

# 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

### 1.1 Difference Logic

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:

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

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

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

## 1.2 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  $\phi$ )
1  model  $\leftarrow \emptyset$ 
2  (inferredAssignments, conflictingClauses)  $\leftarrow$  PROPAGATE( $\phi$ , model)
3  conflictHasArisen  $\leftarrow$  conflictingClauses  $\neq \emptyset$ 
4  if conflictHasArisen
5      then return (UNSAT, NIL)
6  model  $\leftarrow$  model  $\cup$  inferredAssignments
7  while True
8      do (nextVariable, value)  $\leftarrow$  DECIDE( $\phi$ , model)
9          allVariablesHaveAlreadyBeenAssigned  $\leftarrow$  nextVariable = NIL
10         if allVariablesHaveAlreadyBeenAssigned
11             then return (SAT, model)
12         model  $\leftarrow$  model  $\cup$  {nextVariable  $\leftarrow$  value}
13         repeat
14             (inferredAssignments, conflictingClauses)  $\leftarrow$  PROPAGATE( $\phi$ , model)
15             model  $\leftarrow$  model  $\cup$  inferredAssignments
16             conflictHasArisen  $\leftarrow$  conflictingClauses  $\neq \emptyset$ 
17             if conflictHasArisen
18                 then (newModelWithoutConflict,  $\phi_a$ )  $\leftarrow$  RESOLVE( $\phi$ , model, conflictingClauses)
19                     conflictHasNotBeenResolved  $\leftarrow$  newModelWithoutConflict = NIL
20                     if conflictHasNotBeenResolved
21                         then return (UNSAT, NIL)
22                     model  $\leftarrow$  newModelWithoutConflict
23                      $\phi \leftarrow \phi \wedge \phi_a$ 
24         until  $\neg$ conflictHasArisen

```

---

The given above 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  $\phi_a$  (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 described above Algorithm 1 a model is represented by a mapping from PL variables of the input PL formula to Booleans. It is a simplification. Real 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.

### 1.3 Solving SAT Problem of Difference Logic

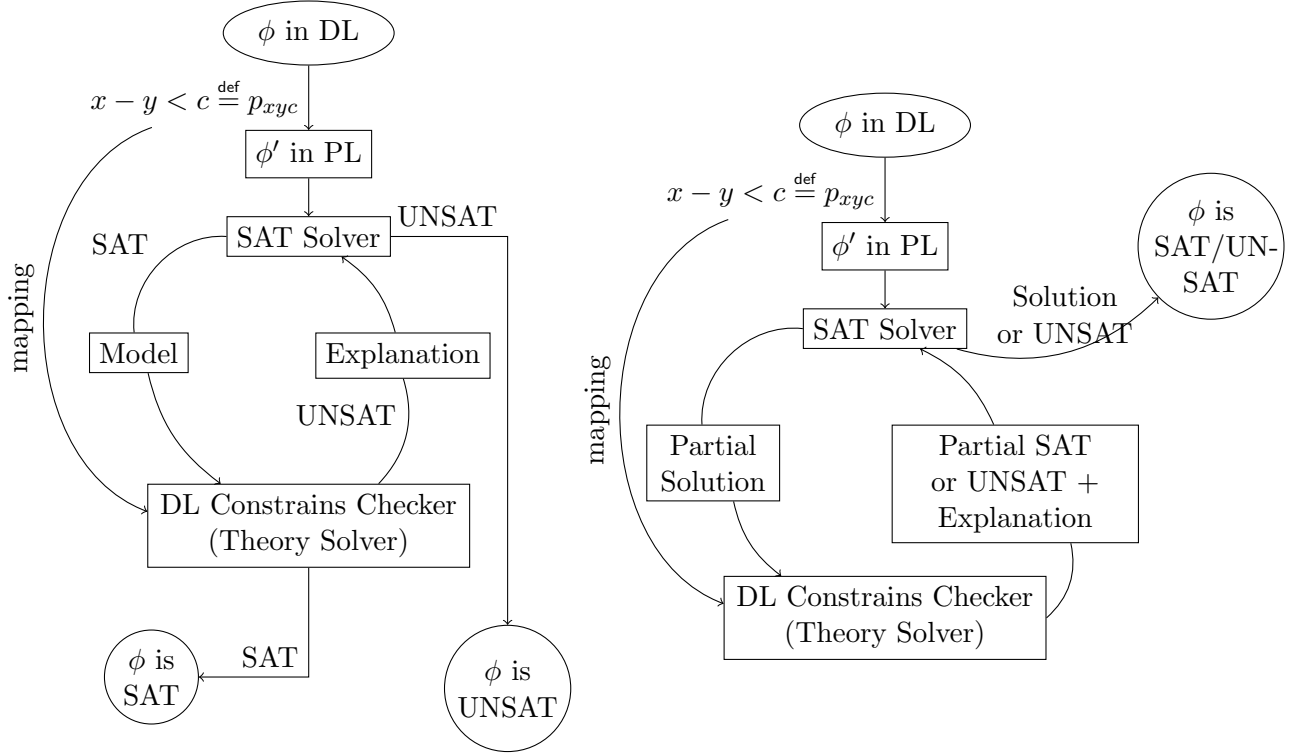


Figure 1: Illustration of the lazy (left) and incremental (right) approaches.

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:

$$(x - y < a) \wedge (y - z < b) \rightarrow (x - z < a + b) \quad (4)$$

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  $\phi'$ .

The PL formula represents the "skeleton", the Boolean abstraction over the original DL formula  $\phi$ . 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.

## 2 Preliminaries

### 2.1 Constraint Graph

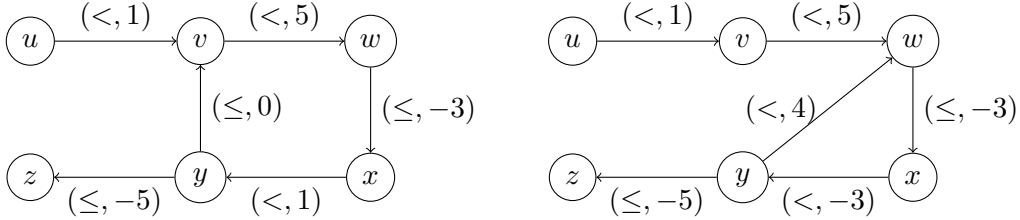


Figure 2: Examples of constraint graphs for Equation 2 (left) and Equation 3 (right).

Constraint Graph (Figure 2) is used by the DL constraints checker (Figure 1). It represents the DL constraints and allows calculate transitivity relations between them (e.g. Equation 4). According to [1] this graph is defined as follows:

**Definition 2.1 (Constraint Graph)** *The constraint graph of a set (or conjunction) of DL constraints  $\Gamma$  is a graph  $\Gamma = (V, E, \xi)$  with one vertex per numeric variable occurring in some DL constraint in  $\Gamma$ , edges  $E = \{(x, y) : (x - y \prec c) \in \Gamma \text{ for some } (\prec, c) \in B\}$  and a function  $\xi : E \mapsto B$  defined by  $(x, y) \mapsto \min\{(\prec, c) : (x - y \prec c) \in \Gamma\}$ .*

Where  $\prec \in \{<, \leq\}$ ,  $B = \{(\prec, c)\}$  is a set of bounds. On  $B$  an order is defined as follows:  $(\prec, c) <_B (\prec', c')$  when  $c < c'$  or when  $c = c'$  and  $\prec$  is  $<$  and  $\prec'$  is  $\leq$ . A couple of examples are given below:

$$\begin{aligned} (\leq, -3) &<_B (<, -1) \\ (<, -1) &<_B (\leq, -1) \end{aligned} \tag{5}$$

### 2.2 Bellman-Ford Algorithm

Bellman-Ford algorithm [5, p.651] calculates shortest paths from a single source vertex to all others. It can also detect negative cycles. In [1] a Goldberg-Radzik [5] variant of the Bellman-Ford algorithm is used. The algorithm is given below:

---

**Algorithm 2** 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*.

---

PROCEDURE SCAN( PL formula  $\phi$ )

```

1   $model \leftarrow \emptyset$ 
2   $(inferredAssignments, conflictingClauses) \leftarrow \text{PROPAGATE}(\phi, model)$ 
3   $conflictHasArisen \leftarrow conflictingClauses \neq \emptyset$ 
4  if  $conflictHasArisen$ 
5    then return (UNSAT, NIL)
6   $model \leftarrow model \cup inferredAssignments$ 
7  while True
8    do  $(nextVariable, value) \leftarrow \text{DECIDE}(\phi, model)$ 
9       $allVariablesHaveAlreadyBeenAssigned \leftarrow nextVariable = \text{NIL}$ 
10     if  $allVariablesHaveAlreadyBeenAssigned$ 
11       then return (SAT,  $model$ )
12      $model \leftarrow model \cup \{nextVariable \leftarrow value\}$ 
13     repeat
14        $(inferredAssignments, conflictingClauses) \leftarrow \text{PROPAGATE}(\phi, model)$ 
15        $model \leftarrow model \cup inferredAssignments$ 
16        $conflictHasArisen \leftarrow conflictingClauses \neq \emptyset$ 
17       if  $conflictHasArisen$ 
18         then  $(newModelWithoutConflict, \phi_a) \leftarrow \text{RESOLVE}(\phi, model, conflictingClauses)$ 
19            $conflictHasNotBeenResolved \leftarrow newModelWithoutConflict = \text{NIL}$ 
20           if  $conflictHasNotBeenResolved$ 
21             then return (UNSAT, NIL)
22            $model \leftarrow newModelWithoutConflict$ 
23            $\phi \leftarrow \phi \wedge \phi_a$ 
24     until  $\neg conflictHasArisen$ 

```

---

## 3 Topic

### 3.1 Negative Cycles Detection

Show how the satisfiability of a DL formula is related to the negative cycle detection.

### 3.2 Implementation Details

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

### 3.3 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.