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. Chapter 2 gives theoretical background on SAT checking. 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 LA logic. [1] and [2, p.5] define DL 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:*

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

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 $\neg$.

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 SAT and returns SAT status of the formula (SAT or UNSAT). It also returns a model i.e. an assignment, which evaluates the formula to *True*, in case if the formula is SAT.

```
CHECK( PL formula $\phi$)
 1  $model \leftarrow \varnothing$
 2  $(inferredAssignments, conflictingClauses) \leftarrow$ PROPAGATE$(\phi, model)$
 3  $conflictHasArisen \leftarrow conflictingClauses \neq \varnothing$
 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              $conflictHasArisen \leftarrow conflictingClauses \neq \varnothing$
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 $\neg conflictHasArisen$
```

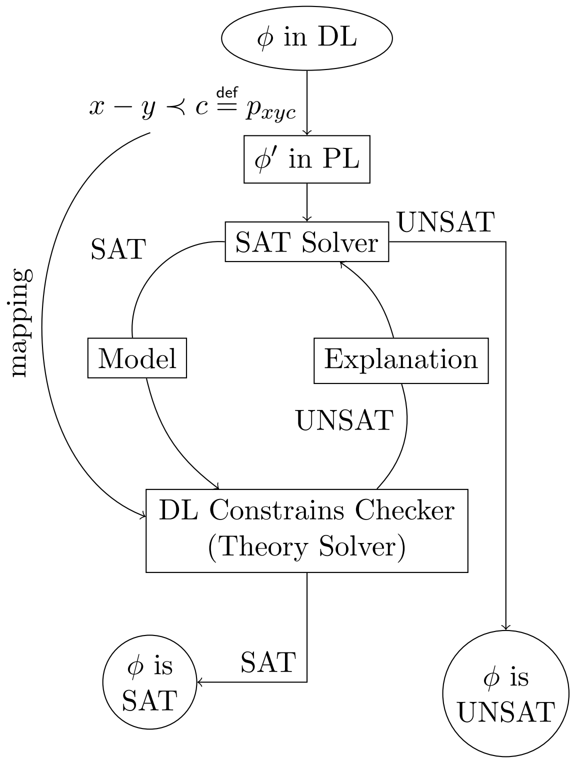
The Algorithm 1 is quite generic. It is more of a template. One needs to plug into this template his own implementations of the following functions:

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.

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.

Additionally, a care should be taken when resolving a conflict and computing a new model. The solver should make sure it visits each state associated with a model once only.

2.2 Solving SAT Problem of Difference Logic



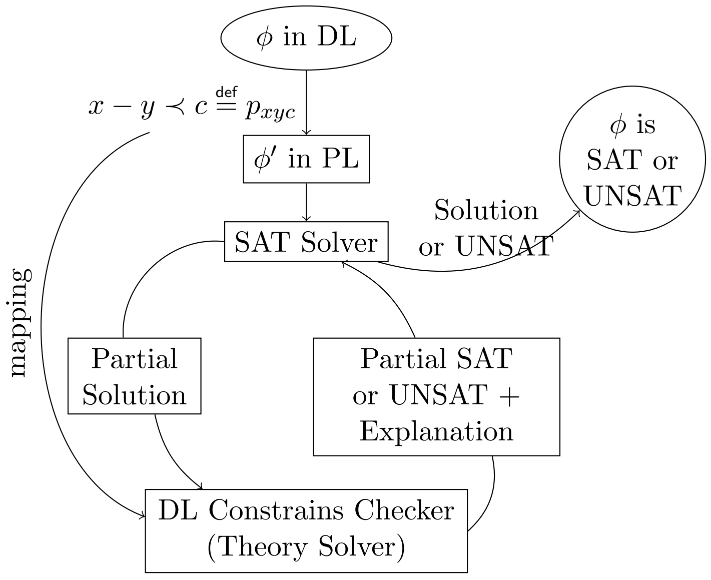

ϕ in DL
$x - y \prec c \overset{\text{def}}{=} p_{xyc}$
ϕ' in PL
ϕ is SAT or UNSAT
Solution or UNSAT
SAT Solver
mapping
Partial Solution
Partial SAT or UNSAT + Explanation
DL Constrains Checker (Theory Solver)

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

[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:

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

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 $x - y \prec c$ 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 Topic

3.1 Notation

Throughout this Chapter the following notation is used:

- ϕ is a *conjunction* of DL constraints. It is being checked for SAT.
- $x - y \prec c$ 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

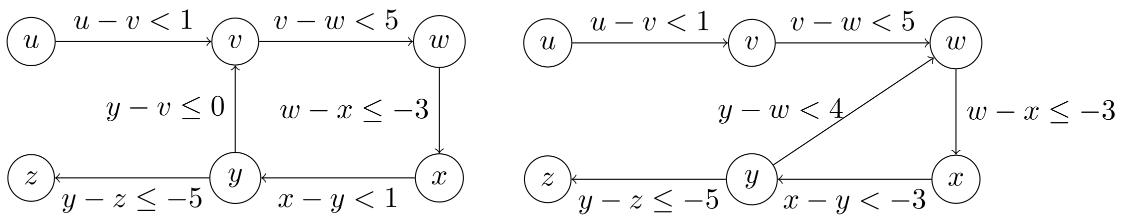


Figure 2: Examples of constraint graphs for Equation 2 (left) and Equation 3 (right).

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:

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

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 an inferred 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 conflicting inequalities:

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

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 $v \in V$.*

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

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 $\Gamma_d$ has a cycle $C$ (DFS on $\Gamma_d$ can be used to check it)
         then $l \leftarrow$ length of $C$ in $\Gamma$
              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 $\Gamma_d$ (BFS on $\Gamma_d$ can be used here)
      $A \leftarrow$ sort $A$ topologically using $\Gamma_d$ as an input graph (DFS on $\Gamma_d$ 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 3.5 (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 7). 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.

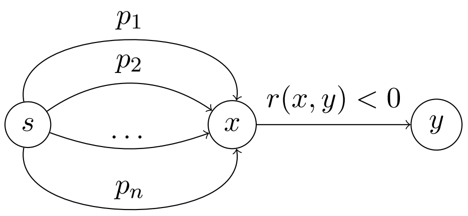


Figure 3: Multiple paths from s to y may be used to improve distance estimate to x.

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.*

Proof 3.1 *$\Rightarrow$ Case 1. Γ has a negative cycle. To prove: Γ_d has a cycle at some distance estimate d_k.*

Suppose Γ_d has no cycles at any distance estimate (the initial assumption). Consider an arbitrary edge $(x, y) \in \Gamma_d$ at some distance estimate d_k for which $r(x, y) < 0$ i.e. this edge can be relaxed. If we apply relaxation to this edge then we change the distance estimate to some d_{k+1} by updating distance estimate for the vertex y to a smaller value:

$$\begin{aligned} r(x, y) < 0 \Rightarrow d_k(x) + weight(x, y) < d_k(y) \\ d_{k+1}(y) = d_k(x) + weight(x, y) \end{aligned} \tag{7}$$

Immediately after this operation we will have that $r(x, y) = 0$ and thus the edge will still be in Γ_d but relaxation cannot be applicable to it. It can be the case, however, that this edge will be relaxed again in some further iterations. Since there are no cycles in Γ_d, multiple updates are only possible when there are some better paths $p_1, p_2, \dots p_n$ from the *source vertex s to x (Figure 3). These paths can be used to update distance estimate for the vertex x and make possible relaxation of the edge (x, y). Number of these paths cannot be infinite because the graph has finite number of vertices and edges. Therefore the edge (x, y) will be relaxed finitely many times. Then $r(x, y)$ will become zero at some iteration and the edge will stay in Γ_d for all further iterations.*

The above reasoning has been applied to an arbitrary edge (x, y) for which we have $r(x, y) < 0$ at some distance estimate d_k during the execution of the algorithm. This reasoning can be repeated for all other edges and it can be concluded therefore that after some finite number of iterations no distance updates will be possible, because for any edge $(x, y) \in \Gamma$ we will have $r(x, y) \geq 0$. Thus, the distance update process will converge. However, in [5, p.653] it has been proven that, in case when Γ has a negative cycle, the distance update process does not converge. Therefore we have a contradiction and the initial assumption is wrong. Thus, Γ_d will have a cycle at some distance estimate.

Case 2. *Γ has no negative cycles but it has a zero weight cycle $Z = (x_0, x_1, \ldots, x_{n-1}, x_n)$ with $x_n = x_0$. To prove: Γ_d has a cycle at some distance estimate d_k.*

In this case the distance estimate process will converge to some function d_k which means that there is no edge that can be relaxed, including the edges of Z:

$$d(x_i) + weight(x_i, x_{i+1}) \geq d(x_{i+1}),\ 0 \leq i < n \tag{8}$$

Let us sum the inequality from Equation 8 along the cycle Z:

$$\begin{aligned}
\sum_{i=0}^{n-1} d(x_i) + \sum_{i=0}^{n-1} weight(x_i, x_{i+1}) \geq \sum_{i=0}^{n-1} d(x_{i+1}) \\
\sum_{i=0}^{n-1} weight(x_i, x_{i+1}) \geq \sum_{i=0}^{n-1} d(x_{i+1}) - \sum_{i=0}^{n-1} d(x_i) \\
Z \text{ has zero weight} \\
0 \geq \sum_{i=0}^{n-1} d(x_{i+1}) - \sum_{i=0}^{n-1} d(x_i) \\
\text{apply index renaming to the first sum} \\
0 \geq \sum_{i=1}^{n} d(x_i) - \sum_{i=0}^{n-1} d(x_i) \\
\text{since } x_0 = x_n \text{ the two sums are actually equal} \\
0 \geq 0
\end{aligned} \tag{9}$$

The final inequality in Equation 9 is valid and is a consequence of the convergence.

Now assume that there is at least one edge (x_i, x_{i+1}) along the cycle Z for which we have $d(x_i) + weight(x_i, x_{i+1}) > d(x_{i+1})$ (the initial assumption). Then, if we do the summation in Equation 9 again, we will get a strict inequality $0 > 0$ which is invalid. Thus, we have a contradiction with the convergence condition and therefore the initial assumption is invalid. Thus $d(x_i) + weight(x_i, x_{i+1}) = d(x_{i+1}),\ 0 \leq i < n$ and $r(x_i, x_{i+1}) = 0$ and therefore all edges of Z are admissible and they are in the graph Γ_d.

$\Leftarrow$ *Γ_d under some distance estimate d_k has a cycle $C = (x_0, x_1, \ldots, x_{n-1}, x_n)$ with $x_0 = x_n$. To prove: there is a negative or zero weight cycle in Γ.*

C is in the graph Γ_d. Therefore its edges are admissible:

$$\begin{aligned}
r(x_i, x_{i+1}) &\le 0,\ 0 \le i < n \\
d(x_i) + weight(x_i, x_{i+1}) - d(x_{i+1}) &\le 0,\ 0 \le i < n \\
d(x_i) + weight(x_i, x_{i+1}) &\le d(x_{i+1}),\ 0 \le i < n \\
&\text{apply summation along the cycle} \\
\sum_{i=0}^{n-1} d(x_i) + \sum_{i=0}^{n-1} weight(x_i, x_{i+1}) &\le \sum_{i=0}^{n-1} d(x_{i+1}) \\
\sum_{i=0}^{n-1} weight(x_i, x_{i+1}) &\le \sum_{i=0}^{n-1} d(x_{i+1}) - \sum_{i=0}^{n-1} d(x_i) \\
&\text{the two sums on the right are equal (see Equation 9)} \\
\sum_{i=0}^{n-1} weight(x_i, x_{i+1}) &\le 0
\end{aligned} \tag{10}$$

The left hand side expression in the final inequality in Equation 10 is the length of the cycle C which is less than or equal to zero.

4 Conclusion

DL logic is widely used to describe timing-related problems. Although, it is possible to use simplex to solve SAT problem of DL constraints, nevertheless there is a more efficient graph-based algorithm.

The algorithm works on a constraint graph which represents a conjunction of DL 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 DL inequalities which, if summed together, will produce a conflicting inequality $0 < 0$ or $0 \prec c$ where c is a negative constant and $\prec \in \{<, \le\}$. The inequalities, which correspond to the cycle, are used to 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.