Stochastic Local Search for SMT

Silvia Tomasi

December 2010

Technical Report # DISI-10-060

Contents

1 Introduction **5**

2 Background **5**

2.1 SAT solvers . . . 5
2.2 Lazy Satisfiability Modulo Theory . . . 7
2.3 Sthocastic Local Search . . . 9
2.3.1 Stochastic Local Search for SAT . . . 12
2.3.2 GSAT Architecture . . . 13
2.3.3 Basic GSAT - GSAT . . . 14
2.3.4 GSAT with Random Walk - GWSAT . . . 16
2.3.5 GSAT with Random Noise - GUWSAT . . . 16
2.3.6 GSAT with Tabu Search - GSAT/Tabu . . . 17
2.3.7 Introduction to Tabu Search . . . 17
2.3.8 GSAT/Tabu . . . 17
2.3.9 GSAT with History Information - HSAT . . . 18
2.3.10 HSAT with Random Walk - HWSAT . . . 18
2.3.11 Restarting GSAT - RGSAT . . . 18
2.3.12 WalkSAT Architecture . . . 19
2.3.13 WalkSAT/SKC . . . 19
2.3.14 WalkSAT with Tabu Search - WalkSAT/Tabu . . . 21
2.3.15 Novelty . . . 21
2.3.16 Novelty$^+$ . . . 22
2.3.17 Novelty$^+$ with Diversification Probability - Novelty^{++} . 22
2.3.18 R-Novelty . . . 23
2.3.19 R-Novelty$^+$ . . . 23
2.3.20 Adaptive Novelty$^+$ . . . 24
2.3.21 Deterministic Adaptive Novelty$^+$ . . . 25
2.3.22 Conflict-Directed Random Walk . . . 25
2.3.23 Conflict-Directed Random Walk for k-SAT . . . 26
2.3.24 Deterministic Conflict-Directed Random Walk . . . 27
2.3.25 Gradient-based Greedy WalkSAT - G^{2}WSAT . . . 27
2.3.26 *adapt*G^{2}WSAT . . . 28
2.3.27 Novelty$^+$ with look-ahead - Novelty^+p . . . 30
2.3.28 Novelty^{++} with look-ahead - Novelty^{++}p . . . 30
2.3.29 *adapt*G^{2}WSAT^+p . . . 30

2.3.30 Gradient-based Greedy WalkSAT with look-ahead - G^2WSAT^+p . . . 32
2.3.31 Variable Weighting Scheme One - VW1 . . . 32
2.3.32 Variable Weighting Scheme Two - VW2 . . . 33
2.3.33 Dynamic Local Search Algorithms for SAT . . . 33
2.3.34 Scaling and Probabilistic Smoothing - SAPS . . . 35
2.3.35 RSAPS . . . 36
2.3.36 De-randomized version of SAPS - SAPS/NR . . . 38
2.3.37 Divide and Distribute Fixed Weights - DDFW . . . 38
2.3.38 Pure Additive Weighting Scheme - PAWS . . . 41

3 Stochastic Local Search for SMT **41**
3.1 A basic WalkSMT procedure . . . 41
3.2 Enhancements to the basic WalkSMT procedure . . . 44
3.2.1 Preprocessing . . . 44
3.2.2 Learning . . . 45
3.2.3 Unit simplification . . . 45
3.2.4 Filtering the assignments given to $\mathcal{T}$-solvers . . . 45
3.2.5 Multiple learning . . . 46
3.3 WalkSMT's Specification . . . 47
3.3.1 Naming Convention for Procedures . . . 49
3.3.2 The InputCreator Component . . . 49
3.3.3 The InputReader . . . 50
3.3.4 The Preprocessor . . . 50
3.3.5 The Wrapper Component . . . 51
3.3.6 The Mapper Component . . . 52
3.3.7 The $\mathcal{LA}$-Solver Component . . . 53
3.3.8 Backend . . . 54
3.3.9 Frontend . . . 57
3.3.10 The UBCSAT Component . . . 60

4 Experimental Evaluation **62**
4.1 Testing Description . . . 62
4.2 SMT-LIB Formulas . . . 63
4.3 Random Generated Formulas . . . 63

5 Conclusion **64**

1 Introduction

This document aims to describe an new SMT procedure integrating a Boolean Stochastic Local Search (SLS) algorithm with a $\mathcal{T}$-solver by lazy approach. The SLS-based SMT solver uses the SLS solver to explore the set of assignments in order to find an optimal truth assignment for the Boolean abstraction of the input formula. Then it invokes the $\mathcal{T}$-solver to find conflicts which are used for guiding the SLS search process. We also implemented a group of techniques in order to improve the efficitiveness of our SMT procedure.

The document is structured as follows. Section 2 gives a set of theoretical concepts and describes the state-of-the-art of the SMT procedures and SLS algorithms. Section 3 introduces a basic version of the SLS-based SMT solver, explain a group of techniques which are thought to improve the synergy between the Boolean and the $\mathcal{T}$-specific components and finally describes the specification of the architecture of the SMT-Solver prototype that we developed, called WalkSMT. Section 4 presents a preliminary experimental evaluation of WalkSMT which is based on the integration of UBCSAT SLS platform with the $\mathcal{LA}(\mathcal{Q})$-solver of MathSAT. In order to evaluate the performance we compare our SMT prototype against MathSAT, a state-of-the-art DPLL-based SMT solver, on both structured industrial problems coming from the SMT-LIB and randomly-generated unstructured problems. Finally, section 5 summarizes our work.

2 Background

2.1 SAT solvers

Given an input Boolean formula ϕ, a SAT solver is a procedure which decides whether ϕ is satisfiable and, if yes, it returns a satisfying assignment μ.

Most state-of-the-art SAT solvers are evolution of Davis-Putnam-Logemann-Loveland [18]. Modern DPLL engines can be divided into two main families:

- *conflict-driven DPLL*, whose search process is driven by the analysis of the conflicts at every failed branch;
- *look-ahead DPLL*, whose search process is driven by a look-ahead procedure which evaluates the reduction effect of the selection of each variable in a group.

A schema of a modern conflict driven DPLL engine can be shown in Algorithm 1. It take as input a Boolean formula φ and the truth assignment μ which is initially empty and then is updated in a stack-based manner.

At the beginning, **preprocess**(φ, μ) rewrites the input formula φ into a simpler and equi-satisfiable formula updating μ if it is the case. It return UNSAT if the resulting formula is unsatisfiable.

In the outer while loop, the procedure **decide-next-branch**(φ, μ) selects an unassigned literal l, called *decision literal*, from φ and adds it to the truth assignment μ. This step is called *decision* and the number of decision literals of μ after this operation is called *decision level* of l. There exists many heuristic criteria for selecting the decision literal. For example, we can pick the literal occurring most often in the minimal-size clauses or we can select a candidate set of literals, performs *Boolean Constraint Propagation (BCP)*, chooses the literal leading to the smallest clause set. It could be useful to provide to the DPLL solver a list of variables (called *privileged variables*) on which branch first. Most conflict-driven DPLL solvers select a decision literal according to a score which is update at the end of a branch and the privileged variables occur in recently-learned clauses.

In the inner while loop, **deduce**(φ, μ) applies unit-propagation, namely it iteratively deduces literals l deriving from the current assignments (namely, $\varphi \wedge \mu \models_p l$) and updates ϕ and μ accordingly. It continues until either no literals can be deduced, μ satisfies φ or μ falsifies φ, returning CONFLICT, SAT and UNKNOWN respectively.

If **deduce**(φ, μ) return SAT then the DPLL solver terminates with SAT. Notice that modern conflict-driven SAT solvers typically return total truth assignments even though the formulas are satisfied by partial assignments. If **deduce**(φ, μ) return CONFLICT then **analyze-conflict**(φ, μ) searches for the subset η, called *conflict set*, of μ which caused the conflict and the decision level to backtrack. When **blevel** $=$ **0** the DPLL solver return UNSAT since a conflict exits. When **blevel** $\neq$ **0** **backtrack**$(blevel, \varphi, \mu)$ adds $\neg\eta$ to the formula ϕ (this technique is called *learning*) and backtracks up to **blevel** (this technique is called *backjumping*) updating φ and μ accordingly. Finally, if **deduce**(φ, μ) return UNKNOWN then the DPLL solver looks for the next decision.

In order to compute the conflict set, **analyze-conflict**(φ, μ) label each literal with its decision level, namely the literal corresponding to the nth decision and the literals derived by unit propagation after that decision are labeled with n. Moreover, it labels each non-decision literal l in μ with a link to the clause C_l causing its unit-propagation, it is called the *antecedent clause* of l. If the clause C is falsified by μ then we say that there is a *conflict* and C is the *conflicting clause*. So **analyze-conflict**(φ, μ) compute the *conflict clause* C' so that it contains only one l_u literals which has been assigned at the last decision level. In particular, it starts from C by iteratively resolving the clause with the antecedent clause C_l of some literal (typically the last-assigned) in C' until a stop criterion is met. For example, the *1st UIP strategy* picks the last-assigned literal in C' and it stops when C' contains only one literal l_u assigned at the last decision level. The *last UIP strategy*, l_u must be the last decision literal. Building a conflict set/clause corresponds to building and analyzing the implication graph corresponding to the current assignment. An implication graph is a DAG where each node represents a variable assignment (literal), the node of a decision literal has no incoming edges, all edges incoming into a non-decision-literal node l are labeled with the antecedent clause. When both l and $\neg l$ occur in the implication graph we have a conflict; given a partition of the graph with all decision literals on one side and the conflict on the other, the set of the source nodes of all arcs intersecting the borderline of the partition represents a conflict set. A node l_u in an implication graph is a *unique implication point (UIP)* for the last decision level if any path from the last decision node to both the conflict nodes passes through l_u. The most recent decision node is a UIP (the last UIP) and the most-recently-assigned UIP is called the 1st UIP. When the conflict clause is computed then **analyze-conflict** add it to the formula and **backtrack**$(blevel, \varphi, \mu)$ pops all assigned literals out of μ up to a decision level **blevel** deriving from C' which can be computed according to different strategies. In modern conflict-clause DPLL, it backtracks to the highest point in the stack where the literal l_u in the learned clause C' is not assigned and unit-propagates l_u.

2.2 Lazy Satisfiability Modulo Theory

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some decidable theory $\mathcal{T}$ [18].

Typically, $SMT(\mathcal{T})$ problems have to test the satisfiability of formulas which are Boolean combination of atomic propositions and atomic expression in some theory $\mathcal{T}$.

In the last decade we have witnessed an impressive advance in the effi-

Algorithm 1 DPLL solver

```
Require: boolean formula $\varphi$, assignment $\&\mu$
 1: if preprocess($\varphi, \mu$) = CONFLICT then
 2:     return UNSAT
 3: end if
 4: while 1 do
 5:     decide-next-branch($\varphi, \mu$)
 6:     while 1 do
 7:         $status$ ← deduce($\varphi, \mu$)
 8:         if status = SAT then
 9:             return SAT
10:         else if $status$ = CONFLICT then
11:             blevel ← analyze-conflict($\varphi, \mu$)
12:             if blevel = 0 then
13:                 return UNSAT
14:             else
15:                 backtrack($blevel, \varphi, \mu$)
16:             end if
17:         else
18:             break
19:         end if
20:     end while
21: end while
```

ciency SAT solvers theory-specific decision procedures which are very important tool in most application areas.

So, during the last ten years, different techniques have been proposed in order efficiently integrate SAT solvers with *theory solver*[1] respectively handling the Boolean and the theory-specific part of reason.

There are two opposite approaches to $SMT(\mathcal{T})$ called *lazy* and *eager*. The dominating is the *lazy* approach which aims to integrate a SAT solver and one (or more) $\mathcal{T}$-solver. The idea is of using the SAT solver to enumerate assignments which satisfy Boolean abstraction of the input formula so that the $\mathcal{T}$-solver is assigned to check the consistency in $\mathcal{T}$ of the set of the set of literals corresponding to the assignments enumerated. Instead, the eager approach is based on the idea of encoding a SMT formula into an equivalently satisfiable Boolean formula whose satisfiability is tested using a SAT solver.

The *lazy* SMT procedures can be partitioned in two main categories: *offline* and *online* procedures. In the *offline* schema, the DPLL procedure is used as a SAT solver which is invoked form scratch whenever an assignment is not $\mathcal{T}$-satisfiable. In the *online* schema, the DPLL procedure is modified to behave like an enumerator of truth assignments whose $\mathcal{T}$-satisfiability is checked by the $\mathcal{T}$-solver.

The Algorithm 2 shown the schema of a lazy online SMT procedure based on modern evolution of DPLL.

Notice that we use the superscript ^{p} to denote the boolean abstraction, namely given a $\mathcal{T}$-formula ζ we write ζ^p to denote $\mathcal{T}2\mathcal{B}(\zeta)$ [2].

2.3 Sthocastic Local Search

Local search (LS) algorithms [8, 7] are widely used approaches 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 neighbouring one where each move is determined by a decision based on information about the local neighbourhood.

[1]Given a theory $\mathcal{T}$, a *theory solver* ($\mathcal{T}$-solver) is a procedure which decides whether any given finite conjunction of quantifier-free literals expressed in $\mathcal{T}$ is $\mathcal{T}$-satisfiable or not.

[2]We define the bijective function $\mathcal{T}2\mathcal{B}$ and its inverse $\mathcal{B}2\mathcal{T}=\mathcal{T}2\mathcal{B}^{-1}$, such that it is a which maps boolean atoms into themselves and non-boolean $\mathcal{T}$-atoms into fresh boolean atoms and distributes with sets and boolean connectives [18].

Algorithm 2 Lazy-SMT-DPLL

```
Require: $\mathcal{T}$-formula $\varphi$, $\mathcal{T}$-assignment &$\mu$
 1: if preprocess($\varphi, \mu$) = CONFLICT then
 2:     return UNSAT
 3: end if
 4: $\varphi^p \leftarrow \mathcal{T}2\mathcal{B}(\varphi)$
 5: $\mu^p \leftarrow \mathcal{T}2\mathcal{B}(\mu)$
 6: while 1 do
 7:     $\mathcal{T}$-decide-next-branch($\varphi^p, \mu^p$)
 8:     while 1 do
 9:         $status \leftarrow \mathcal{T}$-deduce($\varphi^p, \mu^p$)
10:         if status = SAT then
11:             $\mu = \mathcal{B}2\mathcal{T}(\mu^p)$
12:             return SAT
13:         else if $status$ = CONFLICT then
14:             blevel $\leftarrow \mathcal{T}$-analyze-conflict($\varphi^p, \mu^p$)
15:             if blevel = 0 then
16:                 return UNSAT
17:             else
18:                 $\mathcal{T}$-backtrack($blevel, \varphi^p, \mu^p$)
19:             end if
20:         else
21:             break
22:         end if
23:     end while
24: end while
```

LS algorithms making use of randomized choices during the search process are called *Stochastic Local search (SLS) algorithms.* Given a problem Π and a probability distribution $\mathcal{D} : S \mapsto \mathbb{R}_0^+$ mapping the elements of S into their probabilities, the components which define a SLS algorithm for solving a problem instance $\pi \in \Pi$ are:

- the *search space* $S(\pi)$ of the problem instance π, which is a finite set of search positions $s \in S(\pi)$ (also called candidate solutions, locations, configurations or states);
- a set of *solutions* $S'(\pi) \subseteq S(\pi)$;
- a *neighbourhood relation* $N(\pi) \subseteq S(\pi) \times S(\pi)$ on $S(\pi)$, which specifies the set of neighbouring positions which can be visited in one local search step for each position in $S(\pi)$;
- a finite set of memory states $M(\pi)$, which can be used to represent information exploited by the algorithm to control the search process (if the SLS algorithm does not use memory then it consists of a single state);
- an *initialization function* $init(\pi) : \emptyset \mapsto \mathcal{D}(S(\pi) \times M(\pi))$, which specifies a probability distribution over initial search position and memory state[3];
- an *step function* $step(\pi) : S(\pi) \times M(\pi) \mapsto \mathcal{D}(S(\pi) \times M(\pi))$, which maps each search position and memory state onto a probability distribution over it neighbouring search positions and memory states;
- a *termination predicate* $terminate(\pi) : S(\pi) \times M(\pi) \mapsto \mathcal{D}(\{T, \bot\})$, which maps each search position and memory state onto a probability distribution over truth values indicating the probability with which the search is to be terminated when a given search state is reached.

The Algorithm 3 shows the general outline of a SLS algorithms for a decision problems Π.

In order to determine the quality of solutions and guide the search, SLS algorithms use an *evaluation function* $g(\pi)(s) : S(\pi) \rightarrow \mathbb{R}$ mapping each

[3]The combination of search position and memory state forms the state of the SLS algorithm. It is called *search state*.

Algorithm 3 Stochastic Local Search Decision Algorithm

```
Require: problem instance $\pi$
1: $(s, m) \leftarrow init(\pi, m)$
2: while not $terminate(\pi, s, m)$ do
3:     $(s, m) \leftarrow step(\pi, s, m)$
4: end while
5: if $s \in S'(\pi)$ then
6:     return $s$
7: else
8:     return $\emptyset$
9: end if
```

search position onto real number so that the global optima of π correspond to the solutions of π.

SLS algorithm are typically incomplete, that is, they do not guarantee that eventually an existing solution is found, so the unsatisfiability of a formula cannot be determined with certainty. Moreover, these algorithms can visit the same position in the search space more than one time. Thus they can get stuck in local minima, namely search positions having no improving neighbours, and plateau regions of the search space, namely regions which do not contain high-quality solutions. This leads to premature stagnation of the search which can be avoided applying special mechanisms. For example, reinitializing the search after a fixed number of steps if no solution is found or performing diversification steps.

2.3.1 Stochastic Local Search for SAT

Recently, SLS algorithms have been successfully applied to the solution of $\mathcal{NP}$-complete decision problem such as the Satisfiability Problem in Propositional Logic (SAT).

SLS algorithms for SAT typically work on propositional formulae in conjunctive normal form[4] (CNF). So, given a SAT instance, namely a CNF formula φ, the search space is defined as the set of all possible variable assignments of φ and the set of solutions is given by the set of models (satisfying assignments). A frequently used neighbourhood relation is the *one-flip neigh-*

[4]A propositional formula φ is in *conjunctive normal form* if, and only if, it is a conjunction over disjunction of literals, namely propositional variables or their negation.

bourhood which defines two variable assignments to be direct neighbours if, and only if, they differ in the truth value of exactly one variable. Therefore, during a search step the truth value of one propositional variable is flipped.

Generally, SLS algorithms for SAT have the same schema of the Figure 4. They initialize the search by generating uniformly at random an initial truth assignment for the input formula φ. Then they iteratively select one variable and flip its 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 to find a model for φ. The main difference in SLS algorithms for SAT is typically the strategy which is applied to select the variable to be flipped.

Algorithm 4 WalkSAT (φ)

```
Require: CNF formula $\varphi$, $MAX\_TRIES$, $MAX\_FLIPS$
 1: for $i = 1$ to $MAX\_TRIES$ do
 2:     $a \leftarrow random\_truth\_assignment(\varphi)$
 3:     for $j = 1$ to $max\_steps$ do
 4:         if $a$ satisfies $\varphi$ then
 5:             return $a$
 6:         else
 7:             $x \leftarrow select\_variable(\varphi)$
 8:             $a \leftarrow flip\_truth\_value(x, a)$
 9:         end if
10:     end for
11: end for
12: return UNKNOWN
```

2.3.2 GSAT Architecture

The GSAT algorithm was one of the first SLS algorithms for SAT.

GAST [8, 7] is a greedy local search procedure which explores the set of assignments (namely, potential solutions) that differ from the current one on only one variable. In particular, GSAT is based on a 1-exchange neighborhood in the space of all complete truth value assignments of the given formula. This means that two variable assignments are neighbors if, and only if, they differ in the truth assignment of exactly one variable

GSAT tries to minimize the number of unsatisfied clauses by a greedy descent in the space of variable assignments. Therefore, it uses an evaluation function[5] $g(F, a)$ that maps each variable assignment a to the number of clauses of the given formulas F unsatisfied under a (models of F are the assignments with evaluation function value zero).

GSAT and most of its variants are iterative improvement[6] methods that flip the truth value of one variable in each search step. The selection of the variable to be flipped is typically based on the score of a variable x under the current assignment a, which is defined as the difference between the number of clauses unsatisfied by a and the assignment obtained by flipping x in a.

Algorithms of the GSAT architecture differ primarily in their variable selection method.

2.3.3 Basic GSAT - GSAT

GSAT [20, 8] consists of a best-improvement search procedure[7] which starts with a randomly generated truth assignment and then flips the assignment of the variable that leads to the largest increase in the total number of satisfied clauses (namely, the variable with maximal score). The variable whose assignment is to be changed is chosen at random from those that would give an equally good improvement (this make it unlikely that the algorithm makes the same sequence of changes over and over). The flips are repeated until either a satisfying assignment is found or a preset maximum number of flips is reached (MAX-FLIPS); this process is repeated until a maximum number of times is reached (MAX-TRIES). GSAT uses this static restart mechanism that re-initialize the search process since it gets easily stuck in local minima[8] of the evaluation function[9].

[5] *Evaluation function* [8] is used for assessing or ranking candidate solutions in the neighborhood of the current search position.

[6] *Iterative Improvement* [8] starts form a randomly selected point in the search space and then tries to improve the current candidate solution w.r.t the evaluation function.

[7] *Iterative Best Improvement* [8] which is based on the idea of randomly selecting in each search step one of the neighboring candidate solutions that achieve a maximal improvement in the evaluation function.

[8] *Local minimum* is defined as a search position without improving neighbors.

[9] There are two main simple mechanisms for escaping from local optima [8]: restart or non-improving steps. Restart means re-initialize the search process whenever a local optimum is encountered (often rather ineffective due to cost of initialization). Non-improving steps means to select candidate solutions with equal or worse evaluation function value when we get stuck in local minima (can lead to long walks in plateaus, namely those

The GSAT procedure requires the settings of the two parameter MAX-FLIPS and MAX-TRIES. MAX-FLIPS is usually set equal to a few times the number of variables and MAX-TRIES will generally be determined by the total amount of time that one wants to spend looking for an assignment, which depends on the application.

Clearly for any fixed number of restarts GSAT is not complete, namely it could fail to find an assignment even if one exists, and severe stagnation behavior is observed on most SAT instances. Even if GSAT, when it was introduced, outperformed the best systematic search algorithms for SAT available at that time, basic GSAT's performance is substantially weaker then that of the algorithm which are described in the following.

The Algorithm 5 shows a pseudo-code representation of the basic GSAT algorithm.

Algorithm 5 GSAT

```
Require: $F, MAX-STEPS, MAX-TRIES$
1: for $i = 1$ to $MAX\_TRIES$ do
2:     randomly generate a truth assignment $a$
3:     for $j = 1$ to $MAX\_STEPS$ do
4:         if $a$ satisfies $F$ then
5:             return $a$
6:         else
7:             randomly select a variable $x$ flipping that minimizes the
               number of unsatisfied clauses
8:             flip the variable $x$ in $a$
9:         end if
10:    end for
11: end for
12: return no solution found
```

regions of search positions with identical evaluation function).

2.3.4 GSAT with Random Walk - GWSAT

GWSAT [19, 8] consists of a randomized[10] best-improvement search procedure which is based on the idea to decide at each local search step with a fixed probability *wp* (called *walk probability* or *noise setting*) whether to do a standard GSAT step or a random walk step, in which a variable is selected uniformly at random from the set of all variables occurring in currently unsatisfied clauses and then flipped. To be more clear the Random Walk Strategy works as follows:

- With probability *wp*, pick a variable occurring in some unsatisfied clause and flip its truth assignment.
- With probability $1 - wp$, follow the standard GSAT scheme, i.e., make the best possible local move.

When using sufficiently high noise settings (the precise value depends on the problem instances) GWSAT does not suffer from stagnation behavior but its performance can decrease uniformly.

2.3.5 GSAT with Random Noise - GUWSAT

A simpler variation of the random walk strategy is not to restrict the choice of a randomly flipped variable to the set of variables that appear in unsatisfied clauses. [19] refers to this modification as the *random noise* strategy and [8] summarizes it like follows:

- With probability *wp*, pick a variable uniformly at random from the set of all variable in the formula.
- With probability $1 - wp$, follow the standard GSAT scheme, i.e., make the best possible local move.

This algorithm is called GUWSAT [11].

[10] *Randomized Iterative Improvement* [8] is based on the idea of introducing a parameter which correspond to the probability of performing a random walk step instead of an improvement step.

[11] GUWSAT is not supported by the UBCSAT tool.

2.3.6 GSAT with Tabu Search - GSAT/Tabu

2.3.7 Introduction to Tabu Search

An approach for escaping from local minima is to use aspects of the search history (memory) rather than random or probabilistic techniques for accepting worsening search steps.

Tabu Search (TS) [8] is a SLS method that systematically utilizes memory for guiding the search process. To select the neighbor of the current candidate solution in each search step it uses a best improvement strategy which in a local optimum can lead to a worsening or plateau steps. In order to prevent the local search to immediately consider candidate solutions which has been already visited, TS forbids steps to recently visited search positions. A parameter *tt*, called *tabu tenure*, is used for determining the duration (namely, the number of search steps) for which the restrictions apply.

This mechanism can also forbid search step which could lead to good and previously unvisited candidate solutions, thus TS make use of *aspiration criterion* which specifies conditions under which the tabu status of candidate solutions or solution components is overridden.

The performance of TS depends on the settings of the tabu tenure parameter *tt*. If *tt* is chosen to too small then search stagnation may occur and if *tt* is too large, the search path is too restricted and high-quality solutions may be missed (moreover, as with GWSAT's noise parameter, very high settings of *tt* can cause an uniformly decrease of the performance).

There exists approaches to make the settings of *tt* more robust (*Robust Tabu Search* repeatedly choose *tt* from given interval) or to adjust *tt* dynamically during the run of the algorithm (*Reactive Tabu Search* dynamically adjust *tt* during search).

2.3.8 GSAT/Tabu

GSAT/Tabu [8] is obtained form basic GSAT by associating a tabu status with propositional variables of the given formula. The idea is that after a variable has been flipped, it cannot be flipped back within the next *tt* steps (where *tt* is the tabu tenure). Therefore, in each search step, the variable to be flipped is selected in as in basic GSAT, except that the choice is restricted to the variables that are currently not tabu.

Using instance-specific optimized tabu tenure for GSAT/Tabu settings and similarly optimized noise settings for GWSAT, GSAT/Tabu typically (but not always as in the case of logistic planning problems) performs significantly better than GWSAT (particularly when applied to large and structured SAT instances).

Finally, GSAT/Tabu can be extended with a random walk mechanism (but typically the extension does not perform better).

2.3.9 GSAT with History Information - HSAT

The basic idea of the HSAT [8, 5] algorithm is that in basic GSAT some variables might never get flipped even if they are frequently suitable to be chose. The stagnation can occur since one of the variables may have to be flipped to improve the search.

Therefore HSAT is based on basic GSAT but it uses historical information (memory) in order to choose deterministically which variable to pick. In particular, when in a search step there are several variables with the same score, HSAT always pick the one that was flipped longest ago, namely the least recently flipped variable.

[8] says that when there are sill variables that have not been flipped, it performs the same random tie breaking between variables with identical score as plain GSAT, instead [5] says that an arbitrary (but fixed) ordering is used to choose between them.

Even if HSAT has superior performance respect to basic GSAT, HSAT is more likely to get stuck in local minima from which it cannot escape since the rule described above restricts the search trajectories. .

2.3.10 HSAT with Random Walk - HWSAT

HWSAT [8, 5] is an extension of HSAT algorithm with the random walk mechanism which is used in GWSAT and its target is to resolve HSAT's problems.

HWSAt shows improved peak performance over GWSAT and compared to GSAT/Tabu, its performance appears to be somewhat better on certain problem instances (like, for instance, Uniform Random 3-SAT) and worse in others cases.

2.3.11 Restarting GSAT - RGSAT

RGSAT [21] uses the same best improvement search method of GSAT (and SAPS), but restarts the search from another random truth assignment whenever it cannot perform an improving search step since a local minimum or plateau of the given search landscape is encountered.

RGSAT has no plateau search capabilities, but it can also never get stuck in a local minimum. Its performance is quite poor, but it provides interesting insights into the hardness of SAT instances for simple local search methods.

2.3.12 WalkSAT Architecture

The WalkSAT architecture [8, 7] is based on a 2-stage variable selection process focused on the variables occurring in currently unsatisfied clauses. For each local search step, the two stages behave like follows:

- first stage: a currently unsatisfied clause is randomly selected.
- second stage: one of the variables (which is selected, for example, at random or according to a greedy heuristic) appearing in the selected clause is then flipped to obtain the new assignment.

WalkSAT algorithms are based on a dynamically determined subset of the GSAT neighborhood relation (while the GSAT architecture is characterized by a static 1-exchange neighborhood relation).

The consequence of the reduced effective neighborhood size is that WalkSAT algorithm can be implemented efficiently without caching and incrementally updating variable scores and achieve lower CPU time per search step then efficient GSAT implementations.

The Algorithm 6 shows a pseudo-code representation of the WalkSAT algorithm family.

2.3.13 WalkSAT/SKC

In WalkSAT/SKC [8, 7] (or also called WSAT for walk sat in [19]), the scoring function $score_b(x)$ counts only the number of clauses (that will be broken) which are currently satisfied but will become unsatisfied by flipping variable x. Using this scoring function, the following variable selection scheme is applied:

- If there is a variable with $score_b(x) = 0$ in the selected clause (namely, if the selected clause can be satisfied without breaking another clause), this variable is flipped (so-called *zero-damage step*).

Algorithm 6 WalkSAT

```
Require: $F, MAX-STEPS, MAX-TRIES, slc$
 1: for $i = 1$ to $MAX\_TRIES$ do
 2:     randomly generate a truth assignment $a$
 3:     for $j = 1$ to $MAX\_STEPS$ do
 4:         if $a$ satisfies $F$ then
 5:             return $a$
 6:         else
 7:             randomly select a clause $c$ unsatisfied under $a$
 8:             pick a variable $x$ from $c$ according to heuristic function $slc$
 9:             flip the variable $x$ in $a$
10:         end if
11:     end for
12: end for
13: return no solution found
```

- If there no exists such variable, with a certain probability p (*noise setting*) the variable with minimal $score_b$ value is selected otherwise one of the variables from the selected clause is picked uniformly at random (*random walk step*).

Conceptually WalkSAT is closely related to GWSAT, but the former has generally superior performance. Both algorithms use the same kind of random walk steps but WalkSAT applies them only under the condition that there is no variable with $score_b(x) = 0$. While, in GWSAT random walk steps are done in an unconditional probabilistic way. Therefore WalkSAT can be considered greedier since random walk steps, which usually increase the number of unsatisfied clauses, are only done when every variable occurring in the selected clause would break some clauses when flipped.

Moreover, WalkSAT chooses from a significantly reduced set of neighbors in a greedy step thanks to the two-stage variable selection scheme, therefore it can be considered to be less greedy than GWSAT.

Finally, because of the different scoring function GWSAT shows a greedier behavior than WalkSA since the former may prefer a variable that breaks some clauses but compensates for this by fixing some other clauses, while in the same situation, WalkSAT would select a variable with a smaller total score, but breaking also a smaller number of clauses.

2.3.14 WalkSAT with Tabu Search - WalkSAT/Tabu

WalkSAT/Tabu [8, 7] extends WalkSAT/SCK in oder to use a simple tabu search mechanism. In particular it uses the same two stage selection mechanism and the same scoring function as WalkSAT but enforces a tabu tenure of tt steps for each flipped variable.

If no zero-damage flip can be made, from all variables which are not tabu, the one with the highest $score_b$ value is picked. When there are several variables with the same maximal score, one of them is selected uniformly at random. It may happen that all variables appearing in the selected clause cannot be flipped because they are tabu (as a result of the two-level variable selection scheme), therefore no variable is flipped (a so-called *null-flip*).

Moreover WalkSAT/TABU has been shown to be essentially incomplete since it can get stuck in local minima regions of the search space.

2.3.15 Novelty

Novelty [8, 7, 22] is a WalkSAT algorithm that uses a history-based variable selection mechanism similar to HSAT. It considers the number of local search step that have been performed since a variable was last flipped (this value is called the variable's *age*).

Differently by WalkSAT/SKC and WalkSAT/Tabu, Novelty uses the same scoring function as GSAT (namely, the number of clauses of the given formula unsatisfied under a given assignment).

After an unsatisfied clause has been chosen, the variable to be flipped is selected as follows:

- if the variable with the highest score does not have minimal age among the variables within the same clause, it is always selected.
- otherwise
  - with a probability of $1-p$ (where p is the noise setting) the variable with the highest score is only selected.
  - otherwise the variable with the next lower score is selected.

In Kautzs and Selmans implementation, if there are several variables with identical score, the one appearing first in the clause is always chosen.

Note that for $p > 0$ the age-based variable selection of Novelty probabilistically prevents flipping the same variable over and over again. At the same time, flips can be immediately reversed with a certain probability if a better choice is not available. Generally, the Novelty algorithm is significantly greedier than WalkSAT, since always one of the two most improving variables from a clause is selected, where WalkSAT may select any variable if no improvement without breaking other clauses can be achieved. Moreover, Novelty is more deterministic than WalkSAT and GWSAT, since its probabilistic decisions are more limited in their scope and take place under more restrictive conditions. Novelty has higher performance then WalkSAT, but it can be shown that Novelty is essentially incomplete since selecting only among the best two variables in a given clause can lead to situations where the algorithm gets stuck in local minima of the objective function.

2.3.16 Novelty$^+$

The Novelty$^+$ [8, 22] algorithm selects the variable to be flipped according to the Novelty mechanism with probability $1 - p$, otherwise it performs a random walk step (as defined for GWSAT, namely it picks a variable occurring in some unsatisfied clause and flip its truth assignment). Small walk probabilities p are generally sufficient to prevent the extreme stagnation behavior that is occasionally observed in Novelty and to achieve substantially superior performance compared to Novelty.

Moreover, Novelty$^+$ is one of the best-performing WalkSAT algorithms currently known and of the best SLS algorithms for SAT available to date [7].

2.3.17 Novelty$^+$ with Diversification Probability - Novelty^{++}

The Novelty^{++} [13] algorithm with probability dp (diversification probability) picks the last recently flipped variable in the selected clause c (diversification), otherwise it performs as Novelty.

Substantially, the random walk in Novelty$^+$ is replaced by the diversification in Novelty^{++} (Novelty^{++} is stronger then Novelty$^+$).

When Novelty gets stuck in local minima, probably there is a clause c that is unsatisfied again and again. The diversification step allows to flip all variables in c by turns during the search, since after the last recently variable in c is flipped, a different variable in c becomes the new last recently flipped.

[13] shows that Novelty^{++} is consistently better than Novelty and Novelty$^+$ in case noise is important and in case stagnation behavior occurs. In other

words, when random walk is needed, diversification systematically does better.

2.3.18 R-Novelty

R-Novelty [8, 7] is a variant of Novelty (they use the same scoring function) which is based on the idea that when deciding between the best and second best variable, the difference of the respective scores should be taken into account. In particular, after an unsatisfied clause has been chosen, the variable to be flipped is selected as follows:

- if the variable with the highest score does not have minimal age among the variables within the same clause, it is always selected
- otherwise the score difference considered:
  - if the score difference is grater then one:
    - with a probability of p (where p is the noise setting) the variable with the higher score is only selected.
    - otherwise the variable with the next lower score is selected.
  - if the score difference is equal to one:
    - with a probability of $1 - p$ (where p is the noise setting) the variable with the higher score is only selected.
    - otherwise the variable with the next lower score is selected.

The R-Novelty algorithm gets too easily stuck in local minima. Therefore it is used a simple loop breaking strategy which randomly chooses a variable from the selected clause and flip it every 100 steps. This mechanism has been shown to be insufficient for effectively escaping from local minima, thus R-Novelty is essentially incomplete.

2.3.19 R-Novelty$^+$

As used in Novelty$^+$, R-Novelty$^+$ selects the variable to be flipped according to the R-Novelty mechanism with probability $1 - p$, otherwise it performs a random walk step (as defined for GWSAT, namely it picks a variable occurring in some unsatisfied clause and flip its truth assignment).

There is some indication that R-Novelty$^+$ and R-Novelty do no reach the performance of Novelty on several classes of structured SAT instances [8].

2.3.20 Adaptive Novelty$^+$

The noise parameter, p, which controls the degree of randomness of the search process, has a major impact on the performance and run-time behavior of Novelty+. Unfortunately, the optimal value of p varies significantly between problem instances, and even small deviations from the optimal value can lead to substantially decreased performance.

The idea behind Adaptive Novelty$^+$ [22, 6, 8] is to use hight noise values only when they are needed to escape from stagnation situations in which the search procedure appear to make no further progress towards finding a solution. It dynamically adjusts the noise setting p based on search progress, as reflected in the time elapsed since the last improvement in the number of satisfied clauses has been achieved.

At the beginning of the search, the search is maximally greedy ($p = 0$). This will typically lead to a series of rapid improvements in the evaluation function value that can be followed by stagnation. In this situation, the noise value is increased. If the resulting increase in the diversification of the search process is not sufficient to escape from the stagnation situation (that is, if it does not lead to an improvement in the number of satisfied clauses within a certain number of steps) the noise value is further increased. Eventually, p should be high enough for the search process to overcome the stagnation situation, at which point the noise can be gradually decreased, leading to an increase in search intensification, until the next stagnation situation is detected or a solution to the given problem instance is found.

As an indicator for search stagnation it is used a predicate that is true if and only if no improvement in objective function value has been observed over the last $\theta \times m$ search steps, where m is the number of clauses of the given problem instance and θ is a parameter.

Every incremental increase in the noise value is realized as $p = p + (1 - p) \times \phi$. The decrements are defined as $p = p - p \times \phi/2$ where ϕ is an additional parameter. For Adaptive Novelty$^+$ the parameters ϕ and θ are the followings: $\phi = 0.2$ and $\theta = 6$. The asymmetry between increases and decreases in the noise setting is motivated by the fact that detecting search stagnation is computationally more expensive than detecting search progress. After the noise setting has been increased or decreased, the current objective function value is stored and becomes the basis for measuring improvement, and hence for detecting search stagnation. As a consequence, between increases in noise level there is always a phase during which the trajectory is monitored for search progress without further increasing the noise. No such delay is enforced between successive decreases in noise level.

Adaptive Novelty$^+$ typically achieves the same performance as Novelty$^+$ with approximately optimal static noise, which renders it one of the best-performing and most robust SLS algorithms for SAT currently available.

2.3.21 Deterministic Adaptive Novelty$^+$

Deterministic Adaptive Novelty$^+$ algorithm has been developed with the intent to derandomize the Adaptive Novelty$^+$ algorithm (in particular three type of random decision: unsatisfied clause selection, random walk steps and noisy variable selection).

In order to select a clause, the algorithm maintains a list of the currently false clauses and simply step through that list, selecting the clause in the list that is the current search step number modulo the size of the list.

For random walk steps, every ($\lfloor 1/wp \rfloor$) steps a variable is selected to be flipped using the same variable selection scheme used by Deterministic Conflict-Direct Random Walk (i.e., the algorithm keeps a counter for each clause, selecting the first variable the first time the clause is selected, the second variable the second time, and so on, returning to the first variable when all have been exhausted).

For the noisy variable selection, the algorithm uses two integer variables n and d. If the ratio (n/d) is less than the current noise setting p a noisy decision is made and n is incremented otherwise if (n/d) is greater than p the greedy decision is made and d is incremented. Whenever the adaptive mechanism modifies the noise parameter p, the values of n and d are reinitialized to $\lfloor 256 \times p \rfloor$ and $(256n)$, respectively.

UBCSAT developers says that this algorithm was developed for academic interest, and is not recommended for practical applications.

2.3.22 Conflict-Directed Random Walk

Conflict-Directed Random Walk algorithm [8, 14] (also known as Papadimitriou's algorithm) starts with a randomly generated truth-assignment and while there are unsatisfied clauses (it performs a sequence of these conflict-random walk steps):

1. selects a currently unsatisfied clause c uniformly at random;

2. selects a variable appearing in c randomly and flips it (to force c to be satisfied).

This algorithm has been proven to solve 2-SAT in quadratic expected time [14] and it has been used to extend basic GSAT and to obtain GWSAT.

2.3.23 Conflict-Directed Random Walk for k-SAT

Conflict-Directed Random Walk for k-SAT [17] (also known as Schning's algorithm) has been designed to solve k-SAT and more generally constraint satisfaction problems. It is the Conflict-Direct Random Walk algorithm described in 2.3.22 with a restart every $3n$ steps, see Algorithm 7.

For any satisfiable k-CNF formula with n variables the conflict-random walk steps has to be repeated only t times (on the average) to find a model for the formula, where t is within a polynomial factor of $(2(1 - 1/k))^n$.

Algorithm 7 Conflict-Directed Random Walk for $k-$SAT

```
Require: $F$
1: for $t \leftarrow 0$ to $3n$ times do
2:     randomly generate a truth assignment $a$
3:     if $a$ satisfies $F$ then
4:         return $a$
5:     else
6:         randomly select a clause $c$ unsatisfied under $a$
7:         randomly pick one of the $\leq k$ literals in $c$ and flip it
8:     end if
9: end for
```

In order to improve this algorithm, [11] combines it with the ResolveSat algorithm [15].

ResolveSat is based on a randomized Davis-Putnam combined with bounded resolution, namely it repeats an exponential number of tries and each try:

1. generate a random initial assignment a;

2. generate a random initial permutation π of $[1, n]$;

3. execute Davis-Putnam based on a and π (which takes at most n steps).

The algorithm in [11] repeats I time the following steps:

1. generate a random initial assignment a;
2. execute a local search 3n steps starting from a and if a satisfying assignment is found then return SAT;
3. execute the step 2 and 3 of the ResolveSat algorithm and if a satisfying assignment is found then return SAT.

If no satisfying assignment is found then it return UNSAT.

2.3.24 Deterministic Conflict-Directed Random Walk

Deterministic Conflict-Directed Random Walk algorithm has been developed with the intent to derandomize the Conflict-Directed Random Walk algorithm (in particular two type of random decision: unsatisfied clause selection, variable selection).

In order to do clause selection (which is uniform, fair and deterministic), the algorithm keeps track of the number of times each clause has been selected t and the number of steps that each clause has been unsatisfied s. Then the algorithm selects the clause that has the smallest ratio (t/s) and on equal values it selects the clause with the smallest index.

For literal selection, the algorithm keeps a counter for each clause, selecting the first literal the first time the clause is selected, the second literal the second time, and so on, returning to the first literal when all have been exhausted.

The Deterministic Conflict-Directed Random Walk algorithm still allows for random decisions at the initialization phase.

UBCSAT developers says that this algorithm was developed for academic interest, and is not recommended for practical applications.

2.3.25 Gradient-based Greedy WalkSAT - G^{2}WSAT

A variable is said decreasing if flipping it would decrease the number of unsatisfied clauses. Let x and y be variables, $x \neq y$, y is not decreasing. If it becomes decreasing after x is flipped, then we say that y is a *promising decreasing variable* after x is flipped.

A *promising decreasing path* is a sequence of moves in which every move flips a promising decreasing variable.

If a variable x is flipped such that the number of clauses is increased, re-flipping x would decrease the number of unsatisfied clauses, namely x is decreasing. However x is not a promising decreasing variable, since re-flipping x would simply cancel a previous move. The idea behind G^{2}WSAT [13, 1] is that such x is never considered when exploiting promising decreasing paths.

Whenever there are promising decreasing variables, G^{2}WSAT (see Algorithm 8) performs as GSAT and deterministically picks the best of them to minimize the total number of unsatisfied clauses (breaking ties in favor of the least recently flipped variable as in HSAT). Otherwise, G^{2}WSAT performs as Walksat and uses Novelty^{++} (it could also use Novelty or Novelty^{+}) to pick the variable to flip from a randomly selected unsatisfied clause. In particular, given an initial assignment, G^{2}WSAT computes the scores for all variables and then uses equation 6 in [13] to maintain a set of promising decreasing variable, update the scores of the neighbors of the flipped variable after each step and flips the best promising decreasing variable if any.

Promising decreasing variables are chosen to flip since they allow local search to explore new promising regions in the search space.

[13] shows that G^{2}WSAT is almost always better than Novelty^{++} (the former needs fewer flips) except for some problems.

UBCSAT also implements a variant in which it uses Novelty+ as the WalkSAT algorithm and selects the oldest (not necessarily the best) decreasing promising variable (this is the algorithm variant used by Adaptive G2WSAT+).

2.3.26 *adapt*G^{2}WSAT

In order to obtain *adapt*G^{2}WSAT, the adaptive noise mechanism of Adaptive Novelty^{+} is implemented in G^{2}WSAT in a way that no parameter have to be manually tubed to solve new problem and achieve good performance.

[1] shows that *adapt*G^{2}WSAT and *adapt*Novelty^{+} achieve good performances (with ϕ and $theta$ fixed values for all problems). Nevertheless, with instance specific noise settings, G^{2}WSAT and Novelty^{+} achieve success rates the same as or higher than *adapt*G^{2}WSAT and *adapt*Novelty^{+}, respectively, for all instances. Moreover, the degradation in performance of *adapt*G^{2}WSAT compared with that of G^{2}WSAT is lower than the degradation in performance of *adapt*Novelty^{+}, compared with that of Novelty^{+}. This observation suggests that the deterministic exploitation of promising decreasing variables enhances the adaptive noise mechanism.

Algorithm 8 G^{2}WSAT

```
Require: $F, MAX-TRIES, MAX-FLIPS, slc$
1: for $i = 1$ to $MAX-TRIES$ do
2:     randomly generate a truth assignment $a$
3:     compute the scores for all variables in $a$
4:     store all decreasing variables in stack $DecVar$
5:     for $j = 1$ to $MAX-FLIPS$ do
6:         if $a$ satisfies $F$ then
7:             return $a$
8:         end if
9:         if $|DecVar| > 0$ then
10:            pick variable $x$ with the highest score, breaking ties in favor
               of the last recently flipped variable
11:        else
12:            randomly selected unsatisfied clause $c$ under $a$
13:            pick a variable $x$ according to Novelty$^{++}$
14:        end if
15:        flip the variable $x$ in $a$
16:        update the scores of the neighbors of the flipped variable $x$
17:        delete variables that are no longer decreasing from $DecVar$
18:        push new decreasing variables into $DecVar$ which are different
           from $x$ and were not decreasing before $x$ is flipped
19:    end for
20: end for
21: return "solution not found"
```

2.3.27 Novelty$^+$ with look-ahead - Novelty^+p

Given a CNF formula and an assignment a, let x be a variable, let b be obtained from a by flipping x, and let x' be the best promising decreasing variable with respect to b. The promising score of a variable x with respect to a is $pscore_a(x) = score_a(x) + score_b(x')$, where $score_a(x)$ is the score of x with respect to a and $score_b(x')$ is the score of x' with respect to b (x' has the highest $score_b(x')$ among all promising decreasing variables with respect to b).

If there are promising decreasing variables with respect to b, the promising score of x with respect to a represents the improvement in the number of unsatisfied clauses under a by flipping x and then x'. In this case, $pscore_a(x) > score_a(x)$. If there is no promising decreasing variable with respect to b, $pscore_a(x) = score_a(x)$.

The computation of $pscore_a(x)$ involves the simulation of flipping x and the searching for the largest score of the promising decreasing variables after flipping x.

Novelty^+p [1] extends Novelty$^+$ in order to exploit limited look-ahead (see Algorithm 9).

2.3.28 Novelty^{++} with look-ahead - Novelty^{++}p

The difference between Novelty^+p and Novelty^{++}p is that, with the random walk probability wp, the former randomly chooses a variable to flip from c, but with the diversification probability dp, the latter chooses a variable in c, whose flip will falsify the least recently satisfied clause.

2.3.29 *adapt*G^{2}WSAT^+p

*adapt*G^{2}WSAT^+p [1] (see Algorithm 10) is a variant of *adapt*G^{2}WSAT which uses Novelty^+p instead of Novelty^{++}. In particular, it maintains a stack $DecVar$ to store all promising decreasing variables in each step. If there are promising decreasing variables then the algorithm chooses the least recently flipped promising decreasing variable among all promising decreasing variables in $|DecVar|$ to flip. Otherwise, the algorithm selects a variable to flip from a randomly chosen unsatisfied clause c, using heuristic Novelty^+p.

This version does not compute the promising scores for the promising decreasing variables with higher scores in $|DecVar|$ but chooses the least

Algorithm 9 Novelty^+p

```
Require: $p, wp, c$
 1: if with probability $wp$ then
 2:     randomly choose a variable $y$ in $c$
 3: else
 4:     $best$ and $second$ are the best and second best variables in $c$ according to the scores, breaking ties in favor of the last recently flipped variable
 5:     if $best$ is the most recently flipped variable in $c$ then
 6:         if with probability $p$ then
 7:             $y$ is the second variable
 8:         else
 9:             if $pscore(second) \geq pscore(best)$ then
10:                 $y$ is the second variable
11:             else
12:                 $y$ is the best variable
13:             end if
14:         end if
15:     else
16:         if $best$ is more recently flipped than $second$ then
17:             if $pscore(second) \geq pscore(best)$ then
18:                 $y$ is the second variable
19:             else
20:                 $y$ is the best variable
21:             end if
22:         else
23:             $y$ is the best variable
24:         end if
25:     end if
26: end if
27: return $y$
```

recently flipped promising decreasing variable among all promising decreasing variables in $|DecVar|$ to flip.

Algorithm 10 *adapt*G^{2}WSAT^+p

```
Require: $F, MAX-TRIES, MAX-FLIPS$
 1: for $i = 1$ to $MAX-TRIES$ do
 2:     randomly generate a truth assignment $a$
 3:     $p = 0$ and $wp = 0$
 4:     store all decreasing variables in stack $DecVar$
 5:     for $j = 1$ to $MAX-FLIPS$ do
 6:         if $a$ satisfies $F$ then
 7:             return $a$
 8:         end if
 9:         if $|DecVar| > 0$ then
10:             the least recently flipped promising decreasing variable
                among all promising decreasing variables in $|DecVar|$
11:         else
12:             randomly selected unsatisfied clause $c$ under $a$
13:             pick a variable $x$ according to Novelty$^+p$($p$,$wp$,$c$)
14:         end if
15:         flip the variable $x$ in $a$
16:         adapt $p$ and $wp$
17:         delete variables that are no longer decreasing from $DecVar$
18:         push new decreasing variables into $DecVar$ which are different
            from $x$ and were not decreasing before $x$ is flipped
19:     end for
20: end for
21: return "solution not found"
```

2.3.30 Gradient-based Greedy WalkSAT with look-ahead - G^{2}WSAT^+p

G^{2}WSAT^+p [1] is a variant of *adapt*G^{2}WSAT^+p which does not use the adaptive noise mechanism.

2.3.31 Variable Weighting Scheme One - VW1

VW1 [16] is an algorithm which adds a new tie-breaking heuristic to Walksat/SKC. Namely it selects flip variables as Walksat/SKC, but break ties (among non-freebies [12]) by preferring the variable that has been flipped least often in the search so far (break further ties randomly). Therefore, the weight of a variable is the number of times it has been flipped and VW1 algorithm select variables with minimal weight as long as this does not conflict with the SKC flip heuristic. This alternative diversification technique based on variable flip histories is called *variable weighting* and it can emulate clause weighting performance.

[16] compares VW1 with five other local search algorithms (SKC, HWSAT, Novelty+, SAPS, TABU) on ternary chains and shows that most algorithms scale exponentially but VM1 scales polinomially.

2.3.32 Variable Weighting Scheme Two - VW2

The Algorithm 11 shows the behavior of the VW2 algorithm which combines continuously smoothed variable weights with heuristics based on Walksat/SKC.

VW2 is identical to SKC except for its flip heuristic. Instead of using weights for tie-breaking they are used to adjust the break counts as follows. From a random violated clause the algorithm selects the variable v with minimum score $b_v + b(w_v - M)$ where b_v is the break count of v, w_v is the current weight of v, M is the current mean weight, and c is a new parameter ($c \geq 0$ and usually $c < 1$). Ties are broken randomly.

11 shows that VW2 currently takes an order of magnitude more flips than the best algorithms on 16-bit parity learning problems, but clause weighting algorithms have undergone several generations of development. Moreover, SAPS has a more efficient smoothing algorithm than most clause weighting algorithms, but continuous smoothing scales better to large problems.

2.3.33 Dynamic Local Search Algorithms for SAT

The goal of Dynamic Local Search (DLS) [8] algorithms is to prevent iterative improvement methods for getting stuck in local optima. The key idea is to modify the evaluation function whenever a local optimum is encountered in such a way that further improvement steps become possible.

This can be done by associating *penalty weights* with solution component. These determine impact of components one evaluation function. In particular, DLS works by performing iterative improvement steps and when it get

[12]Freebies are flips that incur no breaks

Algorithm 11 VW2

```
1: initialize all variables to randomly selected truth values
2: initialize all variable weights to 0
3: while no clause is violated do
4:     randomly select a violated clause $c$
5:     if $c$ contains freebie variables then
6:         randomly flip one of them
7:     else
8:         if with probability $p$ then
9:             flip a variable in $c$ chosen randomly
10:        else
11:            flip a variable in $c$ chosen by the new heuristic
12:        end if
13:    end if
14:    update and smooth the weight of the flipped variable
15: end while
```

stuck in a local optima it increase penalties of some solution components (this lead to a degradation in the evaluation function value of the current candidate solution) until improving steps became available (namely, until it is higher than the evaluation function values of some of its neighbor). The schema of a DSL algorithm is the following:

- determine the initial candidate solution s
- initialize penalties
- while termination criterion is not satisfied:
  - compute modified evaluation function g' form g based on penalties
  - perform subsidiary local search on s using evaluation function g'
  - update penalties based on s

In case of SAT, the solution components that are being selectively penalized are the clauses of the given formula. In particular, the modified evaluation function is $g'(F,a) = g(F,a) + \sum_{c \in CU(F,a)} clp(c)$, where $CU(F,a)$ is the set of all clauses in the formula F that are unsatisfied under the assignment a and $clp(c)$ denotes the penalty associated with the clause c.

The main difference between DLS algorithm for SAT are in the subsidiary local search and in the scheme used for updating the clause penalties.

2.3.34 Scaling and Probabilistic Smoothing - SAPS

The SAPS [8, 3, 9] algorithm (see Algorithm 12) is closely related to the Exponentiated Sub-Gradient [8, 9] (ESG) algorithm.

SAPS assigns a penalty *clp* to each clause, and the search evaluation function of SAPS is the sum of the clause penalties of unsatisfied clauses.

Initially it randomly selects an initial assignment assignment and initializes all clause weights to 1. It uses a best improvement search method (like GSAT) and whenever a local minimum occurs (no step improvement in the evaluation function greater than $SAPS_{thresh}$ is possible):

- With probability p, a random walk step occurs.
- With probability $1 - p$, a *scaling step* occurs, where the penalties for unsatisfied clauses are multiplied by the scaling factor α (namely $clp' = \alpha \times clp$). After a scaling step:
  - with probability p_{smooth}, a *smoothing step* occurs therefore all penalties are adjusted according to the mean penalty value $\overline{clp}$ and the smoothing factor ρ (namely, $clp' = clp + (1 - \rho) \times \overline{clp}$).

The performance of SAPS algorithm depend on α, ρ and p_{smooth} parameters. The SAPS algorithm escapes from local minima by scaling the weights of unsatisfied clauses, whereas smoothing the weights back towards uniform values acts as an intensification of the search; complete smoothing ($\rho = 0$) results in basic GSAT behavior without noise.

Moreover, to understand the its performance it is useful to study the evolution of clause weights over time. If two clauses were unsatisfied at only one local minimum each, then the relative weights of these clauses depend on the order in which they were unsatisfied. Since the weights are scaled back towards the clause weight average at each smoothing stage, the clause that has been unsatisfied more recently has a larger weight. So scaling and smoothing can be seen as a mechanism for ranking the clause weights based on search history. Clearly, the distribution of clause weights, which is controlled by the settings of α, ρ and p_{smooth} has a major impact on the variable selection underlying the primal search steps.

[3] has found that SAPS, RSAPS and SAPS/NR are amongst the state-of-the-art SLS SAT solvers, and each typically performs better than ESG, and the best WalkSAT variants (for example, Novelty+). Moreover, [3] shows that SAPS is similarly effective on MAX-SAT problem instances.

2.3.35 RSAPS

RSAPS [8, 3, 9] is a reactive variant of SAPS that reactively changes the smoothing parameter ρ during the search process whenever search stagnation is detected, using the same adaptive mechanism as Adaptive Novelty+.

The performance of SAPS depends on the settings of its parameters and there can be hard and time-consuming to determine this settings manually. Therefore the basic idea of RSAPS is to reactively use higher noise levels, leading to more search diversification, if and only if there is evidence for search stagnation. In particular, if search stagnation is detected then more noise is introduced, otherwise the noise value is gradually decrease.

In order to control reactively the search intensification it is possible to adapt either ρ or p_{smooth}. Intuitively, it makes much sense to adapt the amount of smoothing since this directly determines the actual extent of search intensification. In order to let changes in ρ effectively control the search, the smoothing probability would have to be rather high. However, in order to achieve superior time performance, we need at least a bias towards low smoothing probabilities. Therefore, it is used a fixed ρ and it is controlled the amount of smoothing by adapting P smooth.

By choosing a rather low value for ρ, large amounts of smoothing and high levels of search intensification can still be achieved, while keeping the average smoothing probability low.

The stagnation criterion is the same as used in Adaptive Novelty^{+}. If the search has not progressed in terms of a reduction in the number of unsatisfied clauses over the last (number of clauses)$\times\theta$ variable flips, the smoothing probability is reduced ($\theta = 1/6$ seems to give uniformly good performance). This reduction of the smoothing probability leads to a diversification of the search, like an increase of the noise value in Adaptive Novelty^{+}. As soon as the number of unsatisfied clauses is reduced below its value at the last change of the smoothing probability, p_{smooth} is increased in order to intensify exploration of the current region of the search space. A bias towards low smoothing probabilities is achieved by decreasing p_{smooth} faster than increasing it. Moreover, after each smoothing operation, p_{smooth} is set to zero (this

Algorithm 12 SAPS

```
1: generate random starting point
2: for each clause $c_i$ do
3:     set clause weight $w_i \leftarrow 1$
4: end for
5: while solution not found and not timed out do
6:     $best \leftarrow \infty$
7:     for each literal $x_{ij}$ appearing in at least one false clause do
8:         $\Delta w \leftarrow$ change in false clause $\Sigma w$ caused by flipping $x_{ij}$
9:         if $\Delta w < best$ then
10:            $L \leftarrow x_{ij}$
11:            $best \leftarrow \Delta w$
12:        else if $\Delta w = best$ then
13:            $L \leftarrow L \cup x_{ij}$
14:        end if
15:    end for
16:    if $best < -0.1$ then
17:        randomly flip $x_{ij} \in L$
18:    else if probability $\leq wp$ then
19:        randomly flip any literal
20:    else
21:        for each false clause $f_i$ do
22:            $w_i \leftarrow w_i \times 1$
23:        end for
24:        if probability $\leq P_{smooth}$ then
25:            $\mu_w \leftarrow$ mean of current clause weights
26:            for each clause $c_i$ do
27:                $w_j \leftarrow w_j \times \rho + (1 - \rho) \times \mu_w$
28:            end for
29:        end if
30:    end if
31: end while
```

happens in procedure Update-Weights). Together, these two mechanisms help to ensure low average values of p_{smooth} for problem instances that do not benefit from smoothing.

2.3.36 De-randomized version of SAPS - SAPS/NR

SAPS/NR [3, 9, 21] is a de-randomized variant of SAPS that eliminates all sources of random decisions throughout the search and which relies upon the initial random variable assignment as the only source of randomness.

In particular SAPS/NR performs:

- Breaking ties deterministically: whenever a tie between variables occurs, the SAPS/NR algorithm deterministically chooses the variable with the lowest index value. Whereas SAPS performs a random tie-breaking, namely when two or more variables would give the identical best improvement when flipped, one of them is chosen at random.

- No random walk steps: The p parameter is always set to zero, so that random walk steps are never performed. Whereas SAPS perform a random walk with probability p when a local minimum is encountered.

- Periodic smoothing: The probabilistic smoothing is replaced with deterministic periodic smoothing, where smoothing occurs every $1/p_{smooth}$ local minima. Whereas in case of SAPS, scaling, which also occurs only when a local minimum is encountered, is followed by smoothing with probability p_{smooth}.

The strongly randomized search mechanisms found in SAT algorithms such as GWSAT or WalkSAT, serve essentially the same purpose as the scaling and smoothing mechanism in SAPS: effective diversification of the search. Moreover, there is no significant difference between the behavior of SAPS and SAPS/NR. It shows chaotic behavior in that the length of successful runs is extremely sensitively dependent on the initial truth assignment.

2.3.37 Divide and Distribute Fixed Weights - DDFW

DDFW algorithm [10] (see Algorithm 13), at the start of the search, uniformly distributes a fixed quantity of weight across all clauses and then escapes from local minimum by transferring weight form satisfied to unsatisfied clauses (instead of increasing weights on false clauses in local minima and decreasing or normalizing weights on all clauses after a series of increases, like SAPS).

In particular, the transfer involves selecting a satisfied clause for each currently unsatisfied clause in a local minimum, reducing the weight on the satisfied clause by an integer amount and adding it to the weight on the unsatisfied clause.

Moreover, DDFW exploits the neighborhood relationships between clauses when it has to decide which pair of clauses will exchange weight.

A clause c_i is a neighbor of clause c_j if there exists at least one literal $l_{im} \in c_i$ and a second literal $l_{jn} \in c_j$ such that $l_{im} = l_{jn}$. Moreover, a clause c_i is a *same sign* neighbor of c_j if the sign of any $l_{im} \in c_i$ is equal to the sign of any $l_{jn} \in c_j$ where $l_{im} = l_{jn}$.

From this it follows that each literal $l_{im} \in c_i$ will have a set of same sign neighboring clauses $C_{l_{im}}$. If c_i is false then all literals $l_{im} \in c_i$ evaluate to false. Therefore flipping any l_{im} will cause it to become true in c_i and also to become true in all $C_{l_{im}}$. This will increase the number of true literals and therefore it increases the overall level of satisfaction for those clauses. Conversely, l_{im} has a corresponding set of opposite sign clauses that would be damaged when l_{im} is flipped. The algorithm adds weight to each false clause in a local minimum, by taking weight away from the most weighted same sign neighbor of that clause. In particular, the weight on a clause is not allowed to fall below $W_{init} - 1$ (W_{init} is the initial weight distributed to each clause initially). If there are no neighboring same sign clause whit sufficient weight to give to a false clause, then a non-neighboring clause with sufficient weight is chosen randomly. If the donating clause has a weight greater than W_{init} then it gives a weight of two, otherwise it givess a weight of one.

The basic idea is that clauses sharing same sign literals should form alliances, because a flip that benefits one of these clauses will always benefit some other members of the group. Therefore, clauses that are connected in this way will form groups that tend towards keeping each other satisfied. However, these groups are not closed, as each clause will have clauses within its own group that are connected by other literals to other groups. Weight is therefore able to move between groups as necessary, rather than being uniformly smoothed.

Algorithm 13 DDFW

```
Require: $F, W_{init}$
 1: set the weight $w_i$ for each clause $c_i \in F$ to $W_{init}$
 2: while solution not found and not timeout do
 3:     find a return list $L$ of literals causing the greatest reduction in
        weighted cost $\Delta w$ when flipped
 4:     if $\Delta w < 0S$ or $\Delta w = 0$ and probablity $\leq 15\%$ then
 5:         randomly flip a literal in $L$
 6:     else
 7:         for each false clause $c_f$ do
 8:             select a satisfied same sign neighbouring clause $c_k$ with
                maximum weight $w_k$
 9:             if $w_k < W_{init}$ then
10:                 randomly select a clause $c_k$ with weight $w_k \geq W_{init}$
11:             else if $w_k > W_{init}$ then
12:                 transfer a weight of two from $c_k$ to $c_f$
13:             else
14:                 transfer a weight of one from $c_k$ to $c_f$
15:             end if
16:         end for
17:     end if
18: end while
```

2.3.38 Pure Additive Weighting Scheme - PAWS

The PAWS algorithm [12] (see Algorithm 14) is a version of SAPS that increase weights additively instead of multiplicatively.

It is controlled by the parameter P_{flat}, which decides whether a randomly selected flat move will be taken (corresponding to wp in SAPS), and Max_{inc}, which determines at which point weight will be decreased (corresponding to P_{smooth} in SAPS).

PAWS differs form SAPS in three aspects. It probabilistically takes a random flat move when no improving move is available instead of allowing cost increasing moves. It deterministically reduces weights after Max_{inc} number of increases instead of reducing weights with probability P_{smooth}. Finally, PAWS allows optimal cost flips that appear in n false clauses to also appear n times in its move list L instead of exactly once.

[12] shows that PAWS is strongly outperforming SAPS on all problems except the most difficult random binary CSP.

3 Stochastic Local Search for SMT

From the point of view of a SAT solver, an SMT problem instance *varphi* can be seen as the problem of solving a *partially-invisible* 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 of the $\mathcal{T}$-lemmas providing the obligations induced by the theory $\mathcal{T}$ on the atoms of φ. So a "lazy" SMT solver can be seen as a DPLL solver which knows φ but not τ^p. The search process works as follows: whenever a model μ^p for φ^p is found, it is passed to a $\mathcal{T}$-solver which knows τ^p and hence checks if μ^p falsifies τ^p. If yes, it returns one clause c^p in τ^p which is falsified by μ^p, which is then used by DPLL to drive the future search and is optionally added to φ^p.

The following section describes the basic procedure integrating a *$\mathcal{T}$-solver* into a SLS algorithm of the WalkSAT family. We called this procedure WalkSMT. Moreover, the next subsections are going to explain the optimizations applied to the basic WalkSMT to improve its performance.

3.1 A basic WalkSMT procedure

In this section we give a high-level description of the pseudo-code of WalkSMT is shown in Algorithm 15. It takes as input a CNF formula φ and the two

Algorithm 14 PAWS

```
1: generate random starting point
2: for each clause $c_i$ do
3:     set clause weight $w_i \leftarrow 1$
4: end for
5: while solution not found and not timed out do
6:     $best \leftarrow \infty$
7:     for each literal $x_{ij}$ in each false clause $f_i$ do
8:         $\Delta w \leftarrow$ change in false clause $\Sigma w$ caused by flipping $x_{ij}$
9:         if $\Delta w < best$ then
10:            $L \leftarrow x_{ij}$
11:            $best \leftarrow \Delta w$
12:        else if $\Delta w = best$ then
13:            $L \leftarrow L \cup x_{ij}$
14:        end if
15:    end for
16:    if $best < 0$ or ($best = 0$ and probability $\leq P_{flat}$) then
17:        randomly flip $x_{ij} \in L$
18:    else
19:        for each false clause $f_i$ do
20:            $w_i \leftarrow w_i + 1$
21:        end for
22:        if # times clause weights increased % $Max_{inc} = 0$ then
23:            for each clause $c_i/w_j > 1$ do
24:                $w_j \leftarrow w_j - 1$
25:            end for
26:        end if
27:    end if
28: end while
```

Algorithm 15 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:             $\langle status, c^p \rangle \leftarrow \mathcal{T}\text{-}solver(\varphi^p, \mu^p)$
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
```

parameters MAX_TRIES and MAX_FLIPS which are used from the SAT solver. The SAT solver explores the set of assignments in order to find a total truth assignment and the $\mathcal{LA}$-*solver* is invoked to find a conflict on the corresponding set of literals.

In lines 3-1, the algorithm rewrites the input formula φ into a $\mathcal{T}$-equivalent one as described in section 3.2. If this process produces some conflict then the algorithm returns $Unsat$. This is an optimization described in 3.2.1.

In line **??**, the procedure InitialTruthAssignment is used to generate an initial truth assignment μ^p to the boolean abstraction of the formula φ. In particular, it assigns the value true to all the variables occurring in the unit clauses of φ^p so that we can save the cost of flipping their literals. Finally, it assigns a random truth value to all the remaining variables.

Recall that we use the superscript ^{p} to denote the boolean abstraction, namely given a $\mathcal{T}$-formula ζ we write ζ^p to denote $\mathcal{T}2\mathcal{B}(\zeta)$.

In line 7, the algorithm checks if μ^p propositionally satisfies φ^p. If yes, then it invokes the *$\mathcal{T}$-solver* on the $\mathcal{LA}$-formula φ and the truth assignment μ for φ. Finally, if the *$\mathcal{T}$-solver* returns a status equal to SAT then also the SMT solver returns SAT in line 10. Otherwise it returns the $\mathcal{T}$-lemma (or conflict clause) ψ which is used to guide the search process. If μ^p does not propositionally satisfy φ^p. then the search process follows the WALKSMT schema by choosing an unsatisfied clause. In both cases, the procedures NEXTTRUTHASSIGNMENT is used to generate a new truth assignment following the variable selection schema of Adaptive Novelty$^+$.

3.2 Enhancements to the basic WalkSMT procedure

The WALKSMT algorithm described above is very simple and we tried to optimize it in several ways.

In this section, we briefly describe some of the most significant optimizations that we have investigated.

3.2.1 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, lines 1-3. It consists of two techniques: *static learning* and *unit propagation*.

We perform a step of *unit propagation*, by substituting each literal occurring as a unit clause in φ with TRUE, repeating this step until a fixpoint is reached, and finally by re-adding to φ the conjunction of all non-propositional unit literals eliminated.

We apply *static learning* [18], which augments the input formula with short $\mathcal{T}$-lemmas generated without invoking the $\mathcal{T}$-solver, having the purpose of detecting a priori in a fast manner obviously $\mathcal{T}$-inconsistent assignments to $\mathcal{T}$-atoms.

After the preprocessing, the original formula is rewritten as $\varphi^{sl}|_{u_i=1} \wedge \bigwedge_i u_i^T$, where φ^{sl} is the formula coming out of the execution of static learning technique, u_i are unit clauses whose atom may be either a boolean variable or a $\mathcal{LA}$-atom and u_i^T are unit clause whose atoms belongs to the theory $\mathcal{LA}$. If some variables occurring in the boolean abstraction of the original formula does not appear in the boolean abstraction of the preprocessed formula then a renaming is performed.

3.2.2 Learning

A very important optimization is the *learning* of the $\mathcal{T}$-lemmas that are generated by the $\mathcal{T}$-solver so that to avoid finding the same $\mathcal{T}$-conflict multiple times (since this might be quite expensive), line 13. This technique is also used in DPLL-based SMT solvers.

3.2.3 Unit simplification

Before learning a $\mathcal{T}$-lemma, line 12, we remove from it (setting them to TRUE) all the literals which occur as unit clauses in the (preprocessed) input problem, as shown in Algorithm 16.

Algorithm 16 Unit-Simplification

```
Require: $\varphi, c$
1: for all unit clause $l$ occurring in $\varphi$ do
2:     $c \leftarrow$ DeleteLiteral($\neg l, c$)
3: end for
4: return $c$
```

3.2.4 Filtering the assignments given to $\mathcal{T}$-solvers

In order to decrease the time spent from the $\mathcal{T}$-solver, we thought to reduce the set of literals on which it is invoked to check the consistency in $\mathcal{T}$. So, we apply some standard filtering techniques to the current truth assignment before invoking the $\mathcal{T}$-solver, such as *pure literal filtering* and *ghost literal filtering* (see [18]).

The idea behind pure literal filtering is that, if we have non-Boolean $\mathcal{T}$-atoms occurring only positively [negatively] in the original formula (learned clauses are not considered), we can drop every negative [positive] occurrence of them from the assignment to be checked by the $\mathcal{T}$-solver [18].

The ghost literal filtering technique states that literals occurring only in original satisfied clauses, called *ghost literals*, can be removed from the assignment to be checked by the $\mathcal{T}$-solver [**?**]. In DPLL-based SMT solvers the presence of ghost $\mathcal{T}$-literals in the assignment μ causes unless extra work to the $\mathcal{T}$-solver. Moreover, it may affect the $\mathcal{T}$-satisfiability of μ forcing unnecessary backtracks and causing unnecessary Boolean search and hence useless calls to the $\mathcal{T}$-solver. In the case of our local search based SMT solver, we cannot realize to have found a satisfiable assignment μ as in the following example: let $\mu = \{l_1, \ldots, l_k, l_{k+1}, \ldots, l_j\}$ be a total truth assignment such that the subassignment $\mu' = \{l_1, \ldots, l_k$ propositionally satisfies the formula, μ and the subassignment $\mu' = \{l_{k+1}, \ldots, l_j$ are inconsistent in $\mathcal{T}$. As we can see, ghost-literals are $l_{k+1}, \ldots, l_j$. Since they cause conflicts with literals occurring in μ', next flips could not involve them. This allows the SMT solver to move away from the solution. In order to compute the set of ghost literals we propose a two-stage process. Let μ^p be the current assignment and $score(\varphi^p, \mu^p)$ be the number of false clause in the formula φ^p under the assignment μ^p. In the first stage, we compute set of candidate ghost literals searching for all the literals l which occurs in *necessary clauses* such that $score(\varphi^p, \mu^p)$ is equal $score(\varphi^p, \mu^p|_{\neg l^p})$. In the second stage, for all candidate ghost literals l we check whether $score(\varphi^p, \mu^p)$ is equal $score(\varphi^p, \mu^p|_{\neg l^p})$ and if yes, it flips the literal l^p in the current assignment μ^p and puts l in the set of ghost literal. This technique was not successfully applied to WalkSMT procedure.

3.2.5 Multiple learning

Unlike with DPLL-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.

In this setting, a truth assignment may be $\mathcal{T}$-inconsistent for several different reasons, 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. In particular, when we find a conflict set η the (unit simplified) $\mathcal{T}$-lemma $\neg\eta$ is used to compute a subassignment μ' s.t. $\mu' \subset \mu$, on which the $\mathcal{T}$-solver is invoked again to find a new conflict set. The subassignment μ' is computed by dividing the current (unit simplified) $\mathcal{T}$-lemma $\neg\eta$ in f parts (where f is a parameter) and removing the variables occurred in the first part of it from μ. This process is repeated until no conflict set is found. We then learn all the $\mathcal{T}$-lemmas generated during the process.

3.3 WalkSMT's Specification

This section aims to provide the specification used as starting point in the development of a tool implementing the Algorithm 15.

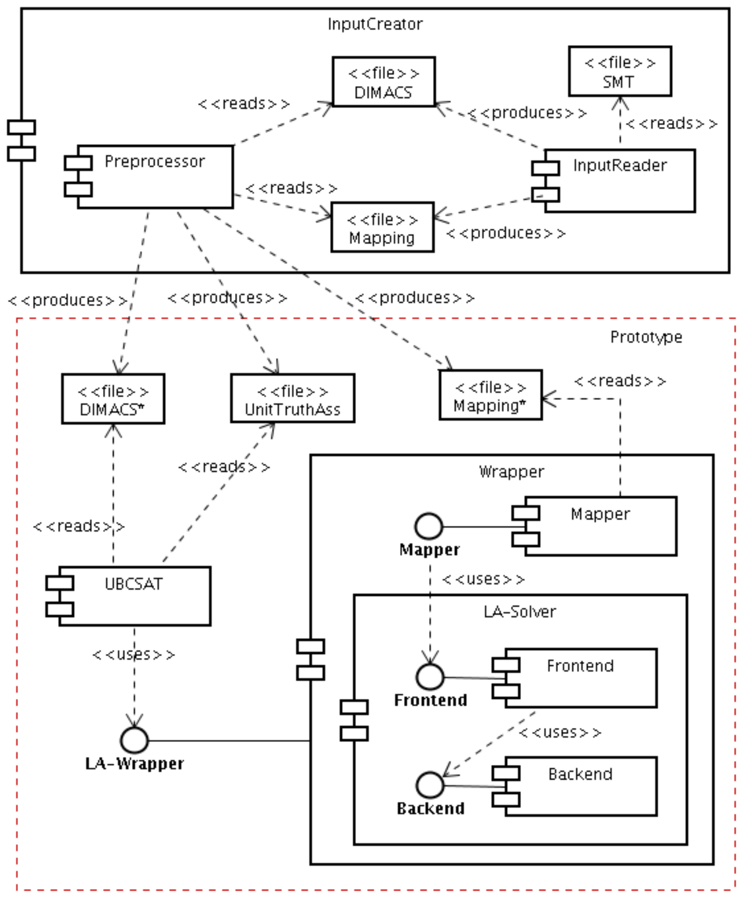


Figure 1: Component Diagram of the SMT-Solver

The Figure 1 shows the architecture of the SMT-Solver which is composed of three main component:

- **InputCreator**: it allows to parse and simplify the $\mathcal{LA}$-formula. Moreover it allows to create the input file of UBCSAT (a DIMACS CNF file and a file containing a truth assignment) and Mapper (Mapping file) components.

- **UBCSAT**: it is the SAT solver which enumerates the truth assignments satisfying the boolean abstraction of the $\mathcal{LA}$-formula. It uses the Wrapper component in order to invoke the $\mathcal{LA}$-Solver on the set of literals in the theory corresponding to the truth assignments.

- **Wrapper**: it allows to interface UBCSAT component, which is written in C language, and the *Mapping* component, which is written in C++ language.

- **Mapper**: it maintains a state which is a mapping between boolean variables and atoms in the theory, provides a set of procedure for access it and allows to invoke the LA-Solver.

- **$\mathcal{LA}$-Solver**, it checks the consistency in the *Linear Arithmetic Theory* of a set of literals.

The dashed line in the Figure 1 denotes the prototype which will be developed at the first stage of the development process. The second stage of the development process will concern the development of the InputCreator component and the testing of the prototype.

The Figure 2 describes the process to check the satisfiability of a $\mathcal{LA}$-formula using the WalkSMT solver.

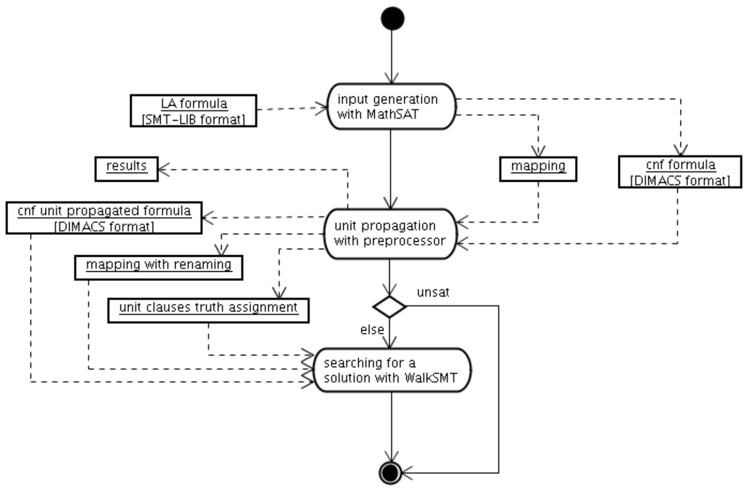

LA formula
[SMT-LIB format]
input generation with MathSAT
mapping
cnf formula
[DIMACS format]
results
cnf unit propagated formula
[DIMACS format]
unit propagation with preprocessor
mapping with renaming
unit clauses truth assignment
unsat
else
searching for a solution with WalkSMT

Figure 2: Component Diagram of the SMT-Solver

The next subsection is going to describe the adopted naming convention for the signature of the interfaces procedures. Moreover, the following subsections are going to describe in details the components above.

3.3.1 Naming Convention for Procedures

The following rules allows to define a naming convention for the signature of the interfaces procedures:

- names are written with small letters;
- words are divided by the underscore character;
- the name of a procedure should be composed by:
  - name of the performed action,
  - name of the object of the performed action (if any);
- use the overloading mechanism so that we can have procedures with the same name but different signature (since the signature depends on the name of the procedure and the type of the parameters);
- there are special cases:
  - if a procedure returns a variable then the signature should be *get_⟨returned variable⟩*;
  - if a procedure sets a variable to a particular value then the signature should be *set_⟨set variable⟩*.

Moreover, class are written using the camel case practice (compound words or phrases are written in a way that the words are joined without spaces and are capitalized within the compound) and constant or enum are written with capital letters and are divided by the underscore character.

3.3.2 The InputCreator Component

The InputCreator component allows to parse and simplify a quantifier-free linear arithmetic formula whose satisfiability has to be proved and, finally, it must create the input of the SMT-Solver. It is composed by two sub components: InputReader and Preprocessor.

3.3.3 The InputReader

The InputReader reads and preprocesses the formula stated in SMT-LIB format. Then it generates the boolean abstraction of the $\mathcal{LA}$-formula and the mapping between boolean variables and atoms in the theory related to it. In particular, the InputReader component performs the following steps:

- reading the formula in SMT-LIB format,
- performing static learning,
- generating the DIMACS file (containing the boolean abstraction of the formula) and the mapping file (containing the mapping between boolean variable and atoms in the theory).

3.3.4 The Preprocessor

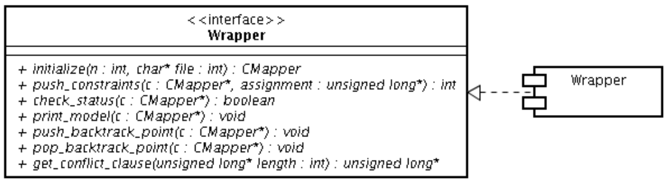
The Preprocessor allows to apply unit propagation to a formula in CNF applying renaming if some variables are deleted from the formula. Other than the resulting formula and a new mapping file, the Preprocessor generates the truth assignment of variables belonging to unit clauses whose atoms are in the theory. In particular, the Preprocessor component performs the following steps:

- it reads the formula in DIMACS CNF format,
- it reads the mapping file related to the formula,
- it performs unit propagation (if it is needed then it renames variables),
- if the formula is unsatisfiable then it generates a file containing the explanation of the conflict.
- otherwise it generates:
  - a file (in DIMACS CNF format) containing the preprocessed formula in conjunction with the unit clauses belonging to the theory;
  - a file containing the new mapping between boolean variables and atoms in the theory related to the preprocessed formula and its abstraction. Moreover, this file contains the list of necessary clauses and specifies for each atom if it is positively/negatively pure or not;

- a file containing the truth assignment of variables belonging to unit clauses whose atoms belong to the theory;
- a file containing the truth assignment of boolean variables belonging to unit clauses and the renaming of the variables.

3.3.5 The Wrapper Component

The Wrapper component follows the mechanisms explained in [2] in order to instantiate the Mapping component and invoke its procedures from the UBCSAT component since the former is written in C++ language and the latter in C language.



```
+ initialize(n : int, char* file : int) : CMapper
+ push_constraints(c : CMapper*, assignment : unsigned long*) : int
+ check_status(c : CMapper*) : boolean
+ print_model(c : CMapper*) : void
+ push_backtrack_point(c : CMapper*) : void
+ pop_backtrack_point(c : CMapper*) : void
+ get_conflict_clause(unsigned long* length : int) : unsigned long*
```

Figure 3: The Mapper component has an interface providing a set of operations to use the Mapping component, which is written in C++ language, in the UBCSAT component, which is written in C language.

The Wrapper component provides to UBCSAT component an interface whose operations are shown in the Figure 3 and have the following meaning:

- *initialize* instantiates the Mapping component (which read the mapping file) and uses it to communicate the atom to the $\mathcal{LA}$-solver.
- *push_constraints* uses Mapping component to assert the set of literals corresponding to the truth assignment in input.
- *check_status* uses Mapping component to check if the current status is consistent
- *get_conflict_clause* uses Mapping component for return the boolean conflict clause (if any).
- *print_model* use the Mapping component to build and print the model using the frontend component.

- *pop_backtrack_point* use the Mapping component to save the current state of the solver for a possible backtrack.
- *pop_backtrack_point* use the Mapping component to restore the last saved state of the solver.

3.3.6 The Mapper Component

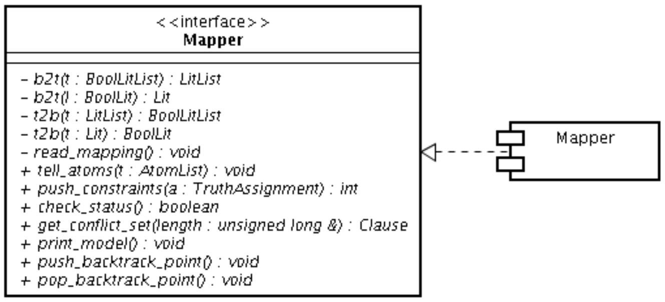

<<interface>>
Mapper
– b2t(t : BoolLitList) : LitList
– b2t(l : BoolLit) : Lit
– t2b(t : LitList) : BoolLitList
– t2b(t : Lit) : BoolLit
– read_mapping() : void
+ tell_atoms(t : AtomList) : void
+ push_constraints(a : TruthAssignment) : int
+ check_status() : boolean
+ get_conflict_set(length : unsigned long &) : Clause
+ print_model() : void
+ push_backtrack_point() : void
+ pop_backtrack_point() : void
Mapper

Figure 4: The Mapper component has an interface providing a set of operations to handle the mapping between boolean variables and atoms.

The Mapper component is used to interface the UBCSAT component, which handles boolean literals, and the $\mathcal{LA}$-Solver component, which handles $\mathcal{LA}$-literals.

This component maintain a state that is an internal representation of the mapping between boolean variables and $\mathcal{LA}$-atoms. It must provide constant-time operations for mapping a boolean literal into a $\mathcal{LA}$-literal and vice-versa. For this reason, the Mapper uses:

- a hash map using the variables (where variables are positive integers) as keys and pointers to atoms as values,
- a vector containing variables indexed by atom's id (where id are positive integers).

The Mapper component provides an interface whose operations are shown in the Figure 4 and have the following meaning:

- *b2t(BoolLitList)* maps a list of boolean literals into the corresponding list of literals in the theory.

- *t2b(BoolLit)* maps a boolean literal into the corresponding literal in the theory.
- *bt2(LitList)* maps a list of literals in the theory into the corresponding list of boolean literals.
- *t2b(Lit)* maps a literal in the theory into the corresponding boolean literal.
- *read_mapping* reads the mapping between boolean variables and atoms in the theory.
- *tell_atoms* communicates to the frontend the set of all possible atoms it might see during the search process.
- *push_constraints* uses the frontend to invoke the $\mathcal{LA}$-solver for asserting the set of literals corresponding to the truth assignment in input.
- *check_status* uses the frontend to invoke the $\mathcal{LA}$-solver for checking if the current status is consistent
- *get_conflict_clause* uses the frontend to invoke the $\mathcal{LA}$-solver for returning the boolean conflict clause (if any).
- *print_model* builds and prints the model using the frontend component.
- *pop_backtrack_point* allows to save the current state of the solver for a possible backtrack.
- *pop_backtrack_point* restores the last saved state of the solver.

3.3.7 The $\mathcal{LA}$-Solver Component

The **$\mathcal{LA}$-Solver** checks the consistency in the *Linear Arithmetic Theory* of the set of literals. The mapping between boolean variable and literals in the theory is responsibility of the Mapper component. It is composed by two subcomponent, frontend and backend, which are described in the following to subsections.

3.3.8 Backend

The Backend component implements a $\mathcal{LA}$-Solver based on the variant of the Simplex algorithm described in [4] .

In this version, the $\mathcal{LA}$-Solver maintains a state that is an internal representation of the atoms asserted so far. So, the $\mathcal{T}$-Solver must provide some operations for updating the state and, in particular, it has to be able to assert new atoms and check whether the state is consistent.

Moreover, in order to interact with the SMT-procedure described in the section the $\mathcal{LA}$-Solver must be able to produce a (minimal) conflict set which is an inconsistent subset of the atoms asserted in the current state.

The $\mathcal{LA}$-Solver state is composed by the following elements:

- a tableau of equations derived from the constrains which are related to the atoms of the linear arithmetic formula. The tableau is written in the following form: $\sum_{x_j \in N} a_{ij} x_j$ for each $x_i \in B$ where N is the set of nonbasic variable and B is the set of basic variable. For example, in the initial state, if $2x - 5 \geq 3y$ is an atom of a given formula and the corresponding positive constraint is $2x - 3y \geq 5$ (whereas the corresponding negative constraint is $2x - 3y < 5$) then derived equation to put in the tableau is $s = 2x - 3y$ where $s \in B$ and $x, y \in N$.

- upper and lower bounds l_i and u_i for every variable x_i known to the $\mathcal{T}$-Solver. In the initial state $u_i = +\infty$ and $l_i = -\infty$.

- a mapping β which assigns a rational value $\beta(x_i)$ to every variable x_i and always satisfies both the bounds on nonbasic variables and the system of equations in the tableau. In the initial state $\beta(x_j) = 0$ for all $j \in N$.

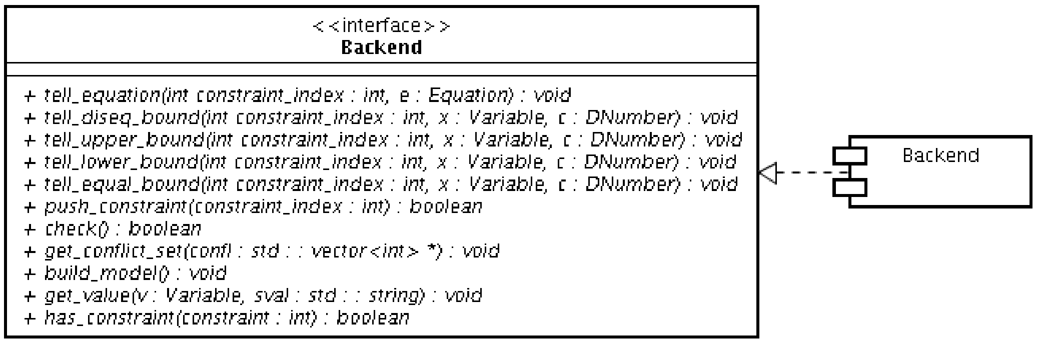
- the set of atoms asserted so far which is initially empty.

Other than the above elements the $\mathcal{T}$-Solver keeps track of:

- the set of range constraints, namely elementary atoms of the form $y \bowtie b$ where $\bowtie \in \{=, \neq, \leq, \geq\}$, derived from the atoms of the $\mathcal{T}$-formula. For example, in the initial state, if we consider the atom, the positive constraint and the equation of the previous example then the range constraint is $s \geq 5$.

- the set of disequality bounds whose consistency has to be checked.

- the mapping between variables and the list of all the range constraint it appears in.
- the status of the current state (namely inconsistent or not).
- the assignment to constrained variables, if the state is consistent.



```
<<interface>>
Backend

+ tell_equation(int constraint_index : int, e : Equation) : void
+ tell_diseq_bound(int constraint_index : int, x : Variable, c : DNumber) : void
+ tell_upper_bound(int constraint_index : int, x : Variable, c : DNumber) : void
+ tell_lower_bound(int constraint_index : int, x : Variable, c : DNumber) : void
+ tell_equal_bound(int constraint_index : int, x : Variable, c : DNumber) : void
+ push_constraint(constraint_index : int) : boolean
+ check() : boolean
+ get_conflict_set(confl : std : : vector<int> *) : void
+ build_model() : void
+ get_value(v : Variable, sval : std : : string) : void
+ has_constraint(constraint : int) : boolean
```

Figure 5: The Backend Interface.

The Figure 5 shows the Backend interface, namely the set of main procedures used to update the $\mathcal{T}$-Solver state.

Initialization of the $\mathcal{LA}$-Solver The $\mathcal{LA}$-Solver needs to a set of procedures which allow to initialize its status for a fixed formula. These procedures must build the initial tableau of equations and the set of range constraints to assert. They are the following:

- *tell_equation*: informs the $\mathcal{LA}$-Solver that the constraint *constraint_index* is an equation equal to e by inserting the equation in the tableau.
- *tell_diseq_bound*: informs the $\mathcal{LA}$-Solver that the constraint *constraint_index* is a disequality (namely a range constraint of the form $x \neq c$) by updating the set of range constraints.
- *tell_lower_bound*: informs the $\mathcal{LA}$-Solver that the constraint *constraint_index* is a lower bound (namely a range constraint of the form $x \geq c$) by updating the set of range constraints.

- *tell_upper_bound*: informs the $\mathcal{LA}$-Solver that the constraint *constraint_index* is an upper bound (namely a range constraint of the form $x \leq c$) by updating the set of range constraints.
- *tell_equal_bound*: informs the $\mathcal{LA}$-Solver that the constraint *constraint_index* is an equality (namely a range constraint of the form $x = c$) by updating the set of range constraints.
- *has_constraint*: checks if the $\mathcal{LA}$-Solver knows the constraint *constraint*.

Atom Assertion The $\mathcal{LA}$-Solver needs to a procedure which allows to assert new atoms in the current state. So, the procedure *push_constraint*, depending on the type of bound constraint, behaves like follows:

- in case of disequality (namely $x \neq c$) bound it remembers the bound in a stack so that it can be checked by the *check* procedure.
- in case of lower bound (namely $x \leq c$) it asserts the bound implementing the procedure AssertLower($x_i \geq c_i$) defined in [4]. If $c_i \leq l_i$, it inserts the bound in the set of atoms asserted so far and returns true to say that the bound is satisfiable. If $c_i > u_i$ then it returns false to say that the bound is unsatisfiable, otherwise l_i is set to c_i and (if x_i is nonbasic) β is updated .
- in case of upper bound (namely $x \geq c$) it asserts the bound in input implementing the procedure AssertUpper($x_i \leq c_i$) defined in [4]. If $c_i \geq u_i$, it inserts the bound in the set of atoms asserted so far and returns true to say that the bound is satisfiable. If $c_i < l_i$ then it returns false to say that the bound is unsatisfiable, otherwise u_i is set to c_i and (if x_i is nonbasic) β is updated.
- in case of equality bound (namely $x = c$) it behaves like both the previous cases.

Checking the Consistency The $\mathcal{LA}$-Solver needs to a procedure which allows to check whether the set of atoms asserted so far α is consistent. So, if β does not satisfies the bounds on some basic variables then $\mathcal{LA}$-Solver searches for a new assignment β such that satisfies all constraints. In particular, relying on a total order on the variables, it selects a basic variable

x_i that does not satisfies its bounds and looks for a variable $x_j \in N$ in the row $x_i = \sum_{x_j \in N} a_{ij}x_j$ of the tableau that can compensate the gap. If no such x_j exists then the (minimal) conflict set has to be built as described in [4] and the status is set to inconsistent otherwise the procedure pivots the two variables and adjusts the bounds. Moreover, this procedure must also check the consistency of disequality bounds. All these operations are performed by the *check* procedure.

Build and Return Conflict Set The $\mathcal{LA}$-Solver needs to build and return the conflict set if the current state is inconsistent. So, the procedure *get_conflict_set* copies the conflict set (built in *check* procedure) for the currently asserted constraints in *confl*.

Build and Return the Model Values The $\mathcal{LA}$-Solver needs to build and return a model if the $\mathcal{LA}$-formula is satisfiable. The procedure *build_model* builds a model assigning a rational value to every constrained variable, given the current mapping β, and the procedure *get_value* returns the value of the variable v (the precondition: model already built).

3.3.9 Frontend

The Frontend component allows to interface the Backend component with the component using it and, in particular, it must codify and communicate the atoms of a given linear arithmetic formula to the Backend component in a way that it is able to handle them.

For any atom the Frontend considers both the corresponding positive and negative constraint and maintain, for each of them, a linear representation of the form *term* $\bowtie c$, where *term* is a vector of pairs variable and coefficient, $\bowtie \in \{\leq, <, \neq, =, \geq, >\}$ and c is a numeric constant. For example, if $2x - 5 \geq 3y$ is an atom then the corresponding positive constraint is $2x - 5 \geq 3y$ and the negative one is $2x - 5 < 3y$. And if $2x - 5 \geq 3y$ is a constraint then the linear representation is $2x - 3y \geq 5$.

In particular, we assume that a positive constraint corresponds to a positive literal in the $\mathcal{LA}$-formula and a negative constraint corresponds to a negative literal.

To handle such representations the Frontend uses the TermMapper component which provides the following mappings:

- a mapping between atoms and the pair of positive and negative constraints,
- a mapping between constraints and their linear representations,
- a mapping between the variables and variable terms,

Since the Backend can accept only equations and bound constraints, the Frontend must codify each non-simple constraint (namely not in the form $x \bowtie c$) in linear representation into an equation and a constraint. For example, if $2x - 3y \geq 5$ is a linear representation of a non-simple constraint then the Frontend builds the equation $s = 2x - 3y$ and the bound constraint $s \geq 5$.

To handle such a coding the Frontend uses the ConstraintMapper component which provides the following mappings:

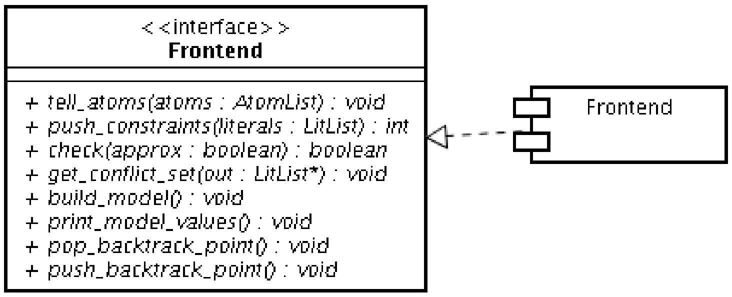
- a mapping between constraints (of the form $a_1x_1 + \cdots + a_nx_n = s$) and variables ($s$),
- a mapping between equations and basic variables.

Since the Backend component is not able to handle strict inequalities, the Frontend uses delta numbers so that a strict inequality $x_i > l_i$ is converted to $x_i \geq l_i + \delta$ and $x_i < u_i$ is converted to $x_i \leq u_i - \delta$, where $\delta = -1$.

In particular the Frontend is incharged of:

- communicating to the backend all the possible atoms after encoding them.
- invoking the backend to:
  - asserts a set of atoms in the current state.
  - checks whether the current state is consistent.
  - build a model for the satisfiable $\mathcal{LA}$-formula.
- printing the model values.

The Figure 6 shows the interface of the Frontend component and the procedures which compose the interface have the following meaning:



<<interface>>
Frontend
+ tell_atoms(atoms : AtomList) : void
+ push_constraints(literals : LitList) : int
+ check(approx : boolean) : boolean
+ get_conflict_set(out : LitList*) : void
+ build_model() : void
+ print_model_values() : void
+ pop_backtrack_point() : void
+ push_backtrack_point() : void
Frontend

Figure 6: The Frontend Interface.

- *tell_atoms* communicates to the backend all the possible atoms it might see during the search. In particular, it considers both the positive and negative constraints related to an atom and for each of them compute the linear representation and update the term mapping. Starting from the linear representation of a constraint the Frontend computes the related equality constraint and the bound constraint (updating the constraint mapping) in order to communicate them to the Backend. Delta numbers are used to represent the constant number of the bound constrains so that a strict inequality $x_i > l_i$ is converted to $x_i \geq l_i + \delta$ and $x_i < u_i$ is converted to $x_i \leq u_i - \delta$, where $\delta = -1$, where $\delta = 1$ (whereas $x_i \geq l_i$ is converted to $x_i \geq l_i + \delta$ and $x_i \leq u_i$ is converted to $x_i \geq u_i - \delta$, where $\delta = 0$).

- *push_constraints*: asserts a set of literals in the current state. In particular, it uses the mapping between literals and constraints to invoke the procedure *push_constraint* of the Backend component on the constraints related the set of literals in input.

- *check*: checks whether the current state is consistent using the procedure *check* of the Backend component.

- *get_conflict_set*: retrieves the conflict set using the mapping between constraints and literals and the procedure *get_conflict_set* of the Backend component.

- *build_model*: allows to compute a model using the procedure *build_model* of the Backend component.

- *print_model_values* prints the value of the variables in the current model using the mapping between the variables and variable terms and the procedure *get_value* of the Backend component.

- *pop_backtrack_point* tells the solver to save its current state for a possible backtrack.

- *pop_backtrack_point* restores the last saved state.

3.3.10 The UBCSAT Component

UBCSAT [23] is a SAT-Solver which allows to check the satisfiability of a boolean formula. In order to find a truth assignment satisfying a formula it provides several stochastic local search algorithms, but we only consider the WalkSAT algorithm.

So, the UBCSAT component is used to enumerate the truth assignments satisfying the boolean abstraction of the $\mathcal{LA}$-formula which are used from the $\mathcal{LA}$-Solver to check the consistency of the atoms of the $\mathcal{LA}$-formula.

UBCSAT has to perform all the steps of the sequence diagram in Figure 7. Once UBCSAT reads the CNF formula, it must:

1. use Wrapper component to instantiate the Mapper component which will read the mapping file;

2. search for a truth assignment satisfying the boolean formula;

3. use Wrapper component to save the state of the solver;

4. use Wrapper component to assert the set of literals corresponding to the truth assignment whenever it satisfies the boolean formula;

5. use Wrapper component to check if the current status is consistent;

6. (if there is a conflict) use Wrapper component to get the conflict clause and then simplify (applying Algorithm 16) and learn it;

7. use Wrapper component to restores the last saved state of the solver;

8. use Wrapper component to build the model and print the values.

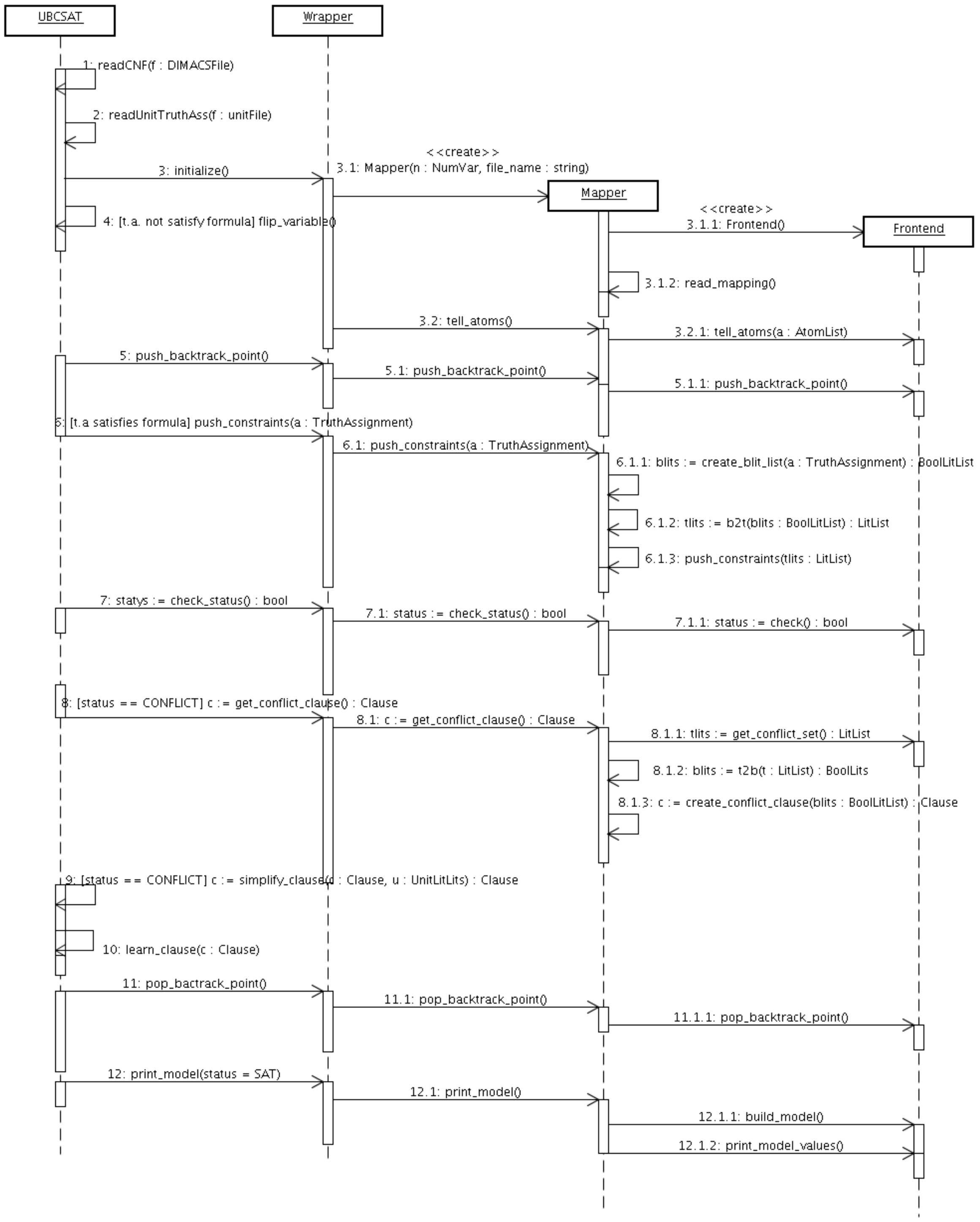

UBCSAT
Wrapper
Mapper
Frontend
1: readCNF(f : DIMACSFile)
2: readUnitTruthAss(f : unitFile)
3: initialize()
<<create>>
3.1: Mapper(n : NumVar, file_name : string)
4: [t.a. not satisfy formula] flip_variable()
<<create>>
3.1.1: Frontend()
3.1.2: read_mapping()
3.2: tell_atoms()
3.2.1: tell_atoms(a : AtomList)
5: push_backtrack_point()
5.1: push_backtrack_point()
5.1.1: push_backtrack_point()
6: [t.a satisfies formula] push_constraints(a : TruthAssignment)
6.1: push_constraints(a : TruthAssignment)
6.1.1: blits := create_blit_list(a : TruthAssignment) : BoolLitList
6.1.2: tlits := b2t(blits : BoolLitList) : LitList
6.1.3: push_constraints(tlits : LitList)
7: statys := check_status() : bool
7.1: status := check_status() : bool
7.1.1: status := check() : bool
8: [status == CONFLICT] c := get_conflict_clause() : Clause
8.1: c := get_conflict_clause() : Clause
8.1.1: tlits := get_conflict_set() : LitList
8.1.2: blits := t2b(t : LitList) : BoolLits
8.1.3: c := create_conflict_clause(blits : BoolLitList) : Clause
9: [status == CONFLICT] c := simplify_clause(c : Clause, u : UnitLitLits) : Clause
10: learn_clause(c : Clause)
11: pop_bactrack_point()
11.1: pop_backtrack_point()
11.1.1: pop_backtrack_point()
12: print_model(status = SAT)
12.1: print_model()
12.1.1: build_model()
12.1.2: print_model_values()

Figure 7: Steps performed by UBCSAT

4 Experimental Evaluation

In this chapter we describe the preliminary experimental evaluation of WalkSMT on the set of formulas we considered.

4.1 Testing Description

In order to evaluate the performance of our SLS-based SMT procedure, we compared it against a state-of-the-art SMT procedure based on DPLL solver.

We considered three version of WalkSMT:

- Basic-WalkSMT, which does not include improvement techniques;
- Learning-WalkSMT, which combines Basic-WalkSMT with the simple learning technique;
- Best-WalkSMT, which combine Basic-WalkSMT with the following techniques: multiple learning with $f = 1$, pure-literal filtering and ghost-literal filtering.

For each version we used the Adaptive Novelty$^+$ algorithm[13] with $(wp, \phi, \theta) = (0.01, 0.2, 1/6)$ [**?**], the largest integer limit of search steps and zero restarts (max_flips and max_tries settings). Moreover, we run WalkSMT with three different random seeds (they are $0, 49, 213$).

We adopted MathSAT as DPLL-based SMT solver and considered the following two configuration:

- MathSAT with static learning, early pruning and $\mathcal{T}$-propagation;
- MathSAT with static learning and without early pruning and $\mathcal{T}$-propagation.

All tests was run setting a timeout of 600 seconds

Finally, the tests suite we used includes both randomly-generated formulas and industrial formulas selected from the SMT-LIB benchmarks[**?**, **?**]. Since SLS algorithms are essentially incomplete and WalkSMT reaches the timeout on unsatisfiable formulas, the tests suit only contains satisfiable formulas.

[13]Remember that Adaptive Novelty$^+$ uses an adaptive mechanism whose input parameter are the walk probability (or noise setting) wp, the rate of change of noise ϕ and the stagnation detection θ.

4.2 SMT-LIB Formulas

The SMT-LIB benchmarks we used to create the tests suite refers to the unquantified real linear arithmetic logic[14] (QF_LRA) and includes the following groups: *sc*, *uart*, *tta_startup*, *TM*, *sal* and *miplib* (see references for their description). In particular, the tests suite contains about 10 satisfiable formulas for each group.

The Figure 12 shows that the simple learning technique leads to a significant improvement in the performance of Basic-WalkSMT. In fact, Basic-WalkSMT has catastrofic results since it almost always reaches the timeout without finding a solution. As we can see in Figure 11, further improvements are due to the usage of the Best-WalkSMT's techniques.

However, the Figures 8 and 10 show that MathSAT is considerably faster than Best-WalkSMT even if we remove early pruning and $\mathcal{T}$-propagation. This could be due to the effectiveness of BCP on industrial problems and to the fact that the $\mathcal{T}$-solver (based on conflict reasoning) could be not well integrated with the SLS paradigm.

4.3 Random Generated Formulas

Formulas are generated in terms of the tuple of parameters $\langle m, k, a, n, p, L\rangle$ where m is the number of clauses, k is the number of disjuncts per clause, a the number of $\mathcal{T}$-atoms, n is the number of $\mathcal{T}$-variables occurring in the formula, p is the number of $\mathcal{T}$-variables per $\mathcal{T}$-atom and L is a positive integer number such that all the numeric constants belong to the interval $[-L, L]$.

Given a tuple $\langle m, k, a, n, p, L\rangle$, formulas are produced by randomly generating m clauses of length k. Each disjunct is randomly selected from a list of a $\mathcal{T}$-atoms and then negated with probability 1/2 (one $\mathcal{T}$-atom can appear only one time within a clause). Each $\mathcal{T}$-atom $c_1 * x_1 + \ldots + c_p * x_p \leq c$ is generated so that c_i and c are randomly selected in the interval $[-L, L]$ and variables x_i are randomly chosen with probability $1/n$ (one variable can appear only one time within a $\mathcal{T}$-atom).

We generated the formulas in the tests suite by taking $L = 100$, $k = 3$ and $p = 4$. The results of the experiments for $n = 20$ are shown in the Figure 13 and for $n = 10, 30$ can be seen at ...

[14]Formulas over unquantified real linear arithmetic logic are Boolean combinations of inequations between linear polynomials over real variables.

Each graph shows curves for Basic-WalkSMT, Learning-WalkSMT, Best-WalkSMT, MathSAT and MathSAT without early pruning and $\mathcal{T}$-propagation. They represents the execution time versus the ratio of clause to $\mathcal{T}$-atoms $r = m/a$. For the three versions of WalkSMT, points are the median value of the median values related to the execution with differend random seeds and computed among 20 randomly generated samples. Instead, for the two versions of MathSAT, points are simply the median value among the 20 samples (since no random seeds was used).

Furthermore, since samples refer to the random generated formulas which are satisfiable, the satisfiability percentage, which is also shown in graphs of Figure 13, is the percentage of formulas we generated to find 20 satisfiable formulas. For example, in the plot located in the first column of the last row of Figure 13 the percentage 0.001% for $r = 5$ means that we had to tests 337631 formulas (using MathSAT and setting a timeout equal to 600 seconds). From plots we can notice that the 50% of $\mathcal{T}$-satisfiable formulas decreases with the complexity of the formula (that is, the number of $\mathcal{T}$-atoms increases), in fact for $a = 30$ it is obtained when $4 \leq r \leq 5$ whereas for $a = 70$ when $3 \leq r \leq 4$.

Results show that there is a very small difference between the performance of Learning-WalkSMT and Best-WalkSMT. Moreover, for the simplest kind of formulas (those having 30 $\mathcal{T}$-atom), we can notice that Basic-WalkSMT has performance similar to the other two "optimized" versions of WalkSMT (except for some instance reaching the timeout). Anyhow, the more complicated formulas became the more degradation in execution time Basic-WalkSMT has.

Differently from the results on SMT-LIB formulas, on random generated formulas there is no winner between Learning-WalkSMT, Best-WalkSMT and MathSAT. However, on the most complicated formulas (that is, those with $a = 70, 80$) MathSAT is slightly better than Best-WalkSMT but the performance of MathSAT without early pruning and $\mathcal{T}$-propagation becomes dreadful reaching the timeout.

5 Conclusion

We described a new SMT procedure, called WalkSMT, which integrates by lazy approach a Boolean SLS solver with a $\mathcal{T}$-solver.

First of all we gave some theoretical concepts and described the state-of-the-art of the SMT procedures and SLS algorithms.

Then we present a basic version of the SLS-based SMT solver and a group of techniques aimed to improve the synergy between the Boolean and the theory specific components and to increase the performance of the solver.

Finally, we evaluate the performance of WalkSMT by comparing it against a state-of-the-art SMT solver based on DPLL, MathSAT. To understand the efficiency of the optimization technologies, we compared different configuration of WalkSMT (from a basic version to a best version). Moreover, in order to comprehend the different factors that influence the performance of a DPLL-based SMT solver, we consider two configuration for MathSAT: one with all the optimization enabled and one in which we disabled two important optimizations that are impossible to apply in an SLS-based algorithm, namely early pruning and $\mathcal{T}$-propagation. We performed our comparison over two distinct sets of problem instances: structured industrial problems coming from the SMT-LIB and randomly-generated unstructured problems. Results show that the performance of the WalkSMT is far from that of the DPLL-based one on SMT-LIB problems and is comparable on random problems.

References

[1] Wanxia Wei Chu Min Li and Harry Zhang. Combining adaptive noise and look-ahead in local search for sat. *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT-07)*, 4501 of Lecture Notes in Computer Science:121–133, 2007.

[2] Stephen Clamage. Mixing c and c++ code in the same program. *http://developers.sun.com/solaris/articles/mixing.html*.

[3] Frank Hutter Dave A. D. Tompkins and Holger H. Hoos. Scaling and probabilistic smoothing (saps). *SAT 2004 Competition Booklet*, 2004.

[4] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for dpll(t). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[5] Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *National Conference on Artificial Intelligence*, pages 28–33, 1993.

[6] H. Hoos. An adaptive noise mechanism for walksat, 2002.

[7] Holger H. Hoos and Thomas Stutzle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

[8] Holger H. Hoos and Thomas Stutzle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann / Elsevier, 2004.

[9] F. Hutter, D. Tompkins, and H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Lecture Notes In Computer Science. Vol. 2470. Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming.*, 2002.

[10] Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, and Duc Nghia Pham. Neighbourhood clause weight redistribution in local search for sat. *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP-05)*, 3709 of Lecture Notes in Computer Science:772–776, 2005.

[11] Kazuo Iwama and Suguru Tamaki. Improved upper bounds for 3-sat. *Proceedings of the 15th ACM-SIAM Symp. on Discrete algorithms (SODA 04)*, page 328328, 2004.

[12] Stuart Bain John Thornton, Duc Nghia Pham and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for sat. *Proceedings of the Ninteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 191–196, 2004.

[13] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, 3569 of Lecture Notes in Computer Science:158–172, 2005.

[14] Christos H. Papadimitriou. On selecting a satisfying truth assignment. *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS-91)*, pages 163–169, 1991.

[15] Ramamohan Paturi, Pavel Pudlàk, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. *Proceedings 39th*

Annual Symposium on Foundations of Computer Science, pages 628–637, 1998.

[16] Steven Prestwich. Random walk with continuously smoothed variable weights. *In Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, 3569 of Lecture Notes in Computer Science:203–215, 2005.

[17] Uwe Schning. A probabilistic algorithm for k-sat and constraint satisfaction problems. *Proceedings of the Fourtieth Annual Symposium on Foundations of Computer Science (FOCS-99)*, page 410, 1999.

[18] Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, Volume 3, 2007.

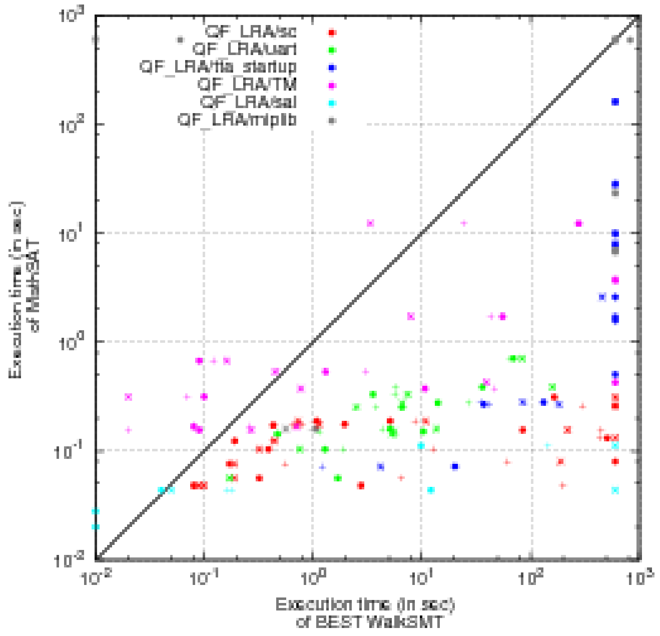
[19] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence.*

[20] B. Selvman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence.*

[21] D. Tompkins and H. Hoos. Warped landscapes and random acts of sat solving, 2004.

[22] Dave A. D. Tompkins and Holger H. Hoos. Novelty+ and adaptive novelty+. *SAT 2004 Competition Booklet*, 2004.

[23] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing: Revised Selected Papers of the Seventh International Conference (SAT 2004, Vancouver, BC, Canada, May 10–13, 2004)*, volume 3542 of *Lecture Notes in Computer Science*, pages 306–320, Berlin, Germany, 2005. Springer Verlag.



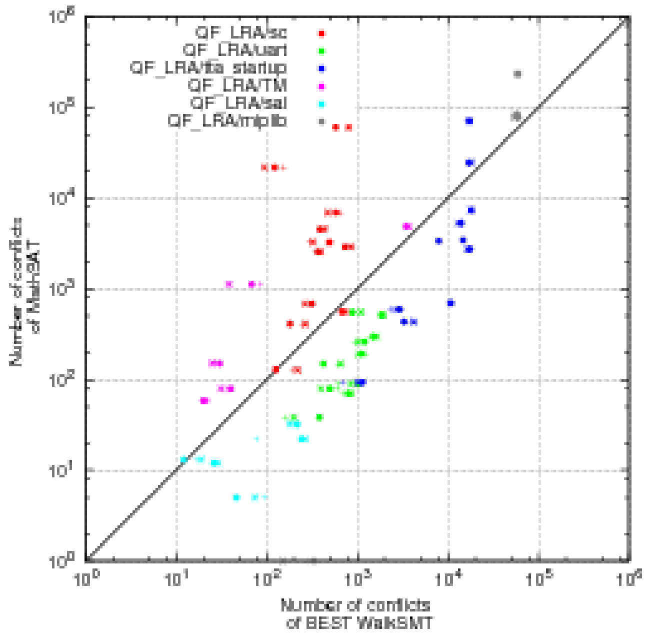
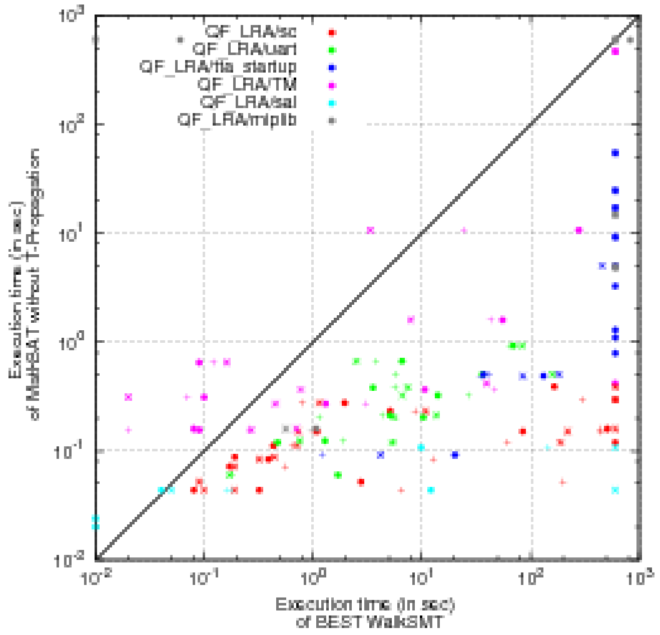


Figure 8: Comparison between Best-WalkSMT and MathSAT. Note that the symbols •, + and $*$ refer to the execution of Best-WalkSMT with respectively $0, 49, 213$ random seeds.)



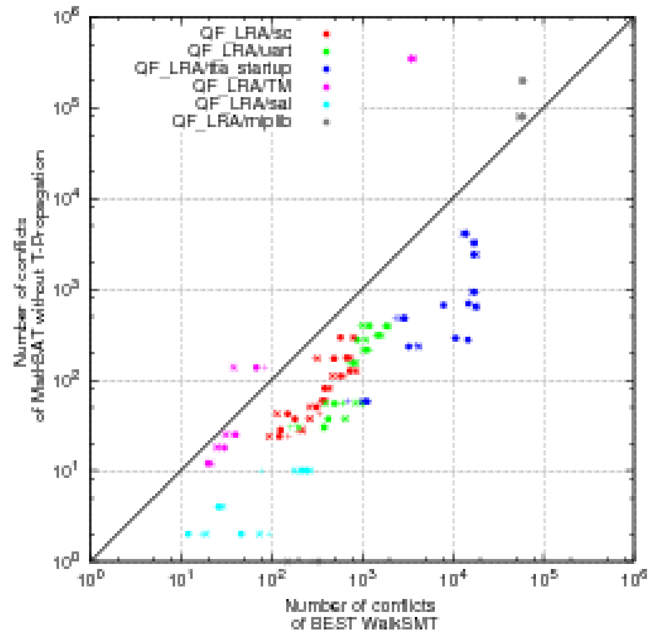
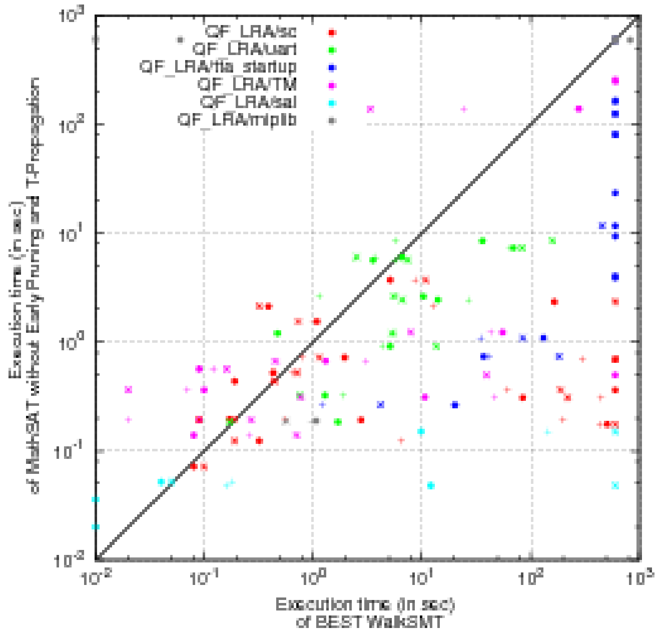


Figure 9: Comparison between Best-WalkSMT and MathSAT without $\mathcal{T}$-Propagation. Note that the symbols $\bullet$, $+$ and $*$ refer to the execution of Best-WalkSMT with respectively $0, 49, 213$ random seeds.



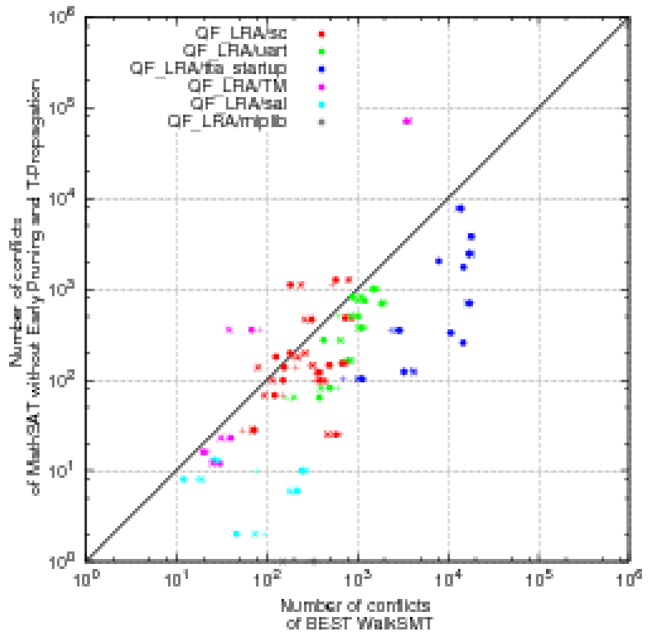
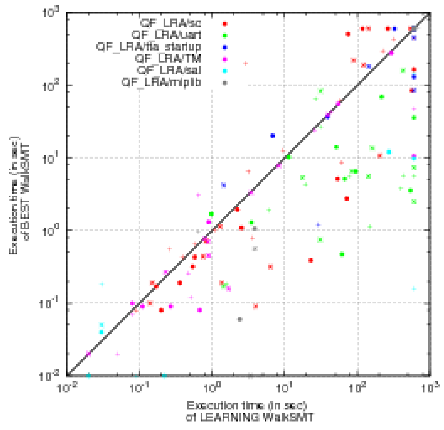
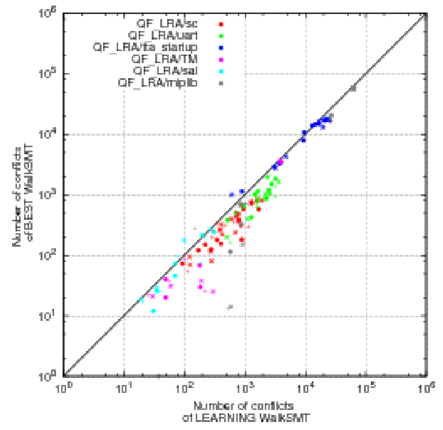


Figure 10: Comparison between BEST-WalkSMT and MathSAT without early pruning and $\mathcal{T}$-Propagation. Note that the symbols $\bullet$, $+$ and $*$ refer to the execution of BEST-WalkSMT with respectively $0, 49, 213$ random seeds.





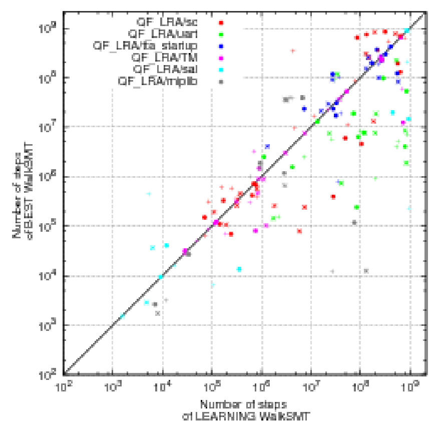


Figure 11: Comparison between LEARNING-WalkSMT and BEST-WalkSMT. Note that the symbols $\bullet$, $+$ and $*$ refer to the execution of LEARNING-WalkSMT and BEST-WalkSMT with respectively $0, 49, 213$ random seeds.

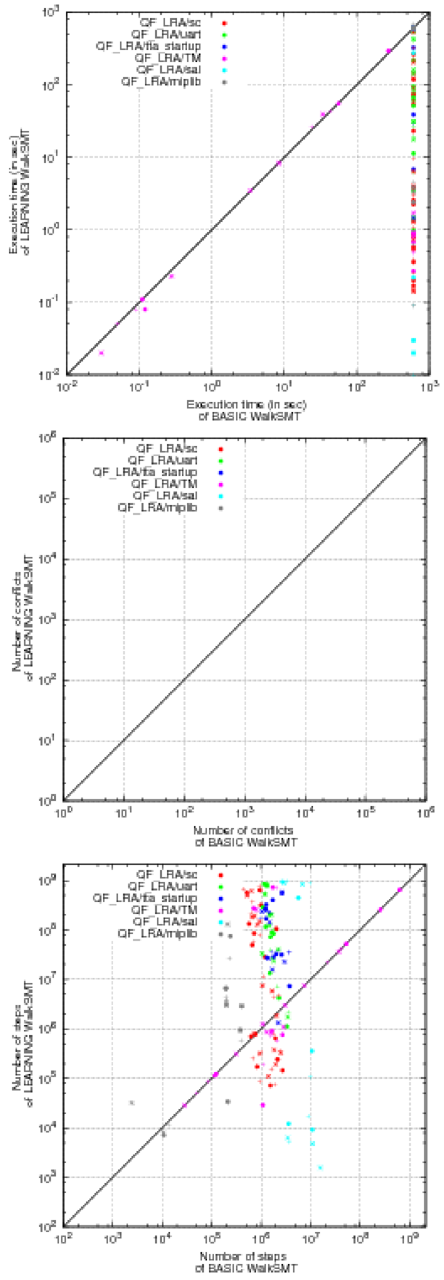


Figure 12: Comparison between Basic-WalkSMT and Learning-WalkSMT. Note that the symbols •, + and $*$ refer to the execution of Basic-WalkSMT and Learning-WalkSMT with respectively $0, 49, 213$ random seeds.

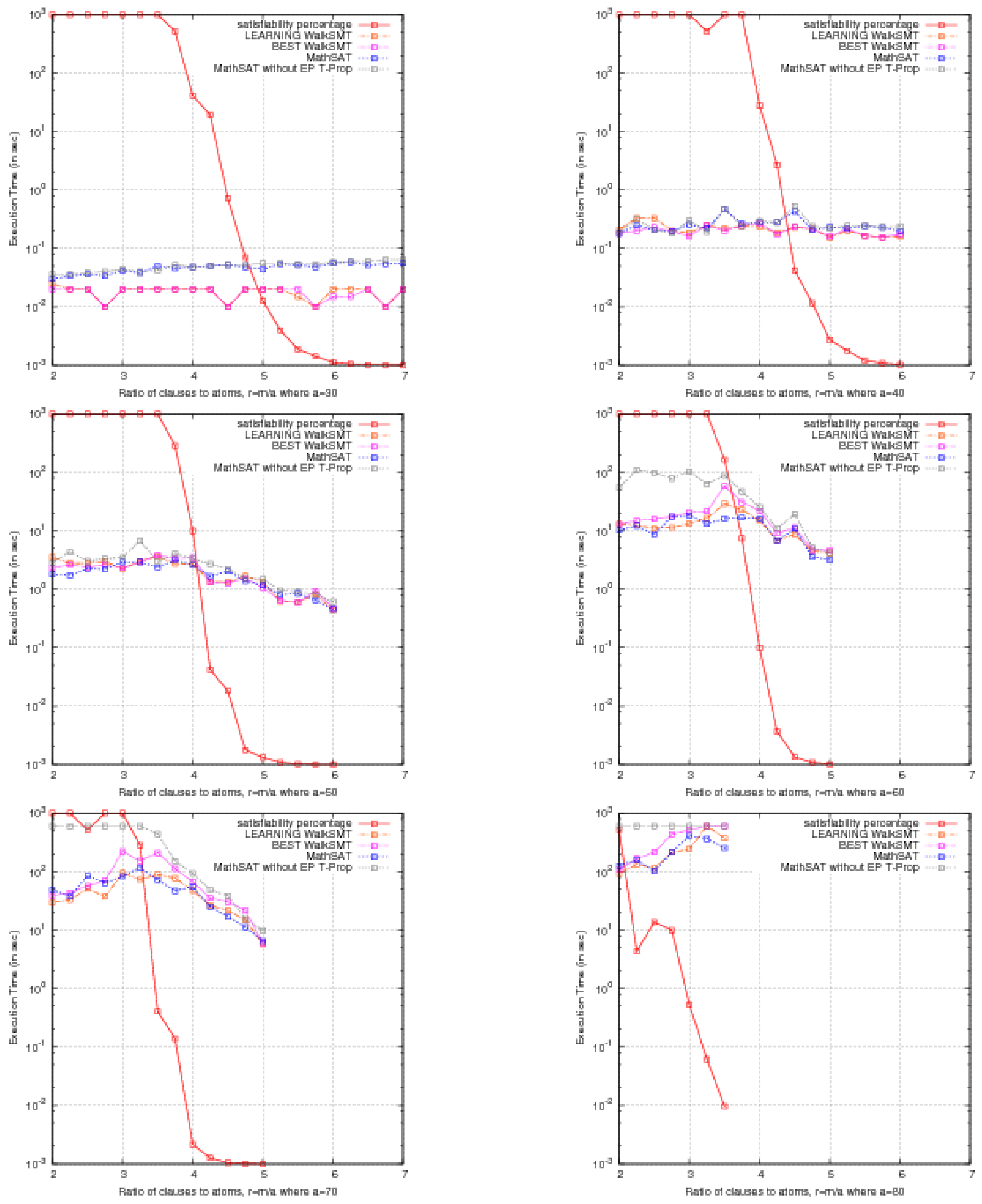


Figure 13: Random Generated Formulas with 20 theory variables and atoms $a = 30, 40, 50, 60, 70, 80$.