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 (DL) 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:*

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

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 $\neg$.

The main difference between linear arithmetic logic 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:

$$\phi_1 = (u-v < 1) \wedge (v-w < 5) \wedge (w-x \leq -3) \wedge (x-y < 1) \wedge (y-z \leq -5) \wedge (y-v \leq 0) \quad (1)$$

$$\phi_2 = (u-v < 1) \wedge (v-w < 5) \wedge (w-x \leq -3) \wedge (x-y < -3) \wedge (y-z \leq -5) \wedge (y-w < 4) \quad (2)$$

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 $zero \notin \mathcal{X}$ instead of zero. An example:

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

Algorithm 2 is compatible with the pseudo-variable *zero* in the sense that introducing *zero* 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:

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

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

$$\begin{aligned}(v = -3) \land (w \neq 4)\\ (\neg((v < -3) \lor (v > -3))) \land ((w < 4) \lor (w > 4))\\ (\neg((v < -3) \lor (-v < 3))) \land ((w < 4) \lor (-w < -4))\\ (\neg((v - 0 < -3) \lor (0 - v < 3))) \land ((w - 0 < 4) \lor (0 - w < -4))\\ (\neg((v - zero < -3) \lor (zero - v < 3))) \land ((w - zero < 4) \lor (zero - w < -4))\end{aligned} \tag{5}$$

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

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

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 $p_{mj} = True$ iff job j is scheduled on machine m. 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, $\mathbb{D} = \mathbb{R}$), p_{mj} are boolean variables from the set $\mathcal{B}$ in Definition 1.1 and t_j are real variables from the set $\mathcal{X}$ 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

Algorithm 1 A basic satisfiability checking algorithm for propositional logic.

```
CHECK()
if ! PROPAGATE()
  then return $UNSAT$
while $True$
do if ! THEORY()
      then if ! RESOLVE()
              then return $UNSAT$
      else  if ! DECIDE()
              then return $SAT$
   while ! PROPAGATE()
   do if ! RESOLVE()
         then return $UNSAT$
```

Most SAT solvers employ a variation of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [3, 4] for deciding satisfiability of propositional logic (PL). Algorithm 1 exemplifies it. It is quite generic. It is more of a template. One needs to plug into this template his own implementations of the following functions:

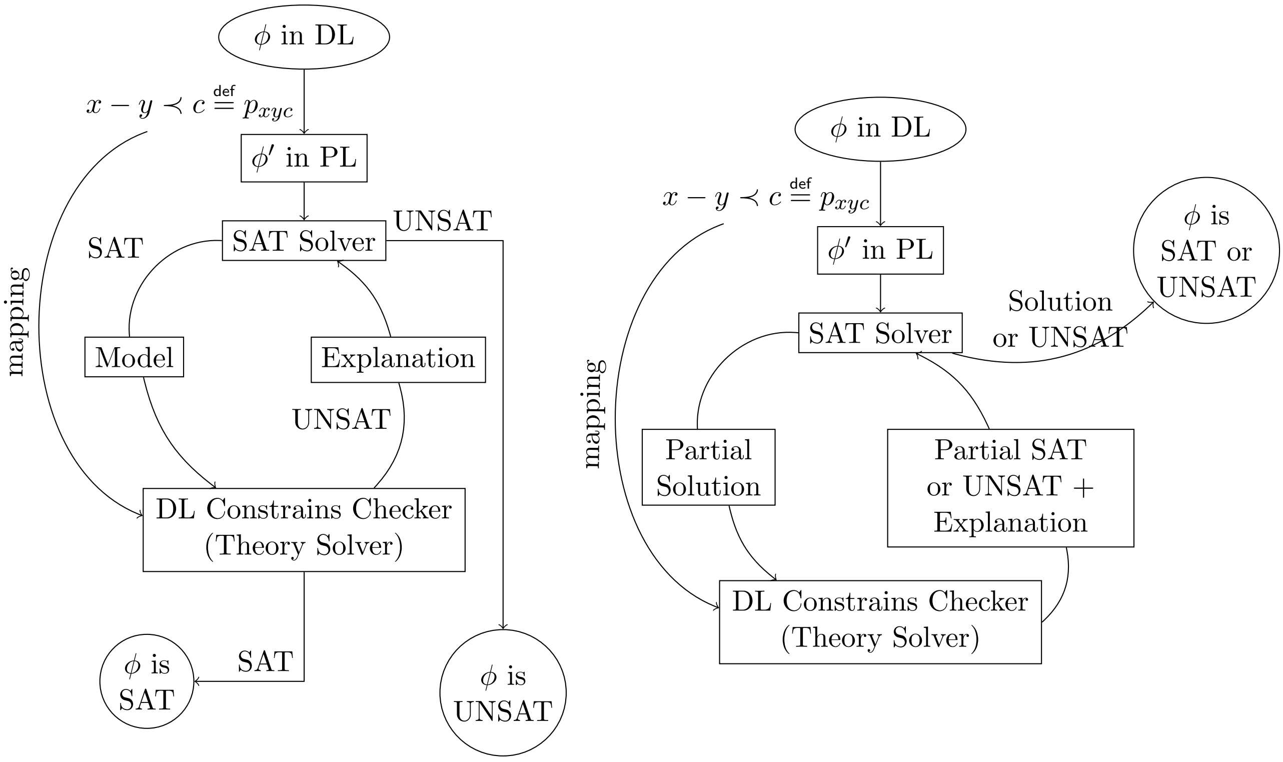
Propagate Propagates a current variables assignment and returns *True* iff the propagation has not run into a conflict.

Decide Tries to extend a current variables assignment by assigning a next variable according to some heuristic and returns *False* iff all variables have been already set (nothing to assign).

Resolve Tries to resolve a conflict and returns *True* iff it has been successfully resolved.

Theory Checks a current variables assignment and returns *True* iff it is consistent with the theory.

2.2 Approaches to Decide Satisfiability of Difference Logic



ϕ in DL
$x - y \prec c \overset{\text{def}}{=} p_{xyc}$
ϕ' in PL
SAT Solver
UNSAT
SAT
mapping
Model
Explanation
UNSAT
DL Constrains Checker (Theory Solver)
ϕ is SAT
SAT
ϕ is UNSAT
ϕ in DL
$x - y \prec c \overset{\text{def}}{=} p_{xyc}$
ϕ' in PL
ϕ is SAT or UNSAT
SAT Solver
Solution or UNSAT
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 satisfiability problem of difference logic:

- Preprocessing approach. This approach suggests transforming a difference logic formula into an equivalent propositional logic formula by encoding all intrinsic dependencies between difference logic constraints in propositional logic. 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 if the resulting equivalent propositional formula is satisfiable. If it is satisfiable then the solution for the original formula can be reconstructed by the reverse transformation.

- Lazy approach (Figure 1 left). This approach suggests substituting each difference logic constraint $x - y \prec c$ with a Boolean variable $p_{xyc} \in \mathbb{B}$ thus yielding a propositional logic formula ϕ'. ϕ' represents the "skeleton", the Boolean abstraction over the original formula ϕ. Then a SAT solver is used in tandem with a difference logic constraints checker (the theory solver) to solve the satisfiability problem. In this approach the SAT solver always computes a complete solution (a variables assignment for all variables) 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 difference logic constraints checker each time it updates its variables assignment and passes a partial solution to the constrains checker. The constraints checker should be able to maintain some internal state of the currently received difference logic constraints and update it incrementally (i.e. add new constraints, delete existing ones). Hence the name of the approach.

3 Theory Solver For Difference Logic

3.1 Notation

Throughout this chapter the following notation is used:

- ϕ is a *conjunction* of difference logic constraints. It is being checked for satisfiability.
- $x - y \prec c$ is a general form of a 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

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

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

Examples of constraint graphs are shown on Figure 2.

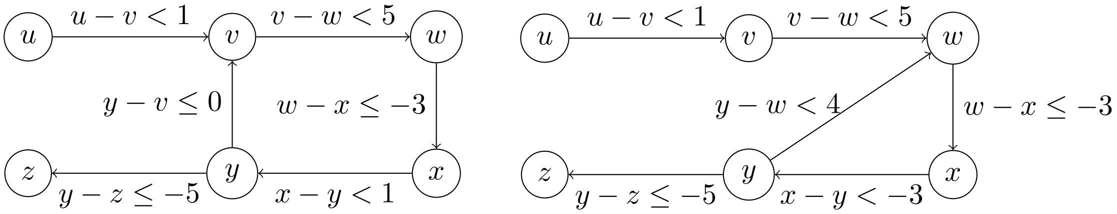


Figure 2: Examples of constraint graphs for Equation 1 (left) and Equation 2 (right).

3.3 Negative Cycles in Constraint Graph

A path in a constraint graph can be used to infer a new inequality by applying transitivity rule (Equation 9). The path $u \to v \to w \to x$ in the left graph on Figure 2 exemplifies it:

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

If at least one strict inequality is present then the resulting inequality will also be strict.

In [1] it is shown that a negative cycle in a constraint graph corresponds to a conflict of the form $0 \prec c$ where $c < 0$. Therefore if a constrain graph has a negative cycle then the corresponding formula ϕ is unsatisfiable. In the same paper it is also shown that if the graph has a zero weight cycle then it corresponds to an inequality $0 \prec 0$ which represents a conflict in case when $\prec\ =\ <$ i.e. when the inequality is strict. In this case ϕ is also unsatisfiable. This case also shows why one needs to map edges of a constraint graph not only into constants but also into types of inequalities (strict or non-strict, function *op* in the 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:

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

The bottomline: ϕ is unsatisfiable iff the corresponding constraint graph has a negative cycle or a zero weight cycle with at least one strict inequality in it.

3.4 Bellman-Ford Algorithm for Constraint Graph

[1] uses a Goldberg-Radzik [5] variant of the Bellman-Ford algorithm [6, p.651] to detect negative and zero weight cycles and thus check if ϕ is satisfiable (Algorithm 2). [5] 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 in the definitions 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.*

Algorithm 2 An algorithm for checking if ϕ which corresponds to the input constraint graph $\Gamma = (V, E, weight, op)$ is satisfiable. 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 [6, p.561]. Goldberg-Radzik heuristic [5], which is used here, suggests to scan a graph in a topological order. This algorithm uses depth first search [6, p.603] (DFS) and breadth first search [6, p.594] (BFS) for auxiliary tasks. Γ_d is the dynamically changing admissible graph from Definition 3.6.

```
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 3.3 (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 [5]) *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 (Admissible Edge) *Edge $(x, y) \in E$ is called admissible if $r(x, y) \leq 0$.*

Definition 3.6 (Admissible Graph) *Admissible graph Γ_d is a subgraph of Γ composed of the admissible edges of Γ.*

Definition 3.7 (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.*

Algorithm 2 relies on the following theorem:

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.*

Theorem 3.1 is proven in [1]. The intuition behind it is the following. Γ_d consists of edges which may potentially improve some current distance estimate function d_k. If Γ_d has a cycle then it means that this cycle may potentially be used to update the distance estimate function d_k infinitely many times thus preventing the process from convergence. As stated in [6, p.645] inability to converge means that the single-source shortest paths problem is not well defined which means that there has to be a negative weight cycle which can always be used to improve distance estimates.

The whole satisfiability checking process of a difference logic formula (Figure 1, right) can be approximately described as follows:

- A boolean skeleton ϕ' is built.
- The SAT solver starts solving ϕ'.
- When it modifies a current variables assignment, it maps it to a conjunction of difference logic inequalities and tells the constraints checker what new inequalities need to be added and what previously passed inequalities need to be deleted.
- The constraints checker modifies its state accordingly (i.e. the constraint graph and other data structures).
- The constraints checker tries to find a negative or zero cycle which would correspond to a conflict (Algorithm 2).
    - If it runs into such a conflict then it informs the SAT checker and returns it the inequalities which constitute the cycle as the explanation of the encountered conflict.

  - Otherwise it informs the SAT checker that the current variables assignment is consistent with the theory (the THEORY call in Algorithm 1) and the SAT checker continues to search for a solution.

The described process ends in two cases. Either a complete solution is found, which is also consistent with the theory. It means that the input difference logic formula is satisfiable. Or the SAT solver has run into a conflict which is impossible to resolve. It means that the initial formula contains a contradiction and thus is unsatisfiable.

As an example consider the difference logic formula in Equation 2 as an input:

- $\phi' = p_{u,v,1} \wedge p_{v,w,5} \wedge p_{w,x,-3} \wedge p_{x,y,-3} \wedge p_{y,z,-5} \wedge p_{y,w,4}$
- The SAT solver starts with a partial assignment $\{p_{u,v,1} = True\}$. It tells the constraints checker: "add $(u - v < 1)$ and check consistency".
- The constraints checker adds the edge (u, v) to the (initially empty) constraints graph, which corresponds to the received inequality. There are no cycles in the graph and therefore the constraints checker informs the SAT checker that the current variables assignment is consistent with the theory.
- The SAT solver continues searching for a solution. It extends a partial assignment to the following assignment: $\{p_{u,v,1} = True, p_{v,w,5} = True\}$. It tells the constraints checker: "add $(v - w < 5)$ and check consistency".
- ...
- At some moment the SAT solver arrives in a state when it has $\{p_{u,v,1} = True, p_{v,w,5} = True, p_{w,x,-3} = True, p_{x,y,-3} = True, p_{y,z,-5} = True\}$ as the current variables assignment and it extends it with an assignment $p_{y,w,4} = True$. It tells the constraints checker: "add $(y - w < 4)$ and check consistency".
- The constraints checker adds the new edge (y, w) and finds a negative cycle $w \to x \to y \to w$ corresponding to a conflict. The set of inequalities $\{(w - x \leq -3), (x - y < -3), (y - w < 4)\}$ correspond to the detected conflict. The constraints checker returns the explanation to the SAT solver: $\neg(p_{w,x,-3} \wedge p_{x,y,-3} \wedge p_{y,w,4})$ which can be rewritten as $(\neg p_{w,x,-3} \vee \neg p_{x,y,-3} \vee \neg p_{y,w,4})$.
- The SAT solver adds the received explanation to ϕ':

  $$\phi'' = \phi' \wedge (\neg p_{w,x,-3} \vee \neg p_{x,y,-3} \vee \neg p_{y,w,4}) \tag{12}$$

  Then it backtracks to the decision level 0 and starts solving ϕ''. It calls PROPAGATE from Algorithm 1 and runs into a conflict. The conflict cannot be resolved because it is detected on the decision level 0. Therefore the SAT solver returns UNSAT as the answer i.e. the input difference logic formula from Equation 2 is unsatisfiable.

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 to find a negative or zero length cycle in the constraint graph. If such a cycle exists then it corresponds to a conflict, the corresponding formula is unsatisfiable and 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] Andrew V. Goldberg and Tomasz Radzik. A heuristic improvement of the bellman-ford algorithm. *Applied Mathematics Letters*, 6(3):3–6, 1993.

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