Finite Instantiations for Integer Difference Logic[*]

Hyondeuk Kim Fabio Somenzi
Department of Electrical and Computer Engineering
University of Colorado at Boulder, CO 80309-0425
{Hyondeuk.Kim,fabio}@Colorado.EDU

Abstract

The last few years have seen the advent of a new breed of decision procedures for various fragments of first-order logic based on propositional abstraction. A lazy satisfiability checker for a given fragment of first-order logic invokes a theory-specific decision procedure (a *theory solver*) on (partial) satisfying assignments for the abstraction. If the assignment is found to be consistent in the given theory, then a model for the original formula has been found. Otherwise, a refinement of the propositional abstraction is extracted from the proof of inconsistency and the search is resumed. We describe a theory solver for *integer difference logic* that is effective when the formula to be decided contains equality and disequality (negated equality) constraints so that the decision problem partakes of the nature of the pigeonhole problem. We propose a reduction of the problem to propositional satisfiability by computing bounds on a sufficient subset of solutions, and present experimental evidence for the efficiency of this approach.

1. Introduction

Decision procedures for fragments of first-order logic have been the subject of intense scrutiny in the last few years. On the one hand, emerging applications like model checking of infinite state systems rely on such decision procedures for tasks like predicate abstraction [1]. On the other hand, algorithmic advances have significantly increased the range of problems that can be tackled, and hence have stimulated interest.

In particular, dramatic increase in performance of propositional satisfiability (SAT) solvers has led to the development of decision procedures that rely on the *propositional abstraction* of formulae from more expressive logics like the logic of linear arithmetic constraints, Presburger arithmetic, or the logic of equality and uninterpreted function symbols (EUF). The propositional abstraction of a formula is obtained by replacing the atomic formulae of the specific theory (e.g., $x-y \leq 5$ or $f(x) = f(y)$, where f is an uninterpreted function symbol) with fresh propositional variables. The satisfying assignments of the abstraction map to conjunctions of literals in the original formula that can be checked for consistency with theory-specific procedures. If such a procedure establishes consistency, then the given formula is satisfiable and the enumeration terminates. Otherwise, from the proof of inconsistency a refinement of the propositional abstraction is extracted and the search is resumed.

There are several ways to combine the propositional reasoning engine with the theory-specific procedures. One broad classification is the one into *lazy* and *eager* approaches. A lazy solver produces an initial propositional approximation that is concise and possibly quite coarse; it relies on the refinements during the enumeration of solutions. By contrast, an eager solver adds constraints to the initial propositional abstraction that embody known relationships among the literals. An example is given by the constraints that encode transitivity of equality. The most effective solvers often adopt elements of both approaches and tailor their strategies to the theory (theories) at hand.

In this paper we focus on *Integer Difference Logic* (IDL), in which arithmetic atomic formulae constrain the difference between the values of pairs of integer variables. This logic finds extensive application to problems involving timing and scheduling constraints, resource allocation, and program analysis. IDL is closely related to *Real Difference Logic* (RDL), to the point that a decision procedure for the latter based on propositional abstraction also works for the former, as long as the coefficients are integers. It is sufficient to rewrite *equality* constraints (of the form $x - y = n$) as the conjunction of two inequalities. However, if an equality constraint is negated, then the conjunction turns into a disjunction, which requires case splitting in the enumeration of the propositional solutions. In contrast, we propose an approach that does not decompose equalities and their negations; rather, it converts the problem of checking satisfiability of a conjunction of arithmetic atomic formulae into a set of propositional satisfiability checks—whose cardinality is bounded by the number of maximal strongly connected components (SCC) of a suitable constraint graph.

The conversion to propositional satisfiability that we propose is based on the ability to bound the values of the integer variables that appear in the formula. While in general such bounds do not exist, we show that to decide satisfiability of a set of constraints whose graph is a single SCC it is sufficient to consider a subset of the solutions for which bounds are easily established. We also show how the general case can be efficiently solved given solutions for the individual SCCs of the constraint graph. Experimental study shows that our new approach greatly improves the efficiency of our decision procedure for problem instances in which disequalities play a significant role, and makes it very competitive with respect to state-of-the-art tools.

The rest of this paper is organized as follows: Section 2 reviews background and introduces notation. Section 3 discusses the bounds on solutions, while Sect. 4 delves into the details of our theory solver. After a brief survey of related work in Sect. 5, experiments are presented in Sect. 6, and conclusions are offered in Sect. 7.

2. Preliminaries

Let P be a set of propositional variables and X a set of integer-valued variables. We define inductively *integer difference logic* (IDL) formulae as follows.

- $p \in P$ is a (propositional atomic) IDL formula.
- $x - y \leq n$ and $x - y = n$ are (arithmetic atomic) IDL formulae, for $x, y \in X$, $n \in \mathbb{Z}$.
- If φ and ψ are IDL formulae, so are $\varphi \wedge \psi$ and $\neg\varphi$.

[*]This work was supported by SRC contract 2005-TJ-920.

The following abbreviations are also defined:

$$\begin{array}{rclrcl} x-y<n & \doteq & x-y \leq n-1 & x-y \neq n & \doteq & \neg(x-y=n) \\ x=y & \doteq & (x-y=0) & x \neq y & \doteq & \neg(x=y) \\ \varphi \vee \psi & \doteq & \neg(\neg\varphi \wedge \neg\psi) \ . \end{array}$$

A literal is an atomic formula, or the negation of an atomic formula. A *clause* is the disjunction of a set of literals, and a formula in *conjunctive normal form* (CNF) is the conjunction of a set of clauses. Note that $x-y=n$ could be defined as an abbreviation $((x-y \leq n) \wedge (x-y \geq n))$. We choose not to do so to stress that our algorithm does not split equalities or disequalities (as mentioned in Sect. 1) and also to keep the definition of clausal formulae simple.

A *model* for an IDL formula φ is a pair of functions $\alpha : X \rightarrow \mathbb{Z}$ and $\beta : P \rightarrow \{\mathsf{false}, \mathsf{true}\}$ such that replacing each variable $x \in X$ with $\alpha(x)$, and every variable $p \in P$ with $\beta(p)$ in φ produces a true statement. A formula is *satisfiable* if it has a model, and is *valid* if every assignment is a model. If a conjunction of literals φ is not satisfiable, then a (minimal) *explanation* for the unsatisfiability of φ is the conjunction of a (minimal) subset of the literals in ψ which is not satisfiable.

Propositional logic is the fragment of IDL obtained by omitting the rule that defines arithmetic atomic formulae. Efficient algorithms to decide the satisfiability of propositional logic formulae are based on the DPLL procedure [7, 6], and exploit techniques like clause recording, conflict analysis, nonchronological backtracking, and fast Boolean constraint propagation [23, 20].

In recent times, decision procedures for IDL, and other fragments of quantifier-free first-order logic, have been based on the DPLL procedure as well. Given a set of propositional variables B such that $B \cap P = \emptyset$, one obtains a propositional formula φ^b from an IDL formula φ by replacing each arithmetic atomic subformula of φ with a distinct variable from B. The resulting formula φ^b is unsatisfiable only if φ is unsatisfiable. Each model of φ^b corresponds to a conjunction of literals of φ. Given a decision procedure for the conjunction of arithmetic atomic propositions in IDL (a *theory solver*), one therefore derives a complete decision procedure for IDL by enumerating the models of φ^b, extracting from each of them the corresponding conjunction of arithmetic atomic propositions and their negations, and checking these conjunctions for satisfiability using the theory solver. In the following, we refer to the conjunction of a set of arithmetic literals as a *set of IDL constraints*.

An edge integer-labeled directed graph is a triple $G = (V, E, \lambda)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $\lambda : E \rightarrow \mathbb{Z}$ is an edge labeling function. A *strongly connected component* (SCC) of G is a subgraph G' of G such that every two nodes of G' are connected by a path in G'. SCC G' is *maximal* if no subgraph of G that is a proper superset of G' is an SCC; it is *trivial* if it consists of one vertex and no arcs. The maximal SCCs of G define a partition of V. The *SCC quotient graph* $\widehat{G} = (\widehat{V}, \widehat{E})$ of G is a directed acyclic graph with one vertex for each maximal SCC of G and an edge $(A, B) \in \widehat{E}$ if and only if there exist $x \in A$ and $y \in B$ such that $(x, y) \in E$.

Given a distinguished source vertex $s \in V$, distances of all vertices from s are well defined provided there exists no *negative cycle* in G; that is, no cycle such that the sum of the labels on the edges along the cycle is negative. The Bellman-Ford algorithm [5] reports negative cycles if they are present, and computes the distance $\delta(x)$ of each vertex in V from the source s otherwise. The *slack* of an edge $(x, y) \in E$ is given by $\sigma((x, y)) = \lambda((x, y)) - (\delta(y) - \delta(x))$. It is easy to see that for all $e \in E$, $\sigma(e) \geq 0$ and that $\sigma((x, y)) = 0$ if and only if (x, y) is on a shortest path from s to y in G. Distances and slacks obviously depend on the choice of source vertex.

Given a (finite) set I of inequality constraints (i.e., of the form $x - y \leq n$), their *constraint graph* $G = (V, E, \lambda)$ is a labeled directed graph defined as follows:

- $V \subseteq X$ is the set of variables appearing in the constraints in I.
- There is an arc $(x, y) \in E$ with $\lambda((x, y)) = n$ if and only if there is a constraint $y - x \leq n$ in I.

It is well known [5] that I is satisfiable if and only if G contains no negative cycle. In fact, adding both sides of the constraints forming a cycle of length w, one gets $0 \leq w$, which is not satisfiable when $w < 0$. If, on the other hand, no negative cycle exists in G, then one can find a model for I by solving a single-source shortest-path problem on an augmented graph G_a, obtained from G by adding a new reference vertex x_r and arcs labeled 0 from x_r to all the other vertices. Let $\delta(x)$ be the distance of $x \in V$ from x_r in G_a. Then δ is a model for I. It is also well known that, given a model of I, $\alpha : V \rightarrow \mathbb{Z}$, and a constant, $c \in \mathbb{Z}$, the assignment $\alpha' : V \rightarrow \mathbb{Z}$ defined by $\alpha'(x) = \alpha(x) + c$ is also a model of I, because $\alpha'(x) - \alpha'(y) = \alpha(x) - \alpha(y)$. This observation allows an easy encoding of *range constraints* in IDL. A set of constraints $\{l_i \leq x_i \leq u_i\}$ is translated to $\{x_i - y \leq u_i\} \cup \{y - x_i \leq -l_i\}$, where y is a fresh variable. The solution α obtained from the constraint graph is then translated so that $\alpha'(y) = 0$.

Since integer labels imply integer distances, if the right-hand sides of the constraints are integer-valued, and the constraints are satisfiable when the variables are real-valued, then an integer-valued solution is also guaranteed to exist. Loosely speaking, the satisfiability problem for *inequalities* is the same for IDL and real difference logic (RDL). Adding *equality* constraints to the inequalities does not change this state of affairs: Given a constraint $x - y = n$, one replaces x by $y + n$; if no immediate inconsistencies arise, one continues with the construction of the constraint graph. In contrast, if disequality constraints (i.e., negations of equalities) are allowed, an unsatisfiable conjunction of IDL constraints may be satisfiable when regarded as an RDL formula. An example is given by $\bigwedge_{1 \leq i \leq p}(1 \leq x_i \leq h) \wedge \bigwedge_{1 \leq i < j \leq p}(x_i \neq x_j)$, which exemplifies the pigeonhole principle.[1]

3. Bounds on Solutions

It was recalled in Sect. 2 that from a solution α to a set of inequality constraints, one can derive a family of solutions $\{\alpha + n\}$. In general, however, not all solutions are obtained one from the other by *translation*. Consider the constraints $\{(x - y \leq 1), (y - x \leq 0)\}$. It is easy to verify that the two assignments $\alpha_1(x) = 0$, $\alpha_1(y) = 0$ and $\alpha_2(x) = 1$, $\alpha_2(y) = 0$ satisfy the constraints, though there is no n such that $\alpha_1 = \alpha_2 + n$. Such solutions are called *independent*. In general, there may be several families of independent solutions, and therefore, multiple distinct solutions that assign a given value to a distinguished variable. The following result characterizes these sets of solutions and forms the basis for our treatment of disequality constraints in IDL.

Theorem 1 *Let I be a set of inequality constraints. Let $G = (V, E, \lambda)$ be the constraint graph associated to I. Suppose that G contains no negative cycle and consists of one maximal SCC. For $x \in V$ and $n \in \mathbb{Z}$, let S_x^n be the set of solutions $\alpha : V \rightarrow \mathbb{Z}$*

[1]This does not contradict what was observed in Sect. 1 because $x \neq y$ translates into $(x < y) \vee (y < x)$ for RDL, but translates into $(x \leq y - 1) \vee (y \leq x - 1)$ for IDL.

to I such that $\alpha(x) = n$. Then, for each vertex $y \in V$, there exist bounds y_l and y_u such that for every solution in S^n_x, $y_l \leq \alpha(y) \leq y_u$:

$$\forall y \in V \, . \, \exists y_l, y_u \in \mathbb{Z} \, . \, \forall \alpha \in S^n_x \, . \, y_l \leq \alpha(y) \leq y_u \enspace .$$

PROOF. By definition of SCC, every vertex in V is reachable from x in G; likewise, x is reachable from any vertex in G. Let δ_{xy} be the distance of y from x (the length of a shortest path). Such a distance is defined because there are no negative cycles in G. Adding both sides of all the constraints along the path yields $y - x \leq \delta_{xy}$. Therefore, for every solution $\alpha \in S^n_x$, it must be $\alpha(y) \leq n + \delta_{xy}$. Said otherwise, $y_u = n + \delta_{xy}$. For the lower bound, let δ_{yx} be the distance of x from y in G. Then, for every solution $\alpha \in S^n_x$, it must be $\alpha(y) \geq n - \delta_{yx}$; that is, $y_l = n - \delta_{yx}$. $\square$

Satisfaction of disequalities is not affected by translation. Therefore, a set of constraints including both inequalities and disequalities is satisfiable if and only if there exists a solution α such that $\alpha(x) = n$. This allows us to limit the search to the set S^n_x. Theorem 1 asserts that solutions in this set are bounded. Hence, we can resort to finite instantiations to find them. Specifically, we can encode each integer variable with binary variables and translate the satisfiability problem for a conjunction of inequality and disequality constraints into a propositional satisfiability problem.

Theorem 1 applies when the constraint graph consists of one maximal SCC. If that is not the case, we examine the SCC quotient graph one SCC at the time. If there is no negative cycle in the constraint graph G, the only reason for unsatisfiability is the inability to satisfy the disequalities within some SCC of G. Therefore, if the finite instantiation of each SCC is satisfiable, the entire set of constraints is satisfiable. This can be shown as follows.

Let G be the constraint graph. Extend G by adding one edge for every disequality constraint $x - y \neq n$ (where n may be 0) such that x and y belong to different SCCs. Let $\preceq$ be the preorder defined by $u \preceq v$ if there is a path in G from u to v. (The preorder is updated after each edge addition.) If $x \preceq y$, add $y - x \leq -n - 1$ to E; if $y \preceq x$, add $x - y \leq n - 1$. If x and y are not comparable in the preorder, add either $y - x \leq -n - 1$ or $x - y \leq n - 1$, but not both. Note that adding these edges does not create cycles, and therefore does not change the SCCs of G. (See Sect. 4.)

Let $\widehat{G} = (\widehat{V}, \widehat{E})$ be the SCC quotient graph of the extended G. Consider the vertices in $\widehat{V}$ starting from the minimal SCCs (those with no predecessors) and proceeding in a chosen topological order. Let A_i be the i-th SCC in that order and let α_i be a solution for the constraints corresponding to its edges. Inductively assume that β_{i-1} is a solution for the constraints in the subgraph induced by $\bigcup_{0<j<i} A_j$. Let k be the maximum amount by which any constraint corresponding to an edge into A_i is violated. (Let $k = 0$ if no such violation exists.) Finally, let $\alpha'_i = \alpha_i - k$. Then, $\beta_i = \beta_{i-1} \cup \alpha'$ is a solution for the constraints in the subgraph induced by $\bigcup_{0<j\leq i} A_i$.

4. Algorithm

We assume a decision procedure for IDL based on propositional abstraction. The given IDL formula φ is translated into a propositional formula φ^b as described in Sect. 2. A *propositional reasoning engine* enumerates the satisfying assignments to φ^b and calls the *theory solver* to determine whether those satisfying assignments correspond to consistent assignments to the integer-valued variables.

The theory solver for IDL is relatively efficient. Therefore, it is advantageous to call it also on partial assignments to terminate the fruitless search of part of the state space, or to learn so-called *theory consequences* [21]. Our implementation follows this approach, though the equality constraints are split and the full check for inconsistencies due to disequalities is applied only to complete assignments. (See lines 38–42 of Fig. 1.) We omit the details of the incremental implementation of the Bellman-Ford algorithm. The interested reader is referred to [26].

4.1 The Theory Solver

The theory solver is called with a collection of arithmetic literals whose corresponding propositional literals are true in a model of the propositional formula φ^b; it then decides whether there is an assignment to the integer-valued variables that satisfies the conjunction of all those literals. The first step is to obtain a set of arithmetic atomic formulae (without negations) from the given set of literals. The given literals are rewritten according to their form:

1. $x - y \leq n$: unchanged;
2. $x = y$: unchanged;
3. $x - y = n$, with $n \neq 0$: split into $(x - y \leq n) \wedge (y - x \leq -n)$;
4. $\neg(x - y \leq n)$: rewritten as $y - x \leq -n - 1$;
5. $\neg(x = y)$: rewritten as $x \neq y$;
6. $\neg(x - y = n)$, with $n \neq 0$: rewritten as $x - y \neq n$.

Constraints of type 1, 3, and 4 are *inequalities* (I). Constraints of type 2 are *equalities* (Q), and finally, constraints of type 5 and 6 are *disequalities* (D). Specifically, constraints of type 5 form the set $D_0 \subseteq D$. Let $C = I \cup Q \cup D$.

The theory solver, whose pseudocode is shown in Figures 1 and 2, adopts the *layered* approach of MathSAT [4]. For IDL, it considers three main layers: equalities, inequalities, and disequalities. Let $X_= \subseteq X$ be the set of integer-valued variables appearing in Q. The theory solver creates an undirected equality graph $\mathcal{Q} = (X_=, \Gamma)$, where

$$\Gamma = \{\{x_i, x_j\} : x_i = x_j \in Q\} \enspace .$$

The vertices of $\mathcal{Q}$ are in the same class if they are made equivalent by the equality constraints. The feasibility of Q with D_0 is checked by comparing the equivalence class of the two vertices of each disequality constraint in D_0. If two vertices are in the same class, an explanation of infeasibility is returned. If the set of equality constraints is feasible, the variables in the same class are merged into a single variable, and some simplified constraints in D_0 and I are dropped from the set.

The algorithm continues by checking the feasibility of the set of inequality constraints. Let $V \subseteq X$ be the set of integer-valued variables appearing in I. The theory solver creates a constraint graph $G = (V, E, \lambda)$ from I as explained in Sect. 2. The Bellman-Ford algorithm is run on G. If a negative cycle is found, the set I is infeasible; a negative cycle with a subset of Q provides the explanation of infeasibility. Equality constraints are involved in the explanation if the constraints on the negative cycle were obtained by simplification in the equality layer. If there is no negative cycle in G, the set $I \cup Q$ is feasible; therefore a solution $\delta : V \to \mathbb{Z}$ is returned by the Bellman-Ford algorithm.[2]

The (simplified) set I combined with D is considered in the next step. Let G_0 be the subgraph of G such that the edges with non-zero slacks for solution δ are removed from G. Since the slacks of

[2]The algorithm is, in principle, applied to the augmented graph G_a described in Sect. 2. In practice, no augmentation of G is required: it suffices to initialize all distances to 0.

```
TheorySolver ($C$) {
    $\mathcal{Q}$ = CreateEqualityGraph ($Q$);
    $Explanation$ = CheckFeasibilityOfEqualityConstraints ($\mathcal{Q}$, $D_0$);
    if ($Explanation \neq SAT$) return $Explanation$;
    else {
        MergeVariablesInSameClass ($\mathcal{Q}$);
        DropSimplifiedConstraints ($C$);
        return CheckFeasibilityOfInequalityConstraints ($I$);
    }
}

CheckFeasibilityOfInequalityConstraints ($I$) {
    $G$ = CreateConstraintGraph ($I$);
    $NegCycle$ = BellmaFordAlgorithm ($G$);
    if ($NegCycle$) {
        return GenerateExplanationFromNegCycle ($NegCycle$);
    } else {
        $SCC$ = GenerateZeroSlackSccOfConstraintGraph ($G$);
        $Explanation$ = CheckFeasibilityOfZeroSlackScc ($SCC$, $D$);
        if ($Explanation \neq SAT$) return $Explanation$;
        else {
            $SCC'$ = GeneratePositiveSlackSccOfConstraintGraph ($G$);
            return CheckFeasibilityOfPositiveSlackScc ($SCC'$, $D$);
        }
    }
}

CheckFeasibilityOfZeroSlackScc ($SCC$, $D$) {
    For each $d \in D$ {
        $Explanation$ = CheckFeasibilityOfDisequalityConstraint ($SCC$, $d$);
        if ($Explanation \neq SAT$) return $Explanation$;
        else DropValidConstraint ($d$,$D$);
    }
    return $SAT$;
}

CheckFeasibilityOfPositiveSlackScc ($SCC'$, $D$) {
    for each $scc' \in SCC'$ {
        $(L, U)$ = GenerateBoundsForEachVariableInScc ($SCC'$);
        $Explanation$ = CheckFeasibilityOfBoundsWithClique($SCC'$, $D$, $L$, $U$);
        if ($Explanation = UNDECIDED$ or $Explanation = PROB\_SAT$ and assignment is complete) {
            $CNF$ = SmallDomainEncodingForConstraintsInScc ($SCC'$, $D$, $L$, $U$);
            $Explanation$ = SatSolver ($CNF$);
            if ($Explanation \neq SAT$) return $Explanation$;
        }
        else return $Explanation$;
    }
    return $SAT$;
}
```

Figure 1: Theory Solver Algorithm

the edges of G_0 are zero, the difference between the values of two variables in the same SCC of G_0 is the same in all solutions to the constraints. In fact, each cycle in G_0 is of length 0 [17]; hence, if x and y are on one cycle of G_0 and the distance from x to y along the cycle is k, then the distance from y to x is $-k$. It follows that every solution to I must satisfy $y - x \leq k$ and $x - y \leq -k$, that is, $y - x = k$. In other words, a maximal SCC of G such that its vertex set induces also a maximal SCC of G_0 has only one family of solutions. (See Sect. 3.)

Each disequality constraint $d \in D$ is checked for feasibility against each SCC of G_0. If the two variables x, y in $x - y \neq n$ (where n may be 0) are in the same SCC of G_0 and $\delta(x) - \delta(y) = n$, then the set $I \cup Q \cup D$ is infeasible. The violated disequality d, together with the cycle that contains x and y and an appropriate subset of Q constitutes the explanation of infeasibility. If the two variables x and y in d are in the same SCC and $\delta(x) - \delta(y) \neq n$, then d is dropped from the set. Disequalities connecting variables in different SCCs of G_0 are simply passed on to the next phase of

```
47 GenerateBoundsForEachVariableInScc ($scc'$) {
48     $x$ = FixValueOfOneVertexInScc ($scc'$);
49     $U$ = ComputeUpperBoundForEachVariableInScc ($scc'$,$x$);
50     $L$ = ComputeLowerBoundForEachVariableInScc ($scc'$,$x$);
51     return $(L, U)$;
52 }

53 ComputeUpperBoundForEachVariableInScc ($scc'$,$x$) {
54     return BellmanFordAlgorithmWithFixedVertex ($scc'$,$x$);
55 }

56 ComputeLowerBoundForEachVariableInScc ($scc'$,$x$) {
57     $rev$ = ReverseDirectionOfEdgesInScc ($scc'$);
58     return BellmanFordAlgorithmWithFixedVertex ($rev$,$x$);
59 }

60 CheckFeasibilityOfBoundsWithClique ($SCC'$, $D$, $L$, $U$) {
61     $V$ = GatherVariablesInDisequalityConstraints($D$);
62     $\Gamma$ = GatherVariablesWithSameBounds ($D$, $L$, $U$);
63     $\rho$ = GetBoundForGatheredVariables ($\Gamma$);
64     $D'$ = CollectRelevantDisequalityConstrints ($D$,$\Gamma$);
65     $\Gamma'$ = RemoveIrrelevantVariableByCheckingDegree ($\Gamma$, $D'$);
66     if ($n(\Gamma') \leq \rho$ and $n(V) = n(\Gamma)$) return $PROB\_SAT$;
67     else if ($n(\Gamma') \leq \rho$ and $n(V) \neq n(\Gamma)$) return $UNDECIDED$;
68     if ($n(D') < ((n(\rho) \cdot (n(\rho) + 1))/2$ and $n(V) = n(\Gamma)$) return $PROB\_SAT$;
69     else if ($n(D') < ((n(\rho) \cdot (n(\rho) + 1))/2$ and $n(V) \neq n(\Gamma)$) return $UNDECIDED$;
70     $C$ = GenerateMaxClique ($\Gamma'$, $D'$);
71     $V'$ = GetVariablesInMaxClique ($C$);
72     if ($n(V') < \rho$ and $n(V) = n(\Gamma)$) return $PROB\_SAT$;
73     else if ($n(V') < \rho$ and $n(V) \neq n(\Gamma)$) return $UNDECIDED$;
74     else return GenerateExplanationFromMaxClique ($SCC'$,$C$);
75 }

76 SmallDomainEncodingForConstraintsInScc ($scc'$, $D$) {
77     $CNF$ = InitializeCNF ();
78     $CNF$ = $CNF \cup$ EncodingForBoundsOfEachVariableInScc ($scc'$);
79     $CNF$ = $CNF \cup$ EncodingForInequalityConstraintsInScc ($scc'$);
80     $CNF$ = $CNF \cup$ EncodingForDisequalityConstraints ($D$);
81     return $CNF$;
82 }
```

Figure 2: Theory Solver Algorithm (continued)

the procedure. If no infeasibility is detected with G_0, a final feasibility check is performed by the small domain encoding method discussed in Sect. 3. For each SCC of G, Theorem 1 is used to compute bounds for each variable as follows.

To compute the upper bound for each variable, a variable in the SCC is chosen arbitrarily as source. (Variable x in Theorem 1.) The distance from it is computed for each variable in the SCC by the Bellman-Ford algorithm. The lower bound for a variable is computed as its distance from the same source variable used to compute the upper bound after reversing the edges in the SCC. (Note that one cannot replace the distances computed by these invocations of the shortest path algorithm with those computed on G_a.)

Some inequalities and disequalities may be automatically satisfied for all values of the variables in their ranges. For instance, if $0 \leq x \leq 1$ and $2 \leq y \leq 3$, then $x \neq y$ and $y - x \leq 4$ are both satisfied. These constraints are therefore ignored in the successive steps, which consist of a quick check based on finding a clique of the disequality graph, possibly followed by propositional encoding and satisfiability check.

The quick check is based on two observations: The first is that if all variables in the SCC have the same range, then the disequalities define a graph whose chromatic number must not exceed the size of the range for the constraints to be satisfiable. (The chromatic number is the least number of colors needed to assign different colors to adjacent vertices in the graph.) The second observation is that the chromatic number of a graph is bounded from below by the size of a clique of the graph and from above by the number of vertices. The process is described in lines 60–75 of Fig. 2. We identify sets of variables that have the same bounds and we check whether there are enough disequalities for the variables in one such set to cause inconsistency. Specifically, suppose a set $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ of variables is found such that all variables in Γ have the same bounds y_l and y_u. Let $\rho = y_u - y_l + 1$ be the range of each variable in Γ. If $p < \rho$ disequalities cannot cause inconsistency of this set of variables. If, on the other hand, the number of variables exceeds their common range, we check whether the disequalities form a clique of size greater than ρ. We first eliminate from Γ all variables that appear in fewer than ρ disequalities of the form $\gamma_i \neq \gamma_j$

($\gamma_i, \gamma_j \in \Gamma$). If Γ is not empty after this process, we greedily grow a clique, adding every time the variable appearing the largest number of disequalities among the surviving members of Γ. This greedy algorithm does not always find the largest clique, but is fast and works well in practice. The check results in one of three results: A suitable clique has been found and inconsistency is declared; a large enough clique was not found because of the heuristic nature of the algorithm; a large enough clique is known not to exist. In the first case, an explanation of inconsistency is derived from the disequalities forming the clique and the inequalities responsible for the bounds. In the last two cases, the result is inconclusive, because the chromatic number of a graph can be arbitrarily larger than the size of even the largest cliques. However, if a large enough clique does not exist in the graph, and the assignment is partial, we avoid a full check for inconsistency, which is rather expensive and likely to fail. (If the assignments to the boolean variables are complete, on the other hand, the consistency check must be performed for the whole decision procedure to be sound.)

In the final step of the theory solver, the constraints and the bounds are converted to a set of clauses whose satisfiability is established by calling a propositional SAT solver.[3] If the clauses are satisfiable, an assignment for the integer variables is extracted from the solution. Otherwise, an explanation for the unsatisfiability is derived as follows from the proof of unsatisfiability returned by the SAT solver, which consists of a subset of the clauses that are found to be unsatisfiable. (The *unsatisfiable core*.)

Every propositional clause is derived from some arithmetic constraint. If a clause appears in the unsatisfiable core, the parent constraint is included in the explanation. The bound constraints on the integer variables also contribute to unsatisfiability. They are accounted for by including all constraints that form the two shortest path spanning trees found during the computation of the bounds.

5. Related Work

Propositional abstraction as an approach to satisfiability modulo theories was proposed in [2]. Notable solvers based on that principle are MathSAT [4, 3], ICS [8], Verifun [10], BarcelogicTools [12, 21], SLICE [26], and SATORI [13]. ASAP [16] takes a dual approach, in which satisfiability of the propositional abstraction guarantees satisfiability of the original quantifier-free Presburger formula, while UCLID [18] is an eager solver. Our propositional enumeration engine is the one of [14, 15].

Finite instantiations for equality logic are studied in [22] and extended to difference logic in [25]; this last work has several points of contact with ours, but also important differences. The approach of [25] is eager, and the ranges are computed once and for all before invoking the propositional SAT solver. In contrast, we advocate a lazy approach and a computation of the ranges that takes place in the theory solver. Because of that, we may compute ranges more than once, but the size of the range for each variable in our algorithm is bounded by the sum of the slacks in the SCC, which is much smaller than $n + maxC$, where $maxC$ is the sum of absolute constants in the formula. In practice, ranges are much smaller in our algorithm. Moreover, we compute ranges by simply finding shortest paths in the constraint graph. The algorithm of [25], on the other hand, enumerates paths in the constraint graph and is exponential in the worst case.

Recent work by Ganai *et al.* [11] presents a polynomial algorithm for the computation of ranges, which improves over the one of [25], but shares the basic approach: ranges are allocated initially, so as to be adequate for every formula built from the given set of difference constraints. Disequalities are converted to disjunctions of inequalities, instead of being retained as such in the formulation of the problem. The theory consistency problem is never converted to propositional satisfiability. Instead, range propagation allows the solver to refine the initial ranges.

MathSAT introduced the notion of layered, incremental theory solver, and that of delayed theory combination; DPLL(t) the idea of exhaustive theory propagation, both of which are included in our implementation. The importance of considering zero-slack SCCs was first pointed out in [17], which deals with RDL. Finally, [26] discusses an efficient way to implement a recursive, backtrackable Bellman-Ford algorithm.

6. Experimental Results

We have implemented the algorithm presented in Sect. 4 in Sateen, a theorem prover for quantifier-free first-order logic that combines the propositional reasoning engine of [14, 15] with theory-specific procedures. A first set of experiments were done with the full set of QF_IDL (Quantifier free integer difference logic) benchmarks from SMT-COMP (Satisfiability Modulo Theories Competition [24]). The experiments were performed on a 1.7 GHz Pentium 4 with 2 GB of RAM running Linux. Time out was set at 3600 seconds. Sateen was compared with BarcelogicTools [9], Yices-0.1.1 [27] and MathSAT 3.3.1 [19]. The compared solvers are the ones that were submitted to SMT-COMP in 2005.

Figures 3–5 show scatterplots comparing BarcelogicTools, Yices and MathSAT to Sateen. Points below the diagonal represent wins for Sateen. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^\eta$, where κ and η are obtained by least-square fitting. Figure 3 shows that Sateen is comparable to BarcelogicTools. In Figures 4 and 5, Sateen shows better results compared to Yices and MathSAT, especially on hard problems. The SMT-COMP benchmark formulae are such that usually the sets of constraints passed to the theory solver either contain few disequality constraints, or are such that the disequality constraints are dealt with by the zero-slack SCC algorithm. The main purpose of these experiments is therefore not to show the effectiveness of the newly proposed algorithm for finite instantiations, but to establish that Sateen is, overall, a competent solver for IDL, comparable to some of the best tools in the field.

To assess the effectiveness of the finite instantiation approach, we have generated two benchmark suites where disequality constraints play a significant role: the Queens Suite and the Job Shop Scheduling Suite. The Queens Suite contains n-Queens problem and n-Super-Queens problem. The n-Queens problem consists of placing n queens on a $n \times n$ board so that they do not attack each other. In the n-Super-Queens problem, each queen's placement is more restricted by allowing it also the knight's moves. The Job Shop Scheduling problem checks the feasibility of processing a number of jobs, each consisting of several tasks, on a given set of machines in a given amount of time. These two sets of benchmarks have disequality constraints that cause pigeonholing problems. In the experiment on these benchmarks, the timeout was set to 1000 seconds.

Figures 6–8 shows that Sateen is often orders of magnitude faster than the other solvers on these problems. The symbol $\times$ represents the experiment on the Queens benchmark, and the symbol $+$ represents the experiment on the Job Shop Scheduling benchmark. We also provide the comparison between Sateen with our proposed algorithm and a version of Sateen that splits disequalities. Figure 9 shows that the finite instantiation algorithm works significantly bet-

[3]Our current encoding of the ranges is rather unsophisticated. We are implementing a heuristic approach to minimizing the total number of encoding bits required.

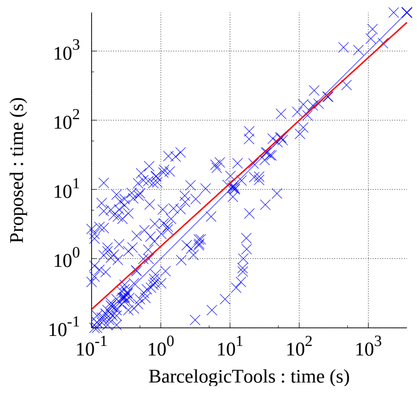

Figure 3: BARCELOGICTOOLS vs. Sateen on QF_IDL

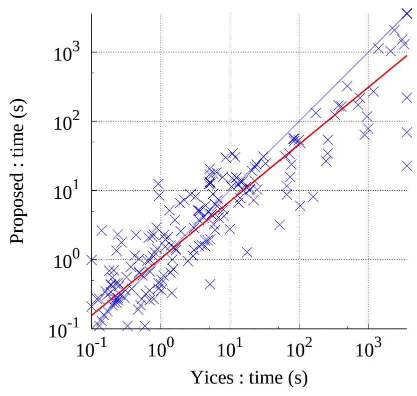

Figure 4: YICES vs. Sateen on QF_IDL

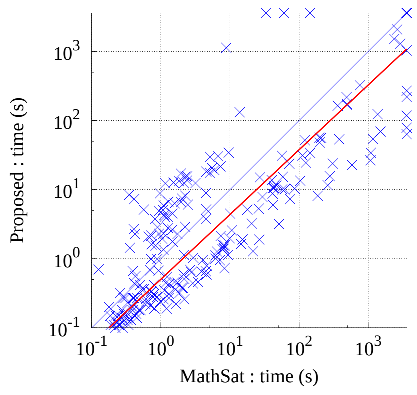

Figure 5: MATHSAT vs. Sateen on QF_IDL

ter than the splitting method. The clique detection algorithm is particularly helpful in Job Shop Scheduling problems.

7. Conclusions

We have presented an approach to solving integer difference logic that is particularly effective when the constraints to be solved are rich in disequalities. By restricting consideration to a small sufficient set of solutions, we are able to compute bounds for the integer variables occurring in the constraints. Experiments indicate that this approach is more effective than splitting disequalities into the disjunction of inequalities. Further improvements in efficiency are expected from a more sophisticated encoding scheme for the finite instances that we are currently developing.

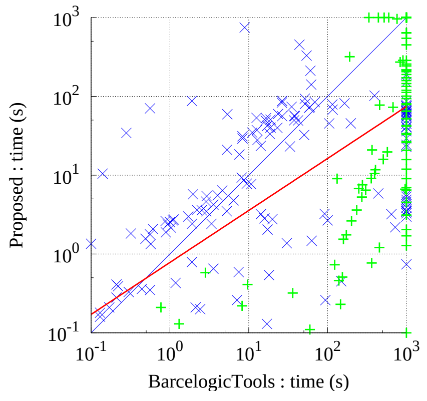

Figure 6: BARCELOGICTOOLS vs. Sateen on Job Shop Scheduling and Queen Suites

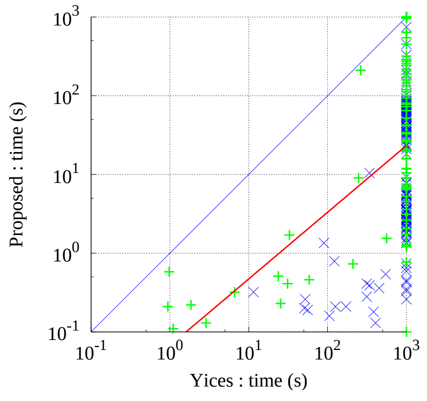

Figure 7: YICES vs. Sateen on Job Shop Scheduling and Queen Suites

Acknowledgment. The authors thank Alberto Oliveras, Alessandro Cimatti, and Leonardo deMoura for their help with BarcelogicTools, MathSAT, and Yices.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, Snowbird, UT, June 2001.

[2] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 236–249. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[3] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Seventeenth Conference on Computer Aided Verification (CAV'05)*, pages 335–349. Springer-Verlag, Berlin, July 2005. LNCS 3576.

[4] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *International Conference on Tools and Algorithms*

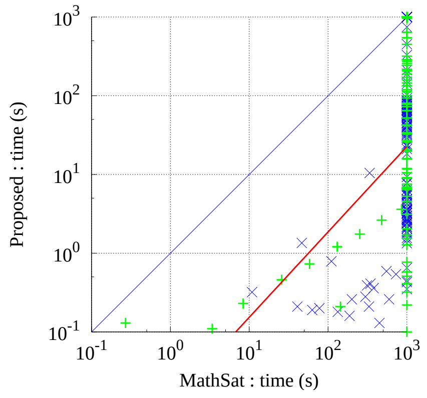

Figure 8: MATHSAT vs. Sateen on Job Shop Scheduling and Queen Suites

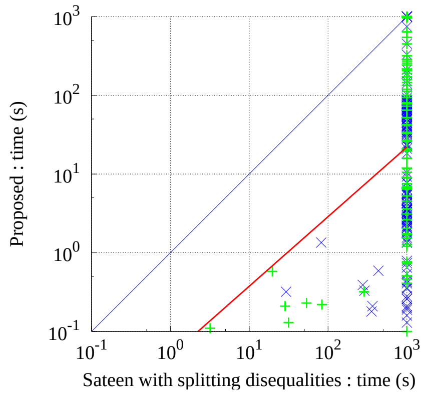

Figure 9: Sateen with splitting disequalities vs. Sateen on Job Shop Scheduling and Queen Suites

for Construction and Analysis of Systems (TACAS'05), pages 317–333, Edinburgh, UK, Apr. 2005. LNCS 3440.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *An Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[8] L. de Moura and H. Reuß. Lemmas on demand for satisfiability solvers. In *Fifth International Symposium on the Theory and Application of Satisfiability Testing (SAT'02)*, Cincinnati, OH, May 2002.

[9] Url: http://www.lsi.upc.edu/ oliveras/bclt-main.html.

[10] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In W. A. Hunt, Jr. and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 355–367. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[11] M. K. Ganay, M. Talupur, and A. Gupta. SDSAT: Tight integration of small domain encoding and lazy abstraction in a separation logic solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06)*, pages 135–150, Vienna, Austria, Mar. 2006. LNCS 3920.

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

[13] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. SATORI – a fast sequential SAT engine for circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 320–325, San Jose, CA, Nov. 2003.

[14] H. Jin, H. Han, and F. Somenzi. Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, pages 287–300, Apr. 2005. LNCS 3440.

[15] H. Jin and F. Somenzi. Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In *Proceedings of the Design Automation Conference*, pages 750–753, Anaheim, CA, June 2005.

[16] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 308–320. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[17] S. Lahiri and M. Musuvathi. An efficient Nelson-Oppen decision procedure for difference constraints over rationals. In *Third International Workshop on Pragmatical Aspects of Decision Procedures in Automated Reasoning (PDPAR'05)*, pages 2–9, Edinburgh, UK, July 2005. To appear in ENTCS.

[18] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 475–478. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[19] Url: http://mathsat.itc.it.

[20] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[21] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Seventeenth Conference on Computer Aided Verification (CAV'05)*, pages 321–334. Springer-Verlag, Berlin, July 2005. LNCS 3576.

[22] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Journal of Information and Computation*, 178(1):279–293, Oct. 2002.

[23] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

[24] Url: http://www.csl.sri.com/users/demoura/smt-comp/.

[25] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allociation for separation logic. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 148–161. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[26] C. Wang, F. Ivancic, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR'2005)*, Montego Bay, Jamaica, Dec. 2005.

[27] Url: http://fm.csl.sri.com/yices.