Solving Sparse Linear Constraints

Shuvendu K. Lahiri and Madanlal Musuvathi

Microsoft Research
{shuvendu, madanm}@microsoft.com

Abstract. Linear arithmetic decision procedures form an important part of theorem provers for program verification. In most verification benchmarks, the linear arithmetic constraints are dominated by simple difference constraints of the form $x \leq y + c$. *Sparse linear arithmetic* (SLA) denotes a set of linear arithmetic constraints with a very few non-difference constraints. In this paper, we propose an efficient decision procedure for SLA constraints, by combining a solver for difference constraints with a solver for general linear constraints. For SLA constraints, the space and time complexity of the resulting algorithm is dominated solely by the complexity for solving the difference constraints. The decision procedure generates models for satisfiable formulas. We show how this combination can be extended to generate implied equalities. We instantiate this framework with an equality generating Simplex as the linear arithmetic solver, and present preliminary experimental evaluation of our implementation on a set of linear arithmetic benchmarks.

1 Introduction

Many program analysis and verification techniques involve checking the satisfiability of formulas containing linear arithmetic constraints. These constraints appear naturally when reasoning about integer variables and array operations in programs. As such, there is a practical need to develop solvers that effectively check the satisfiability of linear arithmetic constraints.

It has been observed [21] that many of the arithmetic constraints that arise in verification or program analysis comprise mostly of *difference* constraints. These constraints are of the form $x \leq y + c$, where x and y are variables and c is a constant. Although efficient polynomial algorithms exist for checking the satisfiability of such constraints, these algorithms cannot be directly used if non-difference constraints, albeit few, are present in the input. In practice, this makes it hard to exploit the efficiency of difference constraints in arithmetic solvers.

Motivated by this problem, we propose a mechanism for solving general linear arithmetic constraints that exploits the presence of difference constraints in the input. We define a set of linear arithmetic constraints as *sparse linear arithmetic(SLA)* constraints, when the fraction of non-difference constraints is very small compared to the fraction of difference constraints.

The main contribution of this paper is a framework for solving linear arithmetic constraints that combines a solver for difference constraints with a general linear arithmetic constraint solver. The former analyzes the difference constraints in the input while the latter processes only the non-difference constraints. These solvers then share relevant facts to check the satisfiability of the input constraints. When used to solve SLA constraints, the time and space complexity of our combination solver is determined solely by the complexity of the difference constraint solver. As a result, our algorithm retains the efficiency of the difference constraint solvers with the completeness of a linear arithmetic solver. Additionally, the combined solver can also generate models (satisfying assignments) for satisfiable formulas.

The second key contribution of this paper is an efficient algorithm for generating the set of implied variable equalities from the combined solver. Generating such equalities is essential when our solver is used in the Nelson-Oppen combination framework [19]. We show that for rationals, the difference and the non-difference solvers only need to exchange equalities with offsets (of the form $x = y + c$) over the shared variables to generate all the implied equalities.

We provide an instantiation of the framework by combining a solver for difference constraints based on *negative cycle detection* algorithms, and a solver for general linear arithmetic constraints based on Simplex [6]. We show that we can modify the Simplex implementation in Simplify [7] (that already generates all implied equalities of the form $x = y$) to generate implied equalities of the form $x = y + c$ without incurring any more overhead. Finally, we provide preliminary experimental results on a set of linear arithmetic benchmarks of varying complexity.

The rest of the paper is organized as follows: In Section 2, we describe the background work including solvers for difference logic. In Section 3, we formally describe the SLA constraints and provide a decision procedure. We extend the decision procedure to generate implied equalities in Section 4.1, and provide a concrete implementation with Simplex in Section 4.2. We present the results in Section 5. In Section 6, we present the related work. Details of the proofs can be found in an extended technical report [16].

2 Background

For a given theory T, a decision procedure for T checks if a formula ϕ in the theory is *satisfiable*, i.e. it is possible to assign values to the symbols in ϕ that are consistent with T, such that ϕ evaluates to `true`.

Decision procedures, nowadays, do not operate in isolation, but form a part of a more complex system that can decide formulas involving symbols shared across multiple theories. In such a setting, a decision procedure has to support the following operations efficiently: (i) *Satisfiability Checking*: Checking if a formula ϕ is satisfiable in the theory. (ii) *Model Generation*: If a formula in the theory is satisfiable, find values for the symbols that appear in the theory that makes it satisfiable. This is crucial for applications that use theorem provers for test-case generation. (iii) *Equality Generation*: The Nelson-Oppen framework for combining decision procedures [19] requires that each theory (at least) produces the set of equalities over variables that are implied by the constraints. (iv) *Proof Generation*: Proof generation can be used to certify the output of a theorem prover [18]. Proofs are also used to construct conflict clauses efficiently in a lazy SAT-based theorem proving architecture [8].

2.1 Linear Arithmetic

Linear arithmetic is the first-order theory where atomic formulas (also called linear constraints) are of the form $\sum_i a_i.x_i \bowtie c$, where x_i is a variable from the set X, each of a_i and c is a constant and $\bowtie \in \{\leq, <, =\}$. When the variables in X range over integers $\mathbb{Z}$, and each of the constants a_i and c is a integer constant, we refer to the theory as integer linear arithmetic. Otherwise, if the variables and the constants range over rationals $\mathbb{Q}$, we refer to it as simply linear arithmetic.

An assignment ρ maps each variable in X to either an integer or a rational value, depending on the underlying theory. A set of linear constraints $\{l_i | l_i \doteq \sum_j a_{i,j}.x_j \bowtie c_i\}$ is *satisfiable*, if there is an assignment ρ such that each l_i evaluates to `true`. Otherwise, the set of linear constraints is said to be *unsatisfiable*.

Given two assignments ρ_A and ρ_B over set of variables A and B respectively (A and B need not be disjoint), we define the resulting assignment $\rho \doteq \rho_A \circ \rho_B$ obtained by composing ρ_A and ρ_B as follows for any $x \in A \cup B$:

$$\rho_A \circ \rho_B(x) = \begin{cases} \rho_A(x) & \text{if } x \in A \\ \rho_B(x) & \text{otherwise} \end{cases}$$

Deciding the satisfiability of a set of integer linear arithmetic constraints is NP-complete [20]. For the rational counterpart, there exists polynomial algorithms for deciding satisfiability [13]. However, in spite of the polynomial complexity, these algorithms have large overhead that make them infeasible on large problems. Instead, Simplex [6] algorithm (that has worst-case exponential complexity) has been found to be efficient for most practical problems. We will describe more about the workings of Simplex in Section 4.2.

2.2 Difference Constraints and Negative Cycle Detection

A particularly useful fragment of linear arithmetic is the theory of *difference constraints*, where the atomic formulas are of the form $x_1 - x_2 \bowtie c$. Constraints of the forms $x \bowtie c$ are converted to the above form by introducing a special vertex x_{orig} to denote the origin, and expressing the constraint as $x - x_{orig} \bowtie c$. The resultant system of difference constraints is equisatisfiable with the original set of constraints. Moreover, if ρ satisfies the resultant set of difference constraints, then a satisfying assignment ρ' to the original set of constraints (that include $x \bowtie c$ constraints) can be obtained by simply assigning $\rho'(x) \doteq \rho(x) - \rho(x_{orig})$, for each variable. A set of difference constraints (both over integers and rationals) can be decided in polynomial time using *negative cycle detection* algorithms.

Given a weighted graph $G(V, E)$, the problem of determining if G has a cycle C, such that sum of the (weight on the) edges along the cycle is negative, is called the negative cycle detection problem. Various algorithms can be used to determine the existence of negative cycles in a graph [4]. Negative cycle detection (NCD) algorithms have two properties:

1. The algorithm determines if there is a negative cycle in the graph. In this case, the algorithm produces a particular negative cycle as a witness.
2. If there are no negative cycles, then the algorithm generates a *feasible* solution $\delta : V \rightarrow \mathbb{Q}$, such that for every $(u, v) \in E$, $\delta(v) \leq \delta(u) + w(u, v)$. Moreover, if all the weights $w(u, v) \in \mathbb{Z}$ for any $(u, v) \in E$, then δ assigns integral values to all vertices.

For example, the Bellman-Ford [3,9] algorithm for single-source shortest path in a graph can be used to detect negative cycles in a graph. If the graph contains n vertices and m edges, the Bellman-Ford algorithm can determine in $O(n.m)$ time and $O(n+m)$ space, if there is a negative cycle in G, and a feasible solution otherwise.

In this paper, we assume that we use one such NCD algorithm. We will define the complexity $O(\mathcal{NCD})$ as the complexity of the NCD algorithm under consideration. This allows us to leverage all the advances in NCD algorithms in recent years [4], which have complexity better than the Bellman-Ford algorithm.

Given a set of difference constraints, we can construct a weighted directed graph by creating a vertex for each variable in the set of constraints, and creating an edge from a vertex x to vertex y with a weight c for each constraint $y - x \leq c$. We will refer to the set of difference constraints and the underlying graph interchangeably in the rest of the paper.

3 Sparse Linear Arithmetic (SLA) Constraints

Pratt [21] observed that most queries that arise in software verification are dominated by difference constraints. Recently, more evidence has been presented strengthening the hypothesis [24], where the authors found more than 95% of the linear arithmetic constraints were restricted to difference constraints for a set of program verification benchmarks. Hence, it is crucial to construct decision procedures for linear arithmetic that can exploit the *sparse* nature of general linear constraints.

Let $\phi \doteq \bigwedge_i \left(\sum_j a_{i,j}.x_j \leq c_i\right)$ be the conjunction of a set of (integer or rational) linear arithmetic constraints over a set of variables X. Let us partition the set of constraints in ϕ into the set of difference constraints ϕ_D and the non-difference constraints ϕ_L, such that $\phi = \phi_D \wedge \phi_L$. Let D be the set of variables that appear in ϕ_D, L be the set of variables that appear in ϕ_L, and let Q be the set of variables in $D \cap L$. We assume that the variable x_{orig} to denote the origin, always belong to D, and any $x \bowtie c$ constraint has been converted to $x \bowtie x_{orig} + c$.

We define a set of constraints ϕ to be *sparse linear arithmetic* (SLA) constraints, if the fraction $|L|/|D| \ll 1$. Observe this also implies that $|Q|/|D| \ll 1$. Our goal is to devise an efficient decision procedure for SLA constraints, such that the complexity is polynomial in D but (possibly) exponential only over L. This would be particularly appealing for solving integer linear constraints, where the complexity of the decision problem is NP-complete. For rational linear arithmetic, the procedure will still retain its polynomial complexity, but will improve the robustness on practical benchmarks by mitigating the effect of the general linear arithmetic solver.

In this section, we describe one such decision procedure for SLA constraints. In Section 4, we show how to generate implied equalities between variable pairs from such a decision procedure and describe its integration with Simplex, for rational linear arithmetic.

3.1 Checking Satisfiability of SLA

We provide an algorithm for checking the satisfiability of a set of SLA constraints that has polynomial complexity in the size of the difference constraints. Moreover, the space complexity of the algorithm is *almost* linear in the size of the difference constraints. Finally, assuming we have a decision procedure for integer linear arithmetic that generates satisfying assignments, the algorithm can generate an integer solution when the input SLA formula is satisfiable over integers.

Let ϕ be a set of linear arithmetic constraints as before, and let Q be the set of variables common to the difference constraints ϕ_D and non-difference constraints ϕ_L. The algorithm (*SLA-SAT*) is simple, and operates in four steps:

1. Check the satisfiability of ϕ_D using a negative cycle detection algorithm.
2. If ϕ_D is unsatisfiable, return unsatisfiable. Else, let $SP(x, y)$ be the weight of the shortest path from the (vertices corresponding to) variable x to y in the graph induced by ϕ_D. Generate the set of difference constraints
$$\phi_Q \doteq \bigwedge \{y - x \leq d \mid x \in Q, y \in Q, SP(x, y) = d\}, \tag{1}$$
over Q.
3. Check the satisfiability of $\phi_L \wedge \phi_Q$ using a linear arithmetic decision procedure. If $\phi_L \wedge \phi_Q$ is unsatisfiable, then return unsatisfiable. Else, let ρ_L be a satisfying assignment for $\phi_L \wedge \phi_Q$ over L.
4. Generate a satisfying assignment ρ_D to the formula $\phi_D \wedge \bigwedge_{x \in Q} (x = \rho_L(x))$, using a negative cycle detection algorithm. Return $\rho_X \doteq \rho_D \circ \rho_L$ as a satisfying assignment for ϕ.

It is easy to see that the algorithm is sound. This is because we report unsatisfiable only when a set of constraints implied by ϕ is detected to be unsatisfiable. To show that the algorithm is complete (for both integer and rational arithmetic), we show that if ϕ_D and $\phi_Q \wedge \phi_L$ are each satisfiable, then ϕ is satisfiable. This is achieved by showing that a satisfying assignment ρ_L for $\phi_L \wedge \phi_Q$ can be extended to an assignment ρ_X for ϕ, such that ϕ is satisfiable.

Lemma 1. *If the assignment ρ_L over L satisfies $\phi_L \wedge \phi_Q$, then the assignment ρ_X over X satisfies ϕ.*

Since a model for ϕ_Q can be extended to be a model for ϕ_D, Lemma 1 also shows another useful fact, which we will utilize later:

Corollary 1. *Let $P \doteq D \setminus Q$ be the set of variables local to ϕ_D. Then ϕ_Q is equivalent to $(\exists P : \phi_D)$, denoted as $\phi_Q \Leftrightarrow (\exists P : \phi_D)$.*

The corollary says that ϕ_Q is the result of quantifier elimination of the variables $D \setminus Q$ local to ϕ_D. Hence, for any constraint ψ over Q, ϕ_D implies ψ (denoted as $\phi_D \Rightarrow \psi$) if and only if $\phi_Q \Rightarrow \psi$. We will make use of this fact throughout the paper.

Theorem 1. *The algorithm* SLA-SAT *is a decision procedure for (integer and rational) linear arithmetic. Moreover, it also generates a satisfying assignment when the constraints are satisfiable.*

Complexity of *SLA-SAT*: Given m difference constraints over n variables, we denote $\mathcal{NCD}(n, m)$ as the complexity of the negative cycle detection algorithm. The space complexity for $\mathcal{NCD}(n, m)$ is $O(n + m)$, and the upper bound of the time complexity is $O(n.m)$, although many algorithms have a much better complexity [4]. Similarly, with m constraints over n variables, we denote $\mathcal{LAP}(n, m)$ as the complexity of the linear arithmetic procedure under consideration. For example, if we use Simplex as the (rational) linear arithmetic decision procedure, then the space complexity for $\mathcal{LAP}(n, m)$ is $O(n.m)$ and the time complexity is polynomial in n and m in practice. Finally, for a set of constraints ψ, let $|\psi|$ denote the the number of constraints in ψ.

Let us try to analyze the complexity of the procedure *SLA-SAT* described in the previous section. Step 1 takes $\mathcal{NCD}(|D|, |\phi_D|)$ time and space complexity. Step 2 requires generating shortest paths between every pair of variables $x \in Q$ and $y \in Q$. This can be obtained by using a variant of Johnson's algorithm for generating all-pair-shortest-paths [5] for a graph. For a graph with n nodes and m vertices, this algorithm has linear space complexity of $O(n+m)$. Assuming we have already performed a negative cycle detection algorithm, the time complexity of the algorithm is only $O(n^2. \log(n))$.

Instead of generating all-pair-shortest-paths for every pair of vertices using Johnson's algorithm, we adapt the algorithm to compute the shortest paths only for vertices in Q, the set of shared variables. This makes the time complexity of Step 2 of the algorithm $O(|Q|.|D|. \log(|D|))$. The space complexity of this step is $O(|\phi_Q|)$ which is bounded by $O(|Q|^2)$.

The complexity of Step 3 is $\mathcal{LAP}(|L|, |\phi_Q| + |\phi_L|)$. Finally, Step 4 incurs another $\mathcal{NCD}(|D|, |\phi_D|)$ complexity, since at most $|Q|$ constraints are added as $x = \rho_L(x)$ constraints to ϕ_D.

4 Equality Generation for SLA

In this section, we consider the problem of generating equalities between variables implied by the constraint ϕ. Equality generation is useful for combining the linear arithmetic decision procedure with other decision procedures in the

Nelson-Oppen combination framework. In Section 4.1, we describe the requirements from the difference and the non-difference decision procedures in *SLA-SAT* to generate all equalities implied by ϕ. In Section 4.2, we describe how to instantiate the framework when combining a negative cycle detection algorithm (as the decision procedure for difference constraints) with Simplex (as the decision procedure for non-difference constraints).

4.1 Equality Generation from *SLA-SAT*

In this section, we extend the basic *SLA-SAT* algorithm to generate all the equalities between pairs of variables, implied by the input formula ϕ. We will describe the procedure in an abstract fashion, without providing an implementation of the individual steps. The algorithm described in this section has only been proved complete for the case when the variables are interpreted over $\mathbb{Q}$; we are currently working on the case of $\mathbb{Z}$.

Throughout this section, we assume that ϕ is satisfiable. We carry the notations (e.g. ϕ_D, ϕ_L etc.) from Section 3. The key steps of the procedure are:

1. Assuming ϕ_D is satisfiable, generate ϕ_Q and solve $\phi_Q \wedge \phi_L$ using linear arithmetic decision procedure.
2. Generate the set of equalities (with offsets) implied by $\phi_Q \wedge \phi_L$

$$\mathcal{E}_1 \doteq \{x = y + c \mid x \in L, y \in L, \text{and}\,(\phi_Q \wedge \phi_L) \Rightarrow x = y + c\}, \qquad (2)$$

   from the linear arithmetic decision procedure.
3. Let $\mathcal{E}_2 \subseteq \mathcal{E}_1$ be the set of equalities over the variables in Q:

$$\mathcal{E}_2 \doteq \{x = y + c \mid x \in Q, y \in Q, x = y + c \in \mathcal{E}_1 \}, \qquad (3)$$

4. Generate all the implied equalities (with offset) from $\mathcal{E}_2$ (interpreted as a formula by conjoining all the equalities in $\mathcal{E}_2$) and ϕ_D:

$$\mathcal{E}_3 \doteq \{x = y + c \mid x \in D, y \in D, (\phi_D \wedge \mathcal{E}_2) \Rightarrow x = y + c\}, \qquad (4)$$

5. Finally, the set of equalities implied by $\mathcal{E}_1$ and $\mathcal{E}_3$ is the set of equalities implied by ϕ:

$$\mathcal{E} \doteq \{x = y \mid x \in X, y \in X, (\mathcal{E}_1 \wedge \mathcal{E}_3) \Rightarrow x = y\} \qquad (5)$$

Before proving the correctness of the equality generating algorithm (Theorem 2), we first state and prove a few intermediate lemmas.

For a set of linear arithmetic constraints $A \doteq \{e_1, \ldots, e_n\}$, we define a *linear combination* of A to be a summation $\sum_{e_j \in A} c_j.e_j$, such that each $c_j \in \mathbb{Q}$ and non-negative.

Lemma 2. *Let ϕ_A and ϕ_B be two sets of linear arithmetic constraints over variables in A and B respectively. If u is a linear arithmetic term over $A \setminus B$ and v is a linear arithmetic term over B such that $\phi_A \wedge \phi_B \Rightarrow u \bowtie v$, then there exists a term t over $A \cap B$ such that*

1. $\phi_A \Rightarrow u \bowtie t$, and
2. $\phi_B \Rightarrow t \bowtie v$,

where $\bowtie$ is either $\leq$ or $\geq$.

For the set of satisfiable difference constraints $\phi_D \doteq \{e_1, \ldots, e_n\}$, we say a linear combination $\sum_{e_j \in \phi_D} c_j.e_j$ contains a *cycle* (respectively, a *path* from x to y), if there exists a subset of constraints in ϕ_D with positive coefficients (i.e. $c_j > 0$), such that they form a cycle (respectively, a path from x to y) in the graph induced by ϕ_D.

Lemma 3. *For any term t over D, if $\phi_D \Rightarrow t \leq 0$, then there exists a linear derivation of $t \leq 0$ that does not contain any cycles.*

Lemma 4 (Difference-Bounds Lemma). *Let $x, y \in D \setminus Q$, t be a term over Q, and ϕ_D a set of difference constraints.*

1. *If $\phi_D \Rightarrow x \bowtie t$, then there exists terms $u_1, u_2, \ldots, u_n$ such that all of the following are true*
   (a) Each u_i is of the form $x_i + c_i$ for a variable $x_i \in Q$ and a constant c_i,
   (b) $\phi_D \Rightarrow \bigwedge_i x \bowtie u_i$, and
   (c) $\phi_D \Rightarrow 1/n. \sum_i u_i \bowtie t$
2. *If $\phi_D \Rightarrow x - y \bowtie t$, then there exists terms $u_1, u_2, \ldots, u_n$ such that all of the following are true*
   (a) Each u_i is either of the form c_i or $x_i - y_i + c_i$ for variables $x_i, y_i \in Q$ and a constant c_i,
   (b) $\phi_D \Rightarrow \bigwedge_i x - y \bowtie u_i$, and
   (c) $\phi_D \Rightarrow 1/n. \sum_i u_i \bowtie t$

where $\bowtie$ is one of $\leq$ or $\geq$.

The proof makes use of a novel trick to split a linear combination of difference constraints to yield the desired results.

Lemma 5 (Sandwich Lemma). *Let $l_1, l_2, \ldots l_m$ and $u_1, u_2, \ldots u_n$ be terms such that $\bigwedge_{i,j} l_i \leq u_j$. Let $l_{avg} = 1/m. \sum_i l_i$ and $u_{avg} = 1/n. \sum_j u_j$ be the respective average of these terms. If l and u are terms such that $l \leq l_{avg}$ and $u_{avg} \leq u$, then*

$$l = u \Rightarrow \bigwedge_{i,j} l_i = u_j = l$$

Now, we can prove the correctness of the equality propagation algorithm.

Theorem 2. *For two variables $x \in X$ and $y \in X$, $\phi \Rightarrow x = y$ if and only if $x = y \in \mathcal{E}$.*

Proof. Case 1: The easiest case to handle is the case when both $x, y \in L$. Thus, $(\exists D \setminus L : \phi) = \phi_Q \wedge \phi_L \Rightarrow x = y$. Therefore, the equality $x = y$ is present in $\mathcal{E}_1$ and thus in $\mathcal{E}$.

Case 2: Consider the case when one of the variables, say, $x \in D \setminus L$ while $y \in L$. We have $\phi \Rightarrow x \leq y \wedge x \geq y$. Applying Lemma 2 twice, there exist terms $t, t' \in Q$ such

$$\phi_D \Rightarrow x \leq t \wedge x \geq t' \tag{6}$$

$$\phi_L \Rightarrow t \leq y \wedge t' \geq y \tag{7}$$

However, $\phi_D \wedge \phi_L \Rightarrow x = y = t = t'$. As $t, t' \in Q$, we have

$$\phi_Q \wedge \phi_L \Rightarrow t = t' = y \tag{8}$$

Using Lemma 4.1 twice on Equation 6, there exist terms $u_1, \ldots, u_m$ and terms $l_1, \ldots, l_n$ all of the form $v + c$ for a variable $v \in Q$ and a constant c such that

$$\phi_D \Rightarrow \left(\bigwedge_i x \leq u_i \wedge 1/m. \sum_i u_i \leq t \right) \wedge \left(\bigwedge_j x \geq l_j \wedge 1/n. \sum_j l_j \geq t' \right)$$

As the terms u_i and l_j are terms over Q, we have

$$\phi_Q \Rightarrow \left(\bigwedge_{i,j} l_j \leq u_i \right) \wedge \left(1/m. \sum_i u_i \leq t \right) \wedge \left(1/n. \sum_j l_j \geq t' \right)$$

Using Lemma 5 and Equation 8, we have

$$\phi_Q \wedge \phi_L \Rightarrow \bigwedge_{i,j} l_j = u_i = t = t' = y$$

All of the above equalities belong to $\mathcal{E}_1$. Moreover, the equalities between l_j and u_i are present in $\mathcal{E}_2$. Thus, the equality $x = l_j = u_i$ is present in $\mathcal{E}_3$. Thus $x = y$ is in $\mathcal{E}$.

Case 3: The final case involves the case when x, y are both in $D \setminus L$. The proof is similar to Case 2. We have $\phi \Rightarrow x - y \leq 0 \wedge x - y \geq 0$. Applying Lemma 2 twice, there exists terms $t, t' \in Q$ such

$$\phi_D \Rightarrow x - y \leq t \wedge x - y \geq t' \tag{9}$$

$$\phi_L \Rightarrow t \leq 0 \wedge t' \geq 0 \tag{10}$$

However, $\phi_D \wedge \phi_L \Rightarrow x - y = 0 = t = t'$. As $t, t' \in Q$, we have

$$\phi_Q \wedge \phi_L \Rightarrow t = t' = 0 \tag{11}$$

Using Lemma 4.2 twice on Equation 9, there exists terms $u_1, \ldots, u_m$ and terms $l_1, \ldots, l_n$ all of the form $u - v + c$ for variables $u, v \in Q$ and a constant c such that

$$\phi_D \Rightarrow \left(\bigwedge_i x - y \leq u_i \wedge 1/m. \sum_i u_i \leq t \right) \wedge \left(\bigwedge_j x - y \geq l_j \wedge 1/n. \sum_j l_j \geq t' \right)$$

As the terms u_i and l_j are terms over Q, we have

$$\phi_Q \Rightarrow \left(\bigwedge_{i,j} l_j \leq u_i\right) \wedge \left(1/m. \sum_i u_i \leq t\right) \wedge \left(1/n. \sum_j l_j \geq t'\right)$$

Using Lemma 5 and Equation 11, we have

$$\phi_Q \wedge \phi_L \Rightarrow \bigwedge_{i,j} l_j = u_i = t = t' = 0$$

All of the above equalities belong to $\mathcal{E}_1$. Moreover, the equalities between l_j and u_i are present in $\mathcal{E}_2$. Thus, the equality $x = l_j = u_i$ is present in $\mathcal{E}_3$. Thus $x = y$ is in $\mathcal{E}$.

4.2 Equality Generation with NCD and Simplex

In this section, we describe an instantiation of the SLA framework, where we use the Simplex algorithm for solving general linear arithmetic constraints. The Simplex algorithm [6] (although has a worst case exponential complexity) remains one of the most practical methods for solving linear arithmetic constraints, when the variables are interpreted over rationals. Although Simplex is incomplete for integers, various heuristics have been devised to solve most integer queries in practice [7].

The main contribution of this section is to show how to generate all equalities with offsets between a pair of variables, i.e. all the $x = y + c$ equalities implied by a set of linear constraints. The implementation of Simplex in Simplify [7] can generate all possible $x = y$ equalities implied by a set of constraints. We show that the same Simplex implementation also allows generating all $x = y + c$, without any additional overhead.[1] Due to space constraints, we only provide an informal high-level description of the algorithm. Details and proofs can be found in an extended technical report [16]. Finally, we also mention how to derive $x = y + c$ equalities from a set of difference constraints using NCD algorithms. Proof generation (for contradictions and the implied equalities with offsets) in Simplex is an easy adaptation of existing proof-generating Simplex [18].

Simplex Tableau. A *Simplex tableau* is used to represent a set of linear arithmetic constraints. Each linear inequality is first converted to linear equality by the introduction of a *slack variable*, which is restricted to be non-negative. The Simplex tableau is a two-dimensional matrix that consists of the following:

- Natural numbers n and m for the number of rows and columns for tableau respectively,
- The identifiers for the rows $y[0], \ldots, y[n]$ and the columns $x[1], \ldots, x[m]$. The column 0 corresponds to the constant column. We use u, u_1 etc. to range over the row and column identifiers.
- A two dimensional array of rational numbers $a[0, 0], \ldots, a[n, m]$.

[1] In fact, readers familiar with the Simplify work [7] can see that Lemma 4 in Section 8 of [7], almost immediately generalizes to give us the desired result.

– A subset of identifiers (representing the slack variables) in $y[0], \ldots, y[n]$, $x[1], \ldots, x[m]$ have a *sign* $\in \{\geq, *\}$, and are called *restricted.* A variable u with *sign* of $*$ is called $*$*-restricted*, and denotes that $u = 0$; otherwise a restricted variable u with sign $\geq$ denotes $u \geq 0$.
– The $y[0]$ of the Simplex tableau is a special row *Zero* to denote the value 0, and has 0 in all columns.

Each row in the tableau represents a *row constraint* of the form:

$$y[i] = a[i, 0] \; + \; \Sigma_{1 \leq j \leq m} a[i, j].x[j] \tag{12}$$

A *feasible* tableau is one where the solution obtained by setting each of the column variables $x[j]$ to 0 and setting each of the $y[i]$ to $a[i, 0]$, satisfies all the constraints (row constraints and sign constraints). A set of constraints is satisfiable iff such a feasible tableau exists. We will not go into the details of finding the feasible tableau, as it is a well-known method [6,7].

Equality Generation from Simplex Tableau. To generate equalities implied by the set of constraints, the tableau has to be constrained further in addition to being feasible. The tableau has to be constrained such that for any restricted variable u, the set of constraints imply $u = 0$, if and only if u is $*$*-restricted* in the tableau. Such a tableau is called a *minimal* tableau. The Simplex implementation in Simplify [7] provides a procedure for obtaining a minimal tableau for a set of constraints. The set of all implied variable equalities (of the form $u_1 = u_2$) can be simply read off the minimal tableau. We show that, in fact, the set of all implied offset equalities (of the form $u_1 = u_2 + c$) can also be read off such a minimal tableau. The basic idea is that in a minimal tableau, the implied equalities do not depend on the $\geq$ sign constraints.

We now state the generalization of Lemma 2 (Section 8.2 [7]) to include offset equalities:

Lemma 6 (Generalization of Lemma 2 in Section 8.2 [7]). *For any two variables u_1 and u_2 in a feasible and minimal tableau, the set of constraints imply $u_1 = u_2 + c$, where c is a rational constant, if and only if at least one of the following conditions hold:*

1. *u_1 and u_2 are both $*$-restricted columns (here c is 0), or*
2. *both u_1 and u_2 are row variables $y[i]$ and $y[j]$ respectively, and apart from the $*$-restricted columns only (possibly) differ in the constant column, such that $a[i, 0] = a[j, 0] + c$, or*
3. *u_1 is a row variable $y[i]$, u_2 is a column variable $x[j]$, and the only non-zero entries in the row i outside the $*$-restricted columns are $a[i, 0] = c$ and $a[i, j] = 1$.*
4. *u_2 is a $*$-restricted column, and u_1 is a row variable $y[i]$, such that $a[i, 0] = c$ is the only non-zero entry outside $*$-restricted columns in row i.*

Therefore, obtaining the minimal tableau is sufficient to derive even $x = y + c$ facts from Simplex. This is noteworthy because the Simplex implementation does not incur any more overhead in generating these more general equalities than simple $x = y$ equalities.

Inferring Equalities from NCD. The algorithm for SLA equality generation described in Section 4.1 requires generating equalities of the form $x = y + c$ from the NCD component of SLA. Lemma 2 in [15] provides such an algorithm. The lemma is provided here.

Lemma 7 (Lemma 2 in [15]). *For an edge e in G_ϕ representing $y \leq x + c$, e can be strengthened to represent $y = x + c$ (called an equality-edge), if and only if e lies in a cycle of weight zero.*

Hence, using Lemma 6, Theorem 2 and Lemma 7, we obtain a complete equality generating decision procedure over rationals.

Theorem 3. *The SLA implementation by combining NCD and Simplex is an equality generating decision procedure for linear arithmetic over rationals.*

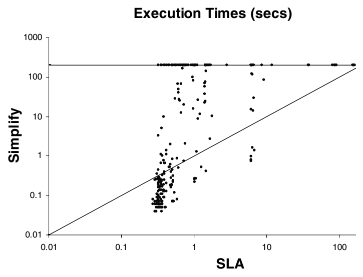
5 Implementation and Results

In this section, we describe our implementation of the SLA algorithm in the Zap [1] theorem prover and report preliminary results from our experiments. The implementation uses the Bellman-Ford algorithm as the NCD algorithm and the Simplex implementation (described in Section 4.2) for the non-difference constraints. We are currently working on the implementation of the proof generation from the SLA algorithm (namely, the proof of implied equalities from NCD [15]) , to integrate it into the lazy proof-generating theorem prover framework [2,8]. Hence, we are currently unable to evaluate our algorithm on more realistic benchmarks (such as the SMT-LIB benchmarks [26]), where we need the proofs to generate conflict clauses to reason about the Boolean structure in the formula. Instead, we evaluate on a set of randomly generated linear arithmetic benchmarks.

We report preliminary results comparing our algorithm with two different implementations for solving linear arithmetic constraints: (i) *Simplify-Simplex*: the linear arithmetic solver in the Simplify [7] theorem prover, and (ii) *Zap-UTVPI*: an implementation of Unit Two Variable Per Inequality (UTVPI) decision procedure [10,12] in Zap.[2] Even though *Zap-UTVPI* is not complete for general linear arithmetic, we chose this implementation to compare a transitive closure based decision procedure (as used by Sheini and Sakallah [25]) to a one based on NCD algorithms.

We generated the random benchmarks as follows. For different values for the total number of variables lying between 100 and 1000, we generated benchmarks with the number of constraints varying from half to five times the number of variables. To measure the effect of the sparseness of the constraints, we varied the ratio of non-difference constraints to difference constraints from 2% to 50%. For each difference constraint we picked the two variables at random. For each non-difference constraint we randomly picked 2 to 5 variables and chose a random

[2] UTVPI constraints are of the form $a.x + b.y \leq c$, where a and $b \in \{-1, 0, 1\}$ and c is an integer constant.



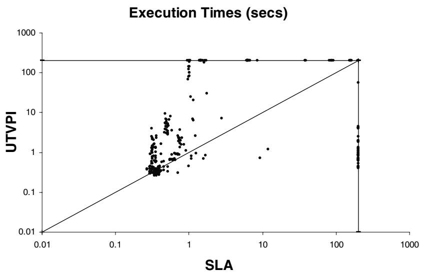


Fig. 1. Comparison of SLA with (a) *Simplify-Simplex* and (b) *Zap-UTVPI* on a set of randomly generated benchmarks

coefficient between −2 and 2. We ensured that the set of benchmarks when run on the SLA implementation involved all of the following: instances where the difference constraints alone were unsatisfiable, instances where the non-difference constraints alone were unsatisfiable, instances that required *both* difference and non-difference reasoning, and finally instances that were satisfiable.

Figure 1 (a) shows the comparison of the execution times of the SLA algorithm against *Simplify-Simplex.* In the graph, we indicate both the runs that took greater than 200 seconds and runs that incurred a crash due to an integer-overflow exception, as timeouts with 200 seconds. The overflow exception happens in Simplex (both in Simplify and Zap) due to the use of machine integers to represent large coefficients in the tableau. The following observations are evident from this graph. On those instances for which Simplify finished within a second, the SLA algorithm also finished within a second, but performed worse than Simplify. This is a result of the constant overhead Zap (implemented in C#) incurs loading the virtual machine of the C# language on every run. On the other hand, SLA solved instances within seconds for which Simplify required orders of magnitude longer time or timed out at 200 seconds. To our surprise, Simplify incurred an integer-overflow exception on many benchmarks for which pure difference reasoning was sufficient to prove the unsatisfiability of the query. The SLA implementation did incur an integer-overflow on certain instances for which Simplify completed successfully. This could be due to the fact that our Simplex implementation is not as optimized as the one in Simplify as we have not implemented the many pivot heuristics of Simplify.

Figure 1 (b) shows the execution time of the UTVPI decision procedure on these benchmarks. SLA performs better than the UTVPI decision procedure on a greater proportion of the instances. The transitive-closure based algorithm for the UTVPI decision procedure has a quadratic space complexity, resulting in orders of magnitude slowdown. There are instances, however, where the SLA algorithm results in an integer-overflow for which the UTVPI algorithm terminates. (Note, the UTVPI algorithm is incomplete for general linear arithmetic.) This suggests a possibility of combining the linear-space UTVPI algorithm [14] with a general linear arithmetic solver, along the lines of SLA. While this is an interesting problem for future work, we are unsure about its value in practice.

6 Related Work

Checking the satisfiability of a set of linear arithmetic constraints over integers is NP-complete [20]. Various algorithms based on branch-and-bound heuristics are implemented in various integer linear programming (ILP) solvers like LP_SOLVE [17] and commercial tools like CPLEX [11] to solve this fragment. These algorithms have a worst-case exponential time complexity. Even for the relaxation of the linear arithmetic problem over rationals (where polynomial time decision procedures exists [13]), most practical solvers use Simplex [6] algorithm that has a worst-case exponential complexity. Gomory cuts [23] can be used to extend Simplex over integers although the algorithm might require exponential space in the worst case. Ruess and Shankar [22] provide one such implementation. Their algorithm also generates equalities over variables. However, unlike our approach, their algorithm does not try to exploit the sparsity in linear arithmetic constraints, and the asymptotic complexity for solving sparse linear arithmetic constraints is still exponential.

Recently attempts have been made to exploit the sparsity in linear arithmetic constraints mostly dominated by difference logic queries. Seshia and Bryant [24] demonstrate that although one might incur a linear blowup for translating a Boolean formula over linear arithmetic constraints (over integers) to an equisatisfiable propositional formula, formulas with only a small number of non-difference constraints can be converted using a logarithmic blowup. This approach however does not help towards improving the complexity of solving a set of linear arithmetic constraints.

The closest approach to ours is the approach of Sheini and Sakallah [25], where they provide a decision procedure for integer linear arithmetic by combining a decision procedure for UTVPI constraints and a general linear arithmetic solver (CPLEX [11] in their case). Their algorithm relies on computing a transitive closure for the UTVPI constraints that incurs cubic time and quadratic space complexity, independent of the sparsity of the constraints. In contrast, our decision procedure retains the efficiency of the NCD algorithms thereby making our procedure robust even for non sparse linear arithmetic benchmarks. This is well demonstrated by our experimental results (Figure 1 (b)). Moreover, their combination does not generate models for satisfiable formulas. Finally, their algorithm does not provide a way to generate implied equalities that are crucial for a Nelson-Oppen framework.

References

1. T. Ball, S. K. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, LNCS 3835, pages 2–22, 2005.
2. C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 236–249, 2002.
3. R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

4. B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, 1996.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
6. G. Dantzig. *Linear programming and extensions.* Princeton University Press, Princeton NJ, 1963.
7. D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, HPL-2003-148, 2003.
8. C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem proving using lazy proof explication. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 355–367, 2003.
9. L. R. Ford, Jr., and D. R. Fulkerson. *Flows in Networks.* 1962.
10. W. Harvey and P. J. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *Proceedings of the 20th Australasian Computer Science Conference (ACSC '97)*, pages 102–111, 1997.
11. ILOG CPLEX. Available at `http://ilog.com/products/cplex`.
12. J. Jaffar, M. J. Maher, P. J. Stuckey, and H. C. Yap. Beyond finite domains. In *PPCP 94: Principles and Practice of Constraint Programming*, LNCS 874, pages 86–94, 1994.
13. Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.
14. S. K. Lahiri and M. Musuvathi. An efficient decision procedure for UTVPI constraints. In *FroCos 05: Frontiers of Combining Systems*, LNCS 3717, pages 168–183, 2005.
15. S. K. Lahiri and M. Musuvathi. An Efficient Nelson-Oppen Decision Procedure for Difference Constraints over Rationals. Number 2 in ENTCS 144, pages 27—41, 2005.
16. S. K. Lahiri and M. Musuvathi. Solving sparse linear constraints. Technical Report MSR-TR-2006-47, Microsoft Research, 2006.
17. LP_SOLVE. Available at `http://groups.yahoo.com/group/lp_solve/`.
18. G. C. Necula and P. Lee. Proof generation in the touchstone theorem prover. In *Conference on Automated Deduction*, LNCS 1831, pages 25–44, 2000.
19. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.
20. C. H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.
21. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, Cambridge, Mass., September 1977.
22. H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, January 2004.
23. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.
24. S. A. Seshia and R. E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. In *LICS 04: Logic in Computer Science*, pages 100–109, July 2004.
25. H. M. Sheini and K. A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *Theory and Applications of Satisfiability Testing (SAT 2005)*, LNCS 3569, pages 241–256, 2005.
26. SMT-LIB: The Satisfiability Modulo Theories Library. Available at `http://combination.cs.uiowa.edu/smtlib/`.