leave $\mathcal{T}_{save}$ unchanged by resetting idx to zero. Otherwise, idx will be the number of literals at the front of $\mathcal{T}_{save}$ that have been moved to $\mathcal{T}$ (or skipped over since they are already on $\mathcal{T}$). We then remove the first idx literals from $\mathcal{T}_{save}$ (line 27), and return the conflict (equal to $\varnothing$ if no conflict was found).

Example 2. Figure 2 provides an example of how $\mathcal{T}_{save}$ is used. Initially the literals l_1 to l_{14} are on the solver's $\mathcal{T}$, and $\mathcal{T}_{save}$ is empty. This is shown in the first two lines of the figure. In the figure the superscript on the literals indicates their decision level, and a superscripted $*$ indicates that the literal is a decision. Hence l_1^{1*} indicates that $decLvl(l_1) = 1$ and that l_1 is a decision.

Then a conflict is found at level 6 and the 1-UIP clause $(\neg l_1, \neg l_3, \neg l_{12})$ is learnt. Thus the solver will backtrack to level 2, where it will add $\neg l_{12}$ as a unit implicant. The next two lines show $\mathcal{T}$ and $\mathcal{T}_{save}$ right after the backtrack to level 2: the backtracked levels have been copied into $\mathcal{T}_{save}$ omitting the conflict level 6.

The new unit $\neg l_{12}$ is now added to $\mathcal{T}$ and unit propagation performed adding l_7 and l_9 to level 2. Since the first literal on $\mathcal{T}_{save}$, l_5, has $reason_{save}(l_5) = \varnothing$ (l_5 was a decision on $\mathcal{T}$ at the time backtrack occurred) and is not yet TRUE, $\mathcal{T}_{save}$ is not helpful at this stage. The status of $\mathcal{T}$ and $\mathcal{T}_{save}$ at this point is shown in the figure.

After unit propagation is finished the solver makes a new decision, which happens to be (but is not forced to be) l_5. Now $\mathcal{T}_{save}$ can be used: l_5 is TRUE so it is removed, l_6 is unassigned so it is added to $\mathcal{T}$, l_7 is TRUE and so removed,

l_8 is unassigned so it is added to $\mathcal{T}$, l_9 is TRUE and removed, and finally l_{10} and l_{11} are unassigned and so are added to $\mathcal{T}$. In this example, $\mathcal{T}_{save}$ is emptied, and cannot contribute more to $\mathcal{T}$.

All of these units are added to $\mathcal{T}$ before the solver starts to unit propagate l_5. Since, new literals have been added to $\mathcal{T}$ before l_5 the solver must propagate l_5 and all of the literals that follow it before making its next decision.

As noted in the previous example unit propagation has to be rerun on all saved literals added to $\mathcal{T}$ from $\mathcal{T}_{save}$. Thus our technique, unlike C-bt, does not completely remove the overhead of reproducing the trail on the solver's redescent. Nevertheless, trail saving improves the efficiency of this redescent in three different ways. First, by adding more forced literals to the trail before continuing propagating the next literal, propagation can potentially gain a quadratic speedup [2,5]. Second, propagation does not need to examine the reason clause of the added literals. If these literals were not added by useSavedTrail, propagation would have to traverse each of these reason clauses to determine that they have in fact become unit. Third, when a conflict is returned by useSavedTrail all further propagations can be halted. The added literals and their reasons will be sufficient to perform clause learning from the conflict returned by useSavedTrail. Since trail saving can sometimes save hundreds or thousands of literals at a time these savings can in sum be significant.

4.1 Correctness

Now we will prove that our use of $\mathcal{T}_{save}$ preserves the SAT solver's soundness. In particular, $\mathcal{T}_{save}$ is only used in the procedure useSavedTrail, in which it either adds new literals to the solver's trail, or returns conflict clauses to the solver. Hence, we only need to show that these new literals are in fact unit implied and the conflicts are in fact falsified by the solver's trail. Since both $\mathcal{T}$ and $\mathcal{T}_{save}$ are sequences of literals (with associated reasons) we can consider their concatenation denoted as $\mathcal{T} + \mathcal{T}_{save}$.

Theorem 1. *If $\mathcal{T} + \mathcal{T}_{save}$ is **reason sound** (Sect. 2) then the following holds. If the first i literals on $\mathcal{T}_{save}$ are all in $\mathcal{T}$ ($\forall j. 0 \leq j < i. \mathcal{T}_{save}[j] \in \mathcal{T}$) and $\mathcal{T}_{save}[i] = l$ is an implied literal with $reason_{save}(l) = C$, then C has been made unit by $\mathcal{T}$ implying l.*

Proof: Since $\mathcal{T} + \mathcal{T}_{save}$ is reason sound, every literal in C other than l appears negated before l in the sequence $\mathcal{T} + \mathcal{T}_{save}$. Thus for $x \in C$ we have $\neg x \in \mathcal{T}$ or $\neg x \in \mathcal{T}_{save}[0] \ldots \mathcal{T}_{save}[i-1]$. But in the later case we also have $\neg x \in \mathcal{T}$. □

This theorem shows that useSavedTrail's processing is sound. In this procedure, an implied literal from $\mathcal{T}_{save}$ is added to $\mathcal{T}$ (line 24) only when all prior literals on $\mathcal{T}_{save}$ are already on $\mathcal{T}$ (i.e., previously on $\mathcal{T}$ or already added to $\mathcal{T}$). Thus each new addition is sound given the inductive soundness of the previous additions, with the base case covered by Theorem 1. If l is to be added, the theorem shows that every other literal in $reason_{save}(l)$ has been falsified by $\mathcal{T}$.

Hence if l is also falsified by $\mathcal{T}$ then $reason_{save}(l)$ is a clause that is falsified by $\mathcal{T}$, thus it is a sound conflict for the solver.

Now we only have to show that $\mathcal{T} + \mathcal{T}_{save}$ is always **reason sound** during the operation of the solver.

Proposition 1. *If $\mathcal{T} + \mathcal{T}_{save}$ is reason sound then $\mathcal{T}' + \mathcal{T}'_{save}$ is reason sound in all of the following cases.*

1. $\mathcal{T}_{save}[0] \in \mathcal{T}$, $\mathcal{T}' = \mathcal{T}$, *and* $\mathcal{T}'_{save} = \mathcal{T}_{save}$.removeFront().
2. $\mathcal{T}' = \mathcal{T} + \mathcal{T}_{save}[0]$ *and* $\mathcal{T}'_{save} = \mathcal{T}_{save}$.removeFront().
3. $\mathcal{T}' = \mathcal{T} + \mathcal{T}_{new}$ *and* $\mathcal{T}'_{save} = \mathcal{T}'_{save}$ *and* $\mathcal{T}'$ *is reason sound.*
4. *We also have that* $\mathcal{T}$ *is reason sound if* $\mathcal{T}$ *was generated by the solver.*

Proof: (1) $\mathcal{T}_{save}[0]$ already appears earlier in the $\mathcal{T}$ so it can be removed without affecting the soundness of any reason following it. (2) is obvious as the sequence is unchanged. (3) the reasons in $\mathcal{T} + \mathcal{T}_{new}$ are sound by assumption. Those in $\mathcal{T}_{save}$ remain sound as they depend only on the literals in $\mathcal{T}$ and prior literals on $\mathcal{T}_{save}$, both of which are unchanged. (4) is obvious from the operation of unit propagation in the solver. □

Theorem 2. *$\mathcal{T}+\mathcal{T}_{save}$ is always reason sound during the operation of the solver.*

Proof: $\mathcal{T}_{save}$ starts off being empty, so $\mathcal{T} + \mathcal{T}_{save} = \mathcal{T}$ is reason sound as it was generated by the solver (4). In procedure BACKTRACK $\mathcal{T} + \mathcal{T}_{save}$ is set to a trail that was previously generated by the solver (4). The solver can add to $\mathcal{T}$ by decisions and propagations without using $\mathcal{T}_{save}$. In this case $\mathcal{T}' = \mathcal{T} + \mathcal{T}_{new}$, and $\mathcal{T}'$ is reason sound by (4), thus the new $\mathcal{T}' + \mathcal{T}_{save}$ is reason sound by (3). Finally, in procedure USESAVEDTRAIL either (a) literals at the front of $\mathcal{T}_{save}$ are discarded since they already appear on $\mathcal{T}$, or (b) literals are moved from $\mathcal{T}_{save}$ to $\mathcal{T}$. Under both of these changes $\mathcal{T} + \mathcal{T}_{save}$ remains reason sound by (1) and (2). □

4.2 Enhancements

We developed three enhancements of the base trail saving method described above. In this section we present these enhancements.

Saving the Trail over Multiple Backtracks. It can often be the case that when the solver finds a conflict and backtracks to L_{back} it might immediately find a another conflict at L_{back} causing a further backtrack. In the procedure BACKTRACK every backtrack causes $\mathcal{T}_{save}$ to be overwritten. Hence, in these cases most of the trail will not be saved—only the portion from the last backtrack. Our first extension addresses this potential issue and also provides more general trail saving in other contexts as well.

This extension is simply to add the latest backtrack to the front of $\mathcal{T}_{save}$ leaving all of the previous contents of $\mathcal{T}_{save}$ intact. Specifically, we replace line 3 of BACKTRACK by the new line:

3. $\mathcal{T}_{save} = \mathcal{T}[[L_{back}{+}1 \dots L_{deep}{-}1]] + \mathcal{T}_{save}$

It is not difficult to show that this change preserves soundness. Only Theorem 2 is potentially affected. However, we know that $\mathcal{T}_{save}$ is unchanged at the level at which a conflict occurs: either the conflict is detected without consulting $\mathcal{T}_{save}$ or if the conflict comes from $\mathcal{T}_{save}$ then useSavedTrail leaves $\mathcal{T}_{save}$ unchanged (line 21). Hence, at the level before the conflict occurred we have inductively that $\mathcal{T}[[0 \dots L_{deep}{-}1]] + \mathcal{T}_{save}$ was reason sound, and hence so is $\mathcal{T}' + \mathcal{T}'_{save}$ with $\mathcal{T}' = \mathcal{T}[[0 \dots L_{back}]]$ and $\mathcal{T}'_{save} = \mathcal{T}[[L_{back}{+}1 \dots L_{deep}{-}1]] + \mathcal{T}_{save}$.

When adding to the front of $\mathcal{T}_{save}$ in this manner $\mathcal{T}_{save}$ can grow indefinitely. So we prune $\mathcal{T}_{save}$ when it gets too large by (a) removing $\mathcal{T}_{save}[i]$ if $\mathcal{T}_{save}[i] = \mathcal{T}_{save}[j]$ for some $j < i$ ($\mathcal{T}_{save}[i]$ is redundant), and (b) removing the suffix of $\mathcal{T}_{save}$ starting at $\mathcal{T}_{save}[i]$ when $\mathcal{T}_{save}[j] = \neg\mathcal{T}_{save}[i]$ for some $j < i$ ($\mathcal{T}_{save}[i]$ will never be useful as its negation, $\mathcal{T}_{save}[j]$, would have to be added to $\mathcal{T}$ first). In this way $\mathcal{T}_{save}$ need never become larger than the number of variables in $\mathcal{F}$.

Lookahead for Conflicts. In useSavedTrail we stop adding literals from $\mathcal{T}_{save}$ to $\mathcal{T}$ once we reach a decision literal d on $\mathcal{T}_{save}$ that is not yet on $\mathcal{T}$ (line 18 of useSavedTrail). This is done so that the solver has full control over variable decisions without interference from the trail saving mechanism (unlike the case with C-bt). However, another option would be to force the solver to use d as its next decision literal, which would then allow us to further add all of d's implied literals on $\mathcal{T}_{save}$ onto $\mathcal{T}$. This can be done for the first k decisions on $\mathcal{T}_{save}$ for any k. But in general, we do not want to remove the solver's autonomy by forcing it to make potentially different decisions than it might have wanted to.

However, if there is a literal $l \in \mathcal{T}_{save}$ for which $\neg l \in \mathcal{T}$, we can observe that forcing the solver to make all of the decisions of $\mathcal{T}_{save}$ that lie above l will immediately generate a conflict in the solver: $reason_{save}(l)$ will be falsified. In fact, in this situation we would not even need to perform unit propagation over the literals added from $\mathcal{T}_{save}$; the literals and their reasons obtained from $\mathcal{T}_{save}$ would be sufficient to perform 1-UIP learning from $reason_{save}(l)$.

We experimented with this "lookahead for conflicts" idea using various values of k. We found that $k = 2$, i.e., forcing up to two decisions from $\mathcal{T}_{save}$ to be made by the solver if this yields a conflict, often enhanced the solver's performance. Limiting the lookahead to only one decision level of $\mathcal{T}_{save}$ was not as good, and looking ahead more than 2 decisions of $\mathcal{T}_{save}$ also degraded performance. This provides some evidence that taking too much control away from the solver and forcing it to make too many decisions from $\mathcal{T}_{save}$ can lead to conflicts that are not as useful to the solver.

Reason Quality. The saved trail can be thought of as remembering the solver's recent trajectory. Sometimes we want to follow the past trajectory, but perhaps sometimes we do not. In particular, when adding literals from $\mathcal{T}_{save}$ to $\mathcal{T}$ we can examine the quality of the saved reasons to see if they are worth using. Once we encounter a literal with a low quality saved reason we stop adding literals from $\mathcal{T}_{save}$ to the solver's trail. In particular, we can change lines 24–25 of useSaved-Trail to the following:

23.5 **if** (lowQuality($reason_{save}(l_{save})$)) **break**
24. $\mathcal{T}$.addToEnd(l_{save})
25. $reason(l_{save}) \leftarrow reason_{save}(l_{save})$

Note that the solver will still set the un-added literals as they are unit implied by $\mathcal{T}$, but it might be able to find better reasons for these implicants. There is of course no guarantee that better reasons will be found, but our empirical results show that sometimes this does happen. We experimented with two quality metrics, clause size and clause LBD, obtaining positive results with both. These results also provides evidence against the argument given in [14] that changing literal reasons is not impactful. With an appropriate clause quality metric the changing of literal reasons can have an impact.

5 Experiments and Results

We implemented our techniques in two different SAT solvers, MapleSAT and Cadical,[3] both of which have finished at or near the top of SAT competitions for the past several years [3,4]. We then ran each solver on the 800 total benchmark instances used in the main tracks of the 2018 SAT Competition and 2019 SAT Race. The experiments were executed on a cluster of 2.7 GHz Intel cores with 5000 s CPU time and 7 GB memory limits for each instance. We chose not to output or verify the proofs generated by any of the solvers. The Par-2 scores obtained and total instances solved by each solver are reported in Figs. 3, 5, and 6. We also show the cactus plot of the new version of cadical in Fig. 4.

In Fig. 3 we used the newest version of cadical (downloaded as of January 1, 2020) as the baseline solver, in Fig. 5 we used the version of cadical published in [8] as the baseline, and in Fig. 6 we used MapleLCMDist [7,15] as the baseline. Each of the baselines were run with standard non-chronological backtracking. We then refer to versions of each solver with additional features implemented on top by adding suffixes. "-chrono" refers to the solver with C-bt enabled (using the solver's default settings), "-trail" refers to the baseline with plain trail saving added (as described in Fig. 1), "-trail-multipleBT" refers to the baseline with trail saving plus the first enhancement of saving over multiple backtracks, "-trail-multipleBT-lookahead" also adds the enhancement of lookahead for conflicts by 2 decision levels, and "-trail-multipleBT-lookahead-reason" also adds the final enhancement to cease trail saving once a reason of "low quality" is reached. For more details on the enhancements, please see Sect. 4.2.

Interestingly, C-bt made the newest version of cadical perform worse than the baseline (Fig. 3). This demonstrates that C-bt is not always beneficial. Trail saving alone did not impact the performance of this solver significantly, but adding all of the enhancements on top of trail saving resulted in solving six more instances and yielding a better Par-2 score than the baseline. The key enhancement for this solver seemed to be the last one where we stop using the saved trail once we detect a reason of "low quality". We tried both clause size

[3] Our implementation is available at https://github.com/rgh000/cadical-trail.

and lbd as the clause quality metric, and both yielded a positive gain, with clause size being slightly more effective.

	Total	SAT	UNSAT	Avg. Par-2
cadical	532	314	218	4025
cadical-chrono	525	308	217	4086
cadical-trail	529	310	**219**	4060
cadical-trail-multipleBT	530	313	217	3930
cadical-trail-multipleBT-lookahead	531	313	218	4020
cadical-trail-multipleBT-lookahead-reason	**538**	**319**	**219**	**3854**

Fig. 3. Table of results for cadical, version pulled from github as of January 1, 2020.

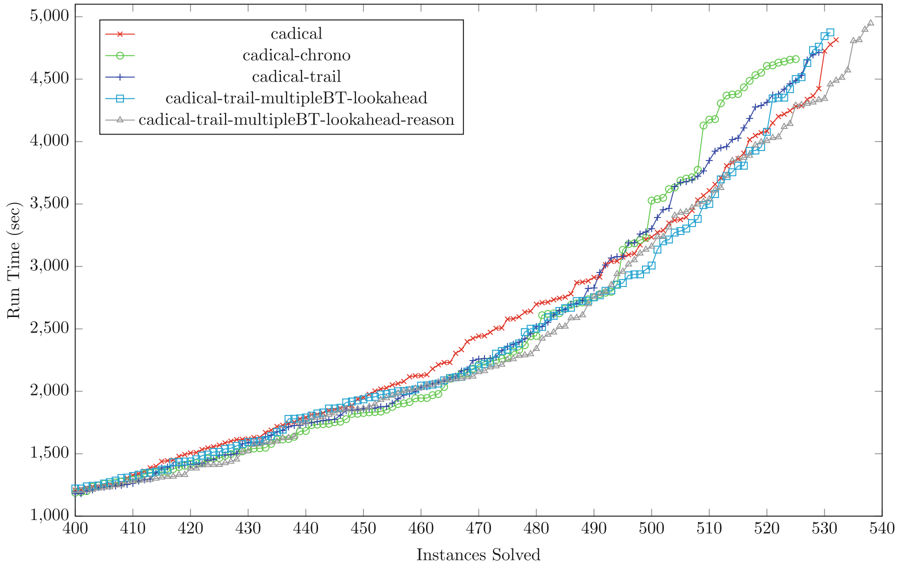

Fig. 4. Cactus plot for the newest version of cadical comparing standard non-chronological backtracking to C-bt and various configurations of trail saving. The first 400 problems were solved in less than 1200 s, so that part of the plot is truncated.

The version of cadical used in Fig. 5 did show benefits from C-bt in agreement with previously published results [8]. Trail saving alone did not significantly impact this solver, but adding all of the enhancements on top of trail saving resulted in solving the same number of instances as the solver with C-bt did, albeit with a slight increase in the Par-2 score.

MapleLCMDist (in Fig. 6) is another solver that benefited from C-bt. In this solver trail saving alone solved two more instances than the solver with C-bt

	Total	SAT	UNSAT	Avg. Par-2
cadical	492	289	203	4378
cadical-chrono	**498**	**292**	**206**	**4300**
cadical-trail	493	290	203	4403
cadical-trail-multipleBT-lookahead	496	**292**	204	4356
cadical-trail-multipleBT-lookahead-reason	**498**	**292**	**206**	4351

Fig. 5. Table of results for cadical or "chrono", version published in [8].

did. Adding the first two enhancements on top of trail saving resulted in solving only one more instance but yielded a better Par-2 score than the solver with C-bt. Adding the last enhancement of ceasing trail saving on a "low quality" reason made the performance worse, whether clause size or lbd was used as the clause quality metric. This suggests that the enhancements to trail saving have different impacts on different solvers.

	Total	SAT	UNSAT	Avg. Par-2
maple	458	259	199	4746
maple-chrono	470	271	199	4613
maple-trail	**472**	271	**201**	4618
maple-trail-multipleBT-lookahead	471	**272**	199	**4597**
maple-trail-multipleBT-lookahead-reason	469	271	198	4633

Fig. 6. Table of results for MapleLCMDist.

6 Conclusion

We have shown that our trail saving technique can speed up two state-of-the-art SAT solvers, cadical and MapleSAT, as or more effectively than chronological backtracking can. We also introduced three enhancements one can implement when using a saved trail and demonstrated experimentally that these enhancements can sometimes improve a solver's performance by a significant amount. We have shown that trail saving and all enhancements we proposed are sound.

There are many avenues that can be pursued in future work, such as using the saved trail to help make inprocessing techniques faster or using the saved trail to learn multiple clauses from a single conflict. It is also possible to combine trail saving with chronological backtracking, but it would require further work to determine whether or not this would be useful and how to best approach it.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, 11–17 July 2009, pp. 399–404 (2009). http://ijcai.org/Proceedings/09/Papers/074.pdf
2. Gent, I.P.: Optimal implementation of watched literals and more general techniques. J. Artif. Intell. Res. **48**, 231–251 (2013). https://doi.org/10.1613/jair.4016
3. Heule, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions. University of Helsinki (2018). http://hdl.handle.net/10138/237063
4. Heule, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Race 2019: Solver and Benchmark Descriptions. University of Helsinki (2019). http://hdl.handle.net/10138/306988
5. Hickey, R., Bacchus, F.: Speeding up assumption-based SAT. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 164–182. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_11
6. Jiang, C., Zhang, T.: Partial backtracking in CDCL solvers. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 490–502. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_33
7. Luo, M., Li, C., Xiao, F., Manyà, F., Lü, Z.: An effective learnt clause minimization approach for CDCL SAT solvers. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 703–711. ijcai.org (2017). https://doi.org/10.24963/ijcai.2017/98
8. Möhle, S., Biere, A.: Backing backtracking. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 250–266. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_18
9. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, 18–22 June 2001, pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
10. Nadel, A.: Understanding and improving a modern SAT solver. Ph.D. thesis, Tel Aviv University (2009)
11. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 111–121. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_7
12. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006). https://doi.org/10.1145/1217856.1217859
13. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-131
14. van der Tak, P., Ramos, A., Heule, M.: Reusing the assignment trail in CDCL solvers. JSAT **7**(4), 133–138 (2011). https://satassociation.org/jsat/index.php/jsat/article/view/89
15. Xiao, F., Luo, M., Li, C.M., Manya, F., Lü, Z.: MapleLRB LCM, Maple LCM, Maple LCM dist, MapleLRB LCMoccRestart and glucose-3.0+ width in SAT competition 2017. In: Balyo, T., Heule, M.J.H., Järvisalo, M.J. (eds.) Proceedings of SAT Competition 2017: Solver and Benchmark Descrptions, pp. 22–23. University of Helsinki (2017). http://hdl.handle.net/10138/224324