Local Search for SMT on Linear Integer Arithmetic

Shaowei Cai[1,2(✉)], Bohan Li[1,2], and Xindi Zhang[1,2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{caisw,libh,zhangxd}@ios.ac.cn
[2] School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Abstract. Satisfiability Modulo Linear Integer Arithmetic, SMT (LIA) for short, has significant applications in many domains. In this paper, we develop the first local search algorithm for SMT (LIA) by directly operating on variables, breaking through the traditional framework. We propose a local search framework by considering the distinctions between Boolean and integer variables. Moreover, we design a novel operator and scoring functions tailored for LIA, and propose a two-level operation selection heuristic. Putting these together, we develop a local search SMT (LIA) solver called LS-LIA. Experiments are carried out to evaluate LS-LIA on benchmarks from SMTLIB and two benchmark sets generated from job shop scheduling and data race detection. The results show that LS-LIA is competitive and complementary with state-of-the-art SMT solvers, and performs particularly well on those formulae with only integer variables. A simple sequential portfolio with Z3 improves the state-of-the-art on satisfiable benchmark sets of LIA and IDL benchmarks from SMT-LIB. LS-LIA also solves Job Shop Scheduling benchmarks substantially faster than traditional complete SMT solvers.

Keywords: SMT · Local Search · Linear Integer Arithmetic · Integer Difference Logic

1 Introduction

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first order logic formula with respect to certain background theories. Inspired by the great success of propositional satisfiability (SAT) solving, SMT attempts to generalize the advances of satisfiability solvers from propositional logic to fragments of first order logic. Typical theories supported by SMT include the theories of integers, real numbers, lists, arrays and bit-vectors. The field of SMT

S. Cai, B. Li and X. Zhang—The authors are listed in alphabetical order, as they all contribute significantly.

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13372, pp. 227–248, 2022.
https://doi.org/10.1007/978-3-031-13188-2_12

has seen significant progress in the past two decades. SMT solvers have become important formal verification engines, with applications in various domains.

In this paper, we focus on the theory of *Linear Integer Arithmetic* (LIA), consisting of arithmetic atomic formulae in the form of $\sum_i a_i x_i + c \bowtie 0$, where $\bowtie \in \{=, \leq\}$, c and a_i's are rational numbers and x_i's are integer variables. Moreover, we are also interested in a popular fragment of LIA, namely *Integer Difference Logic* (IDL), consisting of arithmetic atomic formulae to constrain the difference between pairs of integer variables in the form of $a - b \leq k$, where a, b are integer variables and k is integer constant. The SMT problem with the background theory of LIA and IDL, is to determine the satisfiability of the Boolean combination of respective arithmetic atomic formulae and propositional variables, and referred to as SMT (LIA) and SMT (IDL).

SMT (LIA) is important in software verification and automated reasoning, since most programs use integer variables and perform arithmetic operation on them [35]. Specifically, SMT (LIA) has various applications in automated termination analysis [16], sequential equivalence checking [34], and state reachability checking under weak memory models [24]. SMT (IDL) has found applications in problems with timing-related constraints [17], such as hardware models with ordered data structures [23], stable models computing [30], and job shop scheduling [40].

Much effort has been devoted to solving SMT (LIA) and SMT (IDL). The most popular approach is the *lazy* approach [3,41], also known as DPLL(T) [38], which is a central development of SMT. Many DPLL(T) solvers have been developed for SMT (LIA) [7,19] and SMT (IDL) [31,37,47]. In this approach, the formula is abstracted into a Boolean formula by replacing arithmetic atomic formulae with fresh Boolean variables. A SAT solver is used to reason about the Boolean structure and solve the Boolean formula, while a theory solver receives assignments from the SAT solver and performs decision procedure to solve the conjunctions of atomic subformulae, including consistency checking of the assignments and theory-based deduction.

The effort in this approach is mainly devoted to producing more effective theory solvers. Simplex-based linear arithmetic solvers that can be integrated efficiently in the DPLL(T) framework were studied [19]. A simplex-based decision procedure that minimizes the sum of infeasibilities of constraints was proposed [32]. A theory solver made use of layering and several heuristics to achieve good performance [26]. A theory solver called SPASS-IQ was designed to efficiently handle unbounded problems [6,8]. According to recent SMT Competitions,[1] almost all state-of-the-art SMT (LIA) and SMT (IDL) solvers are based on the lazy approach, including MathSAT5 [15], CVC5 [2], Yices2 [21], Z3 [18], SMTInterpol [14] and SPASS-SATT [7].

The other approach is the *eager* approach, where the formula is reduced to an equi-satisfiable Boolean formula and then solved by a SAT solver. This approach works well for SMT (IDL). Typically, all intrinsic dependencies between integer variables are computed and encoded as Boolean constraints. Encoding to

[1] https://smt-comp.github.io/.

Boolean formula is done either by deriving adequate ranges for formula variables (a.k.a. small domain encoding) [9,39,45], or by deriving all possible transitivity constraints (a.k.a per-constraint encoding) [44]. A hybrid method combining the strengths of two encoding scheme showed robust performance [43].

Local search is an incomplete method which plays an important role in many combinatorial problems [28]. Local search algorithms move from solution to solution in the space of candidate solutions by applying local changes. It has been successfully applied to Boolean Satisfiability (SAT) problem [1,4,12,13,33] and is competitive with CDCL solvers on certain types of instances. However, very limited effort has been devoted to local search for SMT. The idea of integrating local search solvers with theory solvers has been explored before, where a local search SAT solver WalkSAT is used to solve the Boolean skeleton of the SMT formula [26]. A pure local search solver [22] was proposed to solve SMT on the theory of *bit vectors* directly on the theory level, by lifting the successful techniques in local search SAT solvers to the SMT level. In [36], a precise propagation based local search for SMT on the theory of *bit vectors* is proposed, by introducing a notion of essential inputs to lift the concept of controlling inputs from the bit-level to the word-level. We are not aware of any work on local search solvers for SMT on integer arithmetic theories.

This work, for the first time, develops a local search solver for SMT (LIA), which directly operates on both Boolean and integer variables, breaking through the traditional approaches. We propose a local search framework, which switches between two modes, namely Boolean mode and Integer mode. Each mode consists of consecutive operations of the same type (either Boolean or integer). Moreover, for the Integer mode, we propose a literal-level operator named *critical move* and a fine-grained scoring function named *distance score* which takes into account the *distance to truth* of literals and *distance to satisfaction* of clauses. A two-level heuristic is proposed to pick a *critical move* operation. By putting these together, we develop a local search solver for SMT (LIA) called LS-LIA.

Experiments are conducted to evaluate LS-LIA on 4 benchmarks, including QF_LIA and QF_IDL benchmarks from SMTLIB (excluding unsatisfiable instances),[2] instances encoded from job shop scheduling (JSP) and instances generated by data race detection system on a real world benchmark [29]. We compare our solver with state of the art SMT solvers including Z3, CVC5, Yices and MathSAT5. Experimental results show that LS-LIA is competitive and complementary with state-of-the-art SMT solvers. Particularly, LS-LIA is good at solving instances without Boolean variables, noting that a large portion in SMTLIB (81.1% for LIA and 44.1% for IDL) belongs to this type. A simple sequential portfolio with Z3 improves the state-of-the-art on satisfiable QF_LIA and QF_IDL benchmarks from SMT-LIB. LS-LIA also solves Job Shop Scheduling benchmarks substantially faster than traditional complete SMT solvers.

[2] http://www.smt-lib.org/.

2 Preliminary

Definition 1. *Linear Integer Arithmetic (LIA): Let $P = \{p_1, p_2, ...p_n\}$ be a set of propositional (Boolean) variables and $X = \{x_1, x_2...x_m\}$ be a set of integer-valued variables. The linear integer arithmetic formulae are inductively defined.*

1) $p \in P$ is a propositional atomic LIA formula.
2) $\sum_i a_i x_i \bowtie k$ is an arithmetic atomic LIA formulae, where $\bowtie \in \{=, \leq\}$, $x_i \in X$, k, and a_i are constant coefficients (rationals or integers).
3) If ψ and φ are LIA formulae, so are $\psi \vee \varphi$, $\psi \wedge \varphi$ and $\neg\varphi$.

In the above definition, we note that with '$\leq$' and '$=$', we other inequalities can also be expressed. Specifically, we can express $\sum_i^n a_i x_i < k$ as $\sum_i^n a_i x_i \leq k-1$, $\sum_i^n a_i x_i > k$ as $\neg(\sum_i^n (a_i x_i) \leq k)$, $\sum_i^n a_i x_i \geq k$ as $\sum_i^n (-a_i x_i) \leq (-k)$ and $(\sum_i^n a_i x_i) \neq k$ as $(\sum_i^n a_i x_i \leq (k-1) \vee \neg(\sum_i^n (a_i x_i) \leq k)$.

A popular fragment of linear integer arithmetic is call *Integer Difference Logic* (IDL), where the arithmetic atomic formulae are in the form of $x_i - x_j \bowtie k$, where $\bowtie \in \{=, \leq\}$, $x_i, x_j \in X$ and k is constant.

Example 1. A typical SMT (LIA) formula F: $(p_1 \vee (x_1 + 2x_2 \leq 2)) \wedge (p_2 \vee (3x_3 + 4x_4 + 5x_5 = 2) \vee (-x_2 - x_3 \leq 3))$, where $X = \{x_1, x_2, x_3, x_4, x_5\}$ and $P = \{p_1, p_2\}$ are the sets of integer-valued and propositional variables respectively.

A literal is an atomic formula, or the negation of an atomic formula. A *clause* is the disjunction of a set of literals, and a formula in *conjunctive normal form* (CNF) is the conjunction of a set of clauses. For an SMT (LIA) formula F, an assignment α is a mapping $X \rightarrow Z$ and $P \rightarrow \{false, true\}$, and $\alpha(x)$ denotes the value of a variable x under α. A *complete assignment* is a mapping which assigns to each variable a value. A literal is a true literal if it evaluates to true under the given assignment, and otherwise it is a false literal. A clause is *satisfied* if it has at least one true literal, and *falsified* if all literals in the clause are false. A complete assignment is a *solution* to an SMT (LIA) formula if it satisfies all the clauses.

When applying local search algorithms to solve a satisfiability problem, the search space consists of all complete assignments, each of which is a candidate solution. Typically, a local search algorithm starts from a complete assignment, and iteratively modifies the assignment by changing the value of one variable, to search for a satisfying assignment.

In local search, an *operator* defines how to modify the candidate solution. When an operator is instantiated by specifying the variable to operate, we obtain an *operation*. For example, a standard operator for SAT is *flip*, which modifies the current assignment by changing the value of a Boolean variable, and *flip*(x_1) is an operation, where x_1 is a Boolean variable in the formula.

Given a formula F, the *cost* of an assignment α, denoted as $cost(\alpha)$, is the number of falsified clauses under α. In dynamic local search algorithms which use clause weighting techniques, however, $cost(\alpha)$ denotes the total weight of all falsified clauses under an assignment α. Given a formula and an assignment α, an operation *op* is said *decreasing* if $cost(\alpha') < cost(\alpha)$, where α' is the resulting assignment by applying *op* to α.

Algorithm 1: Local Search of Mode X

```
/* X can be Integer or Boolean */
while non_impr_steps ≤ L × P_X do
    if α satisfies F then return α if ∃ decreasing X operations then
        op := a decreasing X operation
    if fail to find decreasing X operation then
        update clause weights;
        op := an X operation from a random falsified clause containing X
        literals;
    perform op to modify α;
```

3 A Local Search Framework for SMT (LIA)

In this section, we introduce a local search framework for SMT (LIA), which switches between integer operations and Boolean operations.

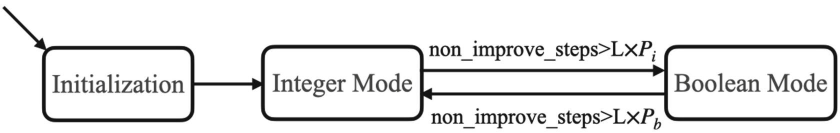


Fig. 1. An SMT Local Search Framework

In the beginning, the algorithm generates a complete assignment α. Then, it iteratively modifies α by performing operations on variables. The algorithm terminates once α becomes a solution to the formula, and outputs "SATISFIABLE" as well as the solution. If the algorithm fails to find a solution within the pre-set time limit, it is cut off and outputs "UNKNOWN".

As depicted in Fig. 1, after the initialization, the algorithm works in two modes, namely Integer mode and Boolean mode. In each mode X (X is Integer or Boolean), an X operation is picked to modify α, where an X operation refers to an operation that works on a variable of data type X. The two modes switches to each other when the number of non-improving steps (denoted as *non_improve_steps*) of the current mode reaches a threshold. The threshold is set to $L \times P_b$ for the Boolean mode and $L \times P_i$ for the Integer mode, where P_b and P_i denote the proportion of Boolean and integer literals to all literals in falsified clauses, and L is a parameter. Note that *non_improve_steps* is set to 0 whenever entering a mode, and then in each following step, it increases by one if $cost(\alpha) \geq cost^*$ in the current step, where $cost^*$ is the cost of the best assignment visited before.

The intuitions of the two mode framework are as follows. When all variables of one type (either Boolean or integer) are fixed, the formula is reduced to a subformula that contains only variables of the other type. Thus, by consecutively performing X (X can be Boolean or Integer) operations in a certain period, the algorithm focuses on dealing with a subformula consisting of only X variables. The switching threshold is set as $L \times P_X$, as we consider that when X literals accounts for larger proportion of all literals in falsified clauses, more steps should be allocated for the corresponding mode.

Local Search in One Mode

No matter the mode in which the algorithm works, it adopts a general procedure as described in Algorithm 1. It prefers to pick a decreasing operation (according to some heuristic) if any. If the algorithm fails to find any decreasing operation, it updates clause weights by increasing the weights of falsified clauses, and then picks an X operation from a random falsified clause containing X literals. Note that we can always pick a falsified clause with X literals (line 7). This is because when the algorithm works in X mode, since $non\_impr\_steps \leq L \times P_X$, we have $P_X > 0$, and so there exists at least one falsified clause with X literals.

As for clause weighting, our algorithm employs the probabilistic version of the PAWS scheme [13,46]. When the clause weighting scheme is activated, the clause weights are updated as follows. With probability $1 - sp$, the weight of each falsified clause is increased by one, and with probability sp, for each satisfied clause whose weight is greater than 1, the weight is decreased by one.

4 The Critical Move Operator and a Two-Level Heuristic

In this section, we introduce key techniques in the Integer mode. We propose a novel operator called critical move, and also a two-level heuristic for choosing a critical move in the Integer mode.

A key and basic component of a local search algorithm is the operator. For handling Boolean variables, our algorithm adopts the typical local search operator for SAT, namely *flip*, which modifies the value of a Boolean variable to the opposite of its current value (from True to False, or from False to True). For handling integer variables, we propose a novel operator called *critical move* which works on the literal level.

4.1 Critical Move

Different from the Boolean operator, an integer operator has two parameters – besides the variable to operate, it also needs to consider the increment (may be positive or negative) on the value.

Let us first consider a simple operator, which motivates us to propose a literal-level operator. A simple integer operator is to modify the value of a variable a by a fixed increment inc, that is, $\alpha(a) := \alpha(a) \pm inc$. The parameter inc needs fine tuning. If inc is too small, it may take many iterations before making any falsified literal become true. If inc is too big, the algorithm may even become problematic that it can never make some literals true and thus essentially unable to solve some formulae.

Example 2. Given a formula $F : (b - a \geq 3) \wedge (b - a \leq 5)$ and the current assignment is $\alpha = \{a = 0, b = 0\}$. If $inc = 1$, it needs at least 3 operations to satisfy the formula. If $inc = 10$, then the formula cannot be satisfied using operations of this type, as the value of $b - a$ would be always a multiple of 10.

In fact, in order to avoid the case that some literals can never become true (when the *inc* is too big), the only acceptable value of *inc* is 1. The main reason accounting for such a drawback is that the above operator ignores the literal-level information. We propose a literal-level operator for integer variables called *critical move*, which is defined below.

Definition 2. *The critical move operator, denoted as $cm(x, \ell)$, assigns an integer variable x to the threshold value making literal ℓ true, where ℓ is a falsified literal containing x. Specifically, for each of the four basic forms of the falsified literal ℓ, let $\Delta = \sum_i a_i x_i - k$, an operation is described below:*

- *$\ell : \sum_i a_i x_i \leq k$. there exists a cm operation $cm(x_i, \ell)$ for each variable x_i: if the coefficient $a_i > 0$, then $cm(x_i, \ell_1)$ decreases $\alpha(x_i)$ by $\left\lceil \left| \frac{\Delta}{a_i} \right| \right\rceil$; if $a_i < 0$, then $cm(x_i, \ell_1)$ increases $\alpha(x_i)$ by $\left\lceil \left| \frac{\Delta}{a_i} \right| \right\rceil$.*
- *$\ell : \neg(\sum_i a_i x_i \leq k)$, that is, $\sum_i a_i x_i > k$. there exists a cm operation $cm(x_i, \ell)$ for each variable x_i: if the coefficient $a_i > 0$, then $cm(x_i, \ell_1)$ increases $\alpha(x_i)$ by $\left\lceil \left| \frac{1-\Delta}{a_i} \right| \right\rceil$; if $a_i < 0$, then $cm(x_i, \ell_1)$ decreases $\alpha(x_i)$ by $\left\lceil \left| \frac{1-\Delta}{a_i} \right| \right\rceil$.*
- *$\ell : \sum_i a_i x_i = k$. There exists an operation $cm(x_i, \ell)$ for each variable x_i with $a_i \mid \Delta$, which increases $\alpha(x_i)$ by $-\frac{\Delta}{a_i}$.*
- *$\ell : \neg(\sum_i a_i x_i = k)$. There exist 2 cm operations for each variable x_i, to +1 or -1 on x_i.*

Given the above definition of the critical move, an issue with this operator is that it may stall on equalities, when there is no such variable with $a_i \mid \Delta$ in ℓ. To address this issue, in this situation, we additionally employ a simple strategy—pick a random variable in that literal and performs $+1$ or -1 to decrease $|\Delta|$.

Example 3. Assume we are given two falsified literals $l_1 : (2b - a \leq -3)$ and $l_2 : (5c - d + 3a = 5)$, and the current assignment is $\alpha = \{a = 0, b = 0, c = 0, d = 0\}$. Then $cm(a, l_1)$, $cm(b, l_1)$, $cm(c, l_2)$, and $cm(d, l_2)$ refers to assigning a to 3, assigning b to -2, assigning c to 1 and assigning d to -5 respectively. Note that there does not exist $cm(a, l_2)$, since $3 \nmid -5$.

An important property of the *cm* operator is that after the execution of a *cm* operation, the corresponding literal must be true. Therefore, by picking a falsified literal and performing a *cm* operation on it, we can make the literal become true.

The critical move operations are analogous to update operations in other linear arithmetic model searching procedures. For example, Simplex for DPLL(T)

[20] also progresses through a sequence of candidate assignments by updating the assignment to a variable to satisfy its bound. The significant distinction of critical moves is only updating input variables and always updating by an integral amount, as we can see from Definition 2.

4.2 A Two-Level Heuristic

In this subsection, we propose a two-level heuristic for selecting a decreasing *cm* operation. We distinguish a special type of decreasing *cm* operations from others, and give a priority to such operations.

From the viewpoint of algorithm design, there is a major difference between *cm* and *flip* operations. A *flip* operation is decreasing only if the flipping variable appears in at least one falsified clause. For a $cm(x, \ell)$ operation to be decreasing, the literal ℓ does not necessarily appear in any falsified clause. This is because integer variables are multi-valued, and a $cm(x, \ell)$ operation that modifies the value of x would have impact on other literals with the same variable x.

Example 4. Given a formula $F = c_1 \wedge c_2 = (a - b \leq 0 \vee b - e \leq -2) \wedge (b - d \leq -1)$, suppose the current assignment is $\alpha = \{a = 0, b = 0, d = 0, e = 0\}$, then c_1 is satisfied and c_2 is falsified. The operation $op1 = cm(b, b - e \leq -2)$ refers to assigning b to -2, and $op2 = cm(b, b - d \leq -1)$ refers to assigning b to -1. The literal of $op1$ does not appear in any falsified clause while the literal of $op2$ appears in a falsified clause c_2. Both operations are decreasing, as either of them would make clause c_2 become satisfied without breaking any satisfied clause.

In order to find a decreasing *cm* operation whenever one exists, we need to scan all *cm* operations on false literals. That is, the candidate set of decreasing operations is $D = \{cm(x, \ell) | \ell$ is a false literal and x appears in $\ell\}$. If $D = \emptyset$, there is no decreasing *cm* operation. We propose to distinguish a special subset $S \subseteq D$ from the rest of D, which is $S = \{cm(x, \ell) | \ell$ appears in at least one falsified clause and x appears in $\ell\}$. Note that any *cm* operation in S would make at least one falsified clause become satisfied. Based on this distinction, we propose a two-level selection heuristic as follows:

- The heuristic prefers to search for a decreasing *cm* operation from S.
- If it fails to find any decreasing operation from S, then it searches for a decreasing *cm* operation from $D \backslash S$.

Besides improving the efficiency of picking a decreasing *cm* operation, there is an important intuition underlying this two-level heuristic. We prefer to pick a decreasing *cm* operation from S, because such operations are conflict driven, as any $cm \in S$ would force a falsified clause become satisfied. This idea can be seen as a LIA version of focused local search for SAT, which has been the core idea of WalkSAT-family SAT solvers [1,4,42].

5 Scoring Functions

Local search algorithms employ scoring functions to guide the search. We introduce two scoring functions, which are used to compare different operations and guide the local search algorithm to pick an operation to execute in each step.

A perhaps most commonly used scoring function for SAT, denoted as *score*, measures the change on the cost of the assignment by flipping a variable. This scoring function indeed can be used to evaluate all types of operations as it only concerns the clauses state (satisfied or falsified). We also employ *score* in our algorithm, for both *flip* and *cm* operations. Formally, the *score* of an operation is defined as

$$score(op) = cost(\alpha) - cost(\alpha'),$$

where α' is obtained from α by applying *op*. Note that, our algorithm employs a clause weighting scheme which associates a positive integer weight to each clause, and thus the *cost* of an assignment is the total weight of falsified clauses. It is easy to see that an operation *op* is *decreasing* if and only if $score(op) > 0$. Our algorithm prefers to choose the operation with greater *score* in the greedy mode, for both Boolean and integer operations.

For integer operations, we propose a more fine-grained scoring function, measuring the potential benefit about pushing a falsified literal towards the direction of becoming true. Firstly, we propose a property for literals to measure this merit.

Definition 3. *Given an assignment α, for an arithmetic literal $\ell : \sum_i a_i x_i \leq k$, its distance to truth is $dtt(\ell, \alpha) = max\{\sum_i a_i \alpha(x_i) - k, 0\}$. For a Boolean literal ℓ and an arithmetic literal $\ell : \sum_i a_i x_i = k$, $dtt(\ell, \alpha) = 0$ if ℓ is true under α and $dtt(\ell, \alpha) = 1$ otherwise.*

Suppose the current assignment is α, for an arithmetic literal $\ell : \sum_i a_i x_i \leq k$, if $\sum_i a_i \alpha(x_i) > k$, then the literal is falsified, and its *dtt* is defined to be $\sum_i a_i \alpha(x_i) - k$. In this case, if we decrease the value of x_i with $a_i > 0$, or increase the value of x_i with $a_i < 0$, the *dtt* of ℓ would decrease. When $\sum_i a_i \alpha(x_i) \leq k$, the literal ℓ is true, and thus its *dtt* is defined to be 0.

The definition of *dtt* for arithmetic literals somehow resembles the violation function for constraint satisfaction problems [27], and the violation operator in the simplex with sum of infeasibilities for SMT [32]. In this work, we extend it to the clause level to measure the distance of a clause away from satisfaction in a fine-grained manner. Based on the concept of *distance to truth* of literals, we define a function to measure the distance of a clause away from satisfaction.

Definition 4. *Given an assignment α, the distance to satisfaction of a clause c is $dts(c, \alpha) = min_{\ell \in c}\{dtt(\ell, \alpha)\}$.*

According to the definition, the *dts* is 0 for satisfied clause, since there is at least one satisfied literal with $dtt = 0$, while *dts* is positive for falsified clauses. It is desirable to lead the algorithm to decrease the *dts* of clauses. To this end, we propose a scoring function to measure the benefit of decreasing the sum of *dts* of all clauses. Additionally, the function takes into account the clause weights as the *score* function.

Definition 5. *Given an LIA formula F, the distance score of an operation op is defined as*

$$dscore(op) = \sum_{c \in F} (dts(c, \alpha) - dts(c, \alpha')) \cdot w(c),$$

where α and α' denotes the assignment before and after performing op.

For Boolean *flip* operations, *dscore* is equal to *score*. For integer operations, however, compared to the *score* function which only concerns the state (satisfied or falsified) transformations of clauses, *dscore* is more fine-grained, as it considers the *dts* of clauses, which are different among falsified clauses.

Example 5. Given a formula $F = c_1 \wedge c_3 \wedge c_3 = (a - b \leq -1) \wedge (a - c \leq -5 \vee a - d \leq -10) \wedge (b - c \leq -5 \vee b - d \leq -10)$. Suppose $w(c_1) = 1, w(c_2) = 2, w(c_3) = 3$, and the current assignment is $\alpha = \{a = 0, b = 0, c = 0, d = 0\}$, and thus all clauses are falsified. Consider two *cm* operations $op1 = cm(a, a - b \leq -1)$ and $op2 = cm(b, a - b \leq -1)$, which assign $\alpha(a) := -1$ and $\alpha(b) := 1$ respectively, leading to α' and α'' respectively. Then $score(op1) = score(op2) = 1$, as they both make c_1 satisfied. Also, $dts(c_2, \alpha) - dts(c_2, \alpha') = 1$, and $dts(c_3, \alpha) - dts(c_3, \alpha'') = -1$, so $dscore(op1) = (dts(c_1, \alpha) - dts(c_1, \alpha')) \cdot w(c_1) + (dts(c_2, \alpha) - dts(c_2, \alpha')) \cdot w(c_2) = 1 \times 1 + 1 \times 2 = 3$ and $dscore(op2) = -2$ by similar calculation. Therefore, $op1$ is a better operation.

Since the computation of *dscore* has considerable overhead, this function is only used when there is no decreasing operation, as the number of candidate operations is limited here, and it is affordable to calculate their *dscore*.

6 LS-LIA Algorithm

Based on the ideas in previous sections, we develop a local search solver for SMT (LIA) called LS-LIA. As described in Sect. 3, after the initialization, the local search works in either Boolean or Integer mode to iteratively modify α until a given time limit is reached or α satisfies the formula F. This section is dedicated to the details of the initialization and the two modes of local search, as well as other optimization techniques.

Initialization: LS-LIA generates a complete assignment α, by assigning the variables one by one until all variables are assigned. All Boolean variables are assigned with True. As for integer variables x_i, if it has upper bound ub and lower bound lb, that is, there exist unit clauses $x_i \leq ub$ and $x_i \geq lb$, it is assigned with a random value in $[lb, ub]$. If x_i only has upper(lower) bound, x_i is assigned with $ub(lb)$. Otherwise, if the variable is unbounded, it is assigned with 0.

Algorithm 2: Local Search of Boolean Mode

1 **while** $non\_impr\_steps \leq L \times P_b$ **do**
2 **if** α *satisfies* F **then** return α
3 **if** $\exists$ *decreasing flip operation* **then**
4   *op* := such an operation with the greatest *score*
5 **else**
6   update clause weights according to the PAWS scheme;
7   c := a random falsified clause with Boolean variables;
8   *op* := a *flip* operation in c with the greatest *score*;
9 $\alpha := \alpha$ with *op* performed;

Algorithm 3: Local Search of Integer Mode

1 **while** $non\_impr\_steps \leq L \times P_i$ **do**
2 **if** α *satisfies* F **then** return α
3 **if** $\exists$ *decreasing cm operation in falsified clauses* **then**
4   *op* := such an operation with the greatest *score*
5 **else if** $\exists$ *decreasing cm operation in satisfied clauses* **then**
6   *op* := such an operation with greatest *score*
7 **else**
8   update clause weights according to the PAWS scheme;
9   c := a random falsified clause with integer variables;
10   *op* := a *cm* operation in c with the greatest *dscore*;
11 $\alpha := \alpha$ with *op* performed;

Boolean Mode (Algorithm 2): If there exist decreasing *flip* operations, the algorithm selects such an operation with highest *score*.

If the algorithm fails to find any decreasing operation, it first updates clause weights according to the weighting scheme described in Sect. 3. Then, it picks a random falsified clause with Boolean literals and chooses a *flip* operation with greatest *score*.

Integer Mode (Algorithm 3): If there exist decreasing *cm* operations, the algorithm chooses a *cm* operation using the two-level heuristic: it first traverses falsified clauses to find a decreasing *cm* operation with greatest *score* (line 9); if no such operation exists, it searches for a decreasing *cm* operation via BMS heuristic (line 10) [10]. Specifically, it samples t *cm* operations (t is a parameter) from the false literals in satisfied clauses, and selects the decreasing one with greatest *score*.

If the algorithm fails to find any decreasing operation, it first updates clause weights similarly to the Boolean mode. Then, it picks a random falsified clause with Integer literals and chooses a *cm* operation with greatest *dscore*.

Restart Mechanism: The search is restarted when the number of falsified clauses has not decreased for $MaxNoImprove$ iterations, where $MaxNoImprove$ is a parameter.

Forbidding Strategies. Local search methods tend to be stuck in suboptimal regions. To address the cycle phenomenon (i.e. revisiting some search regions), we employ a popular forbidding strategies, called the tabu strategy [25]. After an operation is executed, the tabu strategy forbids the reverse operations in the following tt iterations, where tt is a parameter usually called *tabu tenure*. The tabu strategy can be directly applied in LS-LIA. (1) If a *flip* operation is performed to flip a Boolean variable, then the variable is forbidden to flip in the following tt iterations. (2) If a cm operation that increases (decreases, resp.) the value of an integer variable x is performed, then it is forbidden to decrease (increase, resp.) the value of x in the following tt iterations.

7 Experiments

We carried out experiments to evaluate LS-LIA on 4 benchmarks, and compare it with state-of-the-art SMT solvers. Also, we combine LS-LIA with Z3 to obtain a sequential portfolio solver, which shows further improvement. Additionally, experiments are conducted to analyze the effectiveness of the proposed ideas.

7.1 Experiment Preliminaries

Implementation: LS-LIA is implemented in C++ and compiled by g++ with '−O3' option. There are 5 parameters in LS-LIA: L for switching phases, tt for the tabu scheme, $MaxNoImprove$ for restart, t (the number of samples) for the BMS heuristic and sp (the smoothing probability) for the PAWS scheme. The parameters are tuned according to suggestions from the literature and our preliminary experiments on 20% sampled instances, and are set as follows: $L = 20$, $t = 45$, $tt = 3 + rand(10)$, $MaxNoImprove = 500000$ and $sp = 0.0003$ for all benchmarks.

Competitors: We compare LS-LIA with 4 state-of-the-art SMT solvers according to SMT-COMP 2021,[3] namely MathSAT5 (version 5.6.6), CVC5 (version 0.0.4), Yices2 (version 2.6.2), and Z3 (version 4.8.14), which are the union of the top 3 solvers (excluding portfolio solvers) of QF_LIA and QF_IDL tracks. The binaries of all competitors are downloaded from their websites.

Benchmarks: Our experiments are carried out with 4 benchmarks.

- SMTLIB-LIA: This benchmark consists of SMT (LIA) instances from SMT-LIB.[4] As LS-LIA is an incomplete solver, UNSAT instances are excluded, resulting in a benchmark consisting of 2942 unknown and satisfiable instances.
- SMTLIB-IDL: This benchmark consists of SMT (IDL) instances from SMT-LIB.[5] UNSAT instances are also excluded, resulting in a benchmark consisting of 1377 unknown and satisfiable instances.

[3] https://smt-comp.github.io/2021.
[4] https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_LIA.
[5] https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_IDL.

– JSP: This benchmark consists of 120 instances encoded from job shop scheduling problem resembling [31]. Note that there exists a mistake in the encoding method of original instances from [31], and we fixed it in new instances.
– RVPredict: these instances are generated by a runtime predictive analysis system called RVPredict [29], which formulates data race detection in concurrent software as a constraint problem by encoding the control flow and a minimal set of feasibility constraints as a group of IDL logic formulae. The author of RVPredict kindly provides us with 15 satisfiable instances by running RVPredict on Dacapo benchmark suite [5].

Instances from SMTLIB-LIA and SMTLIB-IDL benchmarks are divided into two categories depending on whether it contains Boolean variables. From the viewpoint of algorithm design, there is a major difference between the operations on Boolean and integer variables. We observe that instances containing only integer variables takes up a large proportion, amount to 81.1% and 44.1%, in these two benchmarks.

Experiment Setup: All experiments are carried out on a server with Intel Xeon Platinum 8153 2.00 GHz and 2048G RAM under the system CentOS 7.9.2009. Each solver is executed one run with a cutoff time of 1200 s (as in the SMT-COMP) for each instance in SMTLIB-LIA, SMTLIB-IDL and JSP benchmarks, as they contain sufficient instances. For the RVPredict benchmark (15 instances), the competitors are also executed one run for each instance as they are exact solvers, while LS-LIA is performed 10 runs for each instance. "#inst" denotes the number of instances in each family. We compare the number of instances where an algorithm finds a model ("#solved"), as well as the run time. The bold value in table emphasizes the solver with greatest "#solved". For RVPredict, LS-LIA solves all instances with 100% success rate and we report the median, minimum and maximum run time among the 10 runs for each instance.

We uploaded our solver as well as JSP and RVPredict benchmarks (along with related information) in the anonymous Github repository.[6]

7.2 Results on SMTLIB-LIA and SMTLIB-IDL Benchmarks

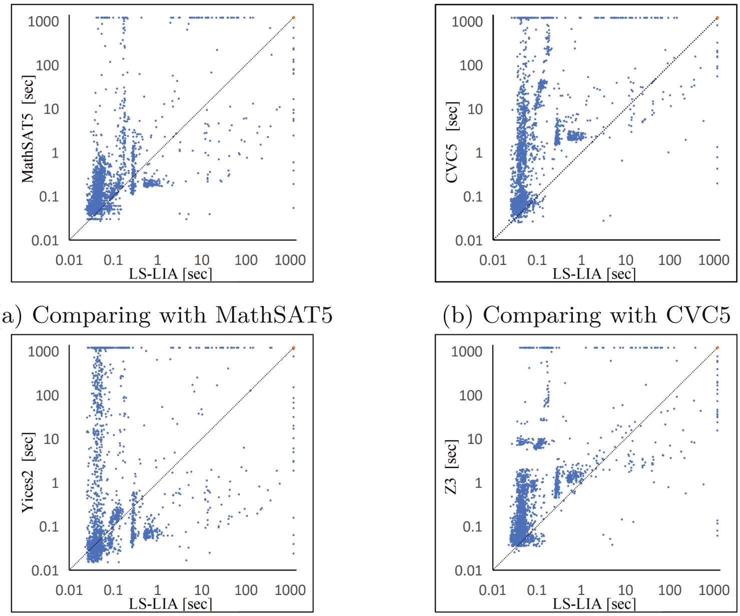
Results on SMTLIB-LIA (Table 1 and Fig. 2). We organize the results into two categories: instances Without Boolean variables, and instances With Boolean variables. LS-LIA outperforms its competitors on the Without Boolean category, solving 2294 out of the 2385 instances. We also present the run time comparisons between LS-LIA and each competitor on the Without Boolean category of SMTLIB-LIA benchmark in Fig. 2. As for the With Boolean category,

[6] https://anonymous.4open.science/r/sls4lia/.

Table 1. Results on instances from SMTLIB-LIA.

Family	Type	#inst	MathSAT5	CVC5	Yices2	Z3	LS-LIA
Without Boolean	20180326-Bromberger	631	538	425	358	532	**581**
	bofill-scheduling	407	**407**	402	**407**	405	391
	CAV_2009_benchmarks	506	**506**	498	396	**506**	**506**
	check	1	1	1	1	1	1
	convert	280	273	205	186	184	**279**
	dillig	230	**230**	**230**	200	**230**	**230**
	miplib2003	16	10	9	11	8	**13**
	pb2010	41	14	5	21	**33**	28
	prime-cone	19	19	19	19	19	19
	RWS	20	11	13	11	**14**	12
	slacks	231	230	**231**	161	230	**231**
	wisa	3	3	3	3	3	3
	Total	2385	2242	2041	1774	2165	**2294**
With Boolean	2019-cmodelsdiff	144	94	**95**	**95**	**95**	51
	2019-ezsmt	108	**84**	79	81	81	54
	20210219-Dartagnan	47	22	22	**23**	**23**	2
	arctic-matrix	100	43	26	59	47	**77**
	Averest	9	**9**	**9**	**9**	**9**	7
	calypto	**24**	**24**	**24**	**24**	**24**	21
	CIRC	18	**18**	**18**	**18**	**18**	3
	fft	5	3	3	3	3	3
	mathsat	21	**21**	**21**	**21**	**21**	13
	nec-smt	1256	1244	425	**1256**	1242	581
	RTCL	2	2	2	2	2	2
	tropical-matrix	108	55	42	71	52	**98**
	Total	1842	1619	766	**1662**	1617	912

the performance of LS-LIA is overall worse than its competitors, but still comparable. A possible explanation is that as local search SAT solvers, LS-LIA is not good at exploiting the relations among Boolean variables. Nevertheless, LS-LIA has obvious advantage in "tropical-matrix" and "arctic-matrix" instances, which are industrial instances from automated program termination analysis [16], showing its complementary performance compared to CDCL(T) solvers.



(a) Comparing with MathSAT5 (b) Comparing with CVC5

(c) Comparing with Yices2 (d) Comparing with Z3

Fig. 2. Run time comparison on Without Boolean category of SMTLIB-LIA

Results on SMTLIB-IDL Benchmark (Table 2 and Fig. 3). Similar to the case for SMTLIB-LIA, our local search solver shows the best performance on IDL instances Without Boolean variables (solving 597 out of the 707 instances), which can be seen from Table 2 and Fig. 3. However, LS-LIA performs worse than its competitors on those With Boolean variables. Overall, LS-LIA cannot rival its competitors on this benchmark, but works particularly well on the instances without Boolean variables.

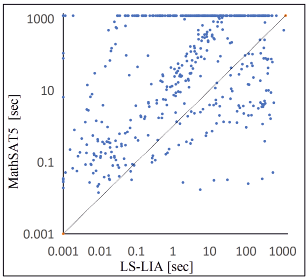
Combination with Z3 and Summary on SMTLIB benchmarks (Table 3). To confirm the complementarity of our local search solver with state of the art SMT solvers, we combine LS-LIA with Z3, by running Z3 with a time limit 600 s, and then LS-LIA from scratch with the remaining 600 s if Z3 fails to solve the instance. This wrapped solver can be regarded as a sequential portfolio solver, denoted as "Z3+LS".

We summarize the results of all solvers, including Z3+LS, on SMTLIB-LIA and SMTLIB-IDL benchmarks in Table 3. Among all single-engine solvers, MathSAT5 solves the most instances of SMTLIB-LIA benchmark, while Z3 solves the most instances of SMTLIB-IDL benchmark. LS-LIA outperforms its competitors on instances Without Boolean variables, indicating that local search is an effective approach for solving SMT (LIA) instances with only integer variables.

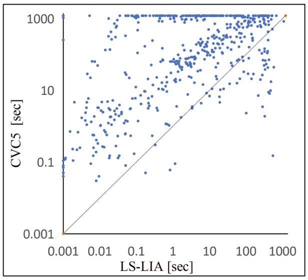
Z3+LS solves more instances than any other solver on both benchmarks, confirming that LS-LIA and Z3 have complementary performance and their

Table 2. Results on instance from SMTLIB-IDL.

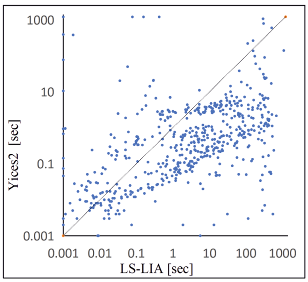
Family	Type	#inst	MathSAT	CVC5	Yices2	Z3	LS-LIA
Without Boolean	20210312-Bouvier	100	4	**44**	21	42	40
	job_shop	108	39	59	74	73	**77**
	n_queen	97	57	86	**97**	92	**97**
	toroidal_bench	32	11	10	12	12	**13**
	super_queen	91	57	86	**91**	**91**	**91**
	DTP	32	32	32	32	32	32
	schedulingIDL	247	100	125	**247**	**247**	**247**
	Total	707	300	442	574	589	**597**
With Boolean	asp	379	147	212	284	**291**	27
	Averest	157	**157**	**157**	**157**	**157**	120
	bcnscheduling	6	3	**4**	**4**	**4**	**4**
	fuzzy-matrix	15	0	0	0	0	**1**
	mathsat	16	**16**	**16**	**16**	**16**	11
	parity	136	130	**136**	**136**	**136**	**136**
	planning	2	**2**	**2**	**2**	**2**	0
	qlock	36	**36**	**36**	**36**	**36**	0
	RTCL	4	4	4	4	4	4
	sal	10	**10**	**10**	**10**	**10**	8
	sep	9	**9**	**9**	**9**	**9**	8
	Total	770	514	586	658	**665**	319



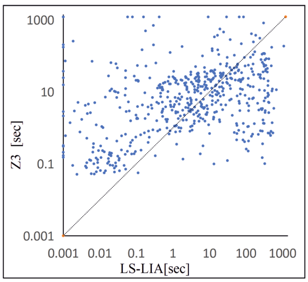
(a) Comparing with MathSAT5



(b) Comparing with CVC5



(c) Comparing with Yices2



(d) Comparing with Z3

Fig. 3. Run time comparison on Without Boolean category of SMTLIB-IDL

Table 3. Summary results on SMTLIB-LIA and SMTLIB-IDL. Instances without and with Boolean variables are denoted by "no_bool" and "with_bool" respectively.

	#inst	MathSAT5	CVC5	Yices2	Z3	LS-LIA	Z3+LS
LIA_no_bool	2385	2242	2041	1774	2165	**2294**	2316
LIA_with_bool	1842	1619	766	**1662**	1617	912	1625
Total	4227	**3861**	2807	3436	3782	3206	3941
IDL_no_bool	707	300	442	574	589	**597**	597
IDL_with_bool	770	514	586	658	**665**	319	661
Total	1477	814	1028	1232	**1254**	916	1258

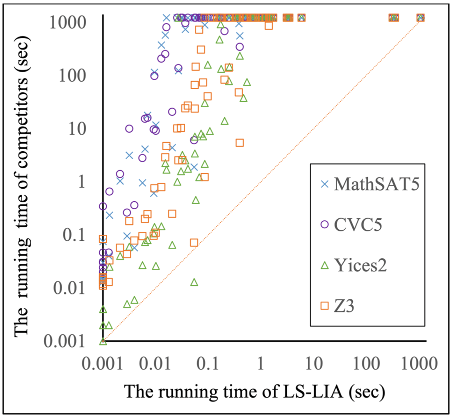


Fig. 4. Run time comparison on job shop scheduling instances.

combination pushes the state of the art in solving satisfiable instances of SMT (LIA). We also combined LS-LIA with Yices in the same manner, resulting in a wrapped solver called YicesLS [11], which won the Single-Query and Model-Validation Track on QF_IDL in SMT-COMP 2021.

7.3 Results on Job Shop Scheduling Benchmark

LS-LIA significantly outperforms the competitors on the JSP benchmark. LS-LIA solves 74 instances, while MathSAT5, CVC5, Yices2, Z3 can only solve 27, 29, 49, 44 instances respectively. The run time comparison on the JSP benchmark are presented in Fig. 4, where the instances that both the competitors and LS-LIA cannot solve are excluded. LS-LIA shows dominating advantage over it competitors on these JSP instances.

7.4 Results on RVPredict Benchmark

Table 4 presents the results on satisfiable instances generated by running RVPredict [29] on Dacapo benchmark suite [5]. LS-LIA solves all the instances

Table 4. The results on RVPredict instances, "#var" and "#clause" denotes the number of variables and clauses respectively. If a solver finds an satisfying assignment, the run time to find the assignment is reported, otherwise 'NA' is reported. For LS-LIA, we report the median (minimum, maximum) run time.

	#var	clause	MathSAT5	CVC5	Yices2	Z3	LS-LIA
RVPredict_1	19782	38262	344.8	410.2	6.3	NA	67.6(56.7,139.4)
RVPredict_2	19782	38262	427.0	429.7	3.3	NA	77.3 (54.2, 107.2)
RVPredict_3	19782	38258	329.5	378.2	9.9	NA	57.8 (56.5, 116.7)
RVPredict_4	19782	38263	333.3	403.5	3.9	NA	80.7 (58.1, 130.5)
RVPredict_5	19782	38262	346.3	412.7	5.8	NA	78.2 (52.3, 124.4)
RVPredict_6	19782	38258	457.2	332.7	2.5	NA	61.1 (43.4, 151.4)
RVPredict_7	19782	38262	541.0	382.7	11.1	NA	68.3 (44.7, 100.6)
RVPredict_8	19782	38259	357.0	405.0	6.9	NA	72.8 (54.5, 131.2)
RVPredict_9	19782	38262	431.3	443.7	12.8	NA	73.2 (41.8, 122.5)
RVPredict_10	19782	38246	460.4	280.7	4.6	NA	56.7 (43.6, 137.3)
RVPredict_11	139	174	0.1	0.1	0.1	0.1	0.1 (0.1, 0.1)
RVPredict_12	460	6309	4.7	5.6	0.1	0.3	1.3 (0.4, 4.5)
RVPredict_13	460	6503	4.1	6.1	0.1	0.3	0.1 (0.1, 0.1)
RVPredict_14	460	6313	4.3	5.8	0.1	0.3	0.7 (0.1, 1.5)
RVPredict_15	460	6313	5.5	5.8	0.1	0.3	0.8 (0.5, 1.7)

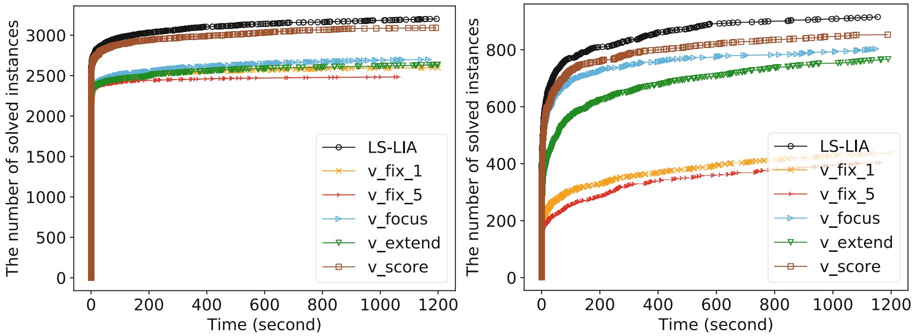
consistently, and ranks second on this benchmark, only slower than Yices2. Particularly, on the 10 large instances RVPredict_1-10, LS-LIA is much faster than competitors except Yices2.

7.5 Effectiveness of Proposed Strategies

To analyze the effectiveness of the strategies in LS-LIA, we modify LS-LIA to obtain 5 alternative versions as follows.

- To analyze the effectiveness of the *cm* operator, we modify LS-LIA by replacing the *cm* operator with the operator that directly modifies an integer variable by a fixed increment *inc*, leading to two versions v_fix_1 and v_fix_5, where *inc* is set as 1 and 5 respectively.
- To analyze the effectiveness of the two level heuristic for picking a decreasing *cm* operation, we modify LS-LIA by choosing a decreasing *cm* operation only from falsified clauses or directly from all false literals, leading to two versions, namely v_focused and v_extend.
- To analyze the effectiveness of *dscore*, we modify LS-LIA to choose a *cm* operation with the highest *score* from the selected clause at local optima, leading to the version v_score.

We compare LS-LIA with these modified version on the SMTLIB-LIA and SMTLIB-IDL benchmarks. The runtime distribution of LS-LIA and its modified versions on the two benchmarks are presented in Fig. 5, confirming the effectiveness of the strategies.



(a) Comparison on SMTLIB-LIA (b) Comparison on SMTLIB-IDL

Fig. 5. Run time distribution comparison

8 Conclusion and Future Work

We developed the first local search solver for SMT (LIA) and SMT (IDL), opening the local search direction for SMT on integer theories. Main features of our solver include a framework switching between Boolean and Integer modes, the critical move operator and a scoring function based on distance to satisfaction. Experiments show that our solver is competitive and complementary to state-of-the-art SMT solvers.

We would like to enhance our solver by improving the performance on instances with Boolean variables. Also, it is interesting to explore deep cooperation with DPLL(T) solvers.

Acknowledgements. This work is supported by NSFC Grant 62122078. We thank the reviewers of CAV 2022 for comments on improving the quality of the paper.

References

1. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 16–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_3
2. Barrett, C., Barbosa, H., Brain, M., et al.: Cvc5 at the SMT competition 2021 (2021)
3. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
4. Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016. Proc. SAT Competition **2016**, 44–45 (2016)
5. Blackburn, S.M., Garner, R., Hoffmann, C., et al.: The DaCapo benchmarks: Java benchmarking development and analysis, pp. 169–190 (2006)

6. Bromberger, M.: Decision procedures for linear arithmetic. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2019)
7. Bromberger, M., Fleury, M., Schwarz, S., Weidenbach, C.: SPASS-SATT. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 111–122. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_7
8. Bromberger, M., Weidenbach, C.: Fast cube tests for LIA constraint solving. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 116–132. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_9
9. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_7
10. Cai, S.: Balance between complexity and quality: local search for minimum vertex cover in massive graphs. In: Proceedings of IJCAI 2015, pp. 747–753 (2015)
11. Cai, S., Li, B., Zhang, X.: YicesLS on SMT COMP2021 (2021)
12. Cai, S., Luo, C., Su, K.: CCAnr: a configuration checking based local search solver for non-random satisfiability. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 1–8. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_1
13. Cai, S., Su, K.: Local search for Boolean satisfiability with configuration checking and subscore. Artif. Intell. **204**, 75–98 (2013)
14. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
15. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
16. Codish, M., Fekete, Y., Fuhs, C., Giesl, J., Waldmann, J.: Exotic semi-ring constraints. In: SMT@ IJCAR 20, pp. 88–97 (2012)
17. Cotton, S., Podelski, A., Finkbeiner, B.: Satisfiability checking with difference constraints. IMPRS Computer Science, Saarbruceken (2005)
18. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
19. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
20. Dutertre, B., De Moura, L.: Integrating simplex with DPLL (T). Computer Science Laboratory, SRI International, Technical Report SRI-CSL-06-01 (2006)
21. Dutertre, B., De Moura, L.: The YICES SMT solver, vol. 2, no. 2, pp. 1–2 (2006). Tool paper at http://yices.csl.sri.com/tool-paper.pdf
22. Fröhlich, A., Biere, A., Wintersteiger, C., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. In: Proceedings of AAAI 2015, vol. 29 (2015)
23. Ganai, M.K., Talupur, M., Gupta, A.: *SDSAT*: tight integration of *small domain encoding* and *lazy* approaches in a separation logic solver. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 135–150. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_9
24. Gavrilenko, N., Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 355–365. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_19

25. Glover, F., Laguna, M.: Tabu search. In: Du, D.Z., Pardalos, P.M. (eds.) Handbook of Combinatorial Optimization. Springer, Boston (1998). https://doi.org/10.1007/978-1-4613-0303-9_33
26. Griggio, A., Phan, Q.-S., Sebastiani, R., Tomasi, S.: Stochastic local search for SMT: combining theory solvers with walkSAT. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 163–178. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24364-6_12
27. Hentenryck, P.V., Michel, L.: Constraint-based local search (2009)
28. Hoos, H.H., Stützle, T.: Stochastic local search: foundations and applications (2004)
29. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. In: Proceedings of PLDI 2014, pp. 337–348 (2014)
30. Janhunen, T., Niemelä, I., Sevalnev, M.: Computing stable models via reductions to difference logic. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS (LNAI), vol. 5753, pp. 142–154. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04238-6_14
31. Kim, H., Jin, H., Somenzi, F.: Disequality management in integer difference logic via finite instantiations. JSAT **3**(1–2), 47–66 (2007)
32. King, T., Barrett, C., Dutertre, B.: Simplex with sum of infeasibilities for SMT. In: Proceedings of FMCAD 2013, pp. 189–196 (2013)
33. Li, C.M., Li, Y.: Satisfying versus falsifying in local search for satisfiability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 477–478. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_43
34. Lopes, N.P., Monteiro, J.: Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. Int. J. Softw. Tools Technol. Transfer **18**(4), 359–374 (2015). https://doi.org/10.1007/s10009-015-0366-1
35. McCarthy, J.: Towards a mathematical science of computation. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds) Program Verification. Studies in Cognitive Systems, vol 14. Springer, Dordrecht. https://doi.org/10.1007/978-94-011-1793-7_2
36. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 199–217. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_11
37. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_33
38. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
39. Pnueli, A., Rodeh, Y., Strichman, O., Siegel, M.: The small model property: how small can it be? Inf. Comput. **178**(1), 279–293 (2002)
40. Roselli, S.F., Bengtsson, K., Åkesson, K.: SMT solvers for job-shop scheduling problems: models comparison and performance evaluation. In: Proceedings of CASE 2018, pp. 547–552 (2018)
41. Sebastiani, R.: Lazy satisfiability modulo theories. JSAT **3**(3–4), 141–224 (2007)
42. Selman, B., Kautz, H.A., Cohen, B., et al.: Local search strategies for satisfiability testing. In: Cliques, Coloring, and Satisfiability, vol. 26, pp. 521–532 (1993)
43. Seshia, S.A., Lahiri, S.K., Bryant, R.E.: A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In: Proceedings of DAC 2003, pp. 425–430 (2003)

44. Strichman, O., Seshia, S.A., Bryant, R.E.: Deciding separation formulas with SAT. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 209–222. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_16
45. Talupur, M., Sinha, N., Strichman, O., Pnueli, A.: Range allocation for separation logic. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 148–161. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_12
46. Thornton, J., Pham, D.N., Bain, S., Ferreira, V., Jr.: Additive versus multiplicative clause weighting for SAT. In: Proceedings of AAAI 2004, pp. 191–196 (2004)
47. Wang, C., Gupta, A., Ganai, M.: Predicate learning and selective theory deduction for a difference logic solver. In: Proceedings of DAC 2006, pp. 235–240 (2006)