An Efficient Nelson-Oppen Decision Procedure for Difference Constraints over Rationals

Shuvendu K. Lahiri Madanlal Musuvathi

May 26, 2005

Technical Report
MSR-TR-2005-61

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

An Efficient Nelson-Oppen Decision Procedure for Difference Constraints over Rationals

Shuvendu K. Lahiri and Madanlal Musuvathi

Microsoft Research
{shuvendu,madanm}@microsoft.com

Abstract. Nelson and Oppen provided a methodology for modularly combining decision procedures for individual theories to construct a decision procedure for a combination of theories. In addition to providing a check for satisfiability, the individual decision procedures need to provide additional functionalities, including equality generation.
In this paper, we propose a decision procedure for a conjunction of difference constraints over rationals (where the atomic formulas are of the form $x \leq y + c$ or $x < y + c$). The procedure extends any negative cycle detection algorithm (like the Bellman-Ford algorithm) to generate (1) equalities between all pair of variables, (2) produce proofs and (3) generates models that can be extended by other theories in a Nelson-Oppen framework. All the operations mentioned above can be performed with only a linear overhead to the cycle detection algorithm, in the average case.

1 Introduction

Difference constraints are a restricted class of linear arithmetic constraints of the form $x \bowtie y + c$, where x, y are variables, $\bowtie \in \{<, \leq\}$ and c is a rational constant. These constraints naturally arise in many applications. For instance, the array bounds' check in a program and the timing constraints in job scheduling can be specified as difference constraints.

There is a well-known, efficient decision procedure for difference constraints. Given a set of difference constraints, one can reduce the problem of checking its satisfiability to the problem of detecting negative cycles in an appropriately generated graph [7]. Then, any of the negative-cycle-detection algorithms (see [6] for a nice survey) can be used to decide the given constraints. For instance, the classic Bellman-Ford [4, 13] algorithm can decide m difference constraints on n variables in $O(n * m)$ time and $O(n + m)$ space complexity.

In this paper, we extend this basic decision procedure to produce an *equality-generating*, *proof-producing*, and *model-generating* decision procedure for difference constraints. The motivations for these extensions are the following: Using the Nelson-Oppen combination framework [16] requires the decision procedure to generate any variable equalities implied by the input constraints. Also, when used in a lazy-proof-explication framework [12, 11, 3], the decision procedure needs to generate proofs, both when reporting unsatisfiability of the input constraints and

when generating any implied equality. Finally, the need for model-generation is motivated by our use of the decision procedure in an unit-testing tool. In this application, an input formula, when satisfiable, represents a feasible path in the program. A model for this formula can then be used to produce a concrete test input that drives the program along that path.

A trivial way to provide the extensions mentioned above is to compute the transitive closure of the input constraints. For instance, the input constraints imply an equality $x = y$ if and only if the transitive closure contains the constraints $x \leq y$ and $y \leq x$. Given m difference constraints in n variables, computing the transitive closure requires $O(n^3)$ time and $O(n^2)$ space. [1] While the worst-case complexity is the same as the complexity of the Bellman-Ford algorithm, our initial experiments with this approach show that computing the transitive closure is very slow in practice and is a major bottleneck for the decision procedure. This is particularly apparent when the input is *sparse*, where m is much less than the maximum possible $O(n^2)$.

In contrast, the decision procedure described in this paper generates equalities, proofs, and models with very little overhead to the basic negative-cycle-algorithm, in average *linear* time and space. Such an algorithm is critical for the following pragmatic reasons. First, by not performing the transitive closure our decision procedure is very fast for sparse difference constraints. Also, the the time and space complexity of the decision procedure is determined only by the negative-cycle-detection algorithm used. Thus, the efficiency of the decision procedure can be further improved by using a negative-cycle-detection algorithm that is optimized for the constraints appearing in a particular domain [6].

1.1 Related Work

Pratt [18] observed that most linear arithmetic queries in software verification are limited to difference logic (DIF) queries. Recently, there has been a renewed interest in solving DIF queries, namely because of the its importance in various hardware [5] and software verification domains [2, 14].

Strichman et al. [22] provided a decision procedure for Boolean combination of DIF constraints by providing an satisfiability-preserving translation to a Boolean formula. The Boolean formula is produced by treating each difference constraint as a Boolean variable and adding *all* the "transitivity constraints" due to the DIF constraints to the formula. This process can introduce an exponential number of such constraints, resulting in a very large Boolean formula. Bryant et al. [5] provide an alternate translation to Boolean formula based on "finite-model" property of DIF logic. Each variable is encoded using a symbolic bit-vector that is sufficient to preserve the satisfiability of the original formula. This results in a polynomial time translation to SAT and is often more efficient in practice. Talupur et al [23] improve upon this idea by providing a method to

[1] Transitive closure can be computed more efficiently using matrix-multiplication based methods. It is not clear how well these methods perform when used in a decision procedure.

identify a small range for each variable to reduce the number of Boolean variables in the final encoding. However, the worst-case complexity of the method to find the range can be exponential in the size of the input formula. The methods in this category are "eager" in translating the formula to SAT, and have been implemented in the tool UCLID [5]. Seshia and Bryant [21] exploit the sparseness of non-difference constraints to provide a decision procedure for a Boolean combination of linear arithmetic constraints.

On the other hand, theorem provers based on a more "lazy" translation to SAT use a decision procedure for a conjunction of DIF constraints. MATH-SAT [1] uses a DIF logic solver as the first step to check the satisfiability of a linear arithmetic constraint before using a more general linear arithmetic decision procedure. Nieuwenhuis et al. [17] use a decision procedure for DIF that can incrementally produce all the constraints implied by a set of constraints. The procedure also produces proofs of unsatisfiability. The implied constraints are used to improve the constraint propagation and conflict analysis of a DPLL [9, 10] style solver for first-order theories. Cotton et al. [8] use a decision procedure for DIF based on negative cycle detection algorithm and integrate conflict analysis of DIF with the conflict analysis of the SAT solver. Unlike our method, these methods do not require producing equalities over variables, as the decision procedure does not operate in a combination framework.

Model generation for linear arithmetic queries in a combination framework has been recently addressed by Ruess et al. [19]. They extend the Simplex decision procedure to generate satisfying assignments (over rationals) in the presence of disequalities. Our contribution is to extend Bellman-Ford algorithm to handle disequalities. For the restricted fragment of DIF, this provides an efficient algorithm to generate such models, while Simplex suffers from a worst-case exponential complexity in solving linear constraints.

2 Background

In this section, we briefly describe some notations and a high-level description of theorem provers based on Nelson-Oppen combination. Since our intention is to explain the decision procedure for difference theory in the Nelson-Oppen setting, we do not elaborate on the technical definitions of theories, signatures etc.; interested readers are refered to excellent survey works [16, 25] for rigorous treatment.

2.1 Premilinaries

Figure 1 defines the syntax of a quantifier-free fragment of first-order logic. An expression in the logic can either be a *term* or a *formula*. A *term* can either be a variable or an application of a function symbol to a list of terms. A *formula* can be the constants `true` or `false` or an atomic formula or Boolean combination of other formulas. Atomic formulas can be formed by an equality between terms or by an application of a predicate symbol to a list of terms. A *literal* is either an

$$
\begin{aligned}
\textit{term} &::= \textit{variable} \mid \textit{function-symbol}(\textit{term}, \ldots, \textit{term}) \\
\textit{formula} &::= \texttt{true} \mid \texttt{false} \mid \textit{atomic-formula} \\
&\mid \textit{formula} \wedge \textit{formula} \mid \textit{formula} \vee \textit{formula} \mid \neg \textit{formula} \\
\textit{atomic-formula} &::= \textit{term} = \textit{term} \mid \textit{predicate-symbol}(\textit{term}, \ldots, \textit{term})
\end{aligned}
$$

Fig. 1. Syntax of a quantifier-free fragment of first-order logic.

atomic formula or its negation. We will often identify a conjunction of literals $l_1 \wedge l_2 \ldots l_k$ with the set $\{l_1, \ldots, l_k\}$.

The function and predicate symbols can either be *uninterpreted* or can be defined by a particular theory. For instance, the theory of integer linear arithmetic defines the function-symbol "+" to be the addition function over integers and "<" to be the comparison predicate over integers. For a theory T, the *signature* Σ denotes the set of function and predicate symbols in the theory. If an expression E involves function or predicate symbols from two (or more) theories T_1 and T_2, then E is said to be an expression over a combination of theories $T_1 \cup T_2$.

A formula F is said to be *satisfiable* if it is possible to assign values to the various symbols in the formula from the domains associated with the theories to make the formula `true`. A formula is *valid* if $\neg F$ is not satisfiable (or unsatisfiable). We say a formula A *implies* a formula B ($A \Rightarrow B$) if and only if $(\neg A) \vee B$ is valid. A satifiability procedure for Σ-theory T checks if a formula ϕ (over Σ) is satisfiable in T.

2.2 Nelson Oppen Combination

Given two *convex* and *stably infinite*[2] theories T_1 and T_2 with disjoint-signatures Σ_1 and Σ_2 respectively (i.e. $\Sigma_1 \cap \Sigma_2 = \{\}$), and a conjunction of literals ϕ over $\Sigma_1 \cup \Sigma_2$, we want to decide if ϕ is satisfiable under $T_1 \cup T_2$. Nelson and Oppen [16] provided a method for modularly combining the satisfiability procedures for T_1 and T_2 to produce a satisfiability procedure for $T_1 \cup T_2$.

The Nelson-Oppen procedure works as follows: The input ϕ is split into the ϕ_1 and ϕ_2 such that ϕ_i only contains symbols from Σ_i and $\phi_1 \wedge \phi_2$ is equisatisfiable with ϕ. Each theory T_i decides the satisfiability of ϕ_i and returns unsatisfiable if ϕ_i is unsatisfiable in T_i. Otherwise, the set of equalities implied by ϕ_i over the variables common to ϕ_1 and ϕ_2 are propagated to the other theory T_j. The theory T_j adds these equalities to ϕ_j and the process is repeated until the set of equalities saturate.

Therefore, in addition to checking the satisfiability of a set of literals, each theory also has to derive all the equalities over variables that are implied by the

[2] We need these restrictions only to state the N-O combination result. The definition of convexity and stably infiniteness can be found in [16, 25].

set of literals. The satisfiability procedure is called *equality generating* if it can generate all such equalities.

If the formula ϕ_i is satisfiable, then theory T_i can generate satisfying assignments for the variables and symbols in ϕ_i. If each theory can generate *diverse* assignments for the shared variables (described in Section 6), then we can generate a satisfying assignment for the symbols and variables in ϕ.

2.3 Lazy Theorem Proving

To check the satisfiability of a formula ψ with arbitrary Boolean structure, many modern theorem provers [12, 3] use a encoding of ψ into a Boolean formula, and use a Boolean Satisfiability (SAT) solver to check the satisfiability of the Boolean formula. We provide a brief description of the method below.

The formula ψ is checked using a Boolean SAT solver, after treating each atomic formula as a Boolean variable. If the SAT solver determines that the formula is unsatisfiable, then ψ is unsatisfiable. Otherwise, the satisfying assignment from SAT ϕ (a conjunction of literals from ψ) is checked for satisfiability using the Nelson-Oppen combination procedure described in the last section. If ϕ is satisfiable over the first-order theories, the formula ψ is satisfiable. Otherwise, a "conflict clause" is derived from the theories that will prevent the same assignment being produced by the SAT solver. One way to derive such as clause is to select the literals that appear in the "proof" of unsatisfiability of ϕ. We call this method the *lazy* theorem proving using proof explication.

Therefore, an additional requirement for the decision procedures working in the Nelson-Oppen procedure (operating in a lazy theorem proving context) is to produce a proof of unsatisfiability. To achieve this, the satisfiability procedure for each theory needs to produce a (1) proof of every equality $x = y$ over the shared variables, and (2) a proof of unsatisfiability if the theory reports unsatisfiable.

3 Difference Logic and Satisfiability

Difference logic is a simple yet useful fragment of linear arithmetic, where the atomic formulas are of the form $x \bowtie y + c$, where x, y are variables, $\bowtie \in \{<, \leq\}$ and c is a rational constant. Any equality $x = y + c$ is represented as a conjunction of $x \leq y + c$ and $y \leq x - c$. Constraints like $x \bowtie c$ are handled by adding a special variable x_0 to denote the constant 0, and rewriting the constraint as $x \bowtie x_0 + c$ [22]. To simplify our discussion, we assume that there are no strict inequalities. This poses no problems as one can simply reduce the bound c by a small amount [20]. The function symbol "+" and the predicate symbols $\{<, \leq\}$ are the interpreted symbols of this theory.

Given a set of difference contraints ϕ, we can construct a graph $G_\phi(V, E)$, where the vertices of the graph are the variables in ϕ and there is a directed edge in the graph from x to y of weight c, if $y \leq x + c \in \phi$. 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 simple *path* P in G_ϕ is a sequence of edges $[e_1, \ldots, e_n]$ such that $d(e_i) = s(e_{i+1})$, for all $1 \leq i \leq n-1$, and no vertex is repeated. We always refer to a simple path as a path, unless otherwise mentioned. 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. $\Sigma_{1 \leq i \leq n} w(e_i)$. A simple cycle C is a sequence of edges $[e_1, \ldots, e_n]$ where $s(e_1) = d(e_n)$ and no vertex appears as a source or destination twice in the path. We use $u \rightsquigarrow v$ in E to denote that there is a path from u to v through edges in E.

It is well known [7] that a set of difference constraints ϕ is unsatisfiable if and only the graph G_ϕ has a simple cycle C, such that $w(C) < 0$. Hence, checking satisfiability can be reduced to checking for negative cycles in the graph G_ϕ. The Bellman-Ford [4, 13] algorithm described below is a way to detect negative cycles in a directed graph. Although the algorithm is well known, we describe it here because it will be used in subsequent sections (e.g. while describing the proof of unsatisfiability).

3.1 Bellman-Ford algorithm

Given a directed graph G (possibly with negative weights), Bellman-Ford algorithm detects the single source shortest path from any given vertex s to all other vertices in the graph. The algorithm returns false when there is a negative cycle in the graph, and true otherwise. For any vertex $v \in V$, $\delta(v)$ denotes the shortest path from s to v, upon the completion of the algorithm with true. In that case, the map $p(v)$ denotes the parent of the vertex v in the shortest path tree rooted at s.

The following algorithm *Bellman-Ford*$(G(V, E), s)$, computes the shortest path from s to each of the vertices in G:

1. Initialize:
   - Set $\delta(s) \leftarrow 0$. For any vertex $v \in V \setminus \{s\}$, set $\delta(v) \leftarrow \infty$. For any vertex $v \in V$, set $p(v) \leftarrow$ nil.
2. For $i = 1$ to $|V| - 1$ do:
   (a) For each edge $(u, v) \in E$
      i. If $\delta(v) > \delta(u) + w(u, v)$ then
         - Set $\delta(v) \leftarrow \delta(u) + w(u, v)$.
         - Set $p(v) \leftarrow u$.
3. For each edge $(u, v) \in E$:
   (a) If $\delta(v) > \delta(u) + w(u, v)$ then
      i. return false.
4. return true.

To detect negative cycles in the graph G_ϕ, the first step is to add a new vertex x_{max} to the graph, and add edges of weight zero from x_{max} to all other vertices in G_ϕ. Let H_ϕ be the new graph. The graph G_ϕ contains a negative cycle if and only if the algorithm *Bellman-Ford*(H_ϕ, x_{max}) returns false. The runtime of the algorithm is $O(|V| * |E|)$.

Proposition 1. *When the Bellman-Ford algorithm returns true, then for any edge* $(u, v) \in E$, $\delta(v) \leq \delta(u) + w(u, v)$.

Let us define the *slack* $sl(u, v)$ for any edge (u, v) (after Bellman-Ford algorithm returns true) as: $sl(u, v) = \delta(u) - \delta(v) + w(u, v)$. It is easy from Proposition 1 to see that $sl(u, v) \geq 0$, for any edge (u, v).

Proposition 2. *For any cycle* $C \doteq [e_1, \ldots, e_n]$ *in* G_ϕ, $w(C) = \Sigma_{e_i \in C} sl(e_i)$.

4 Equality Generation for Difference Constraints

In this section, we illustrate how to generate all the variable equalities implied by the constraint ϕ. We assume that ϕ is satisfiable — i.e. that the Bellman-Ford algorithm has returned true on the graph $G_\phi(V, E)$ constructed as shown in Section 3. Now, one can always produce such equalities by performing a transitive closure of the constraints in ϕ and checking if $x \leq y$ and $y \leq x$ have been derived. However, the algorithm suffers from worst case $O(|V|^3)$ time and $O(|V|^2)$ space complexity. We show an algorithm to derive all such equalities in $O(|V| + |E|)$ average case space and time, after the completion of the Bellman-Ford algorithm. This algorithm assumes an average constant time for hash table insertions and lookups. Without this assumption, our algorithm has a time complexity $O(|V| * logC + |E|)$ where C is the weight of the maximum weighted path in G_ϕ.

We now describe the algorithm $EqGen(G_\phi, \delta)$ for generating equalities over the variables in V.

1. Let E' be set of edges in G such that an edge $e \in E'$ if and only if $sl(e) = 0$.
2. Create the induced subgraph $G'_\phi(V, E')$ from $G_\phi(V, E)$.
3. Group the vertices in G'_ϕ into *strongly connected components* (SCCs). Vertices u and v are in the same SCC if and only if $u \rightsquigarrow v$ and $v \rightsquigarrow u$ in E'. This can be done in linear time [24].
4. For each SCC S, let $V_d^S = \{x \mid x \in S \text{ and } \delta(x) = d\}$. This can be done in average linear time using a hash table.
5. For each $V_d^S = \{x_1, \ldots, x_k\}$, where $k \geq 2$, generate the equalities $x_1 = x_2, x_2 = x_3, \ldots, x_{k-1} = x_k$.

Theorem 1. *Let* $\mathcal{E}$ *be the set of equalities generated by the* $EqGen(G_\phi, \delta)$ *procedure. For any equality* $x = y$ *over* V, $\phi \Rightarrow x = y$ *if and only if* $\mathcal{E} \Rightarrow x = y$.

We will use a few intermediate lemmas before proving the above theorem.

Lemma 1. *A cycle* $C \doteq [e_1, \ldots, e_n]$ *in* G_ϕ *has* $w(C) = 0$, *if and only if* C *is a cycle in* G'_ϕ.

Lemma 2. *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.*

Lemma 2 implies that the equality edges in G_ϕ are exactly those edges in G'_ϕ that are present in the SCCs of G'_ϕ. Since SCCs preserve the cycles in G'_ϕ, any edge e in G_ϕ lies in a zero-weight cycle if and only if e lies in some SCC in G'_ϕ.

Lemma 3. *For two variables x and y in V, $\phi \Rightarrow x = y$ if and only if x and y lie in some SCC of G'_ϕ and $\delta(x) = \delta(y)$.*

Finally, the proof of Theorem 1 follows easily from Lemma 1 and Lemma 3. Note that for any pair of vertices x and y, x and y lie in a cycle of weight zero and $\delta(x) = \delta(y) = d$ if and only if $\{x, y\} \subseteq V_d^S$, for some SCC S. Therefore, either $x = y$ is present in $\mathcal{E}$, or follows from $\mathcal{E}$ by using symmetry and transitivity.

5 Proof generation

When Bellman-Ford algorithm returns false, there is a negative cycle in the graph G_ϕ. This negative cycle is the proof of unsatisfiability. The cycle can be obtained by simply traversing the p pointers. Let v be the vertex such that $\delta(v) > \delta(u) + w(u, v)$ in step 3(i). Let $u_1 \doteq p(u), u_2 \doteq p(u_1), \ldots, u_k \doteq p(u_{k-1}), \ldots$ be the vertices that are obtained by following the parent pointer p. For this sequence, there exists $1 \leq i < j$, such that $u_j = u_i$. The set of edges $[(u_j, u_{j-1}), (u_{j-1}, u_{j-2}), \ldots, (u_{i+1}, u_i)]$ forms a negative cycle in G_ϕ.

One the other hand, when the procedure generates an equality $u = v$, we have to provide a proof for the equality. We will use the SCC computation phase to enable generating the cycle in G'_ϕ containing u and v. To do this, we briefly look at the main steps of the algorithm for generating SCCs.

The following algorithm *SCC*$(G(V, E))$, computes the SCCs for a graph $G(V, E)$. Similar to the Bellman-Ford algorithm, we maintain *parent* pointers p for each vertex.

1. Perform a depth-first search (DFS) on G.
   - Record the *finishing time* $f[v]$ for each vertex v in G. Intuitively, the finishing times indicate the order in which the vertices were popped from the stack (after exploring all its descendants) during the traversal of the graph.
   - During the DFS, update the parent pointer p of any vertex v, $p(v) \leftarrow u$, where v was first visited through the edge $(u, v) \in E$.
2. Construct $G^T \doteq G(V, E^T)$, where $E^T = \{(v, u) \mid (u, v) \in E\}$. Let p^T be the parent pointers for the graph G^T.
3. Perform DFS on G^T in the order of decreasing $f[v]$. Update the parent pointers $p^T(u)$ for each vertex as before.
4. The DFS on G^T induces a set of trees. The set of nodes in each tree represents an SCC.

For an SCC S, let $root(S)$ denote the root of the tree for S (that also corresponds to the node with the highest f value in S). For any vertex u, define $u_1 \doteq p(u)$, $u_2 \doteq p(u_1), \ldots$ and $u_{-1} \doteq p^T(u)$, $u_{-2} \doteq p^T(u_{-1}), \ldots$.

Lemma 4. *For any SCC S (with size at least two), and for any pair of distinct vertices u and v in S, the sequence of edges $[(u, u_{-1}), (u_{-1}, u_{-2}), \ldots, (u_{-k}, root(S))]$ followed by $[(root(S), v_m), (v_m, v_{m-1}), \ldots, (v_1, v)]$ forms a path (not necessarily simple) from u to v in E.*

Lemma 5. *If the SCC algorithm is run on G'_ϕ, then for every pair of vertices u and v in an SCC S with $\delta(u) = \delta(v)$, a path from u to v (and from v to u) with weight zero can be constructed using p and p^T pointers in $O(|V|)$ time.*

Thus, if u and v belong to the same SCC in the graph G'_ϕ, and $\delta(u) = \delta(v)$, then the path from u to v of weight zero will derive $v \leq u$ and the path from v to u of weight zero will derive $u \leq v$. This will constitute the proof of $u = v$.

Theorem 2. *For every equality $u = v$ implied by ϕ, the proof of $u = v$ can be generated in $O(|V|)$ time.*

If there are q (bound by $|V|$) irredundant equalities implied by ϕ, the time for generating the proof for the equalities is bound by $O(q * |V|)$. The algorithm is a linear output-sensitive algorithm.

6 Model Generation

For a conjunction of difference constraints ϕ, the δ values computed by the Bellman-Ford algorithm satisfies all the constraints in ϕ. However, this is not sufficient to produce a model in the Nelson-Oppen combination framework. Consider the following example where a formula involves the logic of equality with uninterpreted functions (EUF) and difference constraints.

Let $\psi = (f(x) \neq f(y) \wedge x \leq y)$ be a formula in the combined theory. Nelson-Oppen framework will add $\psi_1 \doteq f(x) \neq f(y)$ to the EUF theory (T_1) and $\psi_2 \doteq x \leq y$ to the difference logic theory (T_2). Since there are no equalities implied by either theory, and each theory T_i is consistent with ψ_i, the formula ψ is satisfiable. Now, the difference logic theory generates the model $\langle x \mapsto 0, y \mapsto 0 \rangle$ for ψ_2. However, this is not a model for ψ.

To generate an assignment for the variables that are shared across two theories, each theory T_i needs to ensure that the variable assignment ρ for T_i assigns two shared variables x and y equal values if and only if the equality $x = y$ is implied by the constraints in theory T_i. We call such assignments as *diverse*.

If we assume that the domain of any Σ-structure is $\mathcal{Q}$, the set of rational numbers, then every function (predicate) is interpreted as a function (relation, respectively) over rationals. Since the symbols $\{+, \leq\}$ are interpreted by the linear arithmetic theory (in this case, the difference logic theory) only, the absolute values assigned to the shared variables are only relevant for this theory. Hence, the diverse model generated by the linear arithmetic theory can be extended by each theory to provide a model for the overall formula. In the above example, any extension of the model $\langle x \mapsto 0, y \mapsto 1, f(0) \mapsto 0, f(1) \mapsto 1 \rangle$ satisfies the formula ψ.

The following section describes a more general problem of generating models when a set of disequalities Γ are *explicitly* specified. Assuming that the Bellman-Form algorithm has run, this algorithm generates a model in $O(|V|+|E|+|\Gamma|)$ time. It is straightforward to modify this algorithm to generate diverse models in $O(|V|+|E|)$ time by *implicitly* specifying disequalities for every pair of variables $x \neq y$, if $x = y$ is not implied by the set of constraints.

6.1 Model Generation with Disequalities

In this section, we describe rational model generation for a set of difference constraints Φ, along with a set of disequalities $\Gamma \doteq \{x_i \neq y_i, \ldots\}$ over variables. We assume that the set of constraints $\Phi \cup \Gamma$ is satisfiable.

We assume that we have run Bellman-Ford algorithm on G_Φ to compute $\delta(v)$ for each vertex $v \in V$. Also, assume that have constructed the graph G'_Φ(described in Section 4) that contains the vertices with zero-slack edges. Finally, we generate the SCC graph G_Φ^{SCC} of G'_Φ as follows. The vertices $S_1, \ldots, S_n$ of G_Φ^{SCC} are the SCCs in the graph G'_Φ. Let $SCC(v)$ be the SCC S to which the vertex $v \in V$ belongs to. For each edge $(x, y) \in E$, G_Φ^{SCC} contains an edge $(SCC(x), SCC(y))$. It is well known that G_Φ^{SCC} is a directed acyclic graph and can generated from G'_Φ in $O(|V|+|E|)$ time.

The values $\delta(v)$ for each vertex $v \in V$ computed by the Bellman-Ford algorithm is a *feasible* solution for Φ as it satisfies all the constraints in Φ. However, these values might not satisfy the disequalities in in Γ, i.e. it is possible for $\delta(x) = \delta(y)$ for $x \neq y \in \Gamma$. To generate a model for $\Phi \cup \Gamma$, we perturb the values $\delta(v)$ for $v \in V$ so as to satisfy the disequalities in Γ as well as the constraints in Φ.

Given the SCC graph G_Φ^{SCC}, the model generation algorithm orders its vertices in topological-sort order $\preceq$, where $S \preceq T$ for every edge (S, T) in G_Φ^{SCC}. [3] Such an ordering can be performed in linear time [15]. The model generation algorithm makes a single traversal in the SCC graph G_Φ^{SCC} in the topological sort order. During this traversal, the algorithm assigns a value $\gamma(S)$ for each S in G_Φ^{SCC}. This value represents the perturbation of all the vertices in the SCC represented by S.

The disequalities in Γ are captured in G_Φ^{SCC} by the relation $\not\sim$ as follows: For vertices S and T in G_Φ^{SCC}, $S \not\sim T$ exactly when there exists $x \in S$, $y \in T$, $T \preceq S$ and $x \neq y \in \Gamma$.

The model generation algorithm follows.

1. Let $\epsilon \doteq min\left(\{|\delta(x) - \delta(y)| \mid x \in V, y \in V, \delta(x) \neq \delta(y)\} \cup \{sl(e) \mid sl(e) \neq 0\}\right)$. If the set of values is $\{\}$, then return.
2. For each $S \in G_\Phi^{SCC}$, set $\gamma(S) \leftarrow 0$
3. For each S in the topological order $\preceq$ in G_Φ^{SCC} do:
   (a) If there exists T such that $S \not\sim T \wedge \gamma(S) = \gamma(T)$

[3] In practice, it is very easy to modify the SCC algorithm to generate the SCCs in topological sort order — a separate pass is not necessary.

   i. $\gamma(S) \leftarrow \gamma(S) + \epsilon/2$
   ii. $\epsilon \leftarrow \epsilon/2$
(b) For each edge $(S, U) \in G_{\Phi}^{SCC}$
   i. $\gamma(U) \leftarrow max(\gamma(U), \gamma(S))$
4. For each $v \in V$, output $\delta'(v) = \delta(v) - \gamma(SCC(v))$

To prove that this algorithm generates a model form $\Phi \cup \Gamma$, we need the following lemmas.

Lemma 6. *For each SCCs S, T in G_{Φ}^{SCC} and for each $x, y \in V$, the following holds*

1. $sl(x, y) \neq 0 \;\Rightarrow\; \epsilon \leq sl(x, y)$
2. $\delta(x) \neq \delta(y) \;\Rightarrow\; \epsilon \leq |\delta(x) - \delta(y)|$
3. $0 \leq \gamma(s) < \epsilon$
4. *For edge (S, T) in G_{Φ}^{SCC}, $\gamma(S) \leq \gamma(T)$*

Lemma 7. *The assignment δ' output by the algorithm satisfies all constraints in Φ.*

Lemma 8. *For $x \neq y \in \Gamma$, $\delta'(x) \neq \delta'(y)$.*

Theorem 3. *The assignment δ' at the end of the above algorithm satisfies $\Phi \cup \Gamma$.*

7 Conclusion

In this paper, we have presented an efficient decision procedure for handling difference constraints in a decision procedure that operates in a SAT-based proof-explicating, Nelson-Oppen combination framework. The procedure also generates models in the presence of disequalities. The overhead of equality generation, proof generation and model generation over satisfiability checking (using Bellman-Ford algorithm) is only $O(|V| + |E|)$ (both time and space) in the average case.

We are currently implementing this idea in ZAP theorem prover. We hope to present some experimental comparison of our method with a decision procedure based on transitive closure.

References

1. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *LNCS*, pages 195–210. Springer Verlag, July 2002.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, June, 2001. *SIGPLAN Notices,* 36(5), May 2001.

3. C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 236–249, July 2002.
4. R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
5. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.
6. Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, 1996.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
8. S. Cotton, E. Asarin, O. Maler, and P. Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS/FTRTFT*, pages 263–276, 2004.
9. M. Davis, G.eorge Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
10. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
11. L. de Moura and H. Rueß. An Experimental Evaluation of Ground Decision Procedures. In *Computer Aided Verification (CAV '04)*, LNCS 3114. Springer-Verlag, 2004.
12. C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem Proving usign Lazy Proof Explication. In *Computer-Aided Verification (CAV 2003)*, LNCS 2725, pages 355–367. Springer-Verlag, 2003.
13. L. R. Ford, Jr., and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Symposium on Principles of programming languages (POPL '02)*, pages 58–70. ACM Press, 2002.
15. Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968. Second edition, 1973.
16. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.
17. R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *Computer Aided Verification, 17th International Conference, CAV 2005 (to appear)*, LNCS. Springer, 2005.
18. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, Cambridge, Mass., September 1977.
19. H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, January 2004.
20. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.
21. S. A. Seshia and R. E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *19th IEEE Symposium of Logic in Computer Science(LICS '04)*. IEEE Computer Society, July 2004.
22. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding Separation Formulas with SAT. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 209–222, July 2002.

23. M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In *Computer Aided Verification, 16th International Conference, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*. Springer, 2004.
24. R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
25. C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop (Munich, Germany)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, March 1996.

A Proofs of various Lemmas and Theorems

A.1 Proof of Proposition 2

Proof. Consider a cycle $C \doteq [e_1, \ldots, e_n]$ in G'_ϕ. For each edge $e_i \in C$, we know $sl(e_i) = \delta(s(e_i)) - \delta(d(e_i)) + w(e_i)$. Since C is a cycle, each vertex $u \in C$ appears as a source and a destination the same number of times, therefore $\Sigma_{e_i \in C} \delta(s(e_i)) - \delta(d(e_i)) = 0$. Therefore $w(C) = \Sigma_{e_i \in C} w(e_i) = \Sigma_{e_i \in C} sl(e_i)$.

A.2 Proof of Lemma 1

Proof. We first prove the above lemma for simple cycles only. Since there are no negative weight cycles in G_ϕ or G'_ϕ, any cycle of weight zero has to be composed of simple cycles each with weight zero.

If there is a simple cycle $C \doteq [e_1, \ldots, e_n]$ in G'_ϕ, then for each edge e_i, we know that $sl(e_i) = \delta(s(e_i)) - \delta(d(e_i)) + w(e_i) = 0$. Using Proposition 2, we know $w(C) = 0$.

On the other hand, consider a cycle $C \doteq [e_1, \ldots, e_n]$ in G_ϕ with $w(C) = 0$. Using Proposition 2, $\Sigma_{e_i \in C} sl(e_i) = 0$. Since any slack $sl(e_i)$ is non-negative, we know that every edge $e_i \in C$ has $sl(e_i) = 0$, and is therefore present in G'_ϕ. Hence C is present in G'_ϕ.

A.3 Proof of Lemma 2

Proof. It is easy to see that if e representing $y \leq x + c$ lies in a simple cycle $[e, P]$ (where P is a path) of weight 0, then $x \leq y - c$ is implied by edges in P. Thereby, the edge e can be strengthened to represent $y = x + c$.

On the other hand, assume that $x \leq y - c$ is implied by ϕ, and $y \leq x + c$ is present ϕ (i.e., e is in G_ϕ). Also assume that e does not lie in any cycle of weight 0 — therefore there is no path from y to x with weight $-c$. Since $x \leq y - c$ is implied by ϕ, there is a path $P_{y,x}$ from y to x, such that $w(P_{y,x}) \leq -c$[4]. Since $w(P_{y,x}) \neq -c$, there is a negative cycle in the graph. Hence, e has to lie in a simple cycle of weight 0.

[4] The formal justification is that a difference constraint $x \leq y + c$ is implied by a set of difference constraints ϕ if and only if $x \leq y + c$ can be derived from ϕ using a unit linear combination of a subset of constraints from ϕ.

A.4 Proof of Lemma 3

Proof. An equality $x = y$ is implied by ϕ if and only if there is a path $P_{x,y}$ from x to y in G_ϕ of weight 0, and there is path $Q_{y,x}$ from y to x of weight 0. We will show that all the edges in these paths are equality-edges.

Since we know that the weight of the cycle $[P_{x,y}, Q_{y,x}]$ is 0, any edge $e_i \in [P_{x,y}, Q_{y,x}]$ has $sl(e_i) = 0$. Moreover, each edge $e \in P_{x,y}$ (or $Q_{y,x}$) lies in the cycle $[P_{x,y}, Q_{y,x}]$ and therefore by Lemma 2, also an equality edge. Hence, by Lemma 1, the cycle $[P_{x,y}, Q_{y,x}]$ is present in some SCC of G'_ϕ.

Now, consider any edge $e_i \in P_{x,y}$. Since $sl(e_i) + \delta(d(e_i)) - \delta(s(e_i)) = w(e_i)$, and $w(P_{x,y}) = 0$, $\delta(y) - \delta(x) = w(P_{x,y}) = 0$.

A.5 Proof of Lemma 4

Proof. Consider any vertex u in an SCC S. After the DFS of G^T in step 3 of the above algorithm, the p^T pointers store a path from $root(S)$ (the root of the tree for S) to u through the edges $[(root(S), u_{-k}), (u_{-k}, u_{-k+1}), \ldots, (u_{-1}, u)]$ in E^T. Since the edges in E^T are reversed E edges, $[(u, u_{-1}), (u_{-1}, u_{-2}), \ldots, (u_{-k}, root(S))]$ forms a path from u to $root(S)$ in G. Since $[(root(S), v_m), (v_m, v_{m-1}), \ldots, (v_1, v)]$ is a path from $root(S)$ to v in E, the concatenation of the two sequences of edges forms a path from u to v in E.

A.6 Proof of Lemma 5

Proof. This follows easily from Lemma 4 and the fact that for any path $P_{u,v}$ in G'_ϕ, $w(P_{u,v}) = \delta(u) - \delta(v) + sl(P_{u,v}) = 0$, since all edges in G'_ϕ has slack zero.

A.7 Proof of Lemma 6

Proof. The proof of statements 1 and 2 follows straight from the choice of ϵ in Line 1 of the algorithm.

For the proof of statement 3, note that $\gamma(x)$ is initially set to 0 for all vertices $x \in V$ and monotonically increases during the execution of the algorithm. However, as ϵ is halved whenever a $\gamma(x)$ is increased for any x, these cumulative additions $\Sigma_{1 \le i} \left(\epsilon/2^i\right)$ cannot exceed ϵ.

The proof of statement 4 relies on the fact that the algorithm traverses S before T as $S \preceq T$. While visiting S, the update in Line 3(b)i ensures that $\gamma(T) \ge \gamma(S)$. After visiting S, the algorithm never modifies $\gamma(S)$ while $\gamma(T)$ can only increase.

A.8 Proof of Lemma 7

Proof. To prove this lemma, we need to show that for every edge (x, y) in G, $\delta'(y) - \delta'(x) \le w(x, y)$. Consider one such edge (x, y). Let $S = SCC(x)$ and $T = SCC(y)$.

Case 1: The slack $sl(x, y) = 0$. If $S = T$ then trivially $\delta'(y) - \delta'(x) = \delta(y) - \delta(x) \leq w(x, y)$. If $S \neq T$ then the edge (S, T) is in G_{Φ}^{SCC} and therefore

$$\begin{aligned} \delta'(y) - \delta'(x) &= \delta(y) - \delta(x) + \gamma(S) - \gamma(T) \\ &\leq \delta(y) - \delta(x) && \text{by Lemma 6.4} \\ &\leq w(x, y) \end{aligned}$$

Case2: The slack $sl(x, y) > 0$. In this case:

$$\begin{aligned} \delta'(y) - \delta'(x) &= \delta(y) - \delta(x) + \gamma(S) - \gamma(T) \\ &< \delta(y) - \delta(x) + \epsilon && \text{by Lemma 6.3} \\ &\leq \delta(y) - \delta(x) + sl(x, y) && \text{by Lemma 6.1} \\ &= w(x, y) && \text{from definition of } sl(x, y) \end{aligned}$$

Thus, we have shown that for every edge (x, y) in $\delta'(y) - \delta'(x) \leq w(x, y)$, proving the lemma.

A.9 Proof of Lemma 8

Proof. Let $S = SCC(x)$ and $T = SCC(y)$. Let us consider the two cases.

Case 1: $\delta(x) = \delta(y)$. Without loss of generality, assume $T \preceq S$. Clearly, $T \neq S$, otherwise either δ did not satisfy Φ or the disequality $x \neq y$ is contradictory with Φ. When visiting S, the algorithm ensures that $\gamma(S) \neq \gamma(T)$. Also, $\gamma(S)$ does not change any after visiting S, and $\gamma(T)$ was already fixed after visiting T in some previous iteration. Thus, $\delta'(x) \neq \delta'(y)$.

Case2: $\delta(x) \neq \delta(y)$. Without loss of generality, assume $\delta(y) > \delta(x)$. Then

$$\begin{aligned} \delta'(y) - \delta'(x) &= \delta(y) - \delta(x) + \gamma(S) - \gamma(T) \\ &\geq \epsilon - \gamma(t) + \gamma(s) && \text{by Lemma 6.2} \\ &> \gamma(s) && \text{by Lemma 6.3} \\ &\geq 0 && \text{by Lemma 6.3} \end{aligned}$$

Thus, we have $\delta'(y) > \delta'(x)$, proving the lemma.