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.

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 [6]. Then, any of the negative-cycle-detection algorithms (see [5] for a nice survey) can be used to decide the given constraints. For instance, the classic Bellman-Ford [3, 12] 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 [11, 10, 2], 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 [5].

A detailed version of the current paper, complete with proofs and other descriptions is available at [14].

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, mainly because of the its importance in various hardware and software verification domains. In this section, we briefly touch upon some of the related works that provide a decision procedure for DIF constraints in a theorem prover.

MATHSAT [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 [8, 9] style solver for first-order theories. Cotton et al. [7] 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. Finally, Strichman et al. [21] and Bryant et al. [4] provide a satisfiability preserving translation to a Boolean formula for a Boolean combination of DIF constraints.

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

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`.

[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.

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:

1. *Satisfiability Checking*: Check if a formula ϕ is satisfiable in the theory.
2. *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.
3. *Equality Generation*: The Nelson-Oppen framework for combining decision procedures [16] requires that each theory (at least) produces the set of equalities over variables that are implied by the constraints.
4. *Proof Generation*: Proof generation can be used to certify the output of a theorem prover [15]. Proofs are also used to construct conflict clauses efficiently in a lazy SAT-based theorem proving architecture [11].

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 $x \bowtie x_0 + c$ [21]. 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 constraints 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 [6] 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 [3, 12] 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 weight of a 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

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.

Fig. 1. *Bellman-Ford*$(G(V, E), s)$: Bellman-Ford Algorithm for computing single-source shortest path.

the vertex v in the shortest path tree rooted at s. Figure 1 describes the the algorithm *Bellman-Ford*$(G(V, E), s)$ to compute the shortest path from s to each of the vertices in G:

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.

Figure 2 describes the algorithm *EqGen*(G_ϕ, δ) for generating equalities over the variables in V.

Theorem 1. *Let* $\mathcal{E}$ *be the set of equalities generated by the EqGen*(G_ϕ, δ) *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*.*

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 [22].
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 $x_1 = x_2, x_2 = x_3, \ldots, x_{k-1} = x_k$.

Fig. 2. Algorithm $EqGen(G_\phi, \delta)$.

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 algorithm $SCC(G(V, E))$ (described in Figure 3), computes the SCCs for a graph $G(V, E)$. Similar to the Bellman-Ford algorithm, we maintain *parent* pointers p for each vertex.

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.*

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.

Fig. 3. Algorithm *SCC*$(G(V, E))$.

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.*

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*.

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 been 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^{SCC}_Φ of G'_Φ as follows. The vertices $S_1, \ldots, S_n$ of G^{SCC}_Φ are the SCCs in the graph G'_Φ. Let $SCC(v)$ be

1. Let $\epsilon \doteq min\,(\{|\delta(x)-\delta(y)| \mid x \in V, y \in V, \delta(x) \neq \delta(y)\} \cup \{sl(e) \mid sl(e) \neq 0\})$. 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 $S \not\sim T \wedge \gamma(S) = \gamma(T)$
      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 $\delta'(v) = \delta(v) - \gamma(SCC(v))$

Fig. 4. Model generation algorithm with disequalities.

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}. [2] Such an ordering can be performed in linear time [13]. 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 $x \neq y \in \Gamma$. Figure 4 describes the model generation algorithm.

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

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

[2] 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.

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.

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 *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *LNCS*, pages 195–210. Springer Verlag, July 2002.
2. 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.
3. R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
4. 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.
5. B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, 1996.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
7. S. Cotton, E. Asarin, O. Maler, and P. Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS/FTRTFT*, pages 263–276, 2004.
8. M. Davis, G.eorge Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
9. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
10. L. de Moura and H. Rueß. An Experimental Evaluation of Ground Decision Procedures. In *Computer Aided Verification (CAV '04)*, LNCS 3114. Springer-Verlag, 2004.
11. 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.
12. L. R. Ford, Jr., and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.
13. Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, 1968. Second edition, 1973.
14. S. K. Lahiri and M. Musuvathi. An efficient nelson-oppen decision procedure for difference constraints over rationals. Technical Report MSR-TR-2005-61, Microsoft Research, 2005.
15. G. C. Necula and P. Lee. Proof generation in the touchstone theorem prover. In *Conference on Automated Deduction*, LNCS 1831, pages 25–44, 2000.
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. 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.
22. R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.