Trail saving in SMT

Milan Banković[1] and David Šćepanović[2]

[1] Faculty of Mathematics, University of Belgrade, Serbia
milan@matf.bg.ac.rs
[2] Faculty of Mathematics, University of Belgrade, Serbia
davidscepanovic96@gmail.com

Abstract

In this paper we discuss and evaluate the method of trail saving on backjumps in CDCL(**T**)-based SMT solvers. The method was originally proposed for CDCL-based SAT solvers at the SAT conference in 2020, showing a positive impact on solving SAT instances. Since a SAT solver tends to follow a similar path down the search tree after a backjump, saving the retracted portion of the trail enables speeding up the inference after the backjump by copying the saved inferred literals to the assertion trail, instead of re-propagating them by the unit-propagation mechanism. A similar behaviour may be expected when SMT solvers are concerned, but since the process of theory propagation within SMT solvers is usually even more expensive, the trail saving technique may potentially have even more significant impact in case of SMT instances. Although the experimental evaluation given in this paper shows some potential of the approach, the obtained results are generally mixed, and depend greatly on the chosen benchmark family, even within the same theory. Further analysis might be needed in order to better understand the behaviour of the method and its effects on the entire solving process.

1 Introduction

SAT and SMT solvers have become indispensable tools in the previous two decades, with applications spanning from hardware and software verification [4, 8, 17, 12], to proving mathematical theorems [9, 15] and solving different kinds of constraint satisfaction problems [5, 16]. SAT solvers decide on the satisfiability of a given propositional formula, typically in *conjunctive normal form* (CNF). Although quite simple, propositional language is powerful enough to enable a great number of problems from different areas to be expressed as problems of propositional satisfiability (or validity) and then solved by a SAT solver. This approach has showed tremendous success in the previous years, thanks to very efficient SAT solvers based on the *conflict-driven-clause-learning* (CDCL) algorithm [10]. On the other hand, SMT solvers decides on the satisfiability of a first-order formula, *modulo* some given *theory* of interest. Typical theories used in practical SMT solving are motivated by the main line of application of SMT solvers in the field of software verification, and include *equality with uninterpreted functions* (EUF), *linear real and integer arithmetic*, *theory of arrays*, *theory of bitvectors*, *theory of inductive datatypes*, and so on. Most modern SMT solvers uses a CDCL SAT solver for dealing with the propositional structure of a given first-order formula, while the theory reasoning is the responsibility of some dedicated theory-specific procedure [3].

New applications pose new challenges to SAT and SMT solvers, and there is a constant pressure to make SAT and SMT technology even more efficient and robust. Therefore, new implementational and algorithmic enhancements of the existing procedures used in SAT and SMT solvers have become a very popular research direction. One such improvement in the area of SAT solving was presented by Hickey and Bacchus [7] at the SAT conference in 2020. Their work considers saving the retracted portion of the *assertion trail* on each *backjump* (that is, non-chronological backtrack). Since SAT solvers tend to follow a similar path while redescending the

search tree after a backjump, the saved literals may be used to speed up that redescent, since they can be simply copied to the trail, instead of being re-propagated by the mechanism of *unit-propagation* [10] . This is especially significant in the context of non-chronological backtracking, because the retracted portions of the trail tend to be very large, and the solver must reproduce a large number of literals at a similar cost as before the backjump. Hickey and Bacchus showed that this approach had a positive impact on the overall performance of the SAT solvers that were used in their experiments [7].

The main goal of this work is to evaluate the potential of using the described technique within SMT solvers. The process of *theory propagation* [14], which is used as the inference mechanism within SMT solvers, is usually much more time-consuming, compared to the unit propagation of SAT solvers, because of more complex decision procedures (and the corresponding data structures) used for the theory reasoning. Therefore, the trail saving method may be even more efficient in the context of SMT solvers. On the other hand, implementing the trail saving method within an SMT solver may pose additional challenges, since a non-trivial modifications of the theory decision procedures might be needed in order to fully exploit the benefits of the technique. In this paper, we aim to discuss all these issues and also to provide an experimental evaluation that will give us some preliminary answers about the effectiveness of the proposed trail saving method in the area of SMT solving. Up to the authors' knowledge, such an analysis was not published in the literature so far.

The rest of the paper is organized as follows. In Section 2 we present the basic concepts and notions needed in the rest of the text. In Section 3 we describe the basic trail saving method, as presented in [7]. Section 4 considers the usage of the trail saving method within SMT solvers, and also discusses some challenges in that context. In Section 5 we discuss our implementation and present the experimental results obtained by our implementation. Finally, in Section 6 we give some conclusions and mention some further work directions.

2 Background

In this paper we assume the standard syntax and semantics of the first order logic with equality, adopting the terminology from [3]. A first order formula F is *satisfiable* with respect to a given theory $\mathbf{T}$ (or $\mathbf{T}$-*satisfiable*) if there is a *model* of $\mathbf{T}$ in which F is *true*. The problem of satisfiability of first order formulae with respect to a given theory is known as the *SMT problem*. We assume that formulae are in conjunctive normal form (CNF), that is, a formula is a conjunction of *clauses*, and each clause is a disjunction of *literals* (first order atoms or their negations). We also assume that all considered formulae are *ground*, i.e. do not contain variables.

We follow the usual approach for solving SMT problems, based on the CDCL($\mathbf{T}$)[1] scheme [14]. That is, we assume that the solver consists of a SAT solving core, implementing the CDCL algorithm [10], and a dedicated procedure called *a theory solver*, which decides on $\mathbf{T}$-satisfiability of the found propositional model, expressed as a conjunction of ground first order literals.

The notation used in the following text is mostly borrowed from [7]. The CDCL algorithm incrementally builds a propositional model of the given formula F, by assigning truth values to the literals of F. A partial propositional valuation v is represented by the *(assertion) trail* M, which is a stack of literals that are *true* in v. The trail M is partitioned into *decision levels* $M[[i]]$, based on the number of *decision literals* that precede each literal on M (that is, $M[[0]]$

[1] In older literature, the name DPLL($\mathbf{T}$) was used for the same procedure instead.

is the sequence of literals on M before the first decision literal, $M[[1]]$ is the sequence of literals starting from the first decision literal up to (but not including) the second decision literal, and so on). The topmost (current) decision level will be denoted by $M[[L_{deep}]]$. Whenever a decision is made, literals *inferred* from M and the clauses of F are pushed on M (the process known as *unit propagation*). The clause responsible for the inference of a literal l is called the *(propagation) reason* of l, and is denoted by $reason(l)$. If a clause becomes falsified, the process of *conflict analysis* uses the falsified clause (called the *conflict reason*) and the reasons of propagated literals on M to determine the real cause of the conflict (expressed as a *backjump clause* C_{back}) and *backjumps* to the appropriate level L_{back}, i.e. restores the trail to the state $M[[0]], M[[1]], \ldots, M[[L_{back}]]$ (denoted by $M[[0 \ldots L_{back}]]$ for short). The level L_{back} is the minimal level such that C_{back} is a *unit clause* with respect to $M[[0 \ldots L_{back}]]$ (i.e. it would trigger a unit propagation immediately after the backjump).

When a CDCL-based SAT solver is coupled with a theory solver for some first order theory $\mathbf{T}$, a conflict may also arise if the theory solver deduces that a subset R of literals from M is $\mathbf{T}$-unsatisfiable. Such a conflict is called a *theory conflict* (or $\mathbf{T}$-conflict), and R is a $\mathbf{T}$-*conflict reason* which is used as a starting point for conflict analysis. The theory solver may also $\mathbf{T}$-*infer* literals (i.e. discover literals of F that are $\mathbf{T}$-consequence of some subsets of literals from M). These literals are pushed on M (the process known as *theory propagation*). If a literal l is $\mathbf{T}$-inferred from the set $E \subseteq M$, then E is called *(*$\mathbf{T}$*-propagation) reason* of l. It is again denoted by $reason(l)$ and may be used during the conflict analysis in the future.

A formula is *unsatisfiable* if a conflict (propositional or $\mathbf{T}$-conflict) happens when $L_{deep} = 0$, and is *satisfiable* if all literals of F have assigned values, none of the clauses are falsified and there is no $\mathbf{T}$-conflict detected by the theory solver.

3 Trail Saving Algorithm

In this section we discuss the basic trail saving algorithm [7], in the context of a CDCL-based SAT solver. The algorithm maintains the list of saved literals retracted from the trail M, denoted by M_{save} (it is initially empty, by assumption). We denote the i-th literal in M_{save} as $M_{save}[i]$. At each backjump, the portion $M[[L_{back} + 1 \ldots L_{deep} - 1]]$ of the trail should be saved to M_{save}, i.e. the retracted part of the trail without the conflicting level $M[[L_{deep}]]$ (the conflicting level is not saved, since it would certainly produce the same conflict in the future). Together with the literals, their reasons must also be saved. The saved reason of a literal l will be denoted by $reason_{save}(l)$. In case of a saved decision literal l, we assume that $reason_{save}(l) = reason(l) = \varnothing$.

In order to preserve the correctness of the CDCL algorithm, the property of *reason soundness* [7] of the saved trail M_{save} must be maintained during the operation of the solver. This property guarantees that if the literals $M_{save}[0]$, $M_{save}[1]$,…,$M_{save}[i-1]$ are true in M, and $M_{save}[i]$ is a saved inferred literal, then all the literals from $reason_{save}(M_{save}[i])$ distinct from $M_{save}[i]$ itself are false in M. This means that it is safe to push $M_{save}[i]$ to M as an inferred literal (if it is undefined in M), or to report a conflict, if $M_{save}[i]$ is false in M.

In the basic variant of the trail saving algorithm [7], the current content of the list M_{save} is always *replaced* by $M[[L_{back} + 1 \ldots L_{deep} - 1]]$. An alternative, which is proposed as one of the enhancements in [7], is to *prepend* $M[[L_{back} + 1 \ldots L_{deep} - 1]]$ to the current state of M_{save}. In this paper, we consider this enhancement as an intergral part of the algorithm.

The basic functions of the trail saving algorithm [7] are given in Algorithm 1, and are discussed in the following paragraphs.

Algorithm 1 Trail saving algorithm

```
1:  SaveTrail(L_back)
2:      if L_deep = L_back^old then
3:          M_save ← ∅                          ▷ Case when concatenation may spoil reason-soundness
4:      L_back^old ← L_back                     ▷ Remember the backjump level
5:      M_save ← M[[L_back + 1...L_deep − 1]] + M_save   ▷ Conflicting L_deep level is dropped
6:      for all l_save ∈ M[[L_back + 1...L_deep − 1]] do   ▷ Saving reasons
7:          reason_save(l_save) ← reason(l_save)
8:      pivot ← 0
9:
10: UseSavedTrail()
11:     while pivot < |M_save| do               ▷ For each unprocessed literal l_save
12:         l_save ← M_save[pivot]
13:         if reason_save(l_save) = ∅ then     ▷ Saved decision literal
14:             if M ⊨ l_save then
15:                 pivot ← pivot + 1           ▷ Passing saved literal that is true on M
16:             else                            ▷ Decisions are not inferred by M
17:                 return ∅
18:         else                                ▷ Saved inferred literal
19:             if M ⊨ l_save then
20:                 pivot ← pivot + 1           ▷ Passing saved literal that is true on M
21:             else if M ⊨ ¬l_save then        ▷ Conflict found on M_save
22:                 return reason_save(l_save)
23:             else                            ▷ Push saved inferred literal to M
24:                 M ← M + l_save ;  reason(l_save) ← reason_save(l_save)
25:                 pivot ← pivot + 1
26:     return ∅
27:
28: ConfirmPropagatedSavedLiterals()
29:     while pivot > 0 do
30:         pop_front_literal(M_save)           ▷ Confirming M_save changes
31:         pivot ← pivot − 1
32:
33: FilterSavedTrail()
34:     M_new ← ∅                               ▷ Filtered version of M_save
35:     i ← 0
36:     while i < |M_save| do
37:         l_save ← M_save[i]
38:         if l_save ∉ M_new then              ▷ Non-duplicates are preserved in M_save
39:             M_new ← M_new + l_save
40:             if ¬l_save ∈ M_new then
41:                 break                       ▷ Cut the portion of M_save after the first conflicting literal
42:         i ← i + 1
43:     M_save ← M_new
```

Saving the trail. The function SaveTrail is invoked by the solver immediately before a backjump to the level L_{back} is performed. It first checks whether the prepending of the newly saved portion of the trail to M_{save} may spoil the reason soundness of M_{save}. As we will discuss later in more detail, this may happen only when a new conflict occurs immediately after a backjump, while we are still at the level L_{back}. Therefore, in such cases we must clear M_{save} before $M[[L_{back}+1 \dots L_{deep}-1]]$ is prepended, in order to stay on the safe side. For this reason, we keep track of the decision level of the previous backjump as L_{back}^{old}, and reset the saved trail if $L_{deep} = L_{back}^{old}$. After that, $M[[L_{back}+1 \dots L_{deep}-1]]$ is prepended to M_{save}. The function SaveTrail also saves the reasons of the saved literals. The variable *pivot* represents the position of the next-to-process literal in the list M_{save}. This variable is reset at the end of

the function SaveTrail.

Example 1. Assume that we have, among others, the following set of clauses: $\{\neg x_{16}, \neg x_{15}, \neg x_5\}$, $\{x_9, \neg x_2, x_8\}$, $\{\neg x_{13}, x_8, x_{10}\}$, $\{x_{19}, x_3, x_{20}\}$, $\{x_{11}, \neg x_1, x_{10}\}$, $\{x_{16}, \neg x_6, \neg x_{15}\}$, $\{x_{14}, x_8, x_{15}\}$, $\{\neg x_{12}, \neg x_2, x_{10}\}$, $\{\neg x_{17}, \neg x_{18}\}$, $\{\neg x_8, \neg x_7\}$, $\{\neg x_{20}, x_7\}$, $\{\neg x_{22}, \neg x_{23}\}$, $\{\neg x_{22}, \neg x_{24}\}$, $\{x_{22}, \neg x_{25}\}$, $\{x_{23}, x_{24}, \neg x_1\}$, $\{\neg x_4, \neg x_{19}, \neg x_{21}\}$, $\{x_{19}, x_8, \neg x_9\}$, $\{\neg x_1, x_{22}, x_{25}\}$. Assume that after the first several decisions the following state of the trail is reached:

$$M = x_0^d\ x_1^d\ x_2^d\ \neg x_3\ \neg x_4^d\ x_5\ x_6\ x_7^d\ \neg x_8\ x_9\ \neg x_{10}^d\ x_{11}\ \neg x_{12}\ \neg x_{13}\ \neg x_{14}^d\ x_{15}\ x_{16}$$

Note that the literals labelled with d in the exponent are the decision literals, so the current decision level is $L_{deep} = 7$. At that moment, a conflict is encountered with the clause $\{\neg x_{16}, \neg x_{15}, \neg x_5\}$. If the reason of $\neg x_{16}$ is the clause $\{x_{16}, \neg x_6, \neg x_{15}\}$, the conflict analysis gives us the backjump clause $\{\neg x_{15}, \neg x_5, \neg x_6\}$, and the backjump level $L_{back} = 4$. Before the backjump, the portion of the trail consisting of the levels 5 and 6 is saved, so after the backjump, we have the following state:

$$\begin{array}{lcl} M & = & x_0^d\ x_1^d\ x_2^d\ \neg x_3\ \neg x_4^d\ x_5\ x_6\ \neg x_{15} \\ M_{save} & = & x_7^d\ \neg x_8\ x_9\ \neg x_{10}^d\ x_{11}\ \neg x_{12}\ \neg x_{13} \end{array}$$

The reasons of saved inferred literals are also saved. For instance, we have $reason_{save}(x_9) = \{x_9, \neg x_2, x_8\}$, $reason_{save}(\neg x_8) = \{\neg x_8, \neg x_7\}$, $reason_{save}(x_{11}) = \{x_{11}, \neg x_1, x_{10}\}$, and finally, $reason_{save}(\neg x_{13}) = \{\neg x_{13}, x_8, x_{10}\}$ (we will need these saved reasons in the later examples).

Using the saved literals. The function UseSavedTrail uses the literals from the list M_{save} to speed up the inference. In the original algorithm [7], this procedure is invoked within the main propagation loop, before processing of each of the *watch lists* [11]. The function UseSavedTrail returns the conflict reason, if one is discovered during the processing of the saved trail, or $\varnothing$ if no conflict occurs. Note that saved decision literals may be passed during the processing of M_{save} only if they are already true in M – otherwise the processing of M_{save} is stopped and may be continued later, if the decision literal in question becomes true in M by some other mechanism (decision or propagation). On the other hand, saved inferred literal l_{save} are either passed (if l_{save} is already true in M), or pushed on M (if l_{save} is undefined in M), or the conflict is reported (if l_{save} is false in M). In the second two cases, $reason_{save}(l_{save})$ is used as a propagation (or conflict) explanation.

Example 2. Continuing the previous example, assume that after the next two decides, the following state is reached:

$$\begin{array}{lcl} M & = & x_0^d\ x_1^d\ x_2^d\ \neg x_3\ \neg x_4^d\ x_5\ x_6\ \neg x_{15}\ x_{17}^d\ \neg x_{18}\ \neg x_{19}^d\ x_{20}\ x_7\ \neg x_8^*\ x_9^* \\ M_{save} & = & x_7^d\ \neg x_8\ x_9\ \neg x_{10}^d\ x_{11}\ \neg x_{12}\ \neg x_{13} \end{array}$$

Note that the literal x_7 is now inferred (from the clause $\{\neg x_{20}, x_7\}$), and the literals $\neg x_8$ and x_9 are copied from M_{save}, once x_7 became true in M (we mark such literals with $*$ in the exponent). The next literal $\neg x_{10}$ cannot be copied from M_{save}, since it is a saved decision literal undefined in M, so it is not a consequence of M at this moment. Note also that M_{save} is not changed by the function UseSavedTrail, i.e. the literals $x_7, \neg x_8, x_9$ are still on M_{save}.

Confirmation of the propagated saved literals. The function ConfirmPropagatedSavedLiterals is invoked before each decision. Namely, as we have seen in the previous example, the literals from M_{save} that are pushed to M must remain on M_{save} until the propagation process at that level is exhausted, in order to maintain the reason-soundness of M_{save}

(otherwise, in case of a conflict, the SaveTrail function would drop the conflicting topmost level, possibly spoiling the reason-soundness of the remaining literals in M_{save}). When we are assured that there is no conflict at the topmost level, we may safely drop the processed prefix of M_{save}, just before the new decision is made.

Example 3. Continuing the previous example, recall that the literals x_7, $\neg x_8$ and x_9 are still on M_{save}, since their transfer to M is not confirmed yet. In the present state of the trail we again have a conflict, this time with the clause $\{x_{19}, x_8, \neg x_9\}$. During the conflict analysis, we first resolve the literals x_9 and $\neg x_8$ (using their saved reasons $\{x_9, \neg x_2, x_8\}$ and $\{\neg x_8, \neg x_7\}$), reaching the clause $\{x_{19}, \neg x_2, \neg x_7\}$, and then resolve the literal x_7 using its reason $\{x_7, \neg x_{20}\}$ (the resolvent being the clause $\{x_{19}, \neg x_2, \neg x_{20}\}$), and the literal x_{20} using its reason $\{x_{19}, x_3, x_{20}\}$, finally obtaining the backjump clause $\{x_{19}, \neg x_2, x_3\}$. The backjump level is now $L_{back} = 3$, and the new state is:

$$\begin{array}{lcl} M & = & x_0^d\ x_1^d\ x_2^d\ \neg x_3\ x_{19} \\ M_{save} & = & \neg x_4^d\ x_5\ x_6\ \neg x_{15}\ x_{17}^d\ \neg x_{18}\ x_7^d\ \neg x_8\ x_9\ \neg x_{10}^d\ x_{11}\ \neg x_{12}\ \neg x_{13} \end{array}$$

Note that the levels 4 and 5 from the trail M were prepended to M_{save} before the backjump. Note also why it is important to keep the used literals on M_{save} until their transfer to M is confirmed: since the conflicting level is never saved, if the literal $\neg x_8$ was not kept on M_{save} when it was copied to M, it would be lost after the backjump, and the literal $\neg x_{13}$ would stay on M_{save} with an unsound saved reason $\{\neg x_{13}, x_8, x_{10}\}$. Now assume that after the next two decides we have the following state:

$$\begin{array}{lcl} M & = & x_0^d\ x_1^d\ x_2^d\ \neg x_3\ x_{19}\ x_{21}^d\ \neg x_4\ x_5^*\ x_6^*\ \neg x_{15}^*\ x_{22}^d\ \neg x_{23}\ \neg x_{24} \\ M_{save} & = & x_{17}^d\ \neg x_{18}\ x_7^d\ \neg x_8\ x_9\ \neg x_{10}^d\ x_{11}\ \neg x_{12}\ \neg x_{13} \end{array}$$

Namely, after the decision x_{21}, the literal $\neg x_4$ was inferred from the clause $\{\neg x_4, \neg x_{19}, \neg x_{21}\}$, which once again unblocked the saved trail and the literals x_5, x_6 and $\neg x_{15}$ were copied from M_{save} to M. Just before the next decision x_{22}, the copied literals from M_{save} were *confirmed* on M, i.e. they were removed from M_{save}. Note that this is a safe operation – since these literals are now on the trail M, but not at its topmost level, they will either be saved to M_{save} or stay on M after the next backjump, preserving the reason soundness of the literals on M_{save}. The obtained state of M is again conflicting, and this time the conflicting clause is $\{x_{23}, x_{24}, \neg x_1\}$. After explaining $\neg x_{24}$ with the clause $\{\neg x_{22}, \neg x_{24}\}$, and $\neg x_{23}$ with the clause $\{\neg x_{22}, \neg x_{23}\}$, we obtain the backjump clause $\{\neg x_1, \neg x_{22}\}$. The backjump level is now $L_{back} = 2$, and we reach the following state:

$$\begin{array}{lcl} M & = & x_0^d\ x_1^d\ \neg x_{22} \\ M_{save} & = & x_2^d\ \neg x_3\ x_{19}\ x_{21}^d\ \neg x_4\ x_5\ x_6\ \neg x_{15}\ x_{17}^d\ \neg x_{18}\ x_7^d\ \neg x_8\ x_9\ \neg x_{10}^d\ x_{11}\ \neg x_{12}\ \neg x_{13} \end{array}$$

Note that the literals $\neg x_4, x_5, x_6, \neg x_{15}$ that were confirmed on M are now again on M_{save}.

Filtering the saved trail. Note that concatenation of saved portions of the trail may result in duplicates in M_{save}. As a consequence, the list M_{save} may grow indefinitely. Moreover, conflicting literals l and $\neg l$ may appear on M_{save}, so the portion of M_{save} after the two conflicting literals may be unreachable and, therefore, useless. For this reason, we should periodically filter the list M_{save}, by removing the duplicates and all the literals after the first conflicting literal in M_{save}. This is done by the function FilterSavedTrail. Note that the first conflicting literal

$\neg l_{save}$ is kept on M_{save}, since it may help in discovering a conflict. The procedure FilterSavedTrail is invoked whenever the length of M_{save} becomes greater than the total number of atoms of the formula F.

Preserving the reason soundness. It can be argued that the CDCL algorithm enhanced with the trail saving mechanism remains correct, provided that the reason soundness of M_{save} is maintained during the operation of the solver. For the proof of this fact, we refer the interested reader to the original work of Hickey and Bacchus [7]. Here we only discuss the unique case when the concatenation of the saved trails may spoil the reason soundness, which is the case when a new conflict happens immediately after a backjump, while we are still at the level L_{back}.[2] This may happen because $L_{deep} = L_{back}$ in such a case, so the level $M[[L_{deep}]] = M[[L_{back}]]$ was not saved on the previous backjump (since then it was the backjump level), and it will not be saved on the next backjump (since now it is the conflicting level). This can make saved reasons of some of the literals in M_{save} invalid after the next backjump.[3] The next (and final) example illustrates this phenomenon.

Example 4. In the state from the previous example, the clause $\{\neg x_1, x_{22}, x_{25}\}$ triggers the unit propagation of the literal x_{25}, which makes the trail M in conflict with the clause $\{x_{22}, \neg x_{25}\}$. We first resolve the literal x_{25} with its reason, and obtain the clause $\{x_{22}, \neg x_1\}$, and then resolve the literal $\neg x_{22}$ with its reason $\{\neg x_1, \neg x_{22}\}$, and obtain the backjump clause $\{\neg x_1\}$. The backjump level $L_{back} = 0$ this time, so after the backjump we reach the following state:

$$\begin{array}{lcl} M & = & \neg x_1 \\ M_{save} & = & x_0^d \end{array}$$

Note that the previous state of M_{save} was cleared before the new portion of M (the literal x_0) was prepended to it. This is because the conflict happened immediately after the backjump, before the next decide. In such a situation, the literals x_1 and $\neg x_{22}$ from the conflicting level were neither saved, nor they were kept on M, which might compromise the reason soundness of some literals on M_{save}. For instance, recall that the saved reason of the literal x_{11} was $\{x_{11}, \neg x_1, x_{10}\}$, and this reason is no longer sound, so the concatenation of the saved portions of the trail is not possible.

4 Employing trail saving in SMT

The described trail saving technique may be naturally extended to be used in the context of CDCL(**T**)-based SMT solvers, since such solvers are driven by a CDCL SAT solving engine. In this section, the CDCL(**T**) algorithm with the trail saving enabled is presented and discussed in detail. The overall structure of the CDCL(**T**) algorithm is not changed – the only thing that should be done to enable the trail saving is to invoke the trail saving functions described in the previous section at the appropriate places (which are mostly the same places as in the CDCL algorithm).

[2] This issue was not discussed by Hickey and Bacchus [7], but they did implement the appropriate check for such a condition in their solver, which means that they were aware of it.

[3] Note that there are cases when a conflict happens at L_{back}, but the reason-soundness still holds after the contatenation of the saved trails. This means that the resetting of the saved trail *each time* a conflict is encountered at L_{back} may be a quite conservative strategy. In our implementation, we try to recognize (some of) such cases in order to avoid unnecessary resets of M_{save}.

Algorithm 2 CDCL(**T**) algorithm with the trail saving enabled: SOLVE function

```
1:  SOLVE(F)
2:    while true do
3:      while M is changed do                          ▷ Unit and theory propagation loop
4:        C ← UNITPROPAGATE()                          ▷ Returns a conflicting clause, or ∅
5:        if C ≠ ∅ then                                ▷ If a conflict is found, exit the propagation loop
6:          break
7:        C ← THEORYPROPAGATE()                        ▷ Returns a theory conflict reason, or ∅
8:        if C ≠ ∅ then                                ▷ If a conflict is found, exit the propagation loop
9:          break
10:     if C ≠ ∅ then                                  ▷ A conflict is discovered during the propagation
11:       if L_deep = 0 then                           ▷ A conflict at the level 0
12:         return UNSAT
13:       (L_back, C_back) ← ANALYZECONFLICT(C, M)     ▷ C_back is the backjump clause
14:       SAVETRAIL(L_back)                            ▷ Save the retracted portion of the trail
15:       M ← M[[0...L_back]]                          ▷ Backjump the trail to the level L_back
16:       RESTORETHEORYSOLVERSTATE(L_back)             ▷ Notify the theory solver
17:       L_deep ← L_back
18:       if |M_save| is greater than the total number of atoms in F then
19:         FILTERSAVEDTRAIL()                         ▷ Filter M_save if needed
20:       F ← F ∪ {C_back}                             ▷ Learning the backjump clause triggers unit propagation
21:     else
22:       if all atoms from F are assigned in M then
23:         return SAT
24:       CONFIRMPROPAGATEDSAVEDLITERALS()             ▷ Confirm the changes of M_save
25:       l^d ← PICKBRANCHINGVARIABLE(M, F)
26:       M ← M + {l^d};  reason(l^d) ← ∅               ▷ Make a new decision
27:       L_deep ← L_deep + 1
```

The main loop of the CDCL(T) algorithm. The main loop of the algorithm is implemented in the function SOLVE (Algorithm 2). It first invokes UNITPROPAGATE and THEORYPROPAGATE procedures to do the inference. These procedures check for unit/theory propagations and conflicts, and will be discussed later in more detail. If a conflict is encountered, a conflict reason C is returned. In that case, the procedure SOLVE starts the conflict analysis, by invoking the ANALYZECONFLICT[4] function which returns the backjump clause C_{back} and the backjump level L_{back}. Before backjumping to the level L_{back}, the procedure first saves the part of the trail that will be retracted (by calling the function SAVETRAIL). Then it performs the backjump by restoring the trail to $M[[0..L_{back}]]$. It also tells the theory solver to restore its state to the level L_{back}, and performs the saved trail filtering if needed (by invoking the function FILTERSAVEDTRAIL). Finally, the backjump clause is learnt, triggering the next propagation cycle. On the other hand, if no conflict is encountered during the propagation, and there are still atoms from the formula F that are unassigned in M, we first confirm the changes of the saved trail by invoking the function CONFIRMPROPAGATEDSAVEDLITERALS. Then we pick a literal for the next decide and push it to M as a decision literal.

[4]The pseudo-code of the function ANALYZECONFLICT is omitted (as well as of several other functions), in order to save space, since it does not invoke any of the trail saving functions.

Propagation functions and using the saved trail. The functions UnitPropagate and TheoryPropagate which are responsible for the propagation are given in Algorithm 3.

Algorithm 3 CDCL(**T**) algorithm with the trail saving enabled: propagation functions

```
1:  UnitPropagate()
2:    while there are unprocessed literals on M do
3:      C ← UseSavedTrail()                                   ▷ First, try to use the saved trail
4:      if C ≠ ∅ then                          ▷ If a conflict is found, return the conflicting clause C
5:        return C
6:      l is next-to-process literal from M
7:      for all C ∈ watchlist(¬l) do                ▷ process the watchlist of the falsified literal ¬l
8:        l_alt ← FindAlternativeWatch(C, ¬l)
9:        if l_alt ≠ ∅ then                                   ▷ A new watch found
10:         l_alt replaces ¬l as a watch literal of C
11:         move C from watchlist(¬l) to watchlist(l_alt)
12:       else                    ▷ All literals of C are false, except, possibly, the other watch l'
13:         if the other watch l' of C is false in M then
14:           return C                                        ▷ C is a conflicting clause
15:         else if the other watch l' of C is undefined in M then
16:           M ← M + {l'}                                    ▷ Unit propagation
17:           reason(l') = C                                  ▷ C is the propagation reason
18:   return ∅
19:
20: TheoryPropagate()
21:   while there are unprocessed literals on M do
22:     C ← UseSavedTrail()                                   ▷ First, try to use the saved trail
23:     if C ≠ ∅ then                          ▷ If a conflict is found, return the conflicting clause C
24:       return C
25:     l_1, ..., l_n is n next-to-process literals from M                ▷ n is theory-dependant
26:     (C, M_inf) ← DoTheTheoryInference(l_1, ..., l_n)
27:     if C ≠ ∅ then          ▷ If a theory conflict is found, return the theory conflict reason C
28:       return C
29:     for all l in M_inf do                ▷ Otherwise, push the inferred literals M_inf to M
30:       M ← M + {l}
31:       reason(l) ← lazy                    ▷ The reason will be generated lazily, if needed
32:   return ∅
```

The function UnitPropagate implements the well-known *two-watched-literals* algorithm for unit propagation [11]. It processes the literals from M one by one in a loop. In each iteration of the loop, it first tries to use the literals from M_{save} if possible, by invoking the function UseSavedTrail. If no conflict is found, then the next unprocessed literal l is read from M and the watch list of its opposite literal $\neg l$ (which is false in M) is processed in the usual fashion. Note that for each propagated literal l, its reason (which is a clause of the formula F) is set.

We assume that the function TheoryPropagate has a similar global structure, although its precise structure may depend on a concrete theory-specific procedure. It also processes the literals from M in a loop, but not necessarily one by one (depending on a concrete procedure, it might process all unprocessed literals at once). The function UseSavedTrail is invoked once per the loop iteration, before the processing of the literals. The theory-specific inference is captured by the function DoTheTheoryInference, which, by assumption, returns either a theory conflict reason C, or the set of literals M_{inf} that are **T**-inferred from the prefix of M up to the literal l, in case when no theory conflict is encountered. In the latter case, the literals from M_{inf} are pushed to M. On the other hand, their reasons are not set, since the reasons of **T**-inferred literals are usually generated *lazily*, i.e. only when needed, during the conflict

analysis. Thus, we set $reason(l)$ to a special value $lazy$, which serves as a placeholder for the true reason which may be calculated later by the theory solver, if needed.

An issue that deserves a discussion here concerns the exact location and the frequency of invocation of the function UseSavedTrail within the propagation functions. In case of the unit-propagation, we exactly followed the approach from the original trail saving algorithm, presented by Hickey and Bacchus [7]. Moreover, we extended the same approach to the case of the theory propagation. That is, in both cases, the function UseSavedTrail is invoked once per iteration of the propagation loop, before the literals from M are processed.

The question remains whether this approach is the best approach. The rationale behind the approach is that the saved trail should be consulted whenever there are new literals on M, since some of them may unblock the next portion of M_{save} (recall that the consumption of M_{save} is stopped when a saved decision literal l that is undefined in M is encountered). Although we cannot guarantee that each iteration of the loop will indeed produce new literals on M, it will be the case very often in practice. On the other hand, too frequent invocation of the function UseSavedTrail should not be an issue, since its unfruitful calls are not expensive (the function will return immediately if the next literal on M_{save} is a saved decision literal undefined in M). For this reason, we decided to stick to that approach. Further analysis of this issue is left for the future work.

Saving the theory propagation reasons. Another issue concerns saving of explanations of **T**-inferred literals. Recall that the function SaveTrail must save the propagation reasons of the inferred literals, together with the literals themselves. In case of unit-propagated literals, this is easy, since their reasons are the clauses of the formula F (or some of the learned clauses), and they are immediately available (recall that they are set in the function UnitPropagate immediately after the propagation). On the other side, the reasons of the **T**-inferred literals may not be known yet at the moment when the function SaveTrail is invoked, since we already said that the theory solvers usually produce reasons of propagations *lazily*, i.e. only when a reason is required during the conflict analysis (recall that the function TheoryPropagate uses the special value $lazy$ to indicate that the reason will be calculated later, on demand). This problem can be resolved in two ways:

- The theory solver may be asked to produce the reasons of *all* saved **T**-inferred literals *eagerly*, when the trail is saved during the execution of the SaveTrail function. The produced reasons are then saved in the same way as the reasons of the unit-propagated literals. This approach is very easy to implement, since it does not require any modification of the theory solver. On the other hand, producing all the reasons eagerly may be too expensive, since only a portion of them will be actually needed during the conflict analysis.

- Instead of saving the reason of a **T**-inferred literal l, we may only save the information that the theory solver is responsible for producing the reason of l later.[5] The reason will be produced *lazily*, when it is needed during the conflict analysis. In other words, the idea is to follow the usual approach used in SMT. The main problem with this approach is that the theory solver may not be able to produce the reason of the literal that is, technically, not pushed to M as a **T**-inferred literal during the last propagation cycle, but it is copied from M_{save} as a saved **T**-inferred literal from some previous propagation cycle. This means that the internal state of the theory solver has been changed meanwhile, possibly

[5] In case of multiple theory solvers, when the combination of theories is considered, this information should contain a reference to the theory solver responsible for producing the reason of l in the future.

multiple times over consecutive backjumps. This also holds for the data structures within the theory solver that are used by the reason-generating procedures. Such data structures may not contain information needed for explaining the literals that are *not propagated* to M by the theory solver in that propagation cycle, even if those literals *are* indeed **T**-consequences of the trail M. In order to resolve this issue, non-trivial modification of the theory solver's data structures may be needed, making this approach much more difficult to implement.

In our work we follow the first approach, due to its simplicity. Examining the second approach is left for the future work.

5 Implementation and Evaluation

For the purpose of evaluation, the trail saving method described in previous sections is implemented within the SMT solver `argosmt`[6], which is an open-source CDCL(**T**)-based SMT solver implemented in the `C++` programming language for research purposes. The solver includes two theory solvers:

- The EUF theory solver based on the *congruent closure* procedure described in [13]. This procedure maintains a data structure called the *proof forest*, which is used for generation of propagation (and conflict) reasons. The state of this data structure is restored when a backjump is performed, so it cannot be used for explaining the propagations from the previous propagation cycles. Therefore, during the trail saving operation, this theory solver must produce all the reasons of the saved literals eagerly, and this is what is done in our implementation. Notice that the eager reason generation approach may induce significant overhead in this case, since the generation of the propagation reasons within the EUF theory solver may be time-consuming.
- The linear arithmetic theory solver based on the *simplex* procedure [6]. This theory solver is optimized for real arithmetic, but it also includes some basic support for integer constraints (based on the *branch-and-bound* method [6]). Note that this theory solver may be used for solving the *difference logic* problems, too. Unlike the EUF theory solver, the reasons for theory propagations produced by the simplex-based theory solver are much cheaper to generate. Therefore, the expectations are that the overhead of the eager reason generation approach will not be as significant in this case.

We evaluated the implementation on SMT-LIB [2] instances. All instances from all non-incremental `QF_LRA`, `QF_RDL` and `QF_UF` benchmark families were included in the evaluation.[7] The experiments were run on a computer with four AMD Opteron 6168 1.6GHz 12-core processors (that is, 48 cores in total), and with 94GB of RAM.

For comparison, each instance was evaluated with and without trail saving enabled (the two versions of the solver are denoted by `argosmt-ts` and `argosmt-nts`, respectively). In both cases, time limit per instance was 1200 seconds. After the results were obtained, several benchmark families were excluded from the further analysis, either because they were too easy for our solver (all instances were solved in less then a second on average by both `argosmt-nts`

[6]The solver is available at github: https://github.com/milanbankovic/argosmt. To experiment with the trail saving, checkout the branch `ts` and then follow the instructions in the file `README.md`.

[7]Instances ara available at: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks

and `argosmt-ts`) or too hard (no instance was solved in the given time limit by neither of the solver's variants).[8] The final results are presented in Table 1.

Family	Logic	# instances	# solved (nts)	avg. time (nts)	# solved (ts)	avg. time (ts)	% deep bjmps	# saved levels	# saved lits	% saved props
uart	*QF_LRA*	*73*	**61**	**508**	*59*	*545*	*81.9*	*11*	*252*	*16.7*
Heizmann	QF_LRA	58	9	2063	9	**2055**	89.5	10.7	52.4	18.1
tta_startup	*QF_LRA*	*72*	**64**	**321.3**	*62*	*376.4*	*76.5*	*11.4*	*430*	*12.3*
miplib	QF_LRA	42	10	1861.5	10	**1853.5**	77.3	7.6	142.8	5.68
DTP-Sched	*QF_LRA*	*91*	*89*	**62**	*89*	*62.3*	*99.3*	*34.3*	*148.8*	*8.8*
clock_synchro	QF_LRA	36	25	749.5	**27**	**658.4**	62.5	10.9	83	12
latendresse	*QF_LRA*	*18*	*15*	*404.7*	*15*	*404.7*	*100*	*14.6*	*32.6*	*0.42*
2019-ezsmt	QF_LRA	105	72	796.1	**73**	**785.5**	82.9	8.07	173.4	4.05
TM	*QF_LRA*	*25*	*19*	*588.8*	**20**	**538.7**	*61*	*7.9*	*180.3*	*8.1*
SV-COMP	QF_LRA	94	21	1959.7	**27**	**1834.6**	88.9	49.3	214.7	11.8
CoopT2	*QF_LRA*	*202*	**54**	**1879**	*48*	*1962.8*	*91*	*86.7*	*426*	*10.5*
sc	QF_LRA	144	132	**246.9**	132	247.1	85	14	205.8	14
Ultimate	*QF_LRA*	*123*	*42*	**1710.1**	**44**	*1713.7*	*92*	*366.7*	*818.1*	*8.5*
Total	**QF_LRA**	1083	613	1113.6	**615**	**1112.5**	84.1	51.1	287.7	10.7
sal	QF_RDL	60	42	765.5	42	**764.8**	46.3	5.5	231.6	7.2
scheduling	*QF_RDL*	*106*	*51*	*1306*	**56**	**1192**	*95.1*	*15.8*	*56.7*	*19.8*
skdmxa	QF_RDL	36	**17**	**1462**	16	1513	61.7	69.4	2472	12.3
temporal	*QF_RDL*	*51*	*49*	*167.2*	*49*	**159.9**	*95.8*	*87*	*235.8*	*10.3*
Total	**QF_RDL**	253	159	970.6	**163**	**928.5**	80.1	40.5	395.7	13
eq_diamond	QF_UF	100	**22**	**1887**	21	1907	17	1.2	2.5	0.9
hwbench	*QF_UF*	*773*	**769**	**17.1**	*766*	*33.4*	*39.7*	*14.6*	*286.4*	*5.5*
QG-class	QF_UF	6396	**6334**	**34.7**	6321	43.7	49.4	4.7	331.2	20.7
SEQ	*QF_UF*	*56*	**48**	**411.5**	*41*	*707.9*	*59.2*	*18.6*	*56.4*	*20.5*
NEQ	QF_UF	48	**34**	**818.9**	26	1249	51.2	58	120.6	11.7
PEQ	*QF_UF*	*47*	**27**	**1162**	*21*	*1423*	*44.7*	*4*	*13.5*	*14.4*
Total	**QF_UF**	7420	**7234**	**72.9**	7196	89.3	48.6	5.6	323.8	19

Table 1: The results of the evaluation of the trail saving method on the selected SMT-LIB benchmark families. The left side of the table shows the numbers of solved instances and average solving times for `argosmt-nts` and `argosmt-ts`. Times are given in seconds. Time limit per instance was 20 minutes. For unsolved instances, twice the timeout was used when the average solving time was calculated. The winners are printed in boldface. The right side of the table shows some trail saving statistics, on average per instance (percent of backjumps by more than one level, average numbers of saved levels and saved literals per SaveTrail call, percents of propagations originating from the saved trail).

The results show that the trail saving technique has a positive effect in case of some `QF_LRA` benchmark families, but there are also `QF_LRA` benchmark families for which the trail saving algorithm does not improve the solver's performance (either it does not have any effect, or even induces some degradation of performance). On average, there is no significant difference between `argosmt-nts` and `argosmt-ts` on `QF_LRA` instances. On the other hand, the results

[8] Too easy: *keymaera*, *meti-tarski*, *sal*, *spider_benchmarks* (QF_LRA), *CLEARSY*, *Rodin* (QF_UF). Too hard: *tropical-matrix* (QF_LRA).

are positive in general for QF_RDL instances, where argosmt-ts performs better on three of four benchmark families. Finally, the trail saving technique exhibits a bad performance on QF_UF instances on all benchmark families. These claims are also supported by the plots given in Figure 1.

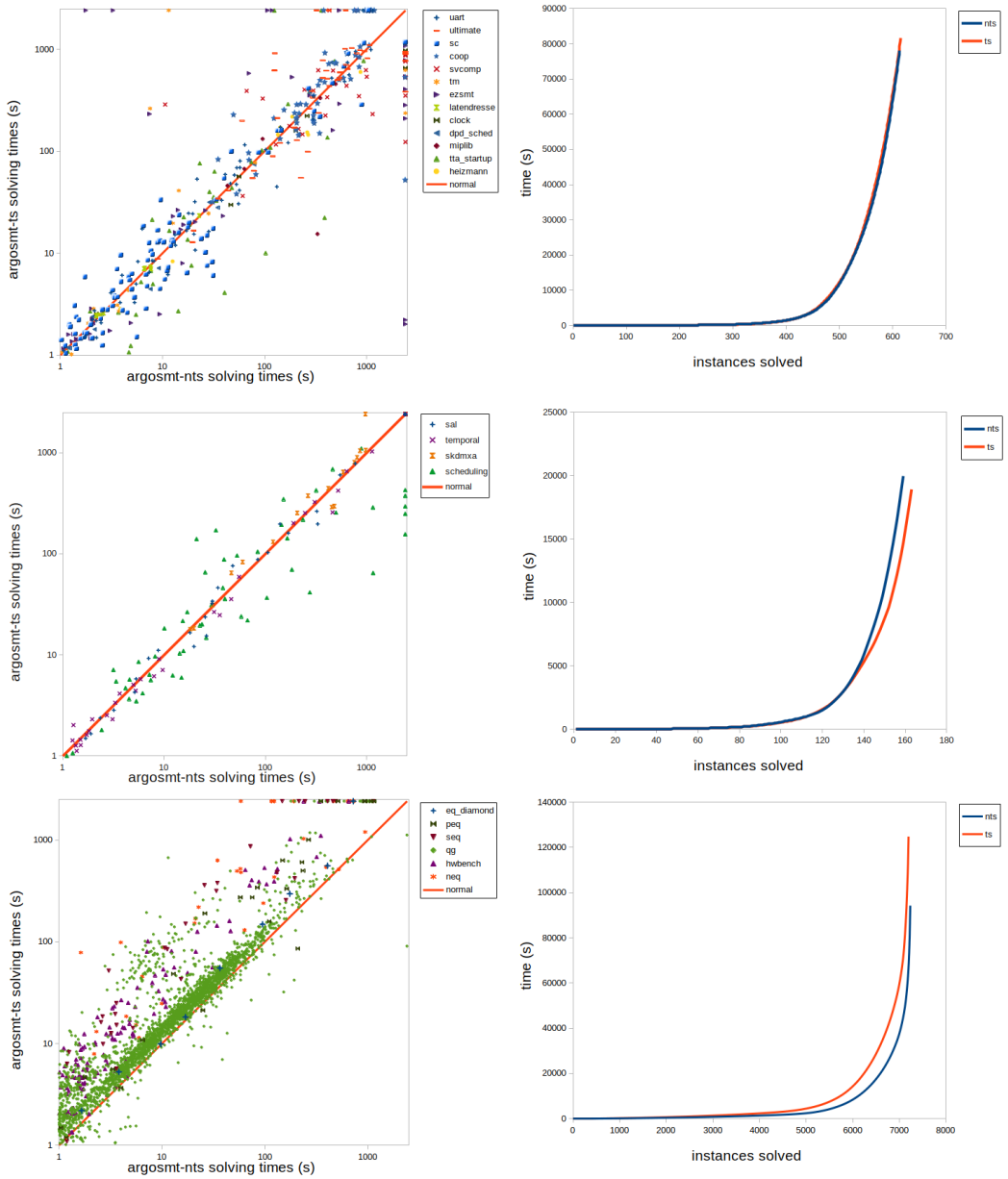

Figure 1: Per instance scatter plots (left) and survival plots (right) for: QF_LRA (top), QF_RDL (middle) and QF_UF (bottom). Times are given in seconds.

The consistently bad performance on `QF_UF` instances is somewhat expected, and may be explained by the expensiveness of the eager reason generation in case of the EUF theory solver. Moreover, the EUF solver is not expected to benefit much from the saved literals, since it must process all these literals again and perform merging of the corresponding congruence classes anyway [13].

On the other hand, prior to the experiments it was expected that the simplex-based arithmetic solver would benefit from the trail saving. Indeed, not only the eager reason generation is cheap, but we can also expect that copying the saved literals from M_{save} instead of propagating them again by the theory solver might help in avoiding some very expensive *pivoting* steps within the simplex procedure [6]. However, such a hypothesis is not supported by our experimental results, since the method does not introduce any consistent improvement on `QF_LRA` benchmarks. Another hypothesis was that the obtained performance improvement should be proportional to the degree of utilization of the saved trail. In order to test this hypothesis, we calculated some statistics concerning the trail saving (also presented in Table 1), such as how often the trail was saved on backjumps (that is, the percent of backjumps for more than one level), and how many levels and literals were saved on average per backjump. Also, the last column in Table 1 shows the average percent of propagations that actually came from the saved trail, which is probably the most accurate measure of the saved trail's utilization during the operation of the solver. Suprisingly, we did not find any correlation between any of these features and the performance of the trail saving algorithm, neither at the per-family level nor at the per-instance level.

In addition to the main experimental evaluation presented in this section, we have also evaluated our solver on all `QF_UFLRA` benchmark categories, in order to see how the trail saving method performs when the combination of the two theories is considered. These results are not presented here, since there were not almost any difference in behaviour between `argosmt-nts` and `argosmt-ts` on `QF_UFLRA` instances.

We also did some preliminary evaluation on `QF_LIA` instances, but taken only from several randomly selected benchmark families, which was not sufficient to draw some general conclusions. However, the obtained results suggest the similar behaviour as on `QF_LRA` instances (that is, the behaviour greatly depends on the benchmark family[9]).

Finally, let us mention that Hickey and Bacchus in their original work [7] also experimented with two additional enhancements of the basic trail saving method (one of them considers the quality of saved reasons and the other considers using the saved decisions as lookaheads for conflicts). We have implemented and evaluated the same enhancements in our solver, but contrary to the results of Hickey and Bacchus, our evaluation on the selected SMT-LIB instances did not show any significant impact of these enhancements.

6 Conclusions and Further Work

In this paper, we have discussed the potential of using the method of trail saving within SMT solvers. For the purpose of evaluation, we have implemented the trail saving method within a CDCL($\mathbf{T}$)-based SMT solver `argosmt`, equipped with two theory solvers which are among the most commonly implemented within modern SMT solvers – the congruence closure EUF theory solver, and simplex-based linear arithmetic theory solver. Our experimental results show that the trail saving exhibits a consistently bad performance on EUF instances, which is probably caused by the expensiveness of the eager reason generation strategy. On the other hand, the

[9] For example, trail saving performed well on the *convert*, *rings* and *tropical_matrix* benchmark families, but not on the *slacks*, *mathsat* and *wisa* families.

effectiveness of the trail saving on linear arithmetic instances greatly depends on the chosen benchmark family, and further investigation is needed in order to discover the exact traits of the benchmarks that determine the behaviour of the trail saving. Our first assumption that the efficiency gains should be proportional to the degree of the utilization of the saved trail was not supported by the obtained experimental results. This means that the main effect of the trail saving may not be in speeding up the propagation, which was our starting assumption. The main effect is probably the influence on the search, since the saved reasons influence the conflict analysis and the learned backjump clauses. The main line of the further work is, therefore, to perform a more detailed analysis of these effects. Another further research direction is to evaluate the method on other important theories used in SMT (such as theory of arrays, bitvectors and so on). Finally, since the implementation details may significantly influence the obtained experimental results, it might be needed to implement the technique within some state-of-the-art SMT solver, in order to more reliably evaluate the potential of the trail saving method in SMT solving.

References

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[3] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press, 2009.

[4] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. In *Handbook of satisfiability*, volume 185, pages 457–481. IOS Press, 2009.

[5] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012.

[6] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006.

[7] Randy Hickey and Fahiem Bacchus. Trail saving on backtrack. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 46–61. Springer, 2020.

[8] Daniel Kroening. Software verification. In *Handbook of Satisfiability*, pages 505–532. IOS Press, 2009.

[9] Filip Marić. Fast formal proof of the Erdős–Szekeres conjecture for convex polygons with at most 6 points. *Journal of Automated Reasoning*, 62(3):301–329, 2019.

[10] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, chapter 4, pages 131–155. IOS Press, 2009.

[11] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Annual ACM IEEE Design Automation Conference*, pages 530–535. ACM, 2001.

[12] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. Hardware verification using software analyzers. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 7–12. IEEE, 2015.

[13] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

[14] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[15] Manfred Scheucher. Two disjoint 5-holes in point sets. *Computational Geometry*, 91:101670, 2020.

[16] Naoyuki Tamura and Mutsunori Banbara. Sugar: A CSP to SAT translator based on order encoding. *Proceedings of the Second International CSP Solver Competition*, pages 65–69, 2008.

[17] Milena Vujošević-Janičić and Viktor Kuncak. Development and evaluation of LAV: an SMT-based error finding platform. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 98–113. Springer, 2012.