A Fast Linear-Arithmetic Solver for DPLL(T)*

Bruno Dutertre and Leonardo de Moura

Computer Science Laboratory, SRI International,
333 Ravenswood Avenue, Menlo Park, CA 94025, USA
{bruno, demoura}@csl.sri.com

Abstract. We present a new Simplex-based linear arithmetic solver that can be integrated efficiently in the DPLL(T) framework. The new solver improves over existing approaches by enabling fast backtracking, supporting a priori simplification to reduce the problem size, and providing an efficient form of theory propagation. We also present a new and simple approach for solving strict inequalities. Experimental results show substantial performance improvements over existing tools that use other Simplex-based solvers in DPLL(T) decision procedures. The new solver is even competitive with state-of-the-art tools specialized for the difference logic fragment.

1 Introduction

Decision procedures for quantifier-free linear arithmetic determine whether a boolean combination of linear equalities, inequalities, and disequalities is satisfiable. Several tools for solving this problem rely on the DPLL(T) approach [1]: they combine boolean satisfiability solvers based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure, and arithmetic solvers capable of deciding the satisfiability of conjunctions of linear constraints. Results of a first satisfiability modulo theories (SMT) competition, comparing several of these tools, are presented in [2]. Several tools (e.g., Barcelogic [21] or Slice [20]) are specialized for the *difference-logic* fragment of linear arithmetic and rely on graph algorithms. For general linear arithmetic, existing tools rely either on Fourier-Motzkin elimination [3] (used by CVClite [4], CVC [5], SVC [6]) or on Simplex methods [7] (used by MathSat [8], ICS [9], Simplics, Yices, ARIO [10]). Fourier-Motzkin elimination explodes on many problems and Simplex is generally superior.

The common methods for integrating a Simplex solver with DPLL rely on incremental versions of Simplex such as described in [11,12,13,14]. A tableau is constructed and updated incrementally: rows are added as DPLL proceeds and are later removed when DPLL backtracks. These frequent addition and removal of rows and the related bookkeeping have a significant cost. For example, backtracking may require pivoting operations. This paper presents a simpler and more efficient solver that considerably reduces this overhead. The approach relies on transforming the original formula Φ into

* This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or the Department of Interior National Business Center (DOI-NBC).

T. Ball and R.B. Jones (Eds.): CAV 2006, LNCS 4144, pp. 81–94, 2006.

an equisatisfiable Φ' such that the satisfiability of Φ' is decided by solving a series of problems of the form

$$\text{find } x \in \mathbb{R}^n \text{ such that } Ax = 0 \text{ and } l_i \leq x_i \leq u_i \text{ for } i = 1, \ldots, n,$$

where the matrix A is fixed and l_i and u_i are bounds on x_i that may vary with each problem. Variants of Simplex can efficiently solve problems in this form. Section 4 presents such a variant designed to be efficient in the DPPL(T) context, and Section 5 shows how to extend it to problems with strict inequalities. Since A is fixed, no row is ever added or removed from the tableau, and backtracking is very cheap. The new solver has additional advantages: it is possible to simplify the problem a priori by eliminating irrelevant variables, and a simple but useful form of theory propagation can be implemented cheaply.

2 Background

Given a quantifier-free theory T, a *T-solver* is a procedure for deciding whether a finite set of atoms of T is satisfiable. If Φ is a formula built by boolean combination of atoms of T, then the satisfiability of Φ can be decided by combining a boolean satisfiability solver and a T-solver. The DPLL(T) approach is an efficient method for such integrations that relies on the DPLL procedure.

2.1 Solvers for DPPL(T)

In the DPLL(T) framework, a T-solver maintains a state that is an internal representation of the atoms asserted so far. This solver must provide operations for updating the state by asserting new atoms, checking whether the state is consistent, and backtracking. Optionally, the solver may also implement *theory propagation*, that is, identify atoms that are implied by the current state. To interact with the DPLL search, the solver must produce *explanations* for conflicts and propagated atoms. In an inconsistent state S, an explanation is any inconsistent subset of the atoms asserted in S. Similarly, an explanation for an implied atom γ is a subset Γ of the asserted atoms such that $\Gamma \models \gamma$. An explanation Γ is *minimal* if no proper subset of Γ is an explanation.

The solver is assumed initialized for a fixed formula Φ and we denote by $\mathcal{A}$ the set of atoms that occur in Φ. The set of atoms asserted so far is denoted by α. The solver also maintains a stack of *checkpoints* that mark consistent states to which the solver can backtrack. We assume that a T-solver implements the following API.[1]

- *Assert*(γ) asserts atom γ in the current state. It returns either ok or unsat$\langle\Gamma\rangle$ where Γ is a subset of α. In the first case, γ is inserted into α. In the latter case, $\alpha \cup \{\gamma\}$ is inconsistent and Γ is the explanation.
- *Check*() checks whether α is consistent. If so, it returns ok, otherwise it returns unsat$\langle\Gamma\rangle$. As previously $\Gamma \subseteq \alpha$ is an explanation for the inconsistency. A new checkpoint is created when ok is returned.

[1] This is similar to the API proposed in [1].

- *Backtrack*() backtracks to the consistent state represented by the checkpoint on the top of the stack.
- *Propagate*() performs theory propagation. It returns a set $\{\langle \Gamma_1, \gamma_1 \rangle, \ldots, \langle \Gamma_t, \gamma_t \rangle\}$ where $\Gamma_i \subseteq \alpha$ and $\gamma_i \in \mathcal{A} \setminus \alpha$. For every pair $\langle \Gamma_i, \gamma_i \rangle$ produced, γ_i is an atom not already asserted that is implied by Γ_i, and Γ_i is a subset of α.

Assert must be sound but is not required to be complete: *Assert*(γ) may return ok even if $\alpha \cup \{\gamma\}$ is inconsistent. Similarly, *Propagate* must be sound but does not have to be exhaustive. On the other hand, function *Check* is required to be sound and complete: if *Check*() $=$ ok then α must be consistent. This model enables several atoms to be asserted in a single "batch", using several calls to *Assert* followed by a single call to *Check*. *Assert* can then implement only inexpensive (and possibly incomplete) consistency checks while *Check* implements a complete (and possibly expensive) consistency-checking procedure. The state S' after executing *Backtrack* must be logically equivalent to the state S when the checkpoint was created, but S' may be different from S.

2.2 Existing Simplex Solvers for DPLL(T)

A quantifier-free linear arithmetic formula is a first-order formula whose atoms are either propositional variables of equalities, disequalities, or inequalities of the form

$$a_1 x_1 + \ldots + a_n x_n \bowtie b,$$

where $a_1, \ldots, a_n$ and b are rational numbers, $x_1, \ldots, x_n$ are real (or integer) variables, and $\bowtie$ is one of the operators $=$, $\leq$, $<$, $>$, $\geq$, or $\neq$. In the DPLL(T) framework, deciding the satisfiability of such formulas requires a linear-arithmetic solver. A common approach is to use incremental forms of Simplex similar to the algorithms described in [11,12,13,14]. Tools based on this approach include our own tools, Yices and Simplics, and others such as MathSat [8].

In these algorithms, a solver state includes a Simplex tableau that is derived from all equalities and inequalities asserted so far. A tableau can be written as a set of equalities of the form

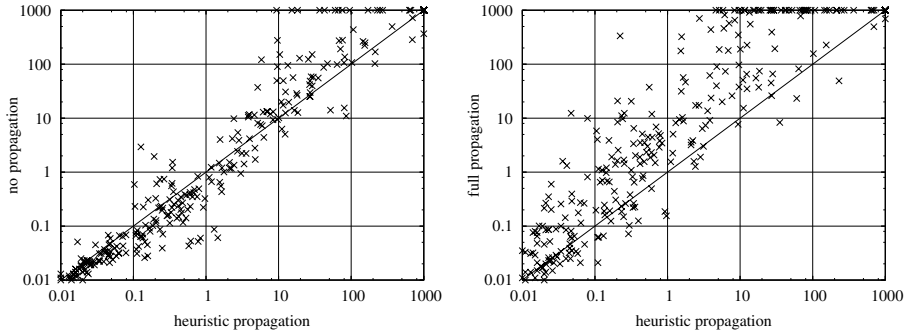
$$x_i = b_i + \sum_{x_j \in \mathcal{N}} a_{ij} x_j, \quad x_i \in \mathcal{B} \tag{1}$$

where $\mathcal{B}$ and $\mathcal{N}$ are disjoint sets of variables. Elements of $\mathcal{B}$ and $\mathcal{N}$ are called *basic* and *nonbasic* variables, respectively. Additional constraints are imposed on some variables of $\mathcal{B} \cup \mathcal{N}$. So-called *slack variables* are required to be non-negative, and the tableau may also contain *zero variables*, which are all implicitly equal to 0. Zero variables are used to generate explanations (cf. [11]).

A pivoting operation *pivot*(x_r, x_s) swaps a basic variable x_r and a nonbasic variable x_s such that $a_{rs} \neq 0$. After pivoting, x_s becomes basic and x_r becomes nonbasic. The tableau is updated by replacing equation $x_r = b_r + \sum_{x_j \in \mathcal{N}} a_{rj} x_j$ with

$$x_s = -\frac{b_r}{a_{rs}} + \frac{x_r}{a_{rs}} - \sum_{x_j \in \mathcal{N} \setminus \{x_s\}} \frac{a_{rj} x_j}{a_{rs}} \tag{2}$$

and then equation (2) is used to eliminate x_s from the rest of the tableau by substitution.



(a)Heuristic vs. No Propagation (b)Heuristic vs. Full Propagation

Fig. 1. Impact of theory propagation in Simplics

Assertion of equalities or inequalities adds new equations to the tableau. For example, let γ be an atom of the form $t \geq 0$ where t is an arithmetic term. The operation *Assert*(γ) involves three steps. First, γ is normalized by substituting any basic variable x_i occurring in t with the term $b_i + \sum_{x_j \in \mathcal{N}} a_{ij} x_j$. The solver checks then whether the resulting inequality $t' \geq 0$ is satisfiable. This step uses the Simplex algorithm to maximize t' subject to the tableau constraints. If t' has a maximum M and M is negative, then $t' \geq 0$ is not satisfiable and an explanation is generated. Otherwise, a fresh slack variable s_k is created and a row of the form $s_k = t''$ is added to the tableau. Some bookkeeping is required to record that s_k is nonnegative and is associated with atom γ. Processing of equalities and strict inequalities follows the same general principles. Backtracking removes rows from the tableau. For example, to retract γ, the solver retrieves the slack variable s_k associated with γ. If s_k is a basic variable in the current state then the corresponding equation is removed from the tableau. Otherwise, a pivoting operation is applied first to make s_k basic.

Disequalities are treated separately since they cannot be incorporated into the tableau. When a disequality $t \neq 0$ is asserted, it is first normalized as before, and then the solver must check whether the current tableau implies $t = 0$. This can be implemented via the *zero-detection procedure* described in [11] for example.

2.3 Performance

Assertions and backtracking have a significant cost in solvers based on incremental Simplex algorithms. Part of this cost (e.g., the pivoting involved in *Assert* operations) cannot be avoided, but there is also significant overhead in the frequent additions and removals of rows, creations and deletions of slack variables, and associated bookkeeping. The remainder of the paper describes a different type of solver, still based on the Simplex method, which significantly reduces this overhead. The new approach is simpler and more uniform than incremental Simplex. It is also more economical as irrelevant variables can be eliminated a priori and fewer slack variables are necessary.

Some of the simplifications are based on lessons we learned from experiments with our previous tools Simplics and Yices:[2]

[2] Both use incremental Simplex and zero detection.

- *Minimal explanations are critical*. Dramatic improvements were observed when comparing Simplics and Yices, which generate minimal explanations, and their predecessor ICS, which does not.
- *Theory propagation is useful if it can be done cheaply*. Figure 1 compares the results of Simplics on the real-arithmetic subset of the SMT-LIB benchmarks [15] using different levels of theory propagation. By default, Simplics uses a heuristic form of propagation that is relatively inexpensive but incomplete (no pivoting is used). This is compared in Figure 1(a) with Simplics running with no propagation at all, and in Figure 1(b) with Simplics running with complete propagation (where pivoting is used). On these benchmarks, full propagation is just too expensive, but no propagation is also a poor choice. Heuristic propagation is clearly superior.
- *Zero detection is expensive and can be avoided*. On a few examples in the SMT-LIB benchmarks, Simplics spends as much as 30% of its time in the zero-detection procedure. A simpler alternative is to rewrite a disequality $t \neq 0$ as the disjunction of two strict inequalities $(t < 0) \vee (t > 0)$. This transformation may seem wasteful since it may entail additional case splits, but it works well in practice. After this transformation, Simplics can solve six problems of the SMT-LIB benchmarks that it cannot solve otherwise.

3 Preprocessing

Incremental Simplex algorithms can be avoided by rewriting a linear arithmetic formula Φ into an equisatisfiable formula of the form $\Phi_A \wedge \Phi'$, where Φ_A is a conjunction of linear equalities, and all the atoms occurring in Φ' are *elementary atoms* of the form $y \bowtie b$, where y is a variable and b is a rational constant. The transformation is straightforward. For example, let Φ be the formula

$$x \geq 0 \wedge (x + y \leq 2 \vee x + 2y - z \geq 6) \wedge (x + y = 2 \vee x + 2y - z > 4).$$

We introduce two variables s_1 and s_2 and rewrite Φ to $\Phi_A \wedge \Phi'$ as follows.

$$\begin{gathered}(s_1 = x + y \wedge s_2 = x + 2y - z) \ \wedge \\ (x \geq 0 \wedge (s_1 \leq 2 \vee s_2 \geq 6) \wedge (s_1 = 2 \vee s_2 > 4))\end{gathered}$$

Clearly, this new formula and Φ are equisatisfiable. In general, starting from a formula Φ, the transformation introduces a new variable s_i for every linear term t_i that is not already a variable and occurs as the left side of an atom $t_i \bowtie b$ of Φ. Then Φ_A is the conjunction of all the equalities $s_i = t_i$ and Φ' is obtained by replacing every term t_i by the corresponding s_i in Φ.

Let $x_1, \ldots, x_n$ be the arithmetic variables of $\Phi_A \wedge \Phi'$, that is, all the variables originally in Φ and m-additional variables $s_1, \ldots, s_m$ introduced by the previous transformation ($m \leq n$). Then formula Φ_A can be written in matrix form as $Ax = 0$, where A is a fixed $m \times n$ rational matrix and x is a vector in $\mathbb{R}^n$. The rows of A are linearly independent so A has rank m. Checking whether Φ is satisfiable amounts to finding an x such that $Ax = 0$ and x satisfies Φ'. In other words, checking the satisfiability of Φ in linear arithmetic is equivalent to checking the satisfiability of Φ' in *linear arithmetic*

modulo $Ax = 0$. Since all atoms of Φ' are elementary, this requires a solver for deciding the consistency of a set of elementary atoms Γ modulo the constraints $Ax = 0$. If Γ contains only equalities and (nonstrict) inequalities, this reduces to searching for $x \in \mathbb{R}^n$ such that

$$Ax = 0 \text{ and } l_j \leq x_j \leq u_j \quad \text{for } j = 1, \ldots, n \tag{3}$$

where l_j is either $-\infty$ or a rational number, and u_j is either $+\infty$ or a rational number.

Since the elementary atoms of Φ' are known in advance, we can immediately simplify the constraints $Ax = 0$ by removing any variable x_i that does not occur in any elementary atom of Φ'. This is done by Gaussian elimination. In practice, this presimplification can reduce the matrix size significantly (cf. [16]).

The variables s_i introduced during the transformation play the same role as the slack variables of standard Simplex. However, the presence of both lower and upper bounds is beneficial. For example, incremental Simplex algorithms need two slack variables to represent a constraint such as $1 \leq x + 3y \leq 4$, whereas a single s_k is sufficient if the general form (3) is used. Overall, rewriting Φ into $\Phi_A \wedge \Phi'$ and relying on the general form leads to problems with fewer variables than the algorithms discussed previously.

4 Basic Solver

We first describe a basic solver that handles equalities and nonstrict inequalities with real variables. Extensions to strict inequalities and integer variables are presented in the next sections. The basic solver decides the satisfiability of problems in form (3) and implements the API of Section 2.1 for integration with a DPLL-based SAT solver.

The solver state includes a tableau derived from the constraint matrix A. We will write such a tableau in the form:

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \quad x_i \in \mathcal{B},$$

where $\mathcal{B}$ and $\mathcal{N}$ denote the set of basic and nonbasic variables, respectively.[3] Since all rows of this tableau are linear combinations of rows of the original matrix A, the equality $x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j$ is satisfied by any x such that $Ax = 0$.

In addition to this tableau, the solver state stores upper and lower bounds l_i and u_i for every variable x_i and a mapping β that assigns a rational value $\beta(x_i)$ to every variable x_i. The bounds on nonbasic variables are always satisfied by β, that is, the following invariant is maintained

$$\forall x_j \in \mathcal{N}, \;\; l_j \leq \beta(x_j) \leq u_j. \tag{4}$$

Furthermore, β satisfies the constraint $Ax = 0$. In the initial state, $l_j = -\infty$, $u_j = +\infty$, and $\beta(x_j) = 0$ for all j.

Figure 2 describes two auxiliary procedures that modify β. Procedure *update*(x_i, v) sets the value of a nonbasic variable x_i to v and adjusts the value of all basic variables so that all equations remain satisfied. Procedure *pivotAndUpdate*(x_i, x_j, v) applies pivoting to the basic variable x_i and the nonbasic variable x_j; it also sets the value of x_i to v and adjusts the values of all basic variables to keep all equations satisfied.

[3] This is the same as (1) with $b_i = 0$ for all $x_i \in \mathcal{B}$.

procedure update(x_i, v)
  for each $x_j \in \mathcal{B}$, $\beta(x_j) := \beta(x_j) + a_{ji}(v - \beta(x_i))$
  $\beta(x_i) := v$

procedure pivotAndUpdate(x_i, x_j, v)
  $\theta := \frac{v-\beta(x_i)}{a_{ij}}$
  $\beta(x_i) := v$
  $\beta(x_j) := \beta(x_j) + \theta$
  for each $x_k \in \mathcal{B} \setminus \{x_i\}$, $\beta(x_k) := \beta(x_k) + a_{kj}\theta$
  pivot(x_i, x_j)

Fig. 2. Auxiliary procedures

4.1 Main Algorithm

The main procedure of our algorithm is based on the dual Simplex and relies on Bland's pivot-selection rule to ensure termination. It relies on a total order on the variables. Assuming an assignment β that satisfies the previous invariants, but where $l_i \leq \beta(x_i) \leq u_i$ may not hold for some basic variables x_i, procedure *Check* searches for a new β that satisfies all constraints. The procedure is shown in Figure 3. It either terminates with a new assignment and basis that satisfy all lower and upper bounds (line 4), or finds the constraints to be unsatisfiable (lines 8 and 13). The body of the main loop selects a basic variable x_i that does not satisfy its bounds (line 3). If x_i is below l_i, then it looks for a variable x_j in the row $x_i = \sum_{x_j \in \mathcal{N}} a_{ij}x_j$ that can compensate the gap in x_i (lines 6-7). If no such x_j exists the problem is unsatisfiable (line 8) because the value of x_i is maximal and is below the lower bound l_i. Otherwise, the procedure pivots x_i and x_j, and x_i is set to l_i (line 9). The case where x_i is above its upper bound (lines 10-14) is symmetrical.

The following property implies the correctness of *Check*; a proof is given in [16].

Theorem 1. *Procedure Check always terminates.*

4.2 Generating Explanations

An inconsistency may be detected by *Check* at line 8 or 13. Let us assume a conflict is detected at line 8. There is then a basic variable x_i such that $\beta(x_i) < l_i$ and for every nonbasic variable x_j we have $a_{ij} > 0 \Rightarrow \beta(x_j) \geq u_j$ and $a_{ij} < 0 \Rightarrow \beta(x_j) \leq l_j$. Let $\mathcal{N}^+ = \{x_j \in \mathcal{N} \mid a_{ij} > 0\}$ and $\mathcal{N}^- = \{x_j \in \mathcal{N} \mid a_{ij} < 0\}$. Since β satisfies all bounds on nonbasic variables, we have $\beta(x_j) = l_j$ for every $x_j \in \mathcal{N}^-$ and $\beta(x_j) = u_j$ for every $x_j \in \mathcal{N}^+$. It follows that

$$\beta(x_i) = \sum_{x_j \in \mathcal{N}} a_{ij}\beta(x_j) \;\; = \sum_{x_j \in \mathcal{N}^+} a_{ij}u_j + \sum_{x_j \in \mathcal{N}^-} a_{ij}l_j.$$

The equation $x_i = \sum_{x_j \in \mathcal{N}} a_{ij}x_j$ holds for any x such that $Ax = 0$. Therefore, for any such x, we have

$$\beta(x_i) - x_i = \sum_{x_j \in \mathcal{N}^+} a_{ij}(u_j - x_j) + \sum_{x_j \in \mathcal{N}^-} a_{ij}(l_j - x_j),$$

1. **procedure** Check()
2. **loop**
3. select the smallest basic variable x_i such that $\beta(x_i) < l_i$ or $\beta(x_i) > u_i$
4. **if** there is no such x_i **then return** *satisfiable*
5. **if** $\beta(x_i) < l_i$ **then**
6. select the smallest nonbasic variable x_j such that
7. $(a_{ij} > 0$ and $\beta(x_j) < u_j)$ or $(a_{ij} < 0$ and $\beta(x_j) > l_j)$
8. **if** there is no such x_j **then return** *unsatisfiable*
9. pivotAndUpdate(x_i, x_j, l_i)
10. **if** $\beta(x_i) > u_i$ **then**
11. select the smallest nonbasic variable x_j such that
12. $(a_{ij} < 0$ and $\beta(x_j) < u_j)$ or $(a_{ij} > 0$ and $\beta(x_j) > l_j)$
13. **if** there is no such x_j **then return** *unsatisfiable*
14. pivotAndUpdate(x_i, x_j, u_i)
15. **end loop**

Fig. 3. Check procedure

from which one can derive the following implication:

$$\bigwedge_{x_j \in \mathcal{N}^+} x_j \leq u_j \;\wedge\; \bigwedge_{x_j \in \mathcal{N}^-} l_j \leq x_j \;\Rightarrow\; x_i \leq \beta(x_i).$$

Since $\beta(x_i) < l_i$, this is inconsistent with $l_i \leq x_i$. The explanation for the conflict is then the following set of elementary atoms:

$$\Gamma = \{x_j \leq u_j \mid j \in \mathcal{N}^+\} \cup \{x_j \geq l_j \mid j \in \mathcal{N}^-\} \cup \{x_i \geq l_i\}.$$

It is easy to see that Γ is minimal. Explanations for conflicts at line 13 are generated in the same way.

4.3 Assertion Procedures

The *Assert* function relies on two procedures shown in Figure 4 for updating the bounds l_i and u_i. Procedure *AssertUpper*$(x_i \leq c_i)$ has no effect if $u_i \leq c_i$ and returns unsatisfiable if $c_i < l_i$; otherwise the current upper bound on x_i is set to c_i. If variable x_i is nonbasic, then β is updated to maintain invariant (4). If an immediate conflict is detected at line 3 then generating a minimal explanation is straightforward.

Procedure *AssertLower*$(x_i \geq c_i)$ does the same thing for the lower bound. An equality $x_i = c_i$ is asserted by calling both *AssertUpper* and *AssertLower*.

4.4 Backtracking

Efficient backtracking is important since the number of backtracks is often very large. In our approach, backtracking can be efficiently implemented. We just need to save the value of u_i (l_i) on a stack before it is updated by the procedure *AssertUpper* (*AssertLower*). This information is used to restore the old bounds when backtracking is performed. Backtracking does not require saving the successive βs on a stack. Only

1. **procedure** AssertUpper($x_i \leq c_i$)
2. **if** $c_i \geq u_i$ **then return** *satisfiable*
3. **if** $c_i < l_i$ **then return** *unsatisfiable*
4. $u_i := c_i$
5. **if** x_i is a nonbasic variable and $\beta(x_i) > c_i$ **then** update(x_i, c_i)
6. **return** *ok*

1. **procedure** AssertLower($x_i \geq c_i$)
2. **if** $c_i \leq l_i$ **then return** *satisfiable*
3. **if** $c_i > u_i$ **then return** *unsatisfiable*
4. $l_i := c_i$
5. **if** x_i is a nonbasic variable and $\beta(x_i) < c_i$ **then** update(x_i, c_i)
6. **return** *ok*

Fig. 4. Assertion procedures

one assignment β needs to be stored, namely, the one corresponding to the last successful *Check*. After a successful *Check*, the assignment β is a model for the current set of constraints and for the set of constraints asserted at any previous checkpoint. Since no pivoting or other expensive operation is used, backtracking is very cheap.

4.5 Theory Propagation

Given a set of elementary atoms $\mathcal{A}$ from the formula Φ', then *unate propagation* is very cheap to implement. For example, if bound $x_i \geq c_i$ has been asserted then any unassigned atom of $\mathcal{A}$ of the form $x_i \geq c'$ with $c' < c_i$ is immediately implied. Similarly, the negation of any atom $x_i \leq u$ with $u < c_i$ is implied. This type of propagation is useful in practice. It occurs frequently in several SMT-LIB benchmarks.

Another method is based on *bound refinement*. Given a row of a tableau, such as $x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j$, one can derive a lower or upper bound on x_i from the lower or upper bounds on the nonbasic variables x_j. These computed bounds may imply unassigned elementary atoms with variable x_i. This is a heuristic technique as the computed bounds may be weaker than the current bounds asserted on x_i (for example, the computed bounds may be $-\infty$ or $+\infty$). However, bound refinement is quite general. It is applicable with any equality $a_1 x_1 + \ldots + a_n x_n = 0$ derived by linear combination of rows of A, not just with rows of a tableau.

4.6 Example

Figure 5 illustrates the algorithm on a small example. Each row represents a state. The columns contain the tableaux, bounds, and assignments. The first row contains the initial state. Suppose $x \leq -4$ is asserted. Then the value of x must be adjusted, since $\beta_0(x) > -4$. Since s_1 and s_2 depend on x, their values are also modified. No pivoting is required since the basic variables do not have bounds, so $A_1 = A_0$. Next, $x \geq -8$ is asserted. Since $\beta_1(x)$ satisfies this bound, nothing changes: $A_2 = A_1$ and $\beta_2 = \beta_1$.

$A_0 = \begin{cases} s_1 = -x + y \\ s_2 = x + y \end{cases}$		$\beta_0 = (x \mapsto 0, y \mapsto 0, s_1 \mapsto 0, s_2 \mapsto 0)$
$A_1 = A_0$	$x \leq -4$	$\beta_1 = (x \mapsto -4, y \mapsto 0, s_1 \mapsto 4, s_2 \mapsto -4)$
$A_2 = A_1$	$-8 \leq x \leq -4$	$\beta_2 = \beta_1$
$A_3 = \begin{cases} y = x + s_1 \\ s_2 = 2x + s_1 \end{cases}$	$-8 \leq x \leq -4$ $s_1 \leq 1$	$\beta_3 = (x \mapsto -4, y \mapsto -3, s_1 \mapsto 1, s_2 \mapsto -7)$

Fig. 5. Example

Next, $s_1 \leq 1$ is asserted. The current value of s_1 does not satisfy this bound, so *Check* must be invoked. *Check* pivots s_1 and y to decrease s_1. The resulting state S_3 is shown in the last row; all constraints are satisfied.

If $s_2 \geq -3$ is asserted in S_3 and *Check* is called then an inconsistency is detected: Tableau A_2 does not allow s_2 to increase since both x and s_1 are at their upper bound. Therefore, $s_2 \geq -3$ is inconsistent with state S_3.

5 Strict Inequalities

The previous method generalizes to strict inequalities using a simple observation.

Lemma 1. *A set of linear arithmetic literals Γ containing strict inequalities $S = \{p_1 > 0, \ldots, p_n > 0\}$ is satisfiable iff there exists a rational number $\delta > 0$ such that $\Gamma_\delta = (\Gamma \cup S_\delta) \setminus S$ is satisfiable, where $S_\delta = \{p_1 \geq \delta, \ldots, p_n \geq \delta\}$.*

This lemma says that we can replace all strict inequalities by nonstrict ones if a small enough δ is known. Rather than computing an explicit value for δ, we treat it symbolically, as an *infinitesimal parameter*. Bounds and variable assignments now range over the set $\mathbb{Q}_\delta$ of pairs of rationals. A pair (c, k) of $\mathbb{Q}_\delta$ is denoted by $c + k\delta$ and the following operations and comparison are defined in $\mathbb{Q}_\delta$:

$$\begin{aligned} (c_1, k_1) + (c_1, k_2) &\equiv (c_1 + c_2, k_1 + k_2) \\ a \times (c, k) &\equiv (a \times c, a \times k) \\ (c_1, k_1) \leq (c_2, k_2) &\equiv (c_1 < c_2) \vee (c_1 = c_2 \wedge k_1 \leq k_2), \end{aligned}$$

where a is a rational. Strict bounds in $\mathbb{Q}$ are converted to nonstrict bounds in $\mathbb{Q}_\delta$: inequality $x_i > l_i$ is converted to $x_i \geq l_i + \delta$, and $x_i < u_i$ is converted to $x_i \leq u_i - \delta$. Then all updates to β used in the previous algorithm can be performed in $\mathbb{Q}_\delta$. The matrix A does not change; all its coefficients are rational numbers.

By this process, a problem $\mathcal{S}$ with strict bounds in the rational is converted into a problem S' in the general form (3) but where the bounds l_i and u_i, and the variables x_i are elements of $\mathbb{Q}_\delta$. If an assignment β' satisfies S' then it can be converted into a rational assignment β that satisfies S. This relies on substituting the symbolic parameter δ with a small enough positive rational number $\delta_0 \in \mathbb{Q}$, which can always be done since

there is a finite number of inequalities in S' (cf. [16]). If S' is unsatisfiable in $\mathbb{Q}_\delta$, then by Lemma 1, S is also unsatisfiable in the rationals.

6 Extensions

The previous solver is sound and complete for the reals. If some or all of the variables x_i are required to be integer, the algorithm is not complete. Nothing ensures that the assignment β constructed by *Check* gives an integer value to integer variables. To be complete in the integer or mixed integer case, we employ a *branch and cut* strategy, that is, the combination of branch-and-bound with a cutting plane generation algorithm [17,18]. The branch-and-bound algorithm works when problems are solved in $\mathbb{Q}_\delta$ rather than $\mathbb{Q}$. In other words, it can be used when strict inequalities are present. The cutting-plane method we use is based on mixed integer Gomory cuts. Such a cutting-plane algorithm is critical as it dramatically accelerates the convergence of branch-and-cut in several cases.

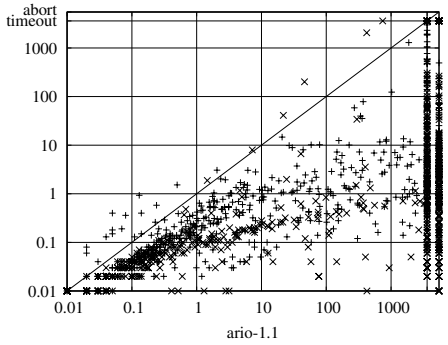
Also, it is possible to integrate the linear-arithmetic solver presented in this paper with solvers for other theories. The simplest method is to perform case-splits on equalities between variables that are shared between different theories. In most cases, the number of such shared variables is small in comparison with the total number of variables and this method is quite efficient. This approach is described in detail at [19]. It can be extended with an opportunistic equality-propagation method [16].

7 Experiments

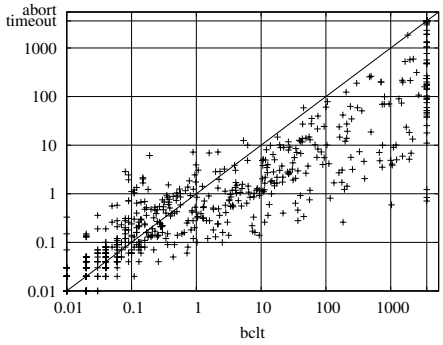
Figure 6 compares a prototype SMT solver that uses the previous algorithms with other tools that participated in last year's SMT competition. The comparison uses all the SMT-LIB benchmarks in the QF_RDL (real difference logic), QF_IDL (integer difference logic), QF_LRA (linear real arithmetic), and QF_LIA (linear integer arithmetic) divisions. The experiments were conducted on identical PCs, all equipped with a 32bit Pentium 4 processor running at 3 GHz. The timeout was set to 1 hour and the memory usage was limited to 1 GB. With these timing and memory constraints, running all the benchmarks required approximately 60 CPU days.

Each point on the graphs represents a benchmark: $+$ denotes a difference logic problem and $\times$ denotes a problem outside the difference-logic fragment. The axes correspond to the CPU time taken by the new solver (y-axis) or the other solver (x-axis) on each benchmark. CPU times are measured in seconds. Points below the diagonal are then SMT-LIB benchmarks where our new solver is faster. Points on the leftmost vertical edge are problems where a solver aborted, typically by running out of memory. The graphs comparing our new solver with Barcelogic and Simplics have fewer points, because Barcelogic supports only difference logic and Simplics does not support integer problems.

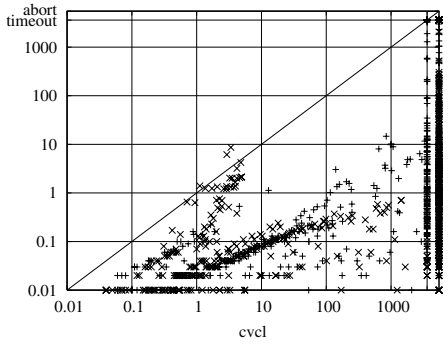
Table 1 summarizes the results. For each tool, it lists the number of instances solved and unsolved, and the total runtime. As can be seen, the new algorithm largely outperforms the other solvers. It is even faster on problems in the difference logic fragment



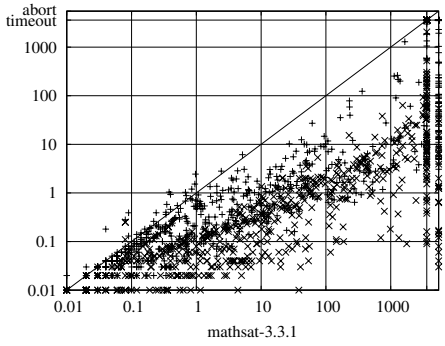
(a)Ario 1.1 vs. New Solver



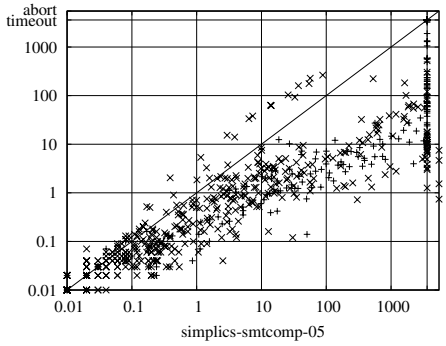
(b)BarcelogicTools vs. New Solver



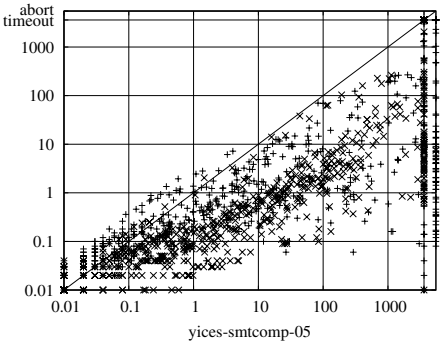
(c)CVC Lite 2.0 vs. New Solver



(d)MathSAT 3.3.1 vs. New Solver



(e)Simplics vs. New Solver



(f)Yices vs. New Solver

Fig. 6. Experimental results

than tools that are specialized for this fragment. The performance improvement is due to efficient backtracking and to the presimplification enabled by our approach, efficient theory propagation based on bound refinement also has a big impact.

Table 1. Experimental results: Summary

	sat	unsat	failed	time (secs)
Ario 1.1	186	640	517	1218371
BarcelogicTools	153	417	92	401842
CVC Lite	117	454	772	1193747
MathSAT 3.3.1	330	779	234	739533
Yices	358	756	229	702129
Simplics	240	351	110	476940
New Solver	**412**	**869**	**62**	**267198**

8 Conclusion

We have presented a new Simplex-based solver designed for efficiently solving SMT problems involving linear arithmetic. The main features of the new approach include the possibility to presimplify the input problem by eliminating variables, a reduction in the number of slack variables, and fast backtracking. A simple but useful form of theory propagation can also be implemented cheaply. Another result of the paper is a simple approach for solving strict inequalities that does not require modification of the basic Simplex algorithm. This approach is more generally applicable to other forms of solvers, such as graph-based solvers for difference logic.

Experimental results show that the new Simplex-based solver outperforms the most competitive solvers from SMT-COMP'05, including specialized solvers on difference logic problems.

Applications for the algorithm presented in this paper go beyond SMT. We are currently extending the solver to support a form of weighted MAX-SMT, that is, the search for an assignment to an SMT problem that maximizes a linear objective function. This MAX-SMT solver will be integrated to SRI's CALO system[4], as part of a module that combines learning and deductive algorithms.

References

1. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In Alur, R., Peled, D., eds.: Int. Conference on Computer Aided Verification (CAV'04). Volume 3114 of LNCS., Springer (2004) 175–188
2. Barrett, C., de Moura, L., Stump, A.: Design and Results of the 1st Satisfiability Modulo Theories Competition (SMT-COMP 2005). To appear in Journal of Automated Reasoning (2006)
3. Dantzig, G., Curtis, B.: Fourier-Motzkin Elimination and its Dual. Journal of Combinatorial Theory (1973) 288–297
4. Barrett, C., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker. In: Int. Conf. on Computer-Aided Verification (CAV). Volume 3114 of LNCS., Springer (2004)
5. Stump, A., Barrett, C., Dill, D.: CVC: A Cooperating Validity Checker. In: Int. Conference on Computer Aided Verification (CAV'02). Volume 2404 of LNCS., Springer (2002)

[4] `http://caloproject.sri.com/`

6. Barrett, C., Dill, D., Levitt, J.: Validity Checking for Combinations of Theories with Equality. In: Int. Conference on Formal Methods in Computer-Aided Design (FMCAD). Volume 1166 of LNCS. (1996) 187–201
7. Chvatal, V.: Linear Programming. W. H. Freeman (1983)
8. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: The MathSAT 3 system. In: Int. Conference on Automated Deduction (CADE). Volume 3632 of LNCS., Springer (2005)
9. Filliâtre, J.C., Owre, S., Rueß, H., Shankar, N.: ICS: Integrated Canonization and Solving. In: Proc. of CAV'01. Volume 2102 of LNCS. (2001)
10. Sheini, H.M., Sakallah, K.A.: A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic. In: SAT'05. Volume 3569 of LNCS., Springer (2005) 241–256
11. Rueß, H., Shankar, N.: Solving Linear Arithmetic Constraints. Technical Report SRI-CSL-04-01, SRI International (2004)
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Labs (2003)
13. Necula, G.: Compiling with Proofs. Technical Report CMU-CS-98-154, School of Computer Science, Carnegie Mellon University (1998)
14. Badros, G., Borning, A., Stuckey, P.: The Cassowary Linear Arithmetic Constraint Solving Algorithm. ACM Transactions on Computer-Human Interaction (TOCHI) **8**(4) (2001) 267–306
15. Ranise, S., Tinelli, C.: The satisfiability modulo theories library (smt-lib) (2006) Available at `http://goedel.cs.uiowa.edu/smtib`.
16. Dutertre, B., de Moura, L.: Integrating Simplex with DPLL(T). Technical report, CSL-06-01, SRI International (2006)
17. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York (1986)
18. Nemhauser, G., Wosley, L.: Integer and Combinatorial Optimization. Wiley (1999)
19. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Ranise, S., Sebastiani, R.: Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In: Int. Conf. on Computer-Aided Verification (CAV). Volume 3576 of LNCS., Springer (2005)
20. Wang, C., Ivancic, F., Ganai, M., Gupta, A.: Deciding Separation Logic Formulae with SAT and Incremental Negative Cycle Elimination. In: Logic for Programming Artificial Intelligence and Reasoning (LPAR). (2005)
21. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In: Int. Conference on Computer Aided Verification (CAV'05), Springer (2005) 321–334