SRI International

CSL Technical Report SRI-CSL-06-01 • May 23, 2006

Integrating Simplex with DPLL(T)

Bruno Dutertre and Leonardo de Moura

This report 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).

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

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.

Contents

List of Figures

List of Tables

Chapter 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 [11]: 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 [3]. In the *difference logic* fragment of linear arithmetic, that is, if all the constraints are of the form $x_i - x_j \leq b$, specialized solvers such as Barcelogic [14] or Slice [19] use graph algorithms such as the Bellman-Ford algorithm. For general linear arithmetic, existing tools rely either on Fourier-Motzkin elimination [8] (used by CVClite [2], CVC [18], SVC [4]) or on Simplex methods [7] (used by MathSat [6], ICS [10], Simplics, Yices, ARIO [17]). In practice, Fourier-Motzkin elimination explodes on many problems and Simplex is generally superior. The worst-case complexity of Simplex is exponential but this worst case is rarely encountered in practice. On the other hand, Fourier-Motzkin elimination tends to generate a very large number of intermediate inequalities which causes it to often fail by running out of memory.

The common methods for integrating a Simplex solver with DPLL rely on incremental versions of Simplex such as described in [15, 9, 12, 1]. A tableau is constructed and updated incrementally: rows are added as DPLL proceeds and are later removed when DPLL bactracks. These frequent addition and removal of rows and the related bookkeeping have a significant cost. For example, backtracking may require pivoting operations. This report presents a simpler and more efficient Simplex-based solver that considerably reduces this overhead. The approach relies on transforming the original formula Φ into 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. This report describes such a variant designed to be efficient in the DPLL(T) context: it has low backtracking overhead and enables a simple but useful form of theory propagation. Furthermore, the new approach makes it possible to simplify the problem a priori by eliminating irrelevant variables. This pre-simplification leads to substantial performance improvements on many examples from the SMT-LIB benchmarks.

We first give an overview of the DPLL(T) approach and discuss some lessons learned from our tools Yices and Simplics. Our new solver relies on a simplex-based procedure for real linear arithmetic described in details in Chapter 3. Extension to integer and mixed-integer problems is discussed in Chapter 4, and experimental results are presented in Chapter 5.

Chapter 2

Background

Given a quantifier-free theory T, a *T-solver* is a procedure for deciding whether a finite sets 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. This can be summarized as follows:

- First Φ is transformed into a propositional formula Φ_0 by replacing its atoms $\phi_1, \ldots, \phi_t$ with fresh propositions $p_1, \ldots, p_t$.
- A boolean valuation for Φ_0 is a mapping b from Φ_0's propositions to $\{0, 1\}$. Any such b defines a set of atoms $\Gamma_b = \{\gamma_1, \ldots, \gamma_t\}$ where γ_i is ϕ_i if $b(p_i) = 1$ and γ_i is $\neg\phi_i$ if $b(p_i) = 0$. Then Φ is satisfiable if there is a b that satisfies Φ_0 (in propositional logic) and for which Γ_b is consistent (in theory T).

The DPLL(T) approach is an efficient method for such integrations that relies on the DPLL procedure.

The efficiency of this approach depends to a large extent on the features of modern DPLL-based SAT solver, such as, fast unit propagation, good heuristics for selecting decision variables, clause learning, and non-chronological backtracking. However, a fast SAT solver is not sufficient, several properties of the T-solver are also important.

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 for 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 $\Gamma \models \gamma$. In both cases, the explanation is *minimal* if no proper subset of Γ is itself 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.
- *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 $\gamma_i \in \mathcal{A} \setminus \alpha$. Every γ_i is implied by Γ_i.

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. For example, a solver without theory propagation can return *Propagate*() $= \emptyset$ for any S. 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 or equalities, disequalities, or inequalities of the form

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

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 [15, 9, 12, 1]. Tools based on this approach include our own tools, Yices and Simplics, and others such as MathSat [6].

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

$$x_i = b_i + \sum_{x_j \in \mathcal{N}} a_{ij} x_j, \quad x_i \in \mathcal{B} \tag{2.1}$$

where $\mathcal{B}$ and $\mathcal{N}$ are disjoint sets of variables. Elements of $\mathcal{B}$ and $\mathcal{N}$ are called *basic* and *non-basic* variables, respectively. Additional constraints are imposed on some variables of $\mathcal{B} \cup \mathcal{N}$. So-called

[1] This is similar to the API proposed in [11].

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. [15]).

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

$$x_s \quad = \quad -\frac{b_r}{a_{rs}} + \frac{x_r}{a_{rs}} - \sum_{x_j \in \mathcal{N} \setminus \{x_s\}} \frac{a_{rj} x_j}{a_{rs}} \tag{2.2}$$

then equation (2.2) is used to eliminate x_s from the rest of the tableau by substitution.

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, then the solver must check whether the current tableau implies $t = 0$. This can be implemented via the *zero-detection procedure* described in [15] 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 the 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]

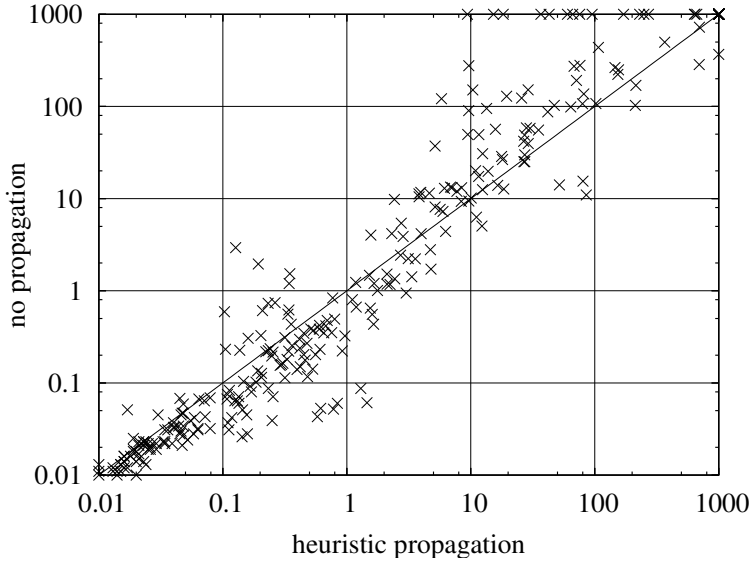
- *Minimal explanations are critical.* Dramatic improvements were observed when comparing Simplics and Yices, which generate minimal explanations, and their predecessor ICS, which does not.

[2] Both use incremental Simplex and zero detection.

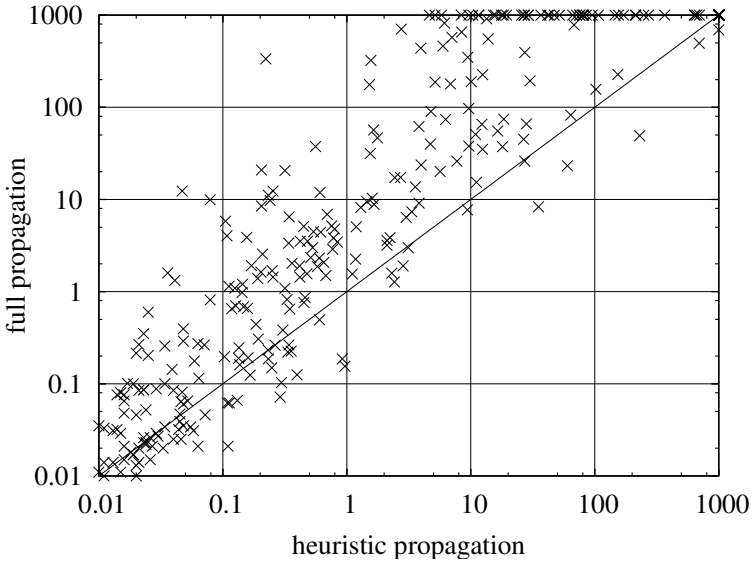
- *Theory propagation is useful if it can be done cheaply*. Figure 2.1 compares the results of Simplics on the real-arithmetic subset of the SMT-LIB benchmarks[3] using different levels of theory propagation. By default, Simplics uses a heuristic form of propagation that's relatively inexpensive but incomplete (no pivoting is used). This is compared in Figure 2.1(a) with Simplics running with no propagation at all, and in Figure 2.1(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]These benchmarks are available at `http://combination.cs.uiowa.edu/smtlib/`. The real-arithmetic subset includes two categories of benchmarks known as QF_RDL and QF_LRA.



(a) Heuristic vs. No Propagation



(b) Heuristic vs. Full Propagation

Figure 2.1: Impact of Theory Propagation in Simplics

Chapter 3

A Linear-Arithmetic Solver for DPLL(T)

3.1 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.

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

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 left-hand 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 (non-strict) inequalities, this reduces to searching for $x \in \mathbb{R}^n$ such that

$$Ax = 0 \ \text{ and } \ l_j \leq x_j \leq u_j \quad \text{for } j = 1, \ldots, n \tag{3.1}$$

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)
  θ := $\frac{v-\beta(x_i)}{a_{ij}}$
  $\beta(x_i)$:= 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)

Figure 3.1: Auxiliary procedures

where l_j is either $-\infty$ or a rational number, and u_j is either $+\infty$ or a rational number. If $l_j = u_j$ then x_j is called a *fixed variable*. If $l_j = -\infty$ and $u_j = +\infty$ then x_j is a *free variable*.

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.

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 (3.1) 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.

3.2 Basic Solver

We first describe a basic solver that handles equalities and non-strict inequalities with real variables. Extensions to strict inequalities and integer variables are presented in the next sections. The basic solver decides the satisfiabilities of problems in form (3.1) 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:

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

where $\mathcal{B}$ and $\mathcal{N}$ denote the set of basic and non-basic variables, respectively.[1] 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 $Ax = 0$.

[1] This is the same as (2.1) with $b_i = 0$ for all $x_i \in \mathcal{B}$.

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 non-basic 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 non-basic 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**

Figure 3.2: Check procedure

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 non-basic variables are always satisfied by β, that is, the following invariant is maintained

$$\forall x_j \in \mathcal{N},\ \ l_j \leq \beta(x_j) \leq u_j. \tag{3.2}$$

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 3.1 describes two auxiliary procedures that modify β. Procedure *update*(x_i, v) sets the value of a non-basic 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 non-basic variable x_j; it also sets the value of x_i to v and adjusts value of all basic variables to keep all equations satisfied.

3.2.1 Main Algorithm

The main procedure of our algorithm is based on the dual Simplex algorithm 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 3.2.

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 correctness of *Check* is a consequene of the following property.

Theorem 1 *Procedure Check always terminates.*

Proof: Every iteration of *Check* modifies the assignment β, the sets of basic and non-basic variables, and the current tableau. However, there is a unique tableau for any set of basic variables $\mathcal{B}$. We can then represent the state of the procedure before the t-th iteration as a pair $S_t = \langle \beta_t, B_t \rangle$, where β_t is the assignment and B_t is the set of basic variables at that point. S_0 denotes then the initial state on entry to the procedure. We also denote by N_t the set of non-basic variables in S_t, that is, $N_t = \{x_1, \ldots, x_n\} \setminus B_t$.

For a non-basic variable $x_j \in N_t$, and any t we have either $\beta_t(x_j) = \beta_0(x_j)$, or $\beta_t(x_j) = l_j$ or $\beta(x_j) = u_j$. For any basic variable $x_i \in B_t$, we have

$$\beta_t(x_i) \quad = \quad \sum_{x_j \in N_t} a_{ij} \beta_t(x_j)$$

where the constants a_{ij} are the tableau coefficients in S_t. Hence, $\beta_t(x_i)$ is uniquely determined by B_t and the values $\beta(x_j)$ for $x_j \in N_t$. There are then finitely many possible assignments β_t for a given B_t. Since the number of variables if finite, there are finitely many possible sets B_t. Therefore, the set of states reachable from S_0 is finite. If *Check* does not terminate, the sequence of states $S_0, S_1, S_2, \ldots$ must contain a cycle, that is, a subsequence $S_k, \ldots, S_t, S_{t+1}$ with $S_{t+1} = S_k$.

Let x_r be the largest variable such that x_r becomes non-basic in one of the states $S_k, \ldots, S_t$. We then have $x_r \in B_l$ and $x_r \in N_{l+1}$ for an index $l \in \{k, \ldots, t\}$. Since x_r becomes non-basic in state S_l, we must have either $\beta_l(x_r) < l_r$ or $\beta_l(x_r) > u_r$. Also, for any variable x_j that is smaller than x_r in the variable ordering we have $l_j \leq \beta_l(x_j) \leq u_j$. This holds if $x_j \in B_l$ since otherwise x_j would be selected to become non-basic rather than x_r (Fig. 3.2, line 3). This also holds if $x_j \in N_l$ by invariant (3.2).

Since the sequence of state is cyclic, x_r must eventually re-enter the basis. Let S_p be the first state after S_{l+1} where x_r becomes basic again: $x_r \in N_p$ and $x_r \in B_{p+1}$. Since x_r stays non-basic in all states $S_{l+1}, \ldots, S_p$ its value does not change, so we have $\beta_p(x_r) = \beta_{l+1}(x_r)$. Let x_s be the basic variable that is pivoted with x_r in S_p and let

$$x_s \quad = \quad \sum_{x_j \in N_p} a_{sj} x_j \tag{3.3}$$

be the corresponding row in S_p's tableau. Since x_s leaves the basis in state S_p, we have either $\beta_p(x_s) < l_s$ or $\beta_p(x_s) > u_s$.

We can now decompose the proof into the following four cases.

1. $\beta_l(x_r) < l_r$ and $\beta_p(x_s) < l_s$.
2. $\beta_l(x_r) < l_r$ and $\beta_p(x_s) > u_s$.
3. $\beta_l(x_r) > u_r$ and $\beta_p(x_s) < l_s$.
4. $\beta_l(x_r) > u_r$ and $\beta_p(x_s) > u_s$.

Let us consider the first case. Since $\beta_l(x_r) < l_r$ we have $\beta_p(x_r) = \beta_{l+1}(x_r) = l_r$. Since x_r is chosen to enter the basis in S_p and $\beta_p(x_s) < l_s$, we must have $a_{sr} > 0$ (Fig. 3.2, line 7).

Equation (3.3) is satisfied by any x such that $Ax = 0$. It is then satified by both β_l and β_p; so we get

$$\beta_l(x_s) - \beta_p(x_s) \quad = \quad \sum_{x_j \in N_p} a_{sj}(\beta_l(x_j) - \beta_p(x_j)) \tag{3.4}$$

By definition of x_r, we know that x_s is smaller than x_r in the variable ordering, so we have $l_s \leq \beta_l(x_s) \leq u_s$. Since we have assumed $\beta_p(x_s) < l_s$, the left-hand side of equation (3.4) is positive. Now, let us consider the terms $a_{sj}(\beta_l(x_j) - \beta_p(x_j))$ that occur in the right-hand side. In all these terms, x_j is a non-basic variable (i.e., $x_j \in N_p$). There are three cases:

- x_j is smaller than x_r in the variable order. As noted previously, this implies $l_j \leq \beta_l(x_j) \leq u_j$. Since x_j was not selected to become basic in state S_p, we must have either $a_{sj} > 0$ and $\beta_p(x_j) \geq u_j$ or $a_{sj} < 0$ and $\beta_p(x_j) \leq l_j$ or $a_{sj} = 0$ (line 7). In all these cases we get

$$a_{sj}(\beta_l(x_j) - \beta_p(x_j)) \quad \leq \quad 0.$$

- $x_j = x_r$. We have $a_{sr} > 0$, $\beta_l(x_r) < l_r$, and $\beta_p(x_r) = l_r$, so we obtain

$$a_{sr}(\beta_l(x_r) - \beta_p(x_r)) \quad < \quad 0.$$

- x_j is larger than x_r in the variable order. In this case, x_j remains non-basic in all the states of the cycle so its value never changes. It follows that $\beta_l(x_j) = \beta_p(x_j)$ and then

$$a_{sj}(\beta_l(x_j) - \beta_p(x_j)) \quad = \quad 0.$$

Thus, for all $x_j \in N_p$ the term $a_{sj}(\beta_l(x_j) - \beta_p(x_j))$ is negative or zero. This contradicts the fact that the left-hand side of equation (3.4) is positive.

A similar contradiction is obtained in the other three cases. We can then conclude that a cyclic sequence of states cannot occur and then that *Check* terminates. □

3.2.2 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 all non-basic 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 non-basic 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

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

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

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

1. **procedure** 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 non-basic 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 non-basic variable and $\beta(x_i) < c_i$ **then** update(x_i, c_i)
6. **return** *ok*

Figure 3.3: Assertion procedures

from which one can derive the following implication:

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

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:

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

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

3.2.3 Assertion Procedures

The *Assert* function relies on two procedures shown in Figure 3.3 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 non-basic, then β is updated to maintain invariant (3.2). 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*.

3.2.4 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 to save the

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

Figure 3.4: Example

successive βs on a stack. Only 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.

3.2.5 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 non-basic 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.

3.2.6 Example

Figure 3.4 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 $\beta_2 = \beta_1$. 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.

3.3 Strict Inequalities

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

Lemma 2 *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 for all δ' such that $0 < \delta' \leq \delta$, $\Gamma_\delta = (\Gamma \cup S_\delta) \setminus S$ 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 non-strict 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 $\mathbb{Q}_\delta$:

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

where a is a rational number.

By the previous definition, if the inequality $(c_1, k_1) \leq (c_2, k_2)$ holds in $\mathbb{Q}_\delta$ then there is a rational number $\delta_0 > 0$ such that the inequality

$$c_1 + k_1\epsilon \quad \leq \quad c_2 + k_2\epsilon$$

is satisfied by positive real ϵ less than δ_0. For example, we can define δ_0 as follows:

$$\begin{array}{rcl} \delta_0 & = & \dfrac{c_2 - c_1}{k_1 - k_2} \quad \text{if } c_1 < c_2 \text{ and } k_1 > k_2 \\ \delta_0 & = & 1 \quad \text{otherwise} \end{array}$$

More generally, consider $2m$ elements of $\mathbb{Q}_\delta$, namely $v_i = (c_i, k_i)$ and $w_i = (d_i, h_i)$ for $i = 1, \ldots, m$. If the m inequalities $v_i \leq w_i$ hold in $\mathbb{Q}_\delta$ then there is a positive rational number δ_0 such that the inequalities

$$\begin{array}{rcl} c_1 + k_1\epsilon & \leq & d_1 + h_1\epsilon \\ & \vdots & \\ c_m + k_m\epsilon & \leq & d_m + h_m\epsilon \end{array}$$

are satisfied for any ϵ such that $0 < \epsilon \leq \delta_0$. One can take

$$\delta_0 \quad = \quad \min \left\{ \frac{d_i - c_i}{k_i - h_i} \mid c_i < d_i \text{ and } k_i > h_i \right\}$$

if the set on the right-hand side is nonempty or set δ_0 to an arbitrary positive rational otherwise.

Now, a linear problem S with strict inequalities in the rationals can be converted into a problem S' in the general form (3.1) but where the bounds and the variables x_i are elements of $\mathbb{Q}_\delta$. Strict bounds in $\mathbb{Q}$ are converted to non-strict bounds in $\mathbb{Q}_\delta$. A strict inequality $x_i > l_i$ is converted to

$x_i \geq l_i + \delta = l'_i$, and $x_i < u_i$ is converted to $x_i \leq u_i - \delta = u'_i$. The matrix A does not change; all its coefficients are rational numbers. The new problem S' can be written

$$\begin{array}{rcl} Ax & = & 0 \\ l'_j \leq & x_j & \leq u'_j \quad \text{for } j = 1, \ldots, n \end{array}$$

and the basic algorithm can be used to determine whether S' is feasible in $\mathbb{Q}_\delta$. This is straighforward; all updates to β used in the previous algorithm can be performed in $\mathbb{Q}_\delta$.

If S' is satisfiable, the algorithm produces a satisfying assignment β' that maps variables to elements of $\mathbb{Q}_\delta$. Then β' can be converted into a satisfying rational assignment β for S. First, the $2n$ inequalities

$$l'_j \leq \beta'(x_j) \leq u'_j \quad \text{for } j = 1, \ldots, n,$$

are satisfied in $\mathbb{Q}_\delta$. Let us set $l'_j = (c_j, k_j)$, $u'_j = (d_j, h_j)$, and $\beta'(x_j) = (p_j, q_j)$. As discussed previously, there is a positive rational number δ_0 such that

$$c_j + k_j\epsilon \leq p_j + q_j\epsilon \leq d_j + h_j\epsilon \quad \text{for } j = 1, \ldots, n \tag{3.5}$$

holds in the reals whenever $0 < \epsilon \leq \delta_0$. In particular, these inequalities hold when $\epsilon = \delta_0$. We can then define β by setting $\beta(x_j) = p_j + q_j\delta_0$ for all x_j and β satisfies all the inequalities (3.5). By construction of S' this implies that β satisfies all the strict and non-strict inequalities in the original problem S. Since β' satisfies the constraints $Ax = 0$ in $\mathbb{Q}_\delta$, it is clear that β satisfies the same constraints $Ax = 0$ in the reals so β is a satisfying assignment for S.

Conversely, if S is satisfiable in the rationals, then any satisfying assignment β for S can be transformed in a straightforward way into a satisfying assignment β' for S'. So if S' is unsatisfiable in $\mathbb{Q}_\delta$, S is not satisfiable either in the rationals.

Chapter 4

Integer and Mixed Integer Problems

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 [16, 13]. 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 accelerate the convergence of branch-and-cut in several cases.

4.1 Branch and Bound

As previously, we consider the constraints

$$\begin{array}{rcll} Ax & = & 0 & \\ l_j \leq & x_j & \leq u_j & \text{for } j = 1, \ldots, n, \end{array}$$

but in addition, some variables are required to be integer valued. The problem is then to find a variable assignment β that satisfies the linear equalities and bound constraints, and such that $\beta(x_j) \in \mathbb{Z}$ for all j in a fixed set $I \subseteq \{1, \ldots, n\}$. We denote this problem by S. We can assume without loss of generality that the bounds l_j and u_j are integer (or infinity) for any j in I.

The branch-and-bound method starts by solving the *linear programming relaxation* (LP relaxation) of S, that is, it searches for a solution β in the reals. In our case, this is done using algorithm *Check* of Section 3.2. If the LP relaxation is infeasible, then S is also infeasible. Otherwise, let β be the solution found by *Check*. If β happens to satisfy all the integer constraints then S is feasible. Otherwise, there is $i \in I$ such that $\beta(x_i)$ is not an integer. There is then a constant $c \in \mathbb{Z}$ such that $c < \beta(x_i) < c + 1$, that is, $c = \lfloor \beta(x_i) \rfloor$. Given c, the original problem S is split into the following two subproblems:

$$\begin{array}{rcrcll} & Ax & = & 0 & & \\ S_0 : & l_j \leq & x_j & \leq u_j & \text{for } j = 1, \ldots, n \text{ and } j \neq i & \\ & l_i \leq & x_i & \leq c & & \end{array}$$

$$
\begin{array}{rrcl}
 & Ax & = & 0 \\
S_1: & l_j \leq & x_j & \leq u_j \quad \text{for } j = 1, \ldots, n \text{ and } j \neq i \\
 & c+1 \leq & x_i & \leq u_i.
\end{array}
$$

Then S_0 or S_1 (or both) are solved recursively using the same procedure. If S_0 or S_1 is satisfiable then S is also satisfiable. If neither is satisfiable then S is also infeasible.

Thus, the branch-and-bound method recursively divides S into a set of smaller subproblems obtained by modifying lower or upper bounds of the integer variables. This recursive division can be efficiently implemented via a depth-first search. Unlike traditional branch-and-bound methods, the algorithm does not attempt to minimize or maximize a linear function but stops when the first integer-feasible solution is found. If a subproblem S' encountered during the search is infeasible, then *Check* produces an explanation, in the form of a minimal set of inconsistent bound constraints. This explanation is exploited to prune the search via non-chronological backtracking.

The branch-and-bound approach works when strict inequalities are present. In such a case the LP-relaxation is solved in $\mathbb{Q}_\delta$ rather than $\mathbb{Q}$. As in the real case, we define $\lfloor \beta(x_i) \rfloor$ as the largest integer less than $\beta(x_i)$. Assuming $\beta(x_i) = s + k\delta$, we then get

$$
\lfloor \beta(x_i) \rfloor = \begin{cases} \lfloor s \rfloor & \text{if } s \notin \mathbb{Z} \\ s & \text{if } s \in \mathbb{Z} \text{ and } k \geq 0 \\ s - 1 & \text{if } s \in \mathbb{Z} \text{ and } k < 0 \end{cases}
$$

This is the only adjustment required for solving interger or mixed integer problems with strict inequalities.

4.2 Gomory Cuts

Branch-and-bound terminates if all the integer variables have a lower and an upper bound, that is, if $-\infty < l_j$ and $u_j < \infty$ for all j in I. Without such bounds, the algorithm may fail to terminate, even on trivial problems. For example, the constraint

$$1 \leq 3x - 3y \leq 2$$

is not satisfiable if x and y are integers, but has unbounded real solutions. On this example, a naïve branch-and-bound implementation loops. Fortunately, we can assume without loss of generality that lower and upper bounds are given for all variables. If S is feasible then it has a solution that is an extreme point of its convex hull, and an explicit bound on the magnitude of such extreme points can be derived from the coefficients of the matrix A (cf. [16, 13]). However, this bound is typically too large to ensure quick termination in practice.

To accelerate convergence, modern integer-programming methods combine branch-and-bound with cutting-plane algorithms. Assume β is a solution to the LP-relaxation P of S but not a solution to S. A *cut* is a linear inequality

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

that is not satisfied by β but is satisfied by any element in the convex hull of S. If one or more such cuts can be found, then they can be added as new constraints to the original problem S. This gives a new problem S' that has the same solutions as S but whose LP-relaxation P' is strictly more constrained than P. In particular, β is a solution to P but it is not a solution to P'. Therefore, rather

than splitting S into subproblems as previously, one can attempt to find a solution β' to P' and iterate the process. Combining cutting-plane and branch-and-bound methods has led to dramatic improvements in the size of mixed-integer programming problems that can be efficiently solved in practice [5].

4.2.1 Derivation of a Gomory Cut

Many cut-generation algorithms have been proposed (see [13]). A general method due to Gomory is simple to implement, widely applicable, and known to be quite effective in practice. In our context, let β be the solution returned by *Check* to the LP-relaxation P of a problem S. Thus, β is obtained from a tableau of the form

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

A *mixed integer Gomory cut* can be constructed from β and the tableau if the following conditions are satisfied:

- There is a basic variable $x_i \in \mathcal{B}$ such that $i \in I$ and $\beta(x_i) \notin \mathbb{Z}$.
- All non-basic variables occurring in the row corresponding to x_i, are assigned to their upper or lower bound:

$$\forall x_j \in \mathcal{N},\ a_{ij} \neq 0 \ \Rightarrow\ \beta(x_j) = l_j \ \vee\ \beta(x_j) = u_j.$$

- All the numbers $\beta(x_i)$, and $\beta(x_j)$ for x_j occurring in row i are rationals (i.e., they are not of the form $c + k\delta$ with $k \neq 0$).

Let $f_0 = \beta(x_i) - \lfloor \beta(x_i) \rfloor$. By the first assumption, f_0 is a positive rational number, so we have $0 < f_0 < 1$. By the second assumption, free variables do not occur in row i. We can also remove the fixed variables: let $\mathcal{N}' = \mathcal{N} \cap \{x_j \mid l_j < u_j\}$. Now, if j is the index of a non-basic variable of $\mathcal{N}'$ then two cases are possible: either $\beta(x_j) = l_i$ or $\beta(x_j) = u_j$. Let us define

$$\begin{aligned} J &= \{j \in I \mid x_j \in \mathcal{N}' \ \wedge\ \beta(x_j) = l_i\} \\ K &= \{j \in I \mid x_j \in \mathcal{N}' \ \wedge\ \beta(x_j) = u_i\}. \end{aligned}$$

The row of x_i, that is, the equation

$$x_i \quad = \quad \sum_{x_j \in \mathcal{N}} a_{ij} x_j \tag{4.1}$$

is satisfied by any x such that $Ax = 0$ and we have $x_i \in \mathbb{Z}$ for any x that satisfies S. For the assignment β, we also have

$$\beta(x_i) \quad = \quad \sum_{x_j \in \mathcal{N}} a_{ij} \beta(x_j). \tag{4.2}$$

Subtracting (4.2) from (4.1) we obtain the following equation

$$x_i - \beta(x_i) \quad = \quad \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j),$$

which can be rewritten as

$$x_i - \lfloor \beta(x_i) \rfloor \quad = \quad f_0 + \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \tag{4.3}$$

This equation holds for any x that satisfies S and, for any such x we know that $x_i - \lfloor \beta(x_i) \rfloor$ is an integer and that the following inequalities are satisfied:

$$\begin{aligned} x_j - l_j &\geq 0 \quad \text{for all } j \in J \\ u_j - x_j &\geq 0 \quad \text{for all } j \in K. \end{aligned}$$

We now consider two cases:

- If $\sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \geq 0$ then we must have

  $$f_0 + \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \quad \geq \quad 1 \tag{4.4}$$

  since the left-hand-side is an integer. Let us split the sets J and K as follows:

  $$\begin{aligned} J^+ &= \{j \in J \mid a_{ij} \geq 0\} \\ J^- &= \{j \in J \mid a_{ij} < 0\} \\ K^+ &= \{j \in K \mid a_{ij} \geq 0\} \\ K^- &= \{j \in K \mid a_{ij} < 0\}. \end{aligned}$$

  Then inequality (4.4) implies

  $$\sum_{j \in J^+} a_{ij}(x_j - l_j) - \sum_{j \in K^-} a_{ij}(u_j - x_j) \quad \geq \quad 1 - f_0,$$

  or, equivalently,

  $$\sum_{j \in J^+} \frac{a_{ij}}{1 - f_0}(x_j - l_j) + \sum_{j \in K^-} \frac{-a_{ij}}{1 - f_0}(u_j - x_j) \quad \geq \quad 1. \tag{4.5}$$

- If $\sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) < 0$, then we must have

  $$f_0 + \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \quad \leq \quad 0,$$

  which implies

  $$- \sum_{j \in J^-} a_{ij}(x_j - l_j) + \sum_{j \in K^+} a_{ij}(u_j - x_j) \quad \geq \quad f_0,$$

  or

  $$\sum_{j \in J^-} \frac{-a_{ij}}{f_0}(x_j - l_j) + \sum_{j \in K^+} \frac{a_{ij}}{f_0}(u_j - x_j) \quad \geq \quad 1. \tag{4.6}$$

Combining (4.5) and (4.6) we obtain the following inequality:

$$\begin{aligned}\sum_{j\in J^+}\frac{a_{ij}}{1-f_0}(x_j-l_j)+\sum_{j\in J^-}\frac{-a_{ij}}{f_0}(x_j-l_j) \quad &+\\ \sum_{j\in K^+}\frac{a_{ij}}{f_0}(u_j-x_j)+\sum_{j\in K^-}\frac{-a_{ij}}{1-f_0}(u_j-x_j) \quad &\geq \quad 1. \end{aligned} \tag{4.7}$$

This is a *mixed-integer Gomory cut*: this inequality is satisfied by any x that satisfies S, but it is not satisfied by the assignment β.

4.2.2 A Stronger Gomory Cut

Inequality (4.7) was derived from the fact that

$$f_0+\sum_{j\in J}a_{ij}(x_j-l_j)-\sum_{j\in K}a_{ij}(u_j-x_j)$$

is an integer for any x that satisfies S. As long as we maintain this property, we can replace the coefficients a_{ij} with other rationals a'_{ij} and derive a different cut. In particular, if x_j is an integer variable then we know that l_j and u_j are integers so we can replace a_{ij} by any a'_{ij} such that $(a'_{ij}-a_{ij})\in\mathbb{Z}$. So we can further split the index sets J and K by distinguishing between integer and non-integer variables:

$$\begin{aligned} J_0 &= J\cap I\\ J_1 &= J\setminus J_0\\ K_0 &= K\cap I\\ K_1 &= K\setminus K_0 \end{aligned}$$

Then we can replace a_{ij} by an a'_{ij} such that $(a'_{ij}-a_{ij})\in\mathbb{Z}$ for any $j\in J_0\cup K_0$. This preserves the essential property, namely the fact that

$$f_0+\sum_{j\in J_0}a'_{ij}(x_j-l_j)+\sum_{j\in J_1}a_{ij}(x_j-l_j)-\sum_{j\in K_0}a'_{ij}(u_j-x_j)-\sum_{j\in K_1}a_{ij}(u_j-x_j) \tag{4.8}$$

is an integer. From this new term, we can derive a mixed integer cut as before.

For $j\in J_0$, the best choice for a'_{ij} is the one that leads to the smallest coefficient of (x_j-l_j) in the cut. If a'_{ij} is positive, this coefficient is $a'_{ij}/(1-f_0)$ otherwise it is $-a'_{ij}/f_0$. Let $f_i=a_{ij}-\lfloor a_{ij}\rfloor$ then a simple analysis shows that the best choice is given by

$$\begin{aligned} a'_{ij} &= f_i = a_{ij}-\lfloor a_{ij}\rfloor \quad \text{if } f_i\leq 1-f_0\\ a'_{ij} &= f_i-1 = a_{ij}-\lceil a_{ij}\rceil \quad \text{if } f_i>1-f_0 \end{aligned}$$

For $j\in K_0$ the best a'_{ij} can be determined in a similar fashion and it is given by

$$\begin{aligned} a'_{ij} &= f_i = a_{ij}-\lfloor a_{ij}\rfloor \quad \text{if } f_i\leq f_0\\ a'_{ij} &= f_i-1 = a_{ij}-\lceil a_{ij}\rceil \quad \text{if } f_i>f_0 \end{aligned}$$

By choosing the coefficients a'_{ij} as above, we obtain the following inequality, which is the strongest possible mixed-integer Gomory cut that can be obtained using the previous techniques:

$$\begin{aligned}
\sum_{j \in J_1^+} \frac{a_{ij}}{1-f_0}(x_j - l_j) + \sum_{j \in J_1^-} \frac{-a_{ij}}{f_0}(x_j - l_j) \quad + \\
\sum_{j \in K_1^+} \frac{a_{ij}}{f_0}(u_j - x_j) + \sum_{j \in K_1^-} \frac{-a_{ij}}{1-f_0}(u_j - x_j) \quad + \\
\sum_{j \in J_0, f_j \leq 1-f_0} \frac{f_j}{1-f_0}(x_j - l_j) + \sum_{j \in J_0, f_j > 1-f_0} \frac{1-f_j}{f_0}(x_j - l_j) \quad + \\
\sum_{j \in K_0, f_j \leq f_0} \frac{f_j}{f_0}(u_j - x_j) + \sum_{j \in K_0, f_j > f_0} \frac{1-f_j}{1-f_0}(u_j - x_j) \quad \geq \quad 1.
\end{aligned}$$

Chapter 5

Experiments

Figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6 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 5.1 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 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.

	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**

Table 5.1: Experimental results: Summary

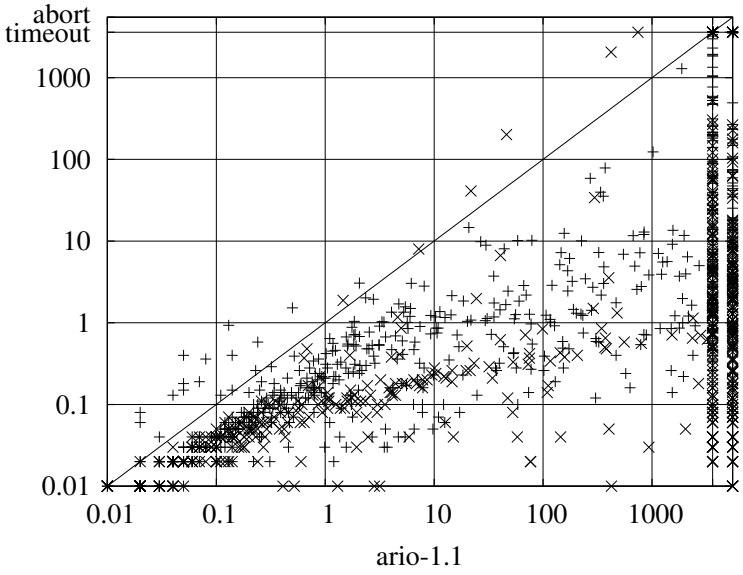


Figure 5.1: Ario 1.1 vs. New Solver

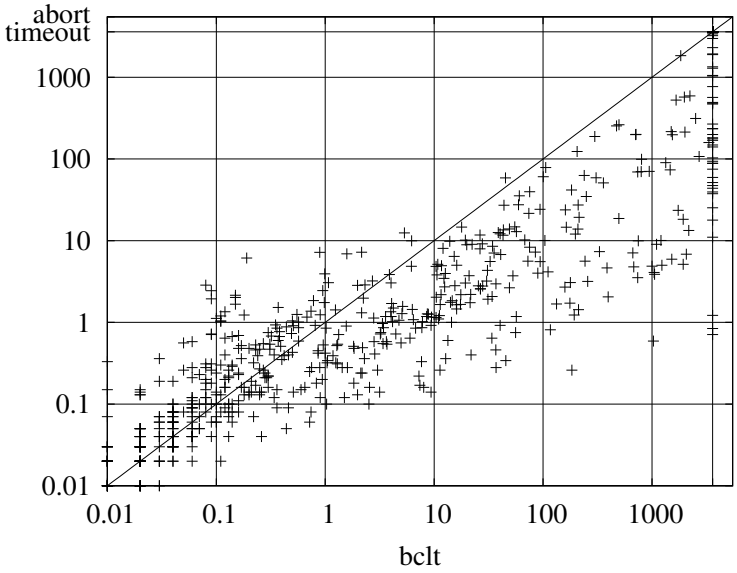


Figure 5.2: BarcelogicTools vs. New Solver

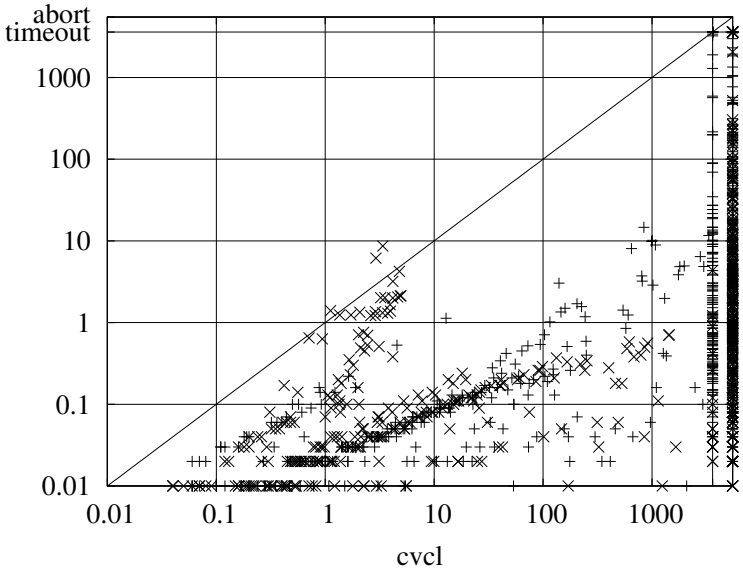


Figure 5.3: CVC Lite 2.0 vs. New Solver

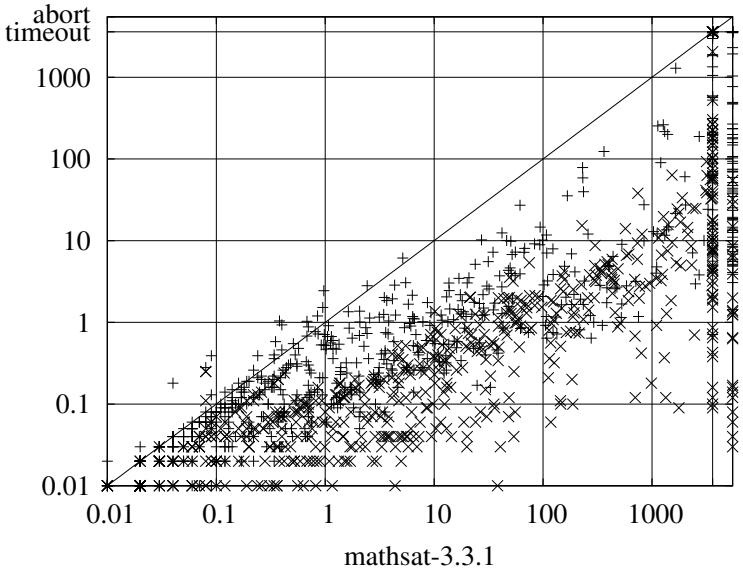


Figure 5.4: MathSAT 3.3.1 vs. New Solver

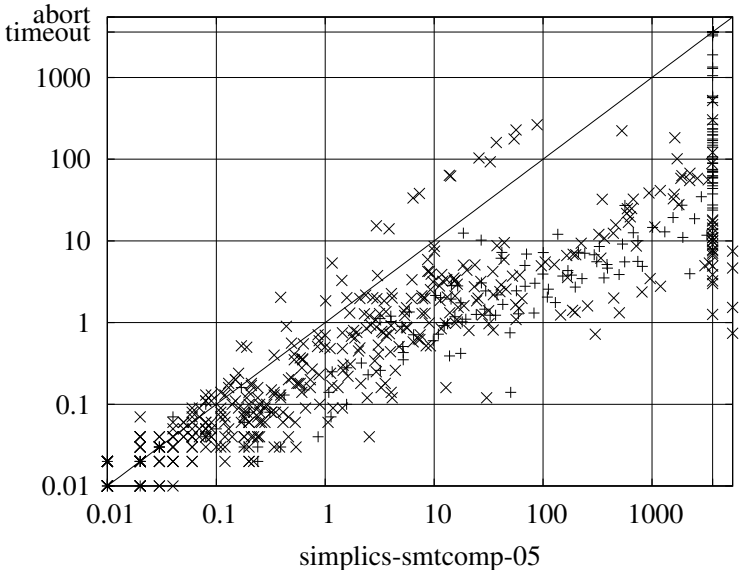


Figure 5.5: Simplics vs. New Solver

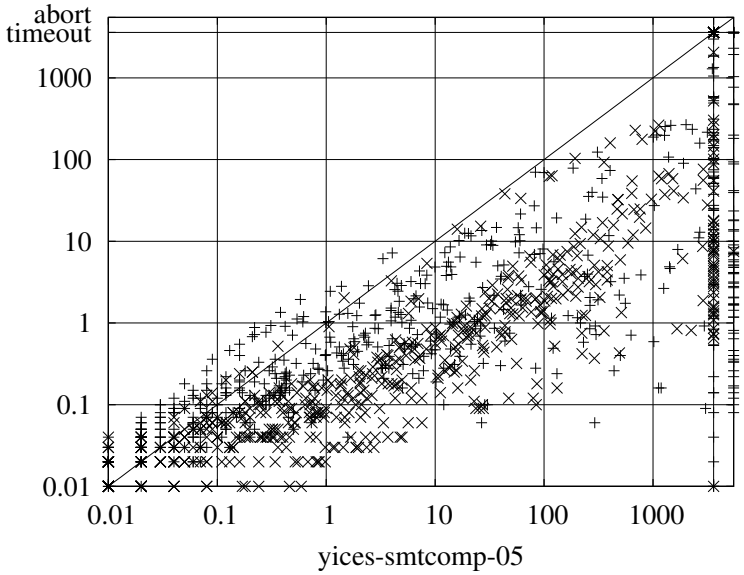


Figure 5.6: Yices vs. New Solver

Chapter 6

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[1], as part of a module that combines learning and deductive algorithms.

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

Bibliography

[1] G.J. Badros, A. Borning, and P.J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, December 2001.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Int. Conf. on Computer-Aided Verification (CAV)*, volume 3114 of *LNCS*. Springer, 2004.

[3] C. Barrett, L. de Moura, and A. Stump. Design and results of the 1st satisfiability modulo theories competition (SMT-COMP 2005). To appear in Journal of Automated Reasoning, 2006.

[4] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Int. Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*, pages 187–201, 1996.

[5] R. E. Bixby, M. Fenelon, Z.gu, E. Rothberg, and R. Wunderling. MIP: theory and practice – closing the gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization: Methods, Theory, and Applications*. Kluwer, 2000.

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. The MathSAT 3 system. In *Int. Conference on Automated Deduction (CADE)*, volume 3632 of *LNCS*. Springer, 2005.

[7] V. Chvatal. *Linear Programming*. W. H. Freeman, 1983.

[8] G.B. Dantzig and B. Curtis. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory*, pages 288–297, 1973.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[10] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In *Proc. of CAV'01*, volume 2102 of *LNCS*, 2001.

[11] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Int. Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[12] G.C. Necula. Compiling with proofs. Technical Report CMU-CS-98-154, School of Computer Science, Carnegie Mellon University, 1998.

[13] G. Nemhauser and L. Wosley. *Integer and Combinatorial Optimization*. Wiley, 1999.

[14] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Int. Conference on Computer Aided Verification (CAV'05)*, pages 321–334. Springer, 2005.

[15] H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report SRI-CSL-04-01, SRI International, 2004.

[16] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, New York, 1986.

[17] H. M. Sheini and K. A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *SAT'05*, volume 3569 of *LNCS*, pages 241–256. Springer, 2005.

[18] A. Stump, C.W. Barrett, and D.L. Dill. CVC: A Cooperating Validity Checker. In *Int. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*. Springer, 2002.

[19] C. Wang, F. Ivancic, M. Ganai, and A. Gupta. Deciding separation logic formulae with SAT and incremental negative cycle elimination. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 2005.