Difference Logic

Satisfiability Checking Seminar

Alex Ryndin
Supervisor: Gereon Kremer

WS 2016/2017

Abstract

This report describes the difference logic (DL) and a graph-based approach for solving satisfiability (SAT) problem of DL formulas. There is of course a simplex-based algorithm which solves SAT problem of any linear arithmetic (LA) constraints. However, it is not efficient compared to the algorithm described here because simplex does not utilize the simple structure of DL constraints.

Efficiently solving SAT problem of DL constraints is very important because a lot of timing related problems can be described by this logic e.g. scheduling problems, detecting race conditions in digital circuits etc.

The report is organized as follows. Chapter 1 introduces difference logic and SAT checking. Chapter 2 gives theoretical background. Chapter 3 describes a graph-based approach to solving SAT problem of DL. Chapter 4 draws a conclusion.

1 Introduction

DL is a special case of an LA logic in which all LA constraints have the form $x - y \prec c$ where x and y are numerical variables, c is a constant and $\prec \in \{<, \leq\}$ is a comparison operator. A more formal definition of DL is given below [1], [2, p.5]:

Definition 1.1 (Difference Logic) *Let $\mathcal{B} = \{b_1, b_2, \dots\}$ be a set of Boolean variables and $\mathcal{X} = \{x_1, x_2, \dots\}$ be a set of numerical variables. The difference logic over $\mathcal{B}$ and $\mathcal{X}$ is called $DL(\mathcal{X}, \mathcal{B})$ and given by the following grammar:*

$$\phi \stackrel{\text{def}}{=} b \mid (x - y < c) \mid \neg\phi \mid \phi \wedge \phi$$

where $b \in \mathcal{B}$, $x, y \in \mathcal{X}$ and $c \in \mathbb{D}$ is a constant. The domain $\mathbb{D}$ is either the integers $\mathbb{Z}$ or the real numbers $\mathbb{R}$.
The remaining Boolean connectives $\vee, \rightarrow, \leftrightarrow, \dots$ can be defined in the usual ways in terms of conjunction and negation.

Examples of DL formulas are given below:

$$\phi_1 = (p \vee q \vee r) \wedge (p \rightarrow (u - v < 3)) \wedge (q \rightarrow (v - w < -5)) \wedge (r \rightarrow (w - x < 0)) \quad (1)$$

$$\phi_2 = (u - v < 1) \wedge (v - w < 5) \wedge (w - x \leq -3) \wedge (x - y < 1) \wedge (y - z \leq -5) \wedge (y - v \leq 0) \quad (2)$$

$$\phi_3 = (u - v < 1) \wedge (v - w < 5) \wedge (w - x \leq -3) \wedge (x - y < -3) \wedge (y - z \leq -5) \wedge (y - w < 4) \quad (3)$$

2 Preliminaries

2.1 Solving SAT Problem of Propositional Logic

Most of the SAT solvers employ a variation of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [3, 4] for solving SAT problem of the propositional logic (PL). One such basic SAT checking algorithm is given below:

Algorithm 1 A basic SAT checking algorithm for solving SAT problem of PL. It takes a PL formula to be checked for satisfiability and returns SAT status of the formula (SAT or UNSAT) and, in case when the formula is SAT, a model i.e. an assignment, which evaluates the formula to $True$.

```
(SAT Status, Model) Check ( PL formula φ)
 1  model ← ∅
 2  (inferredAssignments, conflictingClauses) ← Propagate(φ, model)
 3  conflictHasArisen ← conflictingClauses ≠ ∅
 4  if conflictHasArisen
 5     then return (UNSAT, nil)
 6  model ← model ∪ inferredAssignments
 7  while True
 8  do (nextVariable, value) ← Decide(φ, model)
 9     allVariablesHaveAlreadyBeenAssigned ← nextVariable = nil
10     if allVariablesHaveAlreadyBeenAssigned
11        then return (SAT, model)
12     model ← model ∪ {nextVariable ← value}
13     repeat
14           (inferredAssignments, conflictingClauses) ← Propagate(φ, model)
15           model ← model ∪ inferredAssignments
16           conflictHasArisen ← conflictingClauses ≠ ∅
17           if conflictHasArisen
18              then (newModelWithoutConflict, φ_a) ← Resolve(φ, model, conflictingClauses)
19                   conflictHasNotBeenResolved ← newModelWithoutConflict = nil
20                   if conflictHasNotBeenResolved
21                      then return (UNSAT, nil)
22                   model ← newModelWithoutConflict
23                   φ ← φ ∧ φ_a
24        until ¬conflictHasArisen
```

The Algorithm 1 is quite generic. It is more of a template. One needs to plug into this template his own implementations of the following functions:

Propagate This function takes a PL formula and a model and returns all assignments that follow from them. It also returns a list of conflicting clauses (if there are any). These clauses will be in conflict with the model if one extends the model by applying the returned assignments.

Decide This function takes a PL formula and a model. Then, by applying some heuristic, it selects a variable and a value for it and returns them.

Resolve This function takes a PL formula, current conflicting model and a list of the clauses which are in the conflict with the current model. It returns a new model which does

not have the conflict and an assertion clause ϕ_a (or a conjunction of assertion clauses) which carries the knowledge about the resolved conflict. If the conflict cannot be resolved, the function returns NIL as the model.

In the Algorithm 1 a model is represented by a mapping from PL variables of the input PL formula to Booleans. It is a simplification. Real-world SAT solvers maintain a lot of additional information such as decision levels, assignment order of variables for every decision level, an implicant for every variable (i.e. the clause from which the variable's value was inferred during the PROPAGATE) etc.

2.2 Solving SAT Problem of Difference Logic

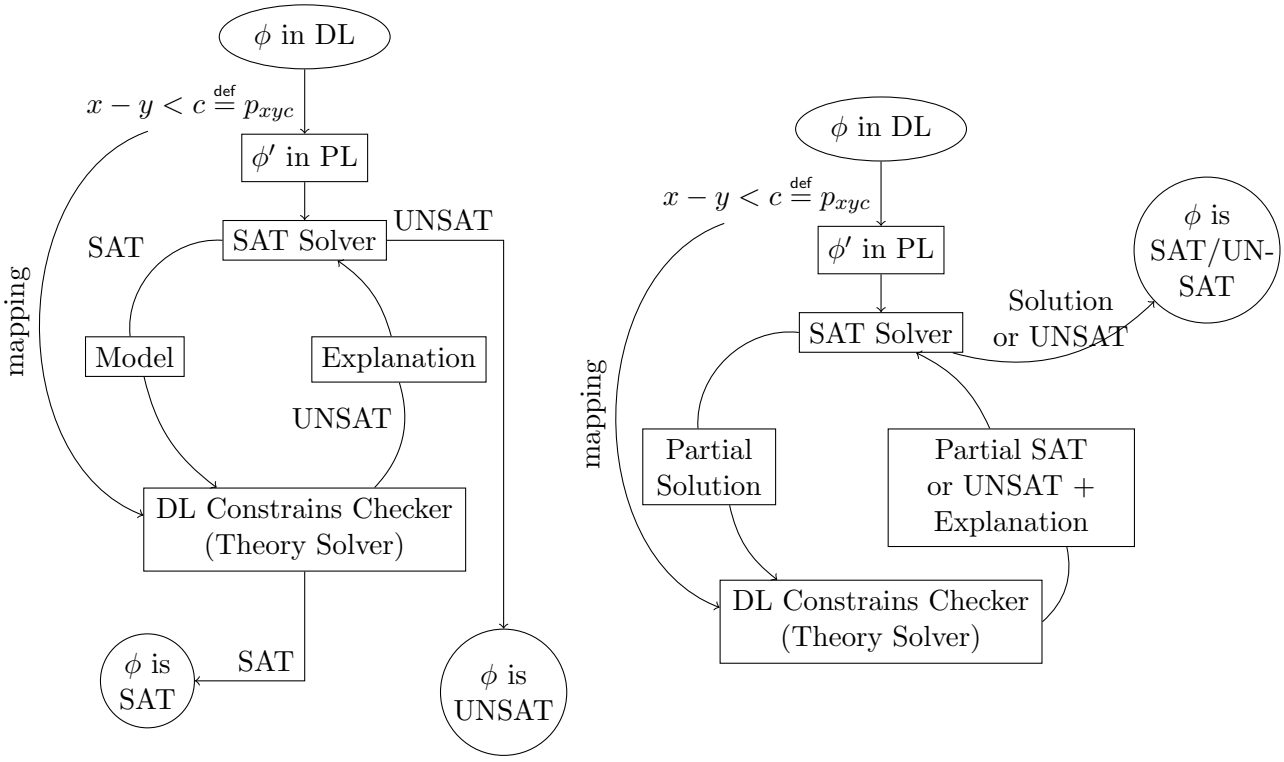

Figure 1: Illustration of the lazy (left) and incremental (right) approaches.

As mentioned in [1] the following main approaches are used for solving the SAT problem of DL:

- Preprocessing approach. This approach suggests transforming a DL formula into an equivalent PL formula by encoding all intrinsic dependencies between DL constraints in PL. An example of such a dependency could be transitivity:

$$(x - y < a) \wedge (y - z < b) \rightarrow (x - z < a + b) \tag{4}$$

  After the transformation a SAT solver can be used to check SAT of the resulting equivalent PL formula. If the PL formula is SAT then the solution for the original DL formula can be constructed by the reverse transformation.

- Lazy approach (Figure 1 left). This approach suggests substituting each DL constraint $x - y < c$ with a Boolean variable $p_{xyc} \in \mathbb{B}$ thus yielding a PL formula ϕ'.

The PL formula represents the "skeleton", the Boolean abstraction over the original DL formula ϕ. Then a SAT solver is used in tandem with a DL constraints checker (the theory solver).

- Incremental approach (Figure 1 right). This approach, like the previous one, also suggests encoding the input DL formula into a Boolean "skeleton" and a SAT solver and a DL constraints checker also work in tandem. However, instead of solving the whole formula and then giving a complete answer to the DL constraints checker, the SAT solver invokes the DL constraints checker each time it updates its model. The DL constraints checker should be able to maintain some internal state of the currently received DL constraints and update it incrementally (i.e. add new constraints, delete existing ones). Hence the name of the approach.

3 Topic

3.1 Constraint Graph

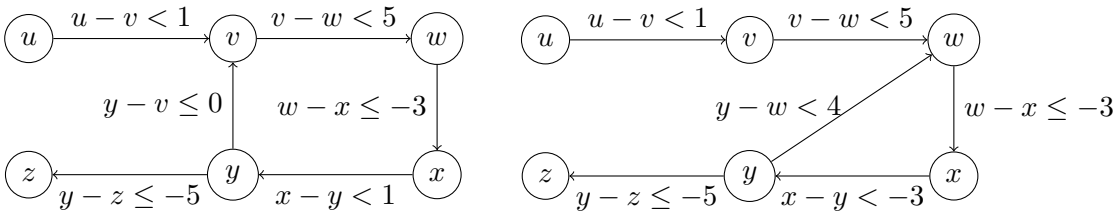

Figure 2: Examples of constraint graphs for Equation 2 (left) and Equation 3 (right).

Constraint graph (Figure 2) is a weighted directed graph which is used by the DL constraints checker (Figure 1) to check if the given DL constraints are satisfiable. Let ϕ be a conjunction of DL constraints (i.e. a DL formula), let $\prec = \{<, \leq\}$ be an operation in a DL constraint $x - y \prec c \in \phi$ and let variables and constants in the constraints be defined over a domain $\mathbb{D}$ (i.e. $x, y, c \in \mathbb{D}$) which can be e.g. Reals ($\mathbb{R}$). Then according to [1] the constraint graph can be defined as follows:

Definition 3.1 (Constraint Graph) *The constraint graph Γ represents a conjunction of DL constraints ϕ. It is a graph $\Gamma = (V, E, weight, op)$ where:*

- *V is a set of vertices. Each vertex $x \in V$ corresponds to one numeric variable occurring in some DL constraint $x - y \prec c \in \phi$.*
- *E is a set of directed edges. Each edge $(x, y) \in E$ corresponds to a DL inequality $x - y \prec c \in \phi$.*
- *$weight : E \mapsto \mathbb{D}$ is a weight function. It maps each edge $(x, y) \in E$ to the constant $c \in \mathbb{D}$ from the corresponding DL inequality $x - y \prec c \in \phi$.*
- *$op : E \mapsto \{<, \leq\}$ is a function which maps each edge $(x, y) \in E$ to the operation $\prec$ from the corresponding DL inequality $x - y \prec c \in \phi$.*

3.2 Negative Cycles in Constraint Graph

If a constraint graph has a negative cycle then the corresponding conjunction of DL constraints ϕ represented by the graph is not SAT.

A path in the graph corresponds to a sum of the corresponding constraints. E.g. the path $u \to v \to w \to x$ in the left graph on Figure 2 corresponds to the following sum of the DL inequalities:

$$\begin{array}{r} u - v < 1 \\ v - w < 5 \\ \hline w - x \leq -3 \\ \hline u - x < 3 \end{array} \tag{5}$$

If at least one strict inequality is present then the resulting inequality will also be strict. This summation along a path can also be expressed with a transitivity constraint (e.g. Equation 4). The transitivity constraint naturally follows from ϕ and therefore must be satisfied in order to satisfy ϕ.

A cycle in the constraint graph corresponds to an inequality $0 \prec c$ which may cause a conflict in the following situations:

- $c < 0$
- $c = 0$ and $\prec$ is $<$ (can be checked with *op* from Definition 3.1)

An example of a conflict can be seen on the right graph on Figure 2. The conflict corresponds to the negative cycle $x \to y \to w \to x$ which corresponds to the following inequalities:

$$\begin{array}{r} x - y < -3 \\ y - w < 4 \\ \hline w - x \leq -3 \\ \hline 0 < -2 \end{array} \tag{6}$$

3.3 Bellman-Ford Algorithm for Constraint Graph

In [1] a Goldberg-Radzik [6] variant of the Bellman-Ford algorithm (Algorithm 2) is applied to a constraint graph in order to detect negative cycles. Important terminology and notation used in the algorithm are given below:

- $d(v) \in \mathbb{D}$ is a distance estimate from the source vertex to the given vertex $v \in V$.
- $\pi(v) \in V$ is a parent of $v \in V$ in a tree of shortest paths. The tree has the source vertex as its root.
- $status(v) = \{unreached, labelled, scanned\}$ denotes if $v \in V$ has or has not been reached yet or has been marked as a scanned i.e. processed vertex.
- $r_d(x, y) = weight(x, y) + d(x) - d(y)$ is a reduction in distance estimate associated with taking a path from the source vertex to $y \in V$ through $x \in V$.
- Edge $(x, y) \in E$ is called admissible if $r_d(x, y) < 0$ i.e. this edge can improve the current distance estimate for the vertex $y \in V$.
- Admissible graph Γ_d is a subgraph of Γ composed out of the admissible edges of Γ.

Algorithm 2 Bellman-Ford algorithm takes a graph and a source vertex and calculates distances to all other reachable vertices. Goldberg-Radzik heuristic, applied to this algorithm, suggests to scan a graph in a topological order.

GOLDBERG-RADZIK(constraint graph $(V, E, weight, op)$, source vertex $s \in V$)
1 INITIALIZE-SINGLE-SOURCE$((V, E, weight, op), s)$
2 $newlyLabelledVertices \leftarrow$ SCAN(s)
3 **while** $newlyLabelledVertices$ *is not empty*
4 **do** $nextNewlyLabelledVertices \leftarrow \varnothing$
5 **for** each vertex v *in* $newlyLabelledVertices$
6 **do** $t \leftarrow$ SCAN(v)
7 $nextNewlyLabelledVertices \leftarrow nextNewlyLabelledVertices \cup t$
8 $newlyLabelledVertices \leftarrow nextNewlyLabelledVertices$
9 **return** $False$

Algorithm 3 This procedure initializes distances, parent pointers etc.

INITIALIZE-SINGLE-SOURCE(constraints graph $(V, E, weight, op)$, source vertex s)
1 **for** each vertex $v \in V$
2 **do** $d(v) = \infty$
3 $\pi(v) =$ NIL
4 $status(v) = unreached$
5 $d(s) = 0$
6 $status(s) = labelled$

Algorithm 4 Given a labelled vertex $v \in V$, this procedure tries to improve the distance estimates by scanning all the edges outgoing from v.

NEWLYLABELLEDVERTICES SCAN(constraints graph $(V, E, weight, op)$, labelled vertex $v \in V$)
1 $newlyLabelledVertices \leftarrow \varnothing$
2 **for** each edge $(v, w) \in E$
3 **do if** $d(v) + weight(v, w) < d(w)$
4 **then return** $(UNSAT,$ NIL$)$
5 **return** $newlyLabelledVertices$

3.4 Implementation Details

Some implementation details (Numeric Conflict Analysis, Reducing Feasibility Checks).

3.5 Experimental Results

Tell a reader about some experimental results.

4 Conclusion

Conclusion on the topic ($\frac{1}{2}$ of a page).

References

[1] Scott Cotton, Eugene Asarin, Oded Maler, and Peter Niebert. Some progress in satisfiability checking for difference logic. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 263–276. Springer, 2004.

[2] Moez Mahfoudh. *Sur la Vérification de la Satisfaction pour la Logique des Différences*. PhD thesis, Université Joseph Fourier (Grenoble), 2003.

[3] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[4] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, third edition, 2009.

[6] Andrew V. Goldberg and Tomasz Radzik. A heuristic improvement of the bellman-ford algorithm. *Applied Mathematics Letters*, 6(3):3–6, 1993.