Stochastic Local Search for SMT: Combining Theory Solvers with WalkSAT [⋆]

Alberto Griggio[1], Quoc-Sang Phan[2], Roberto Sebastiani[2] and Silvia Tomasi[2]

[1] FBK-Irst, Trento, Italy
[2] DISI, University of Trento, Italy

Abstract. A dominant approach to Satisfiability Modulo Theories (SMT) relies on the integration of a Conflict-Driven-Clause-Learning (CDCL) SAT solver and of a decision procedure able to handle sets of atomic constraints in the underlying theory $\mathcal{T}$ ($\mathcal{T}$*-solver*). In pure SAT, however, Stochastic Local-Search (SLS) procedures sometimes are competitive with CDCL SAT solvers on satisfiable instances. Thus, it is a natural research question to wonder whether SLS can be exploited successfully also inside SMT tools.
In this paper we investigate this issue. We first introduce a general procedure for integrating a SLS solver of the WalkSAT family with a $\mathcal{T}$*-solver*. Then we present a group of techniques aimed at improving the synergy between these two components. Finally we implement all these techniques into a novel SLS-based SMT solver for the theory of linear arithmetic over the rationals, combining UBCSAT/UBCSAT++ and MathSAT, and perform an empirical evaluation on satisfiable instances. The results confirm the potential of the approach.

1 Introduction

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a (typically quantifier-free) first-order formula with respect to some decidable theory $\mathcal{T}$. A dominant approach to SMT, called *lazy approach*, relies on the integration of a Conflict-Driven Clause-Learning (CDCL) SAT solver and of a decision procedure able to handle sets of atomic constraints in the underlying theory $\mathcal{T}$ ($\mathcal{T}$*-solver*) (see, e.g., [13, 5]). In pure SAT, however, Stochastic Local-Search (SLS) procedures (see [11]) sometimes are competitive with or even outperform CDCL SAT solvers on satisfiable instances, in particular when dealing with unstructured problems. Therefore, it is a natural research question to wonder whether SLS can be exploited successfully also inside SMT tools. In this paper we start investigating this issue.

Remarkably, CDCL and SLS SAT solvers are very different in the way they perform search. CDCL SAT solvers reason on *partial* truth assignments, which are updated in a stack-based manner. Moreover, they intensively use techniques like *boolean constraint-propagation (BCP)*, *conflict-directed backtracking (backjumping)* and *learning*, which

⋆ A. Griggio is supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION. R. Sebastiani is supported in part by SRC/GRC under Custom Research Project 2009-TJ-1880 WOLFLING. We wish to thank H. Hoos, D. Tompkins, A. Belov and Z. Stachniak for their help with their tools and for useful insights.

are heavily exploited in the lazy-SMT paradigm and allow for very-efficient SMT optimization techniques like *early pruning*, *theory-propagation*, *theory-driven backjumping and learning* (see [13, 5]). SLS SAT solvers, instead, reason on *total* truth assignments, which are updated by swapping the phase of single literals according to some mixed greedy/stochastic strategy. Moreover, they typically do not use BCP, backjumping and learning. Therefore, the problem of an effective integration of a $\mathcal{T}$-solver with a SLS SAT solver is not a straightforward variant of the standard integration with a CDCL solver in lazy SMT. Moreover, the standard SMT optimization techniques mentioned above cannot be applied in a straightforward way.

In order to cope with these problems, we perform the following steps. First, inspired by the idea of "partially-invisible" SAT formulas, we present a novel and general architecture for integrating a $\mathcal{T}$-*solver* with a Boolean SLS solver based on the widely-used WalkSAT algorithm, resulting in a basic SLS-based SMT solver, which we call WALKSMT. Second, we analyze the differences between the interaction of a $\mathcal{T}$-solver with a CDCL-based and a SLS-based SAT solver, and we introduce and discuss a group of optimization techniques aimed at improving the synergy between an SLS solver and the $\mathcal{T}$-solver. Third, we present an implementation of WALKSMT with the optimization techniques above, which is based on the integration of the UBCSAT [17] and UBCSAT++ [6] SLS solvers with the $\mathcal{LA}(\mathbb{Q})$-solver of MATHSAT [7]. Finally, we perform an extensive experimental evaluation of our implementation. We consider satisfiable industrial problems coming from the SMT-LIB, and we evaluate the effects of the various optimization techniques, also comparing them against MATHSAT. We observe that (i) the basic "naive" version of WALKSMT was not able to solve any problem within a 600s timeout; (ii) the optimization techniques drastically improve the performances of the basic version, allowing the optimized WALKSMT to solve 149/225 problems; (iii) as a comparison, MATHSAT solved 208/225 problems. We also compare the optimized WALKSMT and MATHSAT on randomly-generated unstructured problems, obtaining small differences in performances.

The rest of the paper is organized as follows. In §2 we introduce the necessary background on SLS and SMT. In §3 and §4 we describe respectively our basic algorithm and the optimization techniques we have conceived for improving its performance. In §5 we experimentally evaluate our approach. In §6 we conclude and highlight directions for future work.

2 Background

2.1 Stochastic Local Search for SAT

Local search (LS) algorithms [11, 10] are widely used for solving hard combinatorial search problems. The idea behind LS is to inspect the search space of a given problem instance starting at some position and then iteratively moving from the current position to a neighboring one where each move is determined by a decision based on information about the local neighborhood. LS algorithms making use of randomized choices during the search process are called *Stochastic Local search (SLS) algorithms*. SLS algorithms have been successfully applied to the solution of many NP-complete decision problems, including SAT. Notice, however, that SLS algorithms typically do not guarantee that

Algorithm 1 WalkSAT (φ)

```
Require: CNF formula $\varphi$, MAX_TRIES, MAX_FLIPS
 1: for $i = 1$ to MAX_TRIES do
 2:     $\mu \leftarrow$ INITIALTRUTHASSIGNMENT($\varphi$)
 3:     for $j = 1$ to MAX_FLIPS do
 4:         if ($\mu \models \varphi$) then
 5:             return SAT
 6:         else
 7:             $c \leftarrow$ CHOOSEUNSATISFIEDCLAUSE($\varphi$)
 8:             $\mu \leftarrow$ NEXTTRUTHASSIGNMENT($\varphi, c$)
 9:         end if
10:     end for
11: end for
12: return UNKNOWN
```

eventually an existing solution is found, so that they cannot verify the unsatisfiability of a problem.

SLS algorithms for SAT typically work with a CNF input formula (namely φ) and share a common high-level schema: (i) they initialize the search by generating an initial truth assignment (typically at random); (ii) they iteratively select one variable and flip it within the current truth assignment. The search terminates when the current truth assignment satisfies the formula φ or after MAX_TRIES sequences of MAX_FLIPS variable flips without finding a model for φ. The main difference in SLS SAT algorithms is typically given by the different strategies applied to select the variable to be flipped.

WalkSAT Algorithms. WalkSAT is a popular family of SLS-based SAT algorithms [11, 10]. The schema of such algorithms is shown in Algorithm 1. Initially, a complete truth assignment μ for the variables of the input problem φ is selected by INITIALTRUTHASSIGNMENT according to some heuristic criterion (e.g., uniformly at random). If this assignment satisfies the formula, then the algorithm terminates. Otherwise, a variable is selected and flipped in μ using a two-stage process. In the first stage, a currently-unsatisfied clause c is selected by CHOOSEUNSATISFIEDCLAUSE according to some heuristic criterion (e.g., uniformly at random). In the second stage, one of the variables occurring in the selected clause c is flipped by NEXTTRUTHASSIGNMENT according to some mixed greedy/random heuristic criterion, so that to generate another truth assignment. The procedure is repeated until either a solution is found, or the limit for the number of tries is reached.

Over the last ten years, several variants of the basic WalkSAT algorithm have been proposed [14, 12, 16], which differ mainly for the different heuristics used for the functions described above —in particular on the degree of greediness and randomness and in the criteria used for selecting the variable to flip in c within NEXTTRUTHASSIGNMENT. From our own empirical experience [15], the best performing WalkSAT-based algorithm for SAT seems to be Adaptive Novelty$^+$ [16]. It adopts the Novelty$^+$'s variable selection heuristic, and it adjusts its degree of greediness according to the search progress. Novelty$^+$ chooses the variable to be flipped from c depending on the score (i.e.

the difference in the total number of satisfied clauses a flip would cause) and the variable's age (i.e. the number of search steps performed since a variable was last flipped). If the variable with the highest score does not have minimal age among the variables in c, then it is selected. Otherwise, it is selected with a probability $1 - p$, where p is a parameter (called *noise setting*). While in the remaining cases p, the variable is picked uniformly at random (*random walk*). Adaptive Novelty$^+$ changes the probability of making greedy choices by increasing the noise setting p only when it needs to escape from situations in which there is no further progress in finding a solution (once the stagnation situation is overcome, the noise setting is gradually decreased). We refer the reader to [11] for a more detailed explanation.

Trimming Variable Selection and Literal Commitment Strategy. A few attempts have been made in order to enhance SLS algorithms with techniques borrowed from CDCL solvers (e.g. [6, 4]). In particular, Belov and Stachniak [6] propose two techniques that exploit the search history to improve the variable selection process of the classic SLS procedures for SAT. They modify the WalkSAT schema by adding a database (DB) that represents a set of constraints that help to guide the search process. It consists in (1) a set of clauses ψ obtained by storing selected unsatisfied-clauses (see line 7 of Algorithm 1) and (2) a partial truth assignment η that records assignments made by the local search heuristic. The goal of the *trimming variable selection* technique is to prune the search by preventing the selection of variables whose flip will cause a conflict in the database. In particular, for every variable v belonging to the selected clause c, the procedure checks the satisfiability of $\psi \wedge \eta'$ by unit propagation, where η' is obtained from η by adding the (flipped) truth assignment of v. If it is unsatisfiable, the variable v cannot be flipped. When all variables cause a conflict, the database is reset (i.e. η is set to $\emptyset$) so that any variable can be chosen by the local search heuristic. Notice that, once the truth value of a variable has been flipped, η is updated accordingly and the clause c is added to the database.

The *literal commitment strategy* aims at exploiting the power of unit propagation inside SLS procedures that naturally work with total truth assignments rather than partial ones. It iteratively deduces literals l in ψ deriving from η (i.e. $\psi \wedge \eta \models l$) and updates the current total truth assignment μ accordingly during a single search step. We refer the reader to [6] for a more detailed explanation.

2.2 Satisfiability Modulo Theory

Let $\mathcal{T}$ be a first-order theory. We call $\mathcal{T}$*-literal* a ground atomic formula in $\mathcal{T}$ or its negation. We call a *theory solver for* $\mathcal{T}$, $\mathcal{T}$*-solver*, a tool able to decide the $\mathcal{T}$-satisfiability of a conjunction/set μ of $\mathcal{T}$-literals. If μ is $\mathcal{T}$-unsatisfiable, then $\mathcal{T}$*-solver* returns UNSAT and the subset η of $\mathcal{T}$-literals in μ which was found $\mathcal{T}$-unsatisfiable; (η is hereafter called a $\mathcal{T}$*-conflict set*, and $\neg\eta$ a $\mathcal{T}$*-conflict clause*.) if μ is $\mathcal{T}$-satisfiable, then $\mathcal{T}$*-solver* returns SAT; it may also be able to return some unassigned $\mathcal{T}$-literal $l \notin \mu$[3] s.t. $\{l_1, ..., l_n\} \models_{\mathcal{T}} l$, where $\{l_1, ..., l_n\} \subseteq \mu$. We call this process $\mathcal{T}$*-deduction* and

[3] Taken from a set of all the available $\mathcal{T}$-literals; when combined with a SAT solver, such set would be the set of all the $\mathcal{T}$-literals occurring in the input formula to solve.

$(\bigvee_{i=1}^{n} \neg l_i \vee l)$ a $\mathcal{T}$*-deduction clause*. Notice that $\mathcal{T}$-conflict and $\mathcal{T}$-deduction clauses are valid in $\mathcal{T}$. We call them $\mathcal{T}$*-lemmas*. Given a $\mathcal{T}$-formula φ, the formula φ^p obtained by rewriting each $\mathcal{T}$-atom in φ into a fresh atomic proposition is the Boolean abstraction of φ, and φ is the *refinement* of φ^p. Notationally, we indicate by φ^p and μ^p the Boolean abstraction of φ and μ, and by φ and μ the refinements of φ^p and μ^p respectively. With a little abuse of notation, we say that μ^p is $\mathcal{T}$-(un)satisfiable iff μ is $\mathcal{T}$-(un)satisfiable.

In a lazy SMT($\mathcal{T}$) solver, the *Boolean abstraction* φ^p of the input formula φ is given as input to a CDCL SAT solver, and whenever a satisfying assignment μ^p is found s.t. $\mu^p \models \varphi^p$, the corresponding set of $\mathcal{T}$-literals μ is fed to the $\mathcal{T}$*-solver*; if μ is found $\mathcal{T}$-consistent, then φ is $\mathcal{T}$-consistent; otherwise, $\mathcal{T}$*-solver* returns the $\mathcal{T}$-conflict set η causing the inconsistency, so that the clause $\neg\eta^p$ (the Boolean abstraction of $\neg\eta$) is used to drive the backjumping and learning mechanism of the SAT solver. Important optimizations are *early pruning* and $\mathcal{T}$*-propagation*: the $\mathcal{T}$*-solver* is invoked also on an intermediate assignment μ: if it is $\mathcal{T}$-unsatisfiable, then the procedure can backtrack; if not, and if the $\mathcal{T}$*-solver* is able to perform a $\mathcal{T}$-deduction $\{l_1, ..., l_n\} \models_{\mathcal{T}} l$, then l can be unit-propagated, and the $\mathcal{T}$-deduction clause $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ can be used in backjumping and learning. The above schema is a coarse abstraction of the procedures underlying all the state-of-the-art lazy SMT tools. The interested reader is pointed to, e.g., [13, 5] for details and further references.

3 Stochastic Local Search for SMT

We start from a simple observation: in principle, from the perspective of a SAT solver, an SMT problem instance φ can be seen as the problem of solving a *partially-invisible* CNF SAT formula $\varphi^p \wedge \tau^p$, s.t. the "visible" part φ^p is the Boolean abstraction of φ and the "invisible" part τ^p is (the Boolean abstraction of) the set τ of all the $\mathcal{T}$-lemmas providing the obligations induced by the theory $\mathcal{T}$ on the $\mathcal{T}$-atoms of φ. (See the example in Fig 1.) Thus, every assignment μ^p s.t. $\mu^p \models \varphi^p$ is $\mathcal{T}$-unsatisfiable iff μ^p falsifies some non-empty set of clauses $\{c_1^p, ..., c_n^p\} \subseteq \tau^p$. To this extent, a traditional "lazy" SMT solver can be seen as a CDCL SAT solver which knows φ^p but not τ^p: whenever a model μ^p for φ^p is found, it is passed to a $\mathcal{T}$*-solver* which (behaves as if it) knows τ^p, and hence checks if μ^p falsifies some clause $c_i^p \in \tau^p$: if this is the case, it returns one (or more) such clause(s) c_i^p, which is then used to drive the future search and which is optionally added to φ^p.

3.1 A basic WalkSMT procedure

The above observation inspired to us a procedure integrating a $\mathcal{T}$*-solver* into a SLS algorithm of the WalkSAT family (WALKSMT hereafter). A high-level description of the pseudo-code of WALKSMT is shown in Algorithm 2. (We present first a basic version of WALKSMT, in which we temporarily ignore steps 1-3 and 12-13, which we will describe in §4, together with other enhancements.) WALKSMT receives in input a SMT($\mathcal{T}$) CNF formula and applies a WalkSAT scheme to its Boolean abstraction φ^p. INITIALTRUTHASSIGNMENT, CHOOSEUNSATISFIEDCLAUSE and NEXTTRUTHASSIGNMENT are the functions described in §2.1. (Notice that their underlying heuristics vary with the different variants of WalkSAT adopted.)

ϕ :	ϕ^p :
$c_1: \{A_1\}$	$c_1: \{A_1\}$
$c_2: \{\neg A_1 \vee (x - z > 4)\}$	$c_2: \{\neg A_1 \vee B_1\}$
$c_3: \{\neg A_3 \vee A_1 \vee (y \geq 1)\}$	$c_3: \{\neg A_3 \vee A_1 \vee B_2\}$
$c_4: \{\neg A_2 \vee \neg(x - z > 4) \vee \neg A_1\}$	$c_4: \{\neg A_2 \vee \neg B_1 \vee \neg A_1\}$
$c_5: \{(x - y \leq 3) \vee \neg A_4 \vee A_5\}$	$c_5: \{B_3 \vee \neg A_4 \vee A_5\}$
$c_6: \{\neg(y - z \leq 1) \vee (x + y = 1) \vee \neg A_5\}$	$c_6: \{\neg B_4 \vee B_5 \vee \neg A_5\}$
$c_7: \{A_3 \vee \neg(x + y = 0) \vee A_2\}$	$c_7: \{A_3 \vee \neg B_6 \vee A_2\}$
$c_8: \{\neg A_3 \vee (z + y = 2)\}$	$c_8: \{\neg A_3 \vee B_7\}$
τ : (all possible $\mathcal{T}$-lemmas on the $\mathcal{T}$-atoms of ϕ)	τ^p :
$c_9: \{\neg(x + y = 0) \vee \neg(x + y = 1)\}$	$c_9: \{\neg B_6 \vee \neg B_5\}$
$c_{10}: \{\neg(x - z > 4) \vee \neg(x - y \leq 3) \vee \neg(y - z \leq 1)\}$	$c_{10}: \{\neg B_1 \vee \neg B_3 \vee \neg B_4\}$
$c_{11}: \{(x - z > 4) \vee (x - y \leq 3) \vee (y - z \leq 1)\}$	$c_{11}: \{B_1 \vee B_3 \vee B_4\}$
$c_{12}: \{\neg(x - z > 4) \vee \neg(x + y = 1) \vee \neg(z + y = 2)\}$	$c_{12}: \{\neg B_1 \vee \neg B_5 \vee \neg B_7\}$
$c_{13}: \{\neg(x - z > 4) \vee \neg(x + y = 0) \vee \neg(z + y = 2)\}$	$c_{13}: \{\neg B_1 \vee \neg B_6 \vee \neg B_7\}$
... ...	... ...

$$B_1 \overset{\text{def}}{=} (x - z > 4),\ B_2 \overset{\text{def}}{=} (y \geq 1),\quad B_3 \overset{\text{def}}{=} (x - y \leq 3),\ B_4 \overset{\text{def}}{=} (y - z \leq 1),$$
$$B_5 \overset{\text{def}}{=} (x + y = 1),\ B_6 \overset{\text{def}}{=} (x + y = 0),\ B_7 \overset{\text{def}}{=} (z + y = 2).$$

φ :	φ^p :
$c_2: \{(x - z > 4)\}$	$c_2: \{B_1\}$
$c_5: \{(x - y \leq 3) \vee \neg A_4 \vee A_5\}$	$c_5: \{B_3 \vee \neg A_4 \vee A_5\}$
$c_6: \{\neg(y - z \leq 1) \vee (x + y = 1) \vee \neg A_5\}$	$c_6: \{\neg B_4 \vee B_5 \vee \neg A_5\}$
$c_7: \{A_3 \vee \neg(x + y = 0)\}$	$c_7: \{A_3 \vee \neg B_6\}$
$c_8: \{\neg A_3 \vee (z + y = 2)\}$	$c_8: \{\neg A_3 \vee B_7\}$
$c_9: \{\neg(x + y = 0) \vee \neg(x + y = 1)\}$	$c_9: \{\neg B_6 \vee \neg B_5\}$

$$\mu_1^p = \{B_1, A_3, \neg A_4, \neg A_5, \neg B_6, B_5, B_3, B_4, B_7\}$$
$$\mu_1 = \{(x - z > 4), \neg(x + y = 0), (x + y = 1), (x - y \leq 3), (y - z \leq 1), (z + y = 2)\}$$

Fig. 1. Top: example of an SMT($\mathcal{LA}(\mathbb{Q})$) formula ϕ as a "partially-invisible" formula $\phi^p \wedge \tau^p$. Middle: the formula φ [resp φ^p] obtained from ϕ [resp ϕ^p] after preprocessing (see §4). Bottom: a truth assignment μ^p satisfying φ^p and violating c_{10}, c_{12} in τ^p, and its refinement μ_1.

Since we are temporarily ignoring steps 1-3 and 12-13, the only significant difference wrt. Algorithm 1 is in steps 7-14. Whenever a total model μ^p is found s.t. $\mu^p \models \varphi^p$, it is passed to $\mathcal{T}$-*solver*. If (the set of $\mathcal{T}$-literals corresponding to) μ^p is $\mathcal{T}$-satisfiable (i.e., $\mu^p \models \varphi^p \wedge \tau^p$) the procedures ends returning SAT. Otherwise, $\mathcal{T}$-*solver* returns CONFLICT and a $\mathcal{T}$-lemma c^p. Notice that this corresponds to say that $\mu^p \not\models \varphi^p \wedge \tau^p$, and that c^p is one of the (possibly-many) clauses in $\varphi^p \wedge \tau^p$ which are falsified by μ^p. Thus, c^p is used by NEXTTRUTHASSIGNMENT as "selected" unsatisfied clause to drive the flipping of the variable. To this extent, $\mathcal{T}$-*solver* plays also the role of CHOOSEUNSATISFIEDCLAUSE on $\varphi^p \wedge \tau^p$ when no unsatisfied clause is found in φ^p (to this extent, see also "Multiple Learning" in §4).

Algorithm 2 WALKSMT (φ)

```
Require: SMT($\mathcal{T}$) CNF formula $\varphi$, MAX_TRIES, MAX_FLIPS
 1: if ($\mathcal{T}$-PREPROCESS ($\varphi$) == CONFLICT) then
 2:     return UNSAT
 3: end if
 4: for $i = 1$ to MAX_TRIES do
 5:     $\mu^p \leftarrow$ INITIALTRUTHASSIGNMENT ($\varphi^p$)
 6:     for $j = 1$ to MAX_FLIPS do
 7:         if ($\mu^p \models \varphi^p$) then
 8:             $\langle status, c^p \rangle \leftarrow \mathcal{T}$-solver($\varphi^p, \mu^p$)
 9:             if ($status$ == SAT) then
10:                 return SAT
11:             end if
12:             $c^p \leftarrow$ UNIT-SIMPLIFICATION($\varphi^p, c^p$)
13:             $\varphi^p \leftarrow \varphi^p \wedge c^p$
14:             $\mu^p \leftarrow$ NEXTTRUTHASSIGNMENT ($\varphi^p, c^p$)
15:         else
16:             $c^p \leftarrow$ CHOOSEUNSATISFIEDCLAUSE ($\varphi^p$)
17:             $\mu^p \leftarrow$ NEXTTRUTHASSIGNMENT ($\varphi^p, c^p$)
18:         end if
19:     end for
20: end for
21: return UNKNOWN
```

Example 1. Suppose WALKSMT is invoked on the formula φ^p in Fig. 1, generating the total truth assignment μ_1^p that satisfies φ^p. Then $\mathcal{T}$*-solver* is invoked on μ_1, which is $\mathcal{T}$-inconsistent due to the the literals $\{(x - z > 4), (x + y = 1), (z + y = 2)\}$, returning UNSAT and the conflict clause $c_1^p = \{\neg B_1 \vee \neg B_5 \vee \neg B_7\}$ (i.e. c_{12} in τ^p). Then NEXTTRUTHASSIGNMENT will flip one of the literals B_1, B_5 or B_7.

Remark: efficient $\mathcal{T}$-solvers for local search. In CDCL-based SMT solvers, the interaction with $\mathcal{T}$*-solvers* is *stack-based*: the truth assignment μ is incrementally extended when performing unit propagation, $\mathcal{T}$-propagation, and when picking an unassigned literal for branching, and it is partly undone upon backtracking, when the most-recently-assigned literals are removed from it. Consequently, $\mathcal{T}$*-solvers* designed for interaction with a CDCL SAT solver are typically optimized for such stack-based invocation. In particular, they are typically *incremental* —when they have to check the consistency of a truth assignment μ' that is an extension of a previously-checked μ, they don't need to restart the computation from scratch— and *backtrackable* —when backtracking occurs, the most-recently-assigned literals that need to be unassigned can be efficiently removed, and the internal state can be efficiently restored to a previous configuration (see [13, 5]).

In local search, instead, a new assignment μ' is obtained from the previous one μ by flipping *an arbitrary* literal (according to some heuristics). In this setting, the conventional backtrackability feature of $\mathcal{T}$*-solvers* is of little use, since there is no notion of most-recently-assigned literals to remove. Instead, it is very desirable to be able to remove *arbitrary* literals from a $\mathcal{T}$*-solver* without the need of resetting its internal state. Such requirement might seem unrealistic, or at least difficult to fulfill. However, at least two state-of-the-art $\mathcal{T}$*-solvers* have this capability: the $\mathcal{T}$*-solver* for $\mathcal{DL}$ of [8] and the $\mathcal{T}$*-solver* for $\mathcal{LA}(\mathbb{Q})$ of [9], which are therefore natural candidates for integration with a SLS-based SAT solver. The MATHSAT solver implements both.

4 Enhancements to the basic WalkSMT procedure

The WALKSMT algorithm described above is very naive. Here we analyze the interaction of a $\mathcal{T}$-solver with a SLS SAT solver, and we present a group of optimization techniques aimed at improving the synergy of their interaction.

4.1 Preprocessing

Before entering the main WALKSMT routine, we apply a *preprocessing* step to the input formula φ in order to make it simpler to solve (steps 1-3 in algorithm 19). This preprocessing consists mainly of two techniques: *Initial BCP* and *Static Learning*.

Initial BCP. Often SMT formulas contain lots of "structural" atomic propositions whose truth value is assigned deterministically (e.g., when the formula derives from a CNF-ization step). Unlike a CDCL solver, an SLS one cannot handle them efficiently. Thus, during preprocessing we first perform a run of BCP to the input formula, simplifying the formula accordingly. In order to preserve correctness, we keep as unit clauses the $\mathcal{T}$-literals $l_1, .., l_n$ which have been assigned to true by BCP. If during this process one of the clauses of ϕ^p is falsified, or if the set of $\mathcal{T}$-literals $l_1, .., l_n$ above is $\mathcal{T}$-inconsistent, the algorithm can exit returning UNSAT. Otherwise, $l_1, .., l_n$ are tagged "unflippable", so that the SLS engine initially assigns them to true and never flips their value.

Static Learning. During preprocessing we also conjoin to the formula φ/φ^p short and "obvious" $\mathcal{T}$-lemmas on the atoms occurring in φ, which can be generated without explicitly invoking the $\mathcal{T}$-solver. (Examples of such $\mathcal{T}$-lemmas are mutual-exclusion lemmas like c_9 in Fig. 1. See also [13].) Thus the $\mathcal{T}$*-solver* is invoked on an assignment μ only if μ^p verifies also these $\mathcal{T}$-lemmas (row 7 in Alg. 2). This prevents WALKSMT from invoking $\mathcal{T}$*-solver* on obviously-$\mathcal{T}$-inconsistent assignments.

Example 2. Consider as input the formula ϕ of Fig. 1 (top). The preprocessing step generates the formula φ of Fig. 1 (bottom). In fact, BCP unit-propagates the literals $A_1, B_1, \neg A_2$, simplifying clause c_7 and eliminating clauses c_1, c_3 and c_4. Clause c_2 survives as an unit clause because B_1 is (the label of) a $\mathcal{T}$-literal. Notice that the $\mathcal{T}$-atom $B_2 \overset{\text{def}}{=} (y \geq 0)$ disappears from the formula because c_3 is satisfied by the unit-propagation of A_1. The $\mathcal{T}$-lemma c_9 is then added to the simplified formula by static learning.

4.2 Single and multiple learning

Learning. SLS SAT solvers typically do not implement learning. This is potentially a major problem with SLS-based SMT, because the SLS solver may generate many total assignments $\mu_1^p, ..., \mu_k^p$ each containing the same $\mathcal{T}$-inconsistent subset η^p, causing thus $k-1$ useless calls to *$\mathcal{T}$-solver*. Thus, like in standard CDCL-based SMT solvers, we conjoin to φ^p the $\mathcal{T}$-lemma c^p returned by the $\mathcal{T}$-solver (step 13). Henceforth *$\mathcal{T}$-solver* is no more invoked on assignments violating c^p.

Unit Resolution. Before learning a $\mathcal{T}$-lemma c, we remove from it all the $\mathcal{T}$-literals whose negation occurs as unit clauses in the input problem (step 12). (Notice that after this step c may be no longer a $\mathcal{T}$-lemma.) We do this in both static and dynamic learning.

Example 3. Consider the scenario of Example 1, assuming learning is implemented. Because of the unit clause c_2 of φ^p, we remove from the conflict clause c_1^p the literal $\neg B_1$, obtaining $c_1^{p\prime} \stackrel{\text{def}}{=} \{\neg B_5 \vee \neg B_7\}$ (i.e., a unit-resolved version of c_{12} in τ^p.), which we add to φ^p. Then NEXTTRUTHASSIGNMENT will flip one of the literals B_5 or B_7. *$\mathcal{T}$-solver* will never be invoked again on assignments containing both B_5 and B_7.

Multiple Learning. Unlike with CDCL-based SMT solvers, which typically use some form of early pruning to check partial truth assignments for $\mathcal{T}$-consistency, in an SLS-based approach $\mathcal{T}$-solvers operate always on *complete* truth assignments μ. In this setting, it is likely that μ contains many different $\mathcal{T}$-inconsistent subsets, often independent from each another. This is the idea at the basis of our *multiple learning* technique, which allows for learning more than one $\mathcal{T}$-lemma for every $\mathcal{T}$-inconsistent assignment. When a conflict set η is found (and simplified via unit-resolution), a given percentage p of its literals are randomly removed from μ, and *$\mathcal{T}$-solver* is invoked again on the resulting set. This process is repeated until no more conflict is found. We then learn all the $\mathcal{T}$-lemmas $c_1^p, ..., c_k^p$ generated during the process. Also, if $k > 1$, then one clause c^p among $c_1^p, ..., c_k^p$ is chosen by CHOOSEUNSATISFIEDCLAUSE to be fed to NEXTTRUTHASSIGNMENT.

Example 4. Consider the scenario of Example 1 and 3, assuming multiple learning is implemented, with $p = 100\%$. After learning the clause $c_1^{p\prime}$, we drop B_5, B_7 from μ_1^p and re-invoke *$\mathcal{T}$-solver* on the set of $\mathcal{T}$-literals $\mu_2 \stackrel{\text{def}}{=} \mu_1 \setminus \{(x+y=1), (z+y=2)\}$, returning UNSAT and the conflict clause $c_2^p \stackrel{\text{def}}{=} \{\neg B_1 \vee \neg B_3 \vee \neg B_4\}$, from which $\neg B_1$ is removed by unit-resolution, so that also the clause $c_2^{p\prime} \stackrel{\text{def}}{=} \{\neg B_3 \vee \neg B_4\}$ is learned (a unit-resolved version of clause c_{10}). After further removing B_3 and B_4 from μ_2 the set of $\mathcal{T}$-literals is found $\mathcal{T}$-consistent by *$\mathcal{T}$-solver*, so that no further clause is learned. Then $c_1^{p\prime}, c_2^{p\prime}$ are fed to CHOOSEUNSATISFIEDCLAUSE which selects one and feed it to NEXTTRUTHASSIGNMENT, which flips one literal among B_5, B_7, B_3 and B_4.

4.3 Literal filterings

Pure-literal Filtering. If some $\mathcal{T}$-atoms occur only positively [resp. negatively] in the original formula (learned clauses and statically-learned clauses are not considered), then we can safely drop every negative [resp. positive] occurrence of them from the assignment μ to be checked by the $\mathcal{T}$-solver [13]. (Intuitively, since such occurrences play no role in satisfying the formula, the resulting partial assignment $\mu^{p\prime}$ still satisfies φ^p.) The benefits of this action is twofold:

(i) reduces the workload for the $\mathcal{T}$*-solver* by feeding it smaller sets;
(ii) increases the chance of finding a $\mathcal{T}$-consistent satisfying assignment by removing "useless" $\mathcal{T}$-literals which may cause the $\mathcal{T}$-inconsistency of μ.

Example 5. Consider the formula φ^p in Fig. 1 and the total truth assignment

$$\mu_4^p = \{B_1, \neg A_3, \neg A_4, \neg A_5, \neg B_6, \neg B_5, B_3, B_4, \neg B_7\}$$

that satisfies φ^p, but is $\mathcal{T}$-inconsistent because of its subset $\{B_1, B_3, B_4\}$ (clause c_{10} in τ^p). Without pure-literal filtering, $\mathcal{T}$*-solver* detects the inconsistency, WALKSMT learns the clause and looks for another assignment. If pure-literal filtering is implemented, instead, since the $\mathcal{T}$-literals $\neg B_5$, B_4 and $\neg B_7$ occur only negatively in the original formula ϕ, they are filtered out from μ_4^p, resulting in the *partial* assignment

$$\eta_4^p = \{B_1, \neg A_3, \neg A_4, \neg A_5, \neg B_6, B_3\},$$

which still satisfies φ^p. $\mathcal{T}$*-solver* is invoked on the corresponding set of $\mathcal{T}$-literals:

$$\eta_4 = \{(x - z > 4), \neg(x + y = 0), (x - y \leq 3)\}.$$

which is $\mathcal{T}$-consistent, from which we can conclude that φ (and ϕ) is $\mathcal{T}$-consistent.

Ghost-literal Filtering. We further enforce the benefits of pure-literal filtering as follows. When a truth assignment μ is found s.t. $\mu^p \models \varphi^p$, before invoking $\mathcal{T}$*-solver* on μ, we check whether any $\mathcal{T}$-atom occurring only positively [resp. negatively] in the original formula and being assigned true [resp. false] in μ can be flipped without falsifying any clause. (This test can be performed very efficiently inside an SLS solver.) If this is the case, then the atom is flipped. This step is repeated until no more such atoms are found, after which the resulting set μ is passed to $\mathcal{T}$*-solver*. This allows for further removing useless $\mathcal{T}$-literals from μ by pure-literal filtering. (Since such literals are a particular case of "ghost literals" [13], we call this enhancement *ghost-literal filtering*.)

Example 6. Consider the formula φ^p in Fig. 1 and the total truth assignment

$$\mu_5^p = \{B_1, A_3, \neg A_4, \neg A_5, \neg B_6, \neg B_5, B_3, \neg B_4, B_7\}$$

that satisfies φ^p. If we apply pure-literal filtering on μ_5^p, then we can filter out only the literal $\neg B_5$ before invoking $\mathcal{T}$*-solver*. By ghost-literal filtering, the literals B_3, $\neg B_4$ and $\neg B_6$ are flipped without falsifying φ^p, resulting in the total truth assignment:

$$\mu_5^{p\prime} = \{B_1, A_3, \neg A_4, \neg A_5, B_6, \neg B_5, \neg B_3, B_4, B_7\}.$$

Now, by pure-literal filtering, we remove from $\mu_5^{p\prime}$ the literals B_3, $\neg B_4$, $\neg B_5$ and $\neg B_6$.

5 Experimental Evaluation

We have implemented two versions of the WALKSMT procedure described above to work for the $\mathcal{LA}(\mathbb{Q})$ theory. The implementation is done on top of MATHSAT4 [7], using part of its preprocessor its $\mathcal{LA}(\mathbb{Q})$-solver [9] and lots of its features. We have implemented two versions, each using one between two SLS-based SAT solvers: UBCSAT [4] [17] and UBCSAT++ [5] [6]. UBCSAT is a SLS platform providing a very-wide range of SLS algorithms for SAT (including the WalkSAT family), with a very flexible architecding the WalkSAT family), with a very flexible architecture. Among the various SLS procedures provided by UBCSAT, we have chosen to use the Adaptive Novelty$^+$ variant of the WalkSAT family because it was the best-performing in a previous extensive empirical evaluation [15]. UBCSAT++ is built on top of UBCSAT and extends its implementation of Adaptive Novelty$^+$ with the Trimming Variable Selection and Literal Commitment Strategy techniques described in §2.1. We partition the enhancements of WALKSMT of §4 into three groups:

– *Preprocessing and Learning (PL)*, including preprocessing (Initial BCP and Static Learning), Learning and Unit Resolution;
– *Multiple Learning (ML)*;
– *Filtering (FI)*, including both Pure-Literal and Ghost-Literal filterings.

Notationally, we use a "+" [resp. "–"] symbol to denote that an option is enabled [resp. disabled]: e.g., "UBCSAT++ BASIC+PL-ML+FI" denotes WALKSMT based on UBCSAT++ with PL and FI enabled and ML disabled. (Notice that ML requires PL, so that we cannot have "...-PL+ML..." configurations.)

In this section, we evaluate the performance of WALKSMT by comparing its two versions (those based on UBCSAT and UBCSAT++ respectively) against the CDCL-based SMT solver MATHSAT4. We ran MATHSAT4 with all the optimizations enabled (the most important ones are early pruning and $\mathcal{T}$-propagation). [6] We performed our comparison over two distinct sets of instances, which are described in the next two sections: the first consists of the set of all satisfiable $\mathcal{LA}(\mathbb{Q})$ formulas in the SMT-LIB 1.2 (`www.smtlib.org`), whereas the second is composed of randomly-generated problems. All tests were executed on 2.66 GHz Xeon machines running Linux, using a timeout of 600 seconds. The correctness of the models found by WALKSMT have been cross-checked by MATHSAT4. In order to make the experiments reproducible, the full-size plots, the tools, the problems, and the results are available at [1].

5.1 WALKSMT on SMT-LIB Instances

In the first part of our experiments, we compare WALKSMT against MATHSAT on all the satisfiable $\mathcal{LA}(\mathbb{Q})$-formulas (QF_LRA) in the SMT-LIB 1.2. These instances are all classified as "industrial", because they come from the encoding of different real-world problems in formal verification, planning and optimization, and they are divided into

[4] UBCSAT is publicly available at `http://www.satlib.org/ubcsat/`.

[5] UBCSAT++ was kindly provided to us by the developers, Belov and Stachniak.

[6] Although more efficient SMT ($\mathcal{LA}(\mathbb{Q})$) solvers exist, including the recent MATHSAT5, here the choice of MATHSAT4 is aimed at minimizing the differences in performance due to the implementation, because WALKSMT is implemented on top of MATHSAT4 (in particular it uses its preprocessor and $\mathcal{T}$-solver for $\mathcal{LA}(\mathbb{Q})$), so that to better highlight the differences between SLS- and CDCL-based approaches.

Solver	SMT-LIB Instances						Total
	`sc`	`uart`	`sal`	`TM`	`tta`	`miplib`	
Total # of Instances	108	36	11	24	24	22	225
WalkSMT UBCSAT Basic–PL–ML–FI	0	0	0	0	0	0	0
WalkSMT UBCSAT++ Basic–PL–ML–FI	0	0	0	0	0	1	1
WalkSMT UBCSAT Basic+PL–ML–FI	59	10	6	13	5	3	96
WalkSMT UBCSAT++ Basic+PL–ML–FI	46	6	7	17	10	1	87
WalkSMT UBCSAT Basic+PL+ML–FI	103	15	6	12	6	3	145
WalkSMT UBCSAT++ Basic+PL+ML–FI	61	6	7	15	9	1	99
WalkSMT UBCSAT Basic+PL–ML+FI	59	32	10	14	9	3	127
WalkSMT UBCSAT++ Basic+PL–ML+FI	62	12	8	18	10	1	111
WalkSMT UBCSAT Basic+PL+ML+FI	78	35	10	14	9	3	149
WalkSMT UBCSAT++ Basic+PL+ML+FI	63	14	8	19	10	2	116
MATHSAT4	108	36	11	21	24	8	208

Table 1. Comparison of the number of instances solved within the 600s timeout by the various configurations of WALKSMT and MATHSAT4. Notice that instances solved by the different solvers might not be the same.

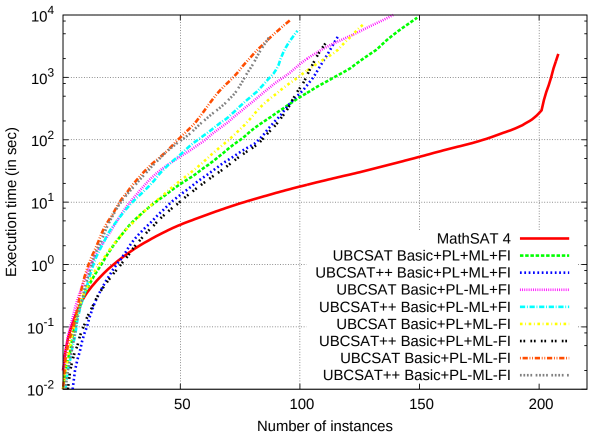

Fig. 2. Cumulative plots of WALKSMT and MATHSAT4 on all SMT-LIB instances.

six categories: `sc`, `uart`, `sal`, `TM`, `tta_startup` ("`tta`" hereafter), and `miplib`.[7] The results of the experiments are reported in Figures 2, 3, 4, 5 and in Table 1. Figure 2 shows the cumulative plots of the execution time for the different configurations of WALKSMT and MATHSAT4 on SMT-LIB instances. (The plots for BASIC-PL-ML-FI are not reported since no formula was solved within the timeout.) Figure 3 compares the best configurations of WALKSMT (BASIC+PL+ML+FI) with UBCSAT (left) and with UBCSAT++ (right) against MATHSAT4 on all instances. Figure 4 shows the rela-

[7] Notice that other SMT-LIB categories like `spider_benchmarks` and `clock_synchro` do not contain satisfiable instances and are thus not reported here.

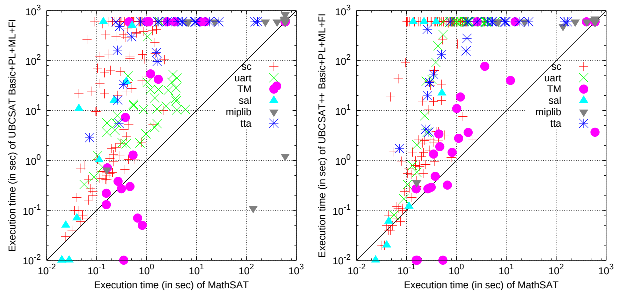

Fig. 3. Comparison of the best configurations of WALKSMT (BASIC+PL+ML+FI) against MATHSAT4 on SMT-LIB instances. Left: with UBCSAT; Center: with UBCSAT++; Right: with UBCSAT++, considering only `miplib` and `TM` benchmarks.

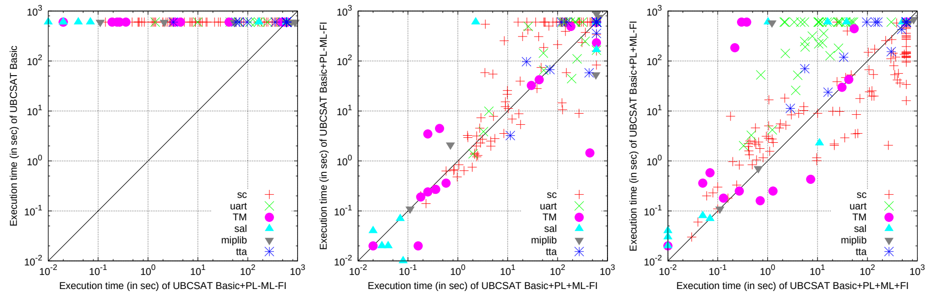

Fig. 4. Pairwise comparison between different configurations of WALKSMT with UBCSAT on SMT-LIB instances, adding increasingly PL, ML and FI to basic WALKSMT.
Left: BASIC-PL-ML-FI vs. BASIC+PL-ML-FI (benefits of adding PL to Basic);
Center: BASIC+PL-ML-FI vs. BASIC+PL+ML-FI (benefits of further adding ML);
Right: BASIC+PL+ML-FI vs. BASIC+PL+ML+FI (benefits of further adding FI).

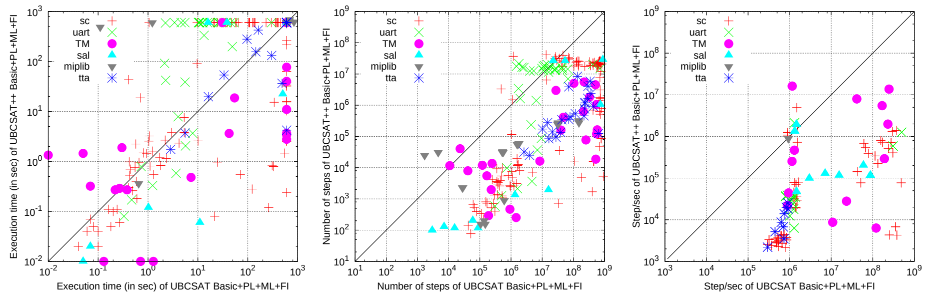

Fig. 5. Comparison between WALKSMT UBCSAT and WALKSMT UBCSAT++ on BASIC+PL+ML+FI versions. Left: CPU time. Center: flip# (on commonly solved instances). Right: average ratio flips#/sec (on commonly solved instances).

tive effects of the different optimizations for WALKSMT with UBCSAT. Figure 5 compares WALKSMT UBCSAT against WALKSMT UBCSAT++ on BASIC+PL+ML+FI versions. The results suggest a list of considerations.

First, the optimizations described in §4 lead to dramatic improvements in performance, sometimes by orders of magnitude. Without them, WALKSMT times out on all instances. (See Table 1 and Figures 2 and 4.):

- PL is crucial for performance, since with PL disabled almost no problem is solved within the timeout. In particular, from our data we see that a key role is played by learning. (Which perhaps is not surprising from an SMT perspective, but we believe may be of interest from an SLS perspective.)
- ML produces significant improvements overall, except for a few cases where it may worsen performances (e.g., with `miplib`).
- FI produces strong improvements in performance in all problem categories, (apparently with the exception of the `sc` benchmarks).

Second, globally WALKSMT seems to perform better with UBCSAT than with UBCSAT++, with some exceptions (`TM`, `tta`). From Figure 5, considering the problems solved by both configurations, we see that the total number of flips performed by UBCSAT++ is dramatically smaller than that performed by UBCSAT, but the average cost of each flip is dramatically higher.

Third, globally MATHSAT4 performs much better than WALKSMT, often by orders of magnitude. This mirrors the typical performance gap between CDCL and SLS SAT solvers on industrial benchmarks.

5.2 WALKSMT on Random Instances

Unlike with SAT, in SMT there is very-limited tradition in testing on random problems (e.g., [2, 3]). However, for a matter of scientific curiosity and/or to leverage to SMT a popular test for SLS SAT procedures, here we present also a brief comparison of WALKSMT vs. MATHSAT4 on randomly-generated, unstructured 3-CNF $\mathcal{LA}(\mathbb{Q})$-formulas. Each 3-CNF formula is randomly generated according to three integer parameters $\langle m, n, a\rangle$ as follows. First, a distinct $\mathcal{T}$-atoms $\psi_1, ..., \psi_a$ are created, s.t. each atom ψ_j is in the form $(\sum_{i=1}^{4} c_{ji}x_{ji} \leq c_j)$, it is generated by randomly picking four distinct variables x_{ji} out of n variables $\{x_1, ..., x_n\}$, and five integer values $c_{j1}, ..., c_{j4}, c_j$ in the interval $[-100, 100]$. Then, m 3-CNF clauses are randomly generated, each by randomly picking 3 distinct $\mathcal{T}$-atoms in $\{\psi_1, ..., \psi_a\}$, negating each with probability 0.5.

Figure 6 shows the run times of several versions of WALKSMT and MATHSAT4 on the generated formulas, for $n = 20$. Each graph shows curves for WALKSMT (in particular, UBCSAT and UBCSAT++ with the best configuration BASIC+PL+ML+FI) and MATHSAT4 on a group of instances with a fixed number a of $\mathcal{T}$-atoms, for $a = 30, 40, 50, 60$. The plots represent the execution time versus the ratio $r = m/a$ of clauses/$\mathcal{T}$-atoms. Each point in the graphs corresponds to the median run-time of each algorithm on 100 different instances of the same size. (For WALKSMT, each value is itself a median value of 3 runs with different seeds.) The plots show also the satisfiability percentage of each group of instances, defined as the ratio between the satisfiable

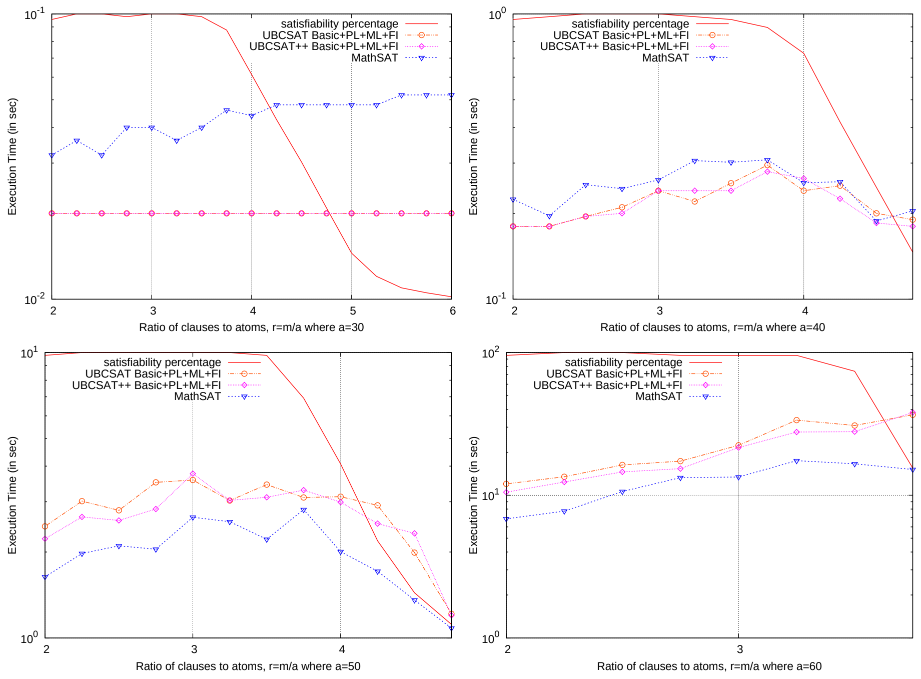

Fig. 6. Comparison of different configurations of WALKSMT and MATHSAT4 on randomly-generated instances with 20 theory variables and atoms $a = 30, 40, 50, 60$.

instances generated and the total number of instances generated, for each value of r. E.g., in the plot in the first column of the first row of Figure 6 the percentage 0.01% for $r = 6$ means that we had to generate and test 10514 formulas (using MATHSAT4 with a timeout of 600 seconds) in order to obtain 100 satisfiable instances.

The results show that, unlike with SMT-LIB formulas, on randomly-generated instances there is a very small difference between the performance of UBCSAT BASIC+PL+ML+FI, UBCSAT++ BASIC+PL+ML+FI and MATHSAT4.

6 Conclusions and future work

In this paper we have investigated the possibility of using an SLS SAT solver instead of a conventional CDCL-based one as propositional engine for a lazy SMT solver. We have presented and discussed several optimizations to the basic architecture proposed, which allowed WALKSMT to solve a significant amount of industrial SMT problems, although it is still much less efficient that the corresponding CDCL-based SMT solver. We believe that the latter fact is not surprising, since optimization techniques for CDCL-based SMT solvers have been investigated and optimized for the last ten years, whilst to the best of our knowledge this is the first attempt of building a SLS-based one.

This research opens the possibility for several interesting future directions. The first obvious option is to port the implementation to the more-efficient MATHSAT5 and to extend the present work to cover other theories typically used in SMT. We would like to concentrate in particular on "hard" theories such as $\mathcal{LA}(\mathbb{Z})$. Second, we plan to investigate the use of SLS techniques for solving/approximating optimization problems, such as Max-SMT. Third, we will explore the possibility of tightening the synergy between the SLS SAT solver and $\mathcal{T}$-solvers, for instance by better exploiting information that can be provided by $\mathcal{T}$-solvers when deciding which variables to flip, or by considering architectures in which the search is more driven by the theory part of the formula rather than by the SAT engine. Finally, we plan to work on the integration/combination between SLS-based and CDCL-based SMT solvers, both using a portfolio-like approach and investigating more tightly-coupled solutions.

References

1. `http://disi.unitn.it/~rseba/frocos11/tests.tar.gz`.
2. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.
3. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. CADE'2002.*, volume 2392 of *LNAI*. Springer, July 2002.
4. G. Audemard, J.-M. Lagniez, B. Mazure, and L. Sais. Boosting Local Search Thanks to CDCL. In *LPAR (Yogyakarta)*, pages 474–488, 2010.
5. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press, 2009.
6. A. Belov and Z. Stachniak. Improving variable selection process in stochastic local search for propositional satisfiability. In *SAT'09*, LNCS. Springer, 2009.
7. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *Proc. of CAV*, volume 5123 of *LNCS*. Springer, 2008.
8. S. Cotton and O. Maler. Fast and Flexible Difference Constraint Propagation for DPLL(T). In *SAT*, volume 4121 of *LNCS*. Springer, 2006.
9. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, 2006.
10. H. H. Hoos and T. Stutzle. Local Search Algorithms for SAT: An Empirical Evaluation. *Journal of Automated Reasoning*, 24(4), 2000.
11. H. H. Hoos and T. Stutzle. *Stochastic Local Search Foundation And Application*. Morgan Kaufmann, 2005.
12. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58(1), 1992.
13. R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, Volume 3, 2007.
14. B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. AAAI*. MIT Press, 1994.
15. S. Tomasi. Stochastic Local Search for SMT. Technical report, DISI-10-060, DISI, University of Trento, 2010. Available at `http://eprints.biblio.unitn.it/`.
16. D. Tompkins and H. Hoos. Novelty+ and Adaptive Novelty+. *SAT 2004 Competition Booklet*, 2004.
17. D. Tompkins and H. Hoos. UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT. In *SAT*, volume 3542 of *LNCS*. Springer, 2004.