Solving Sparse Linear Constraints

Shuvendu K. Lahiri Madanlal Musuvathi

April 19, 2006

Technical Report
MSR-TR-2006-47

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

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 [20] 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 [18]. 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.

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 [18] 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 [17]. 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 [19]. 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 graph $G(V, E)$, the problem of determining if G has a cycle C, such that sum of 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.

3 Sparse Linear Arithmetic (SLA) Constraints

Pratt [20] observed that most queries that arise in software verification are dominated by difference constraints. Recently, more evidence has been presented strengthening the hypothesis [23], 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 length 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.*

Proof of the lemma can be found in Section B.1 in the Appendix. 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 $\phi_Q \Leftrightarrow (\exists P : \phi_D)$.*

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, $\phi_D \Rightarrow \psi$ 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 $\phi_Q \wedge \phi_L$
$$\mathcal{E}_1 \doteq \{x = y + c \mid x \in L, y \in L, \text{and}\,(\phi_Q \wedge \phi_L) \Rightarrow x = y + c\}, \tag{2}$$
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:
$$\mathcal{E}_2 \doteq \{x = y + c \mid x \in Q, y \in Q, x = y + c \in \mathcal{E}_1\ \}, \tag{3}$$
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:
$$\mathcal{E}_3 \doteq \{x = y + c \mid x \in D, y \in D, (\phi_D \wedge \mathcal{E}_2) \Rightarrow x = y + c\}, \tag{4}$$
5. Finally, the set of equalities implied by $\mathcal{E}_1$ and $\mathcal{E}_3$ is the set of equalities implied by ϕ:
$$\mathcal{E} \doteq \{x = y \mid x \in X, y \in X, (\mathcal{E}_1 \wedge \mathcal{E}_3) \Rightarrow x = y\} \tag{5}$$

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* $\geq$.

Proof can be found in Section B.2 in the Appendix.

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. $c_j > 0$) in the derivation, 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.*

The proof can be found in Section B.3 of Appendix.

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 detailed proof of this lemma can be found Section B.4 in the Appendix. 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*

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

Proof can be found in Section B.5 in Appendix.

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 $\phi \Rightarrow x \le y \wedge x \ge y$. Applying Lemma 2 twice, there exists terms $t, t' \in Q$ such

$$\phi_D \Rightarrow x \le t \wedge x \ge t' \tag{6}$$
$$\phi_L \Rightarrow t \le y \wedge t' \ge y \tag{7}$$

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 exists 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

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

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

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

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 $\phi \Rightarrow x - y \le 0 \wedge x - y \ge 0$. Applying Lemma 2 twice, there exists terms $t, t' \in Q$ such

$$\phi_D \Rightarrow x - y \le t \wedge x - y \ge t' \tag{9}$$
$$\phi_L \Rightarrow t \le 0 \wedge t' \ge 0 \tag{10}$$

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

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

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 betwen 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 arithemtic 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. Readers familiar with the work can see that Lemma 4 in Section 8 of [7], almost immediately generalizes to give us the desired result. For lack of space, we only present enough description to state the generalization of the lemma; a more complete description is present in the Appendix. We refer the readers to Section 8 in [7] for complete details the Simplex implementation in Simplify. Finally, we also mention how to derive $x = y + c$ equalties from a set of difference constraints using NCD algorithms.

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.* The Simplex tableau is a two-dimnesional matrix that consists of the following:

– Natural numbers n and m for the number of rows and columns for tableau respectively, and a column *dcol*, where $0 \leq dcol \leq m$,
– The identifiers for the rows $y[0], \ldots, y[n]$ and the identifiers for these columns $x[1], \ldots, x[m]$, where each row or column identifier corresponds to a variable in the constraints, including the slack variables. The column 0 corresponds to the constant column. We use u, u_1 etc. to range over the row and column identifiers.

– A two dimenstional array of rational numbers $a[0,0],\ldots,a[n,m]$.
– A subset of identifiers in $y[0],\ldots,y[n],x[1],\ldots,x[m]$ also have a *sign* $\geq$.
– 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:

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

For any identifier u with a sign $\geq$, the *sign constraint* represents $u \geq 0$. Such an identifier u is said to be *restricted*. Finally, for all $1 \leq j \leq dcol$, $x[j] = 0$, represents the *dead column constraints*.

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. 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. Two variables (row or column) u_1 and u_2 are defined to be *manifestly equal* in the tableau, if and only if either (i) both u_1 and u_2 are row variables and their rows are identical except for the dead columns, or (ii) both u_1 and u_2 are dead columns, or (iii) u_1 is a row variable $y[i]$ and u_2 is a column variable $x[j]$ and $a[i,j] = 1$ is the only non-zero entry for row i outside the dead columns, or (iv) one of u_1 or u_2 is dead column variable, and the other is a row variable whose all non dead column entries are 0.

A tableau is *minimal* if and only if every row or column variable u is either manifestly equal to *Zero* or has a positive value in some solution. The Simplex implementation in Simplify [7] provides a procedure for obtaining a minimal tableau for a set of constraints. It is outside the scope of this work to describe the details of the algorithm. We now state the generalization of Lemma 2 (Section 8.2 [7]) that allows us to extend equality generation to include offsets:

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 manifestly equal (in this case $c = 0$), or*
2. *both u_1 and u_2 are row variables $y[i]$ and $y[j]$ respectively, and apart from the dead columns only 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 row i are $a[i,0] = c$ and $a[i,j] = 1$.*

Proof can be found in Section B.6 of Appenidx. 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]). *An edge e in G_ϕ representing $y \leq x + c$, e_i 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, 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 [25]), 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. [1] 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 [24]) 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 coefficient between -2 and 2. We ensured that the set of benchmarks when run

[1] 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.

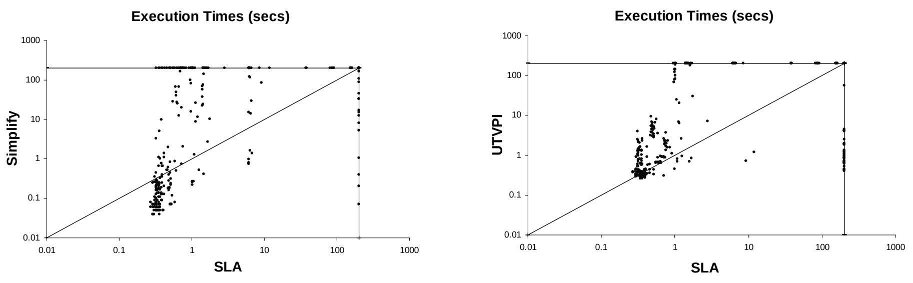

Fig. 1. Comparison of SLA with (a) *Simplify-Simplex* and (b) *Zap-UTVPI* on a set of randomly generated benchmarks.

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 in NP-complete [19]. Various algorithms based on branch-and-bound heuristics are implemented in various integer linear programming (ILP) solvers like LP_SOLVE [16] 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 [22] can be used to extend Simplex over integers although the algorithm might require exponential space in the worst case. Ruess and Shankar [21] 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 [23] 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-differnce 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 [24], 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. *Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005)*, 144(2):27—41, 2005.
16. LP_SOLVE. Available at `http://groups.yahoo.com/group/lp_solve/`.
17. G. C. Necula and P. Lee. Proof generation in the touchstone theorem prover. In *Conference on Automated Deduction*, LNCS 1831, pages 25–44, 2000.
18. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.
19. C. H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.
20. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, Cambridge, Mass., September 1977.
21. H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, January 2004.
22. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.
23. 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.
24. 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.
25. SMT-LIB: The Satisfiability Modulo Theories Library.
26. G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *CADE 05: Conference on Automated Deduction*, LNCS 3632, pages 353–368, 2005.

A Graph formalisms

Let $G(V, E)$ be a directed graph with vertices V and edges E. For each edge $e \in E$, we denote $s(e)$, $d(e)$ and $w(e)$ to be the source, destination and the weight of the edge. A *path* P in G is a sequence of edges $[e_1, \ldots, e_n]$ such that $d(e_i) = s(e_{i+1})$, for all $1 \leq i \leq n-1$. For a path $P \doteq [e_1, \ldots, e_n]$, $s(P)$ denotes $s(e_1)$, $d(P)$ denotes $d(e_n)$ and $w(P)$ denotes the sum of the weights on the edges in the path, i.e. $\sum_{1 \leq i \leq n} w(e_i)$. A cycle C is a sequence of edges $[e_1, \ldots, e_n]$ where $s(e_1) = d(e_n)$. We use $u \rightsquigarrow v$ in E to denote that there is a path from u to v through edges in E.

B Proofs of theorems and lemmas

B.1 Proof of Lemma 1

Proof. First, observe that for any variable $x \in Q$, $\rho_D(x) = \rho_L(x)$, since ρ_D has to satisfy $\bigwedge_{x \in Q} (x = \rho_L(x))$. Hence, if we can show that in Step 4, $\phi_D \wedge \bigwedge_{x \in Q} (x = \rho_L(x))$ is satisfiable, then ρ_X satisfies ϕ_D and ϕ_L.

Let us assume that $\psi \doteq \left[\phi_D \wedge \bigwedge_{x \in Q} (x = \rho_L(x))\right]$ is unsatisfiable. Consider the graph $G_\psi(V, E)$ induced by the formula ψ (as described in Section 2): First, we add an edge for each constraint in ϕ_D. Secondly, each constraint $x = \rho_L(x)$ is broken up into a pair of constraints $x - x_{orig} \leq \rho_L(x)$ and $x_{orig} - x \leq -\rho_L(x)$, and then the corresponding edges are added to the graph. Let $T \subseteq E$ be the set of edges in G_ψ resulting from the addition of edges for all the $x = \rho_L(x)$ constraints.

Since ψ is unsatisfiable, there has to be a (simple) cycle C with negative weight in the graph. Moreover, we know that ϕ_D is satisfiable. Therefore, the negative cycle contains at least one edge from T. Since C is a simple cycle, at most two edges from T can be present in C. Let the edges in C be $[(x_{orig}, x_1), (x_1, x_2), \ldots, (x_k, x_{orig})]$. Let $w_1, w_2, \ldots, w_{k+1}$ be the weights of these edges in C, such that $\sum_i w_{i \in [1,k+1]} < 0$.

Let us first assume that C contains *exactly* one edge $t \in T$, such that either (x_{orig}, x_1) or (x_k, x_{orig}) belong to T. Consider the two subcases:

1. Let $(x_k, x_{orig}) \in T$, representing the constraint $x_{orig} - x_k \leq w_{k+1}$. This means that the constraint $x_{orig} - x_k = w_{k+1}$, or otherwise $\rho_L(x_k) = -w_{k+1}$, is present in ψ. Since C forms a negative cycle, $\sum_i w_{i \in [1,k]} < -w_{k+1}$. Therefore, there is a path from x_{orig} to x_k in G_ψ (without any T edges) with a weight less that $-w_{k+1}$. This implies that the constraint $x_k - x_{orig} < -w_{k+1}$, or equivalently $x_k < -w_{k+1}$ was implied by ϕ_D, and therefore was implied by ϕ_Q. Since ρ_L satisfies ϕ_Q, this leads to a contradiction.
2. Let $(x_{orig}, x_1) \in T$, meaning that $\rho_L(x_1) = w_1$. By a reasoning similar to the previous subcase, we can show that the graph G_ψ (without any T edges) implies the constraint $x_1 > w_1$, which leads to a contradiction.

Now consider the case when both (x_{orig}, x_1) and (x_k, x_{orig}) belong to T. This means that $\rho_L(x_1) = w_1$, and $\rho_L(x_k) = -w_{k+1}$. However, C implies that there is a path from x_1 to x_k of length $\sum_i w_{i\in[2,k]} < -w_1 - w_{k+1}$, implying $x_k - x_1 < -w_1 - w_{k+1}$. This contradicts with the assignment ρ_L, since ϕ_Q should imply $x_k - x_1 < -w_1 - w_{k+1}$.

This completes the proof.

B.2 Proof of Lemma 2

Proof. This lemma is a variation of the lemma that appears in [26]. Accordingly, the proof is similar. Let us consider the case when $\bowtie$ is $\leq$ (the reasoning is the same for $\geq$). Let $\phi_A \doteq \{e_1, \ldots, e_n\}$ and $\phi_B \doteq \{e'_1, \ldots, e'_m\}$ be the set of constraints. As $\phi_A \wedge \phi_B \Rightarrow u \leq v$, there exists a set of non-negative constants $\{c_1, \ldots, c_n, c'_1, \ldots, c'_m\}$ such that

$$u \leq v \equiv \sum_{e_j\in\phi_A} c_j.e_j + \sum_{e'_j\in\phi_B} c'_j.e'_j \tag{13}$$

Since the set of constraints in ϕ_A and ϕ_B only share variables over $A \cap B$, it is easy to see that $\sum_{e_j\in\phi_A} c_j.e_j \equiv (u - t \leq 0)$ and $\sum_{e'_j\in\phi_B} c'_j.e'_j \equiv (-v + t \leq 0)$, where t is a linear term over $A \cap B$. Therefore $\phi_A \Rightarrow u \leq t$ and $\phi_B \Rightarrow t \leq v$.

B.3 Proof of Lemma 3

Proof. Let us assume that the derivation of $t \leq 0$ from ϕ_D contains $A \doteq \{e'_1, \ldots, e'_m\} \subseteq \phi_D$, that appear with non-zero coefficients $c'_1, \ldots, c'_m$ respectively, and $[e'_1, \ldots, e'_m]$ forms a cycle in the graph induced by ϕ_D.

Since ϕ_D is satisfiable, we know that any cycle in the graph has a non-negative weight. Therefore, $\sum_{e_j\in A} c'_j.e'_j \equiv 0 \leq d$, for some $d \geq 0$. Hence, we can derive $t \leq 0$, by reducing the coefficients of each $e'_j \in A$ by the minimum c'_j in this set and adding the constraint $0 \leq 1$ with the same coefficient. We can repeat this process until we do not have any cycles in the derivations.

B.4 Proof of Lemma 4

Proof. We will illustrate the proof for the cases when $\bowtie$ is $\leq$, the case for $\geq$ is similar. Let us first make two observations that will be used for proving both parts of the lemma:

1. It is well known [22], that for a set of linear contraints $\varphi \doteq \{e_1, \ldots, e_k\}$ over $\mathbb{Q}$, and a linear arithmetic constraint ψ, $\varphi \Rightarrow \psi$ if and only if there exists a set of non-negative constants $\{m_1, \ldots, m_k\}$ in $\mathbb{Q}$, such that

$$\psi \equiv \sum_{e_i\in\varphi} m_i.e_i$$

This means that if $\varphi \Rightarrow \psi$, then there exists a set of non-negative integer constants $\{n_1, \ldots, n_k\}$ and a non-negative constant n such that

$$n.\psi \equiv \sum_{e_i \in \varphi} n_i.e_i$$

Let p_i/q_i be the rational representation of m_i. The constant n is simply the least common multiple of $q_1, \ldots, q_k$, and each n_i is $n.p_i/q_i$.

2. For the set of difference constraints $\phi_D \doteq \{e_1, \ldots, e_k\}$, we can represent an integer linear combination of the constraints $\sum_{e_i \in \phi_D} n_i.e_i$ (where each n_i is a non-negative constant in $\mathbb{Z}$), as a multi-graph G, where there are n_i (possibly 0) copies of the edge corresponding to e_i. For any vertex $z \in D$, the *indegree*(z) is the number of edges of the form $z - w_i \le c_i$, where $w_i \in D$; similarly, the *outdegree*(z) is the number of edges of the form $w_i - z \le c_i$ in the graph.

Case 1: Now let us look at the proof of case 1 of the lemma. Let us assume that $\phi_D \Rightarrow x \le t$, for some $x \in D \setminus Q$. Therefore there exists non-negative integer constants $\{n_1, \ldots, n_k\}$ and n, such that

$$\left(\sum_{e_i \in \phi_D} n_i.e_i \right) \equiv (n.x - n.t \le 0) .$$

Now, let us look at the multigraph G induced by the linear combination $\sum_{e_i \in \phi_D} n_i.e_i$. It is not hard to see that the following statements are true for this graph G:

- For any vertex $z \in (D \setminus Q) \setminus \{x\}$, *indegree*$(z) =$ *outdegree*(z).
- For the vertex x, *indegree*$(x) -$ *outdegree*$(x) = n$.
- Since ϕ_D is satisfiable and we have assumed that the derivation is irredundant, we can assume (using Lemma 3) that there are no cycles in this multigraph.

We will now describe a process of iteratively enumerating a set of n paths from the graph G, where each path corresponds to a constraint $x \le x_i + c_i$, where $x_i \in Q$. After each path has been enumerated, the edges in the path are removed from the graph. For each path, we start with an edge $x \le z_1 + d_1$ and extend it until we reach a vertex in Q. This is possible since all the vertices in $(D \setminus Q) \setminus \{x\}$ have equal indegree and outdegree, and there are no cycles. Let the path enumerated have the sequence of edges $x \le z_1 + d_1, z_1 \le z_2 + d_2, \ldots z_l \le x_i + c_i$, where $x_i \in Q$. Observe that the indegree and outdegrees of the intermediate vertices $\{z_i\}$ remain equal even after the path has been removed, and the measure *indegree*$(x) -$ *outdegree*(x) decreases by 1.

By repeating the above process n times, we obtain a set of n paths that sum up to $x \le x_i + c_i$, for $1 \le i \le n$, where $x_i \in Q$. Let G' be the final graph obtained after removing the paths from G. Since the graph G represented the sum $n.x - n.t$, and the n paths represent the sum $n.x - \sum_i (x_i + c_i)$, the constraints in the graph G' represent the sum $\sum_i (x_i + c_i) - n.t$. Moreover since each constraint is ≤ 0, the resultant constraints are ≤ 0.

Hence, we can split the linear combination $\sum_{e_i \in \phi_D} n_i.e_i$ into two linear combinations that represent (a) $x \leq x_i + c_i$, for $1 \leq i \leq n$, such that $x_i \in Q$ and (b) $\sum_i (x_i + c_i) - n.t \leq 0$ or $1/n. \sum_i (x_i + c_i) \leq t$. The terms $x_i + c_i$ are the desired u_is in the case 1 of the Lemma.

Case 2: The second case is similar to the first case in many respect and uses the same methodology to split the linear combination into two linear combinations.

Let $\phi_D \doteq \{e_1, \ldots, e_k\}$. Since $\phi_D \Rightarrow x - y \leq t$, there exists non-negative integer constants $\{n_1, \ldots, n_k\}$ and n such that:

$$\left(\sum_{e_i \in \phi_D} n_i.e_i \right) \equiv (n.x - n.y - n.t \leq 0)\,.$$

Once again, let us look at the graph G induced by the linear comination. This graph now has the following properties:

- For the vertex x, *indegree*(x) − *outdegree*$(x) = n$.
- For the vertex y, *outdegree*(y) − *indegree*$(y) = n$.
- For any other vertex $z \in (D \setminus Q) \setminus \{x, y\}$, *indegree*$(z)$ = *outdegree*(z).
- There are no cycles in the graph G.

We can now enumerate paths from x starting with edges of the form $x \leq z_i + c_i$, until we either reach (i) a vertex $x_i \in Q$ or (ii) y. Similarly, we can enumerate paths (ending at y) from y starting with edges of the form $z_i + c_i \leq y$, until we reach (i) a vertex $y_i \in Q$ or (ii) x.

Let us first enumerate the paths from y to x. Each of these paths represent the constraint $x - y \leq c_i$ for some constant c_i representing the weight of the path. Removal of these paths reduce the *indegree*(x) − *outdegree*(x) and the *outdegree*(y) − *indegree*(y) measure by 1, keeping the *indegree*(z) = *outdegree*(z) for all other $z \in D \setminus Q$. Let there be m such paths between y and x.

From the remainder graph, we can enumerate $n - m$ paths representing $x \leq x_i + c'_i$ and $y_i + d'_i \leq y$ for $1 \leq i \leq n - m$ where $\{x_i, y_i\} \subseteq Q$. For each $1 \leq i \leq n-m$, the sum of the constraints representing $x \leq x_i + c'_i$ and $y_i + d'_i \leq y$ will yield the desired constraints $x - y \leq x_i - y_i + c_i$, where $c_i \doteq (c'_i + d'_i)$. Therefore ϕ_D imply $x - y \leq u_i$, where $u_i \in \{c_i, x_i - y_i + c_i\}$ for $1 \leq i \leq n$. Moreover, since the sum of the constraints removed from G sums upto $n.x - n.y - \sum_i u_i$, the sum of the remaining constraints in the graph would yield $\sum_i u_i - n.t \leq 0$, or $1/n. \sum_i u_i \leq t$.

B.5 Proof of Lemma 5

Proof. Let $l = u$. To prove by contradiction, without loss of generality assume $l_1 < u_1$. Thus, $n.l_1 < \sum_j u_j$ and $n.l_i \leq \sum_j u_j$ for $1 < i \leq m$. Therefore,

$n.\sum_i l_i < m.\sum_j u_j$, which implies that $l < u$, a contradiction. As a result, $l_i = u_j$ for every i, j. Moreover, we have $l_{avg} = u_{avg} = l_1$. Thus, $l = l_1$ proving the lemma.

B.6 Equality generation with NCD and Simplex (Detailed Description)

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 arithemtic 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. Readers familiar with the work can see that Lemma 4 in Section 8 of [7], almost immediately generalizes to give us the desired result. We present some details of the procedure to make this document self-contained.

The rest of this section presents high level details and key lemmas from [7], that will allow us to present the generation of $x = y + c$ equalities from Simplex. We first describe the Simplex *tableau* data structure, the invariants it represents, and how to check for satisfiability of a set of linear constraints using Simplex. We next describe the sufficient condition for generating $x = y$ variable equalities from a *minimal* tableau, and finally present the generalization to generate $x = y + c$ constraints from Simplex. The presentation of this section is a little informal; for more rigorous treatment, we refer the reader to Section 8 of the following work [7].

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*. The Simplex tableau is a two-dimnesional matrix that consists of the following:

– Natural numbers n and m for the number of rows and columns for tableau respectively, and a column *dcol*, where $0 \leq dcol \leq m$,
– The identifiers for the rows $y[0], \ldots, y[n]$ and the identifiers for the columns $x[1], \ldots, x[m]$, where each row or column identifier corresponds to a variable in the constraints, including the slack variables. The column 0 corresponds to the constant column. We use u, u_1 etc. to range over the row and column identifiers.
– A two dimenstional array of rational numbers $a[0, 0], \ldots, a[n, m]$.

– A subset of identifiers in $y[0], \ldots, y[n], x[1], \ldots, x[m]$ also have a *sign* $\geq$.

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

$$y[i] = a[i,0] \ + \ \Sigma_{1 \leq j \leq m} a[i,j] * x[j] \tag{14}$$

For any identifier u with a sign $\geq$, the *sign constraint* represents $u \geq 0$. Such an identifier u is said to be *restricted*. Finally, for all $1 \leq j \leq dcol$, $x[j] = 0$, represents the *dead column constraints*.

The set of solutions to the row constraints and the dead column constraints, together termed as *hyperplane* constraints (i.e. ignoring the sign constraints) is refered to as *hyperplane* solution set (*HPlaneSoln*) and the set of solutions considering all the three types of constraints (together called the *tableau* constraints) is refered to as the *tableau* solution set (*TablSoln*).

Testing Consistency of Simplex Tableau For a tableau T, a row variable $y[i]$ is *manifestly maximized* if every non-zero entry $a[i,j]$ for $j > dcol$, is negative and lies in a column whose variable $x[j]$ is restricted. It is not hard to see that if a tableau is in a state where a restricted row variable $y[i]$ is manifestly maximized, but $a[i,0] < 0$, then the set of constraints is infeasible. Such a tableau is called *infeasible*.

The *sample point* of a tableau corresponds to the solution obtained by setting each of the column variables $x[j]$ to 0 and setting each of the $y[i]$ to $a[i,0]$. If a set of constraints is satisfiable, then there exists a cofiguration of the tableau, where the sample point satisfies belongs to *TablSoln*.

Assuming that we have a feasible tableau for a set of satisfiable constraints, we only need to worry about reaching a feasible tableau after adding a new constraint. If the sample point after adding the new constraint does not satisfy all the sign constraints, then an operation called *pivot* can be repeatedly applied until (a) the sample point satisfies all the sign constraints, or (b) the tableau becomes infeasible (denotes unsatisfiability). The pivot operation swaps a row and a column of the tableau, but preserves each of the constraints. This is the high-level algorithm for checking the satisfiability of a set of constraints using Simplex.

Equality Generation from Simplex Tableau A set of constraints is satisfiable if and only if there is a feasible tableau for the constraints. It is possible to further infer some classes of affine equalities from the Simplex tableau. In addition to being feasible, the tableau needs additional constraints to generate certain classes of equalities.

Two variables (row or column) u_1 and u_2 are defined to be *manifestly equal* in the tableau, if and only if either (i) both u_1 and u_2 are row variables and their rows are identical except for the dead columns, or (ii) both u_1 and u_2 are dead columns, or (iii) u_1 is a row variable $y[i]$ and u_2 is a column variable $x[j]$ and $a[i,j] = 1$ is the only non-zero entry for row i outside the dead columns, or (iv) one of u_1 or u_2 is dead column variable, and the other is a row variable whose all non dead column entries are 0.

Let us first assume that the $y[0]$ of the Simplex tableau is the special row *Zero* to denote the value 0. This row is identically 0 in every configuration of the tableau. Now, we define a tableau to be *minimal* if and only if every row or column variable u is either manifestly equal to *Zero* or has a positive value in some solution in *TablSoln*. The Simplex implementation in Simplify [7] provide an implementation of obtaining a minimal tableau for a set of constraints. It is outside the scope of this work to describe the details of the algorithm. The key lemma that we exploit in this paper for minimal tableau is the following:

Lemma 8 (Lemma 3 in Section 8.2 [7]). *For a feasible and minimal tableau, and any affine function f over the row and column variables,*

$$f = 0 \ \textit{over TablSoln} \Leftrightarrow f = 0 \ \textit{over HPlaneSoln} \tag{15}$$

We now present the generalization of Lemma 2 (Section 8.2 [7]) that allows us to extend equality generation to include offsets:

Lemma 9 (Generalization of Lemma 2 in Section 8.2 [7]). *For any two variables u_1 and u_2 in the tableau, the hyperplane 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 manifestly equal (in this case $c = 0$), or*
2. *both u_1 and u_2 are row variables $y[i]$ and $y[j]$ respectively, and apart from the dead columns only 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 (possibly dead) column variable $x[j]$, and the only non-zero entries in row i are $a[i, 0] = c$ and $a[i, j] = 1$.*

Proof. The key idea of the proof (same as [7]) is that each of the row or column variable can be expressed as an affine combination over the live (non-dead) column variables. For each row $0 \leq i \leq n$:

$$y[i] = a[i, 0] \ + \ \Sigma_{dcol<j\leq m} a[i, j] * x[j]. \tag{16}$$

Similarly, for each live column $dcol < j \leq m$

$$x[j] = 0 \ + \ \Sigma_{dcol<k\leq m} \delta_{j,k} * x[k]. \tag{17}$$

where $\delta_{j,k} = 1$ if $j = k$ and 0 otherwise.

Finally, for any dead column in $0 \leq j \leq dcol$

$$x[j] = 0 \ + \ \Sigma_{dcol<k\leq m} 0 * x[k]. \tag{18}$$

Since the variables in the live column are independent and can assume any arbitrary values, two variables u_1 and u_2 differ by a constant c (including $c = 0$)

if and only if the coefficients of the live column variables are identical in the two variables, and only the constant column differs by c. Note that the Lemma 2 in Section 8.2 [7] is restricted to $c = 0$.

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. The only additional work that has to be done is to identify the pairs u_1 and u_2 that satisfy Lemma 9.

B.7 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 10 (Lemma 2 in [15]). *An edge e in G_ϕ representing $y \leq x + c$, e_i 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 10, Lemma 9 and Theorem 2, we obtain a complete equality generating decision procedure over rationals.

Theorem 4. *The SLA implementation by combining NCD and Simplex is an equality generating decision procedure for linear arithmetic over rationals.*