Difference Logic

Satisfiability Checking Seminar

Alex Ryndin
Supervisor: Gereon Kremer

WS 2016/2017

Abstract

This report describes difference logic, which is a special case of linear arithmetic logic, and a graph-based algorithm for deciding satisfiability (SAT) of a conjunction of difference logic flormulas. There is of course a simplex-based algorithm, which can also do that. However, it has exponential complexity whereas the graph-based algorithm, described here, has polynomial complexity.

Difference logic constraints appear in many timing related problems such as e.g. scheduling problems, detecting race conditions in digital circuits etc. Therefore it is very important to have an efficient algorithm to decide whether they are satisfiable or not.

1 Introduction

Difference logic is a special case of linear arithmetic logic and it is defined in [1] and [2, p.5] as follows:

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 over a domain $\mathbb{D}$. The domain $\mathbb{D}$ is either the Integers $\mathbb{Z}$ or the Reals $\mathbb{R}$. The difference logic over $\mathcal{B}$ and $\mathcal{X}$ is called $DL(\mathcal{X}, \mathcal{B})$ and given by the following grammar:*

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

where $b \in \mathcal{B}$, $x, y \in \mathcal{X}$, $c \in \mathbb{D}$ is a constant and $\prec \in \{<, \leq\}$ is a comparison operator.

The remaining Boolean connectives $\vee, \rightarrow, \leftrightarrow, \dots$ can be defined in the usual ways in terms of conjunction $\wedge$ and negation $\neg$.

The main difference between linear arithmetic and difference logic is that in the latter constraints contain only two variables. A difference logic constraint is essentially a comparison of a difference of those two variables and a constant. This form of constraints naturally emerges when describing a delay between e.g. starting times of two processes or events, which are described by the corresponding numerical variables. It might be the reason why many timing related problems can be described by difference logic. Examples of difference logic formulas are given below:

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

$$\phi_3 = (u - v < 1) \wedge (v - w < 5) \wedge (w - x \leq -3) \wedge (x - y < -3) \wedge (y - z \leq -5) \wedge (y - w < 4) \quad (2)$$

Difference logic can also describe constraints $x \prec c$ and $-x \prec c$. One rewrites them as $x - 0 \prec c$ and $0 - x \prec c$ respectively and then introduces a pseudo-variable z instead of zero. An example:

$$\begin{aligned} (v < 3) \wedge (-w \leq -4) \\ (v - 0 < 3) \wedge (0 - w \leq -4) \\ (v - z < 3) \wedge (z - w \leq -4) \end{aligned} \tag{3}$$

Algorithm 2 is compatible with the pseudo-variable z in the sense that introducing this pseudo-variable z does not break the reasoning, on which the algorithm is based. Since this algorithm constitutes the core of the satisfiability checking of difference logic, the above-described constraints can be incorporated into a difference logic formula.

Constraints $\pm x \succ c$ can be rewritten as $\mp x \prec -c$ and thus can also be described by difference logic. An example:

$$\begin{aligned} (v > -3) \wedge (-w \geq 4) \\ (-v < 3) \wedge (w \leq -4) \\ (0 - v < 3) \wedge (w - 0 \leq -4) \\ (z - v < 3) \wedge (w - z \leq -4) \end{aligned} \tag{4}$$

Constraints $x = c$ and $x \neq c$ can be rewritten as $\neg((x < c) \vee (x > c))$ and $((x < c) \vee (x > c))$ respectively and thus can also be described by difference logic. An example:

$$\begin{aligned} (v = -3) \wedge (w \neq 4) \\ (\neg((v < -3) \vee (v > -3))) \wedge ((w < 4) \vee (w > 4)) \\ (\neg((v < -3) \vee (-v < 3))) \wedge ((w < 4) \vee (-w < -4)) \\ (\neg((v - 0 < -3) \vee (0 - v < 3))) \wedge ((w - 0 < 4) \vee (0 - w < -4)) \\ (\neg((v - z < -3) \vee (z - v < 3))) \wedge ((w - z < 4) \vee (z - w < -4)) \end{aligned} \tag{5}$$

One may employ even more sophisticated problem-specific transformations like in the following example:

$$\begin{aligned} (u - 2w + v > 3) \wedge (2w - v - u \geq 2) \\ (u - w - w + v > 3) \wedge (w - v - u + w \geq 2) \\ ((u - w) - (w - v) > 3) \wedge ((w - v) - (u - w) \geq 2) \\ \text{introduce new variables } x = w - v \text{ and } y = u - w \\ (y - x > 3) \wedge (x - y \geq 2) \\ (x - y < -3) \wedge (y - x \leq -2) \end{aligned} \tag{6}$$

A job scheduling problem is a motivation behind difference logic. The problem is formulated as follows. Given N jobs with processing times $\tau_1, \ldots, \tau_N$ and M identical machines, each of which can process only one job at any given time moment, the following question must be answered. Is it possible to schedule the jobs on the machines such that the overall processing time will not exceed T? An example setting:

$$\begin{aligned} N = 4, M = 2, T = 6.5 \\ \tau_1 = 1.6, \tau_2 = 1.1, \tau_3 = 4.6, \tau_4 = 5.1 \end{aligned} \tag{7}$$

Let $p_{mj} = True$ if job j is scheduled on machine m and $p_{mj} = False$ otherwise. Let also t_j be the start time of the job j. Then the job scheduling problem for the given example setting can be encoded by the following formula:

$$
\begin{aligned}
&\phi = \bigwedge_{j=1}^{4} (p_{1j} \vee p_{2j}) \quad \wedge \\
&\text{each task is executed on at least one machine} \\
&\bigwedge_{j=1}^{4} ((p_{1j} \rightarrow \neg p_{2j}) \wedge (p_{2j} \rightarrow \neg p_{1j})) \quad \wedge \\
&\text{each task can be scheduled on one machine only} \\
&\bigwedge_{j=1}^{4} (t_j \geq 0) \ \wedge \ \bigwedge_{j=1}^{4} (t_j \leq T - \tau_j) \quad \wedge \\
&\text{general time constraints} \\
&\bigwedge_{m=1}^{2} \bigwedge_{i=1}^{3} \bigwedge_{j=i+1}^{4} ((p_{mi} \wedge p_{mj}) \rightarrow ((t_i - t_j \leq -\tau_i) \vee (t_j - t_i \leq -\tau_j))) \\
&\text{jobs } i \text{ and } j \text{ must not overlap if scheduled on the same machine } m
\end{aligned}
\tag{8}
$$

Where τ_j and T are real constants (the $c \in \mathbb{D}$ in Definition 1.1), p_{mj} are boolean variables from the $\mathcal{B}$ set in Definition 1.1 and t_j are real variables from the $\mathcal{X}$ set in Definition 1.1 ($1 \leq m \leq 2, 1 \leq j \leq 4$).

All the linear arithmetic constraints in Equation 8 can be expressed in difference logic. Thus this problem can be solved by a difference logic SAT solver.

The rest of the report is organized as follows. Chapter 2 gives theoretical background on SAT checking. Chapter 3 describes a graph-based algorithm to solving SAT problem of DL. Chapter 4 draws a conclusion.

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.

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 calculates all assignments that follow from a given PL formula and a given model. It also returns a list of conflicting clauses (if there are any). These clauses will be in conflict if one extends the model by the returned assignments.

DECIDE This function applies some heuristic and selects a variable to be set next and a value.

RESOLVE This function tries to resolve a conflict. It returns a new model without the conflict and an asserting clause ϕ_a. If the conflict cannot be resolved, the function returns NIL instead of the model.

THEORY This function tries to resolve a conflict. It returns a new model without the conflict and an asserting clause ϕ_a. If the conflict cannot be resolved, the function returns NIL instead of the model.

Algorithm 1 A basic satisfiability checking algorithm for deciding satisfiability problem of propositional logic.

```
CHECK()
 1  if ! PROPAGATE()
 2    then return UNSAT
 3  while True
 4  do if ! THEORY()
 5        then if ! RESOLVE()
 6                then return UNSAT
 7        else  if ! DECIDE()
 8                then return SAT
 9     while ! PROPAGATE()
10     do if ! RESOLVE()
11           then return UNSAT
```

2.2 Approaches to Decide Satisfiability of Difference Logic

[1] mentions the following main approaches for solving 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 is transitivity:

$$(x - y < a) \wedge (y - z < b) \rightarrow (x - z < a + b) \tag{9}$$

  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 $x - y \prec c$ with a Boolean variable $p_{xyc} \in \mathbb{B}$ thus yielding a PL formula ϕ'. ϕ' 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) to solve SAT problem. In this approach the SAT solver always computes a complete solution which is then passed to the theory solver.

- Incremental approach (Figure 1 right). In [1] this approach is used. It is very similar to the lazy one. However, instead of computing a complete solution, 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 Theory Solver ormFor Difference Logic

3.1 Notation

Throughout this Chapter the following notation is used:

- ϕ is a *conjunction* of DL constraints. It is being checked for SAT.

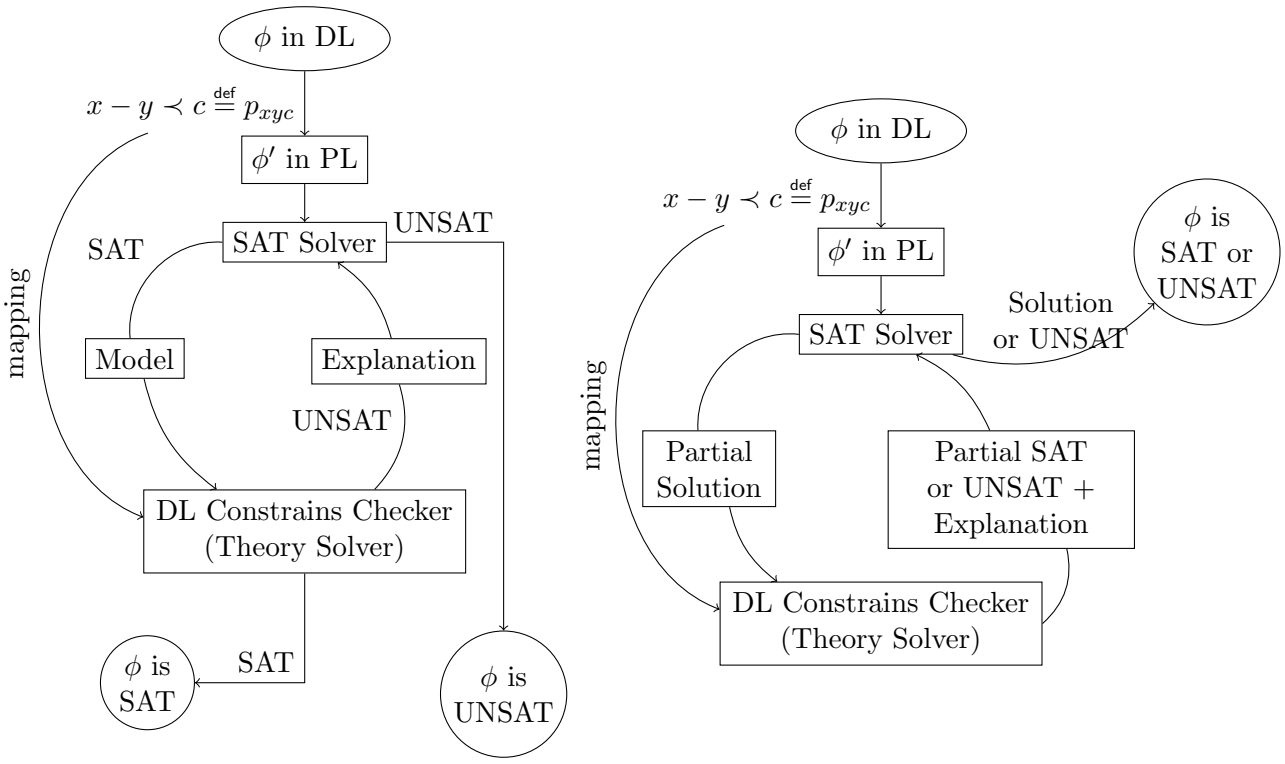

Figure 1: Illustration of the lazy (left) and incremental (right) approaches.

- $x - y \prec c$ is a general form of a DL constraint in ϕ where $\prec \in \{<, \leq\}$.
- $\mathbb{D}$ is a domain over which the variables and constants in ϕ are defined (e.g. $\mathbb{R}$).

3.2 Constraint Graph

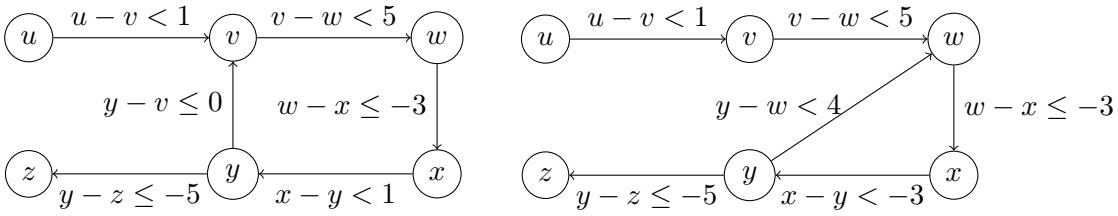

Figure 2: Examples of constraint graphs for Equation 1 (left) and Equation 2 (right).

Constraint graph (Figure 2) is a weighted directed graph which represents ϕ and which is used by a DL constraints checker (Figure 1) to test if ϕ is SAT. In [1] it is defined as follows:

Definition 3.1 (Constraint Graph) *The constraint graph 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 $x - y \prec c$.*

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

3.3 Negative Cycles in Constraint Graph

There is a direct correspondence between a negative cycle in a constraint graph and SAT of ϕ represented by this graph.

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:

$$\begin{array}{r} u - v < \quad 1 \\ v - w < \quad 5 \\ \underline{w - x \leq -3} \\ u - x < \quad 3 \end{array} \tag{10}$$

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 an inferred transitivity constraint (e.g. Equation 9). 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 conflicting inequalities:

$$\begin{array}{r} x - y < -3 \\ y - w < \quad 4 \\ \underline{w - x \leq -3} \\ 0 < -2 \end{array} \tag{11}$$

3.4 Bellman-Ford Algorithm for Constraint Graph

[1] uses a Goldberg-Radzik [6] variant of the Bellman-Ford algorithm [5, p.651] to detect negative cycles and thus check ϕ for SAT (Algorithm 2). [6] states that the algorithm has the same worst-case complexity $O(|V| \cdot |E|)$ as Bellman-Ford algorithm but is superior in practice. Terminology and notation used in the algorithm are given below.

Definition 3.2 (Source Vertex) *The source vertex $s \in V$ is a vertex from which the shortest paths to other vertices are computed.*

Definition 3.3 (Distance Estimating Function [1]) *The distance estimating function $d(v) : V \mapsto \mathbb{D}$ is a function which returns an upper bound on the length of the shortest path from the source vertex to the given vertex $v \in V$.*

Algorithm 2 An algorithm for checking if ϕ which corresponds to the input constraint graph $\Gamma = (V, E, weight, op)$ is SAT. It returns SAT or UNSAT status and a set of DL constraints corresponding to a conflict (in case of UNSAT). It is based on Bellman-Ford algorithm [5, p.561]. Goldberg-Radzik heuristic [6], which is used here, suggests to scan a graph in a topological order. This algorithm uses depth first search [5, p.603] (DFS) and breadth first search [5, p.594] (BFS) for auxiliary tasks.

GOLDBERG-RADZIK(constraint graph $\Gamma = (V, E, weight, op)$, source vertex $s \in V$)

for each vertex $x \in V$
do $d(x) = \infty$
    $status(x) = unreached$
$d(s) = 0$
$status(s) = labeled$
$A \leftarrow \varnothing$
$B \leftarrow \{s\}$
repeat
        if Γ_d has a cycle C (DFS on Γ_d can be used to check it)
          then $l \leftarrow$ length of C in Γ
                if $l < 0$
                  then return ($UNSAT$, DL constraints corresponding to L)
                if $\exists\ (x, y) \in L$ such that $op(x, y) = <$
                  then return ($UNSAT$, DL constraints corresponding to L)
        for each vertex $x \in B$
        do if x has no outgoing admissible edges
              then $B \leftarrow B \setminus \{x\}$
                    $status(x) = scanned$
        $A \leftarrow$ set of unexplored vertices reachable from B in Γ_d (BFS on Γ_d can be used here)
        $A \leftarrow$ sort A topologically using Γ_d as an input graph (DFS on Γ_d can be used here)
        $B \leftarrow \varnothing$
        for each vertex $x \in A$
        do $status(x) = labeled$
            for each edge $(x, y) \in E$
            do if $d(x) + weight(x, y) < d(y)$
                  then $d(y) \leftarrow d(x) + weight(x, y)$
                        if $status(y) = unreached$
                          then $B \leftarrow B \cup \{y\}$
                        $status(y) \leftarrow labeled$
                        $status(x) \leftarrow scanned$
  until A *is empty*
return $(SAT, \varnothing)$

Definition 3.4 (Reduced Cost Function [6]) *The reduced cost function* $r(x, y) : V \mapsto \mathbb{D}$ *is defined as follows:* $r(x, y) = weight(x, y) + d(x) - d(y)$.

Definition 3.5 (Vertex Status) *The vertex status* $status(x) = \{unreached, labeled, scanned\}$ *is a function on vertices which shows a current state of a vertex* $x \in V$. $status(x) = unreached$ *means* x *has not been explored yet.* $status(x) = labeled$ *means* x *has been explored i.e. the distance estimate for it has been updated at least once and potentially it can be used to improve distance estimates to other vertices.* $status(x) = scanned$ *means* x *has been completely explored and will not be considered further for improving distance estimates.*

Definition 3.6 (Admissible Edge) *Edge* $(x, y) \in E$ *is called admissible if* $r_d(x, y) \leq 0$.

Definition 3.7 (Admissible Graph) *Admissible graph* Γ_d *is a subgraph of* Γ *composed of the admissible edges of* Γ.

In Algorithm 2 distance estimates are iteratively updated. In [5, p.648] the act of updating a distance estimate using an edge is called "edge relaxation" (Equation **??**). This iterative process can also be seen as a series of different distance estimating functions $(d_0, d_1, d_2, d_3, \dots)$. Each d_i in this series describes which distance estimates have the vertices at some iteration of the Algorithm 2.

The idea of the Algorithm 2 is to use the dynamically changing graph Γ_d (it changes whenever d changes) to detect negative or zero cycles in the original graph Γ. The following theorem from [1] expresses this idea more formally.

Theorem 3.1 *Given a constraint graph* Γ *and a series of distance estimating functions* $(d_0, d_1, d_2, d_3, \dots)$, Γ *has a negative or zero cycle if and only if* Γ_d *has a cycle under some distance estimate* d_k.

4 Conclusion

Difference logic is widely used to describe timing-related problems. Since difference logic is a special case of linear arithmetic logic, it is possible to employ simplex to decide satisfiability of difference logic constraints. However, simplex has exponential complexity whereas the graph-based algorithm, described in this report, has polynomial complexity.

The algorithm works on a constraint graph which represents a conjunction of difference logic constraints. The idea of the algorithm is based on the fact that negative or zero length cycles in the constraint graph correspond to a sequence of difference logic inequalities which, if summed together, will produce a conflicting inequality $0 < 0$ or $0 \prec c$ where c is a negative constant and $\prec \in \{<, \leq\}$. The inequalities, which correspond to the cycle, form the explanation for the SAT solver.

The ingenious thing about the algorithm is that it does not enumerate all the cycles in a constraint graph but rather detects any cycle in a dynamic admissible graph and therefore achieves Bellman-Ford algorithm's complexity $O(|E| \cdot |V|)$. Enumerating all cycles is prohibitively expensive in terms of computational complexity because there can be exponentially many of them in a graph.

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.