BACHELOR OF SCIENCE THESIS

IMPLEMENTING AN INCREMENTAL SOLVER FOR DIFFERENCE LOGIC

Christopher David Lösbrock

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Joost-Pieter Katooen
Additional Advisors:
Gereon Kremer
Matthias Volk

Aachen, 23.08.2018

Abstract

Satisfiability modulo theories (SMT) describes the problem of finding a satisfiable assignment for a formula from a theory over a first-order logic. The theory of *quantifier-free difference logic* is a subset of linear arithmetic containing only constraints of the form $x - y \leq c$. Difference logic is especially interesting for timed systems and scheduling problems because it allows an easy declaration of orders over its variables.

This thesis aims to expand the existing SMT-RAT framework by two distinguished theory solvers for the theory of quantifier-free difference logic using a graph-based approach. The implemented solvers are based on the Bellman-Ford algorithm and the Floyd-Warshall algorithm for finding shortest paths in a graph, which are already used in a number of state-of-the-art SMT-solvers.

Both implemented solvers were compared against SMT-RAT's existing simplex solver resulting in a better runtime in most cases.

Contents

Chapter 1

Introduction

The satisfiability modulo theories problem (SMT) describes the problem of deciding whether a formula of first-order-logic over some theory is satisfiable by finding a satisfiable assignment for its variables or determining its unsatisfiability. SMT-solving is a heavily researched topic as shown by the annual held SMT-COMP[1], where different SMT-solvers compete against one another on various theories.

This thesis aims to expand SMT-RAT [CKJ+15] by two distinguished theory solvers for solving *quantifier-free difference logic* formulas. Difference logic describes the first-order logic comprised solely of constraints of the form $x - y \leq c$ and Boolean variables and operators. Compared to other first-order logics like linear arithmetic, of which difference logic is a fragment, it is relatively simple. An important property of difference logic is that orders over variables can be naturally expressed by differences, making it useful for scheduling problems or timed systems. Two concrete examples would be the verification of timed automata [NMA+02] or dynamic fault trees (DFTs) [VJK18]. DFTs are used to determine, whether a system will fail, depending on the prior failure of a number of its basic elements. In addition to a number of standard benchmarks, the newly implemented solvers will also be tested on a set of DFT problems. Another advantage of difference logic is the fact that a set of constraints can be represented as a directed graph. The problem of deciding the satisfiability for conjunctions of difference constraints then amounts to searching for negative cycles in a graph. This is a well-known problem in graph theory, solvable by Bellman-Ford, Floyd-Warshall or Dijkstra's algorithm, to name a few. Variations of the first two algorithms are implemented in the state-of-the-art solver Z3 [dMB08], that performed well in prior competitions. Inspired by that, we will implement both algorithms as separate solvers. For the Bellman-Ford algorithm, we will implement an incremental algorithm proposed in [WIGG05]. The Floyd-Warshall solver will be based on a combination of [RHK15] and [HRK17].

The thesis is structured as follows: In the next chapter, we describe the general problem and the definition of difference logic, as well as, the basics of the graph-based approach. In Chapters 3 and 4, we first describe the shortest-paths algorithms and their application to SMT-solving, followed by their implemented incremental versions. After that, we describe the implemented preprocessing method and two of SMT-RAT's already existing preprocessing techniques, that are applicable to difference logic. In

[1]Website of SMT-COMP: `http://smtcomp.sourceforge.net`

Chapter 6 we present our experimental results, comparing the newly implemented solvers against SMT-RAT's prior existing simplex solver and Z3. Furthermore we will analyse the composition of the running times of the newly implemented solvers. Finally, we draw a conclusion and discuss possible further improvements in Chapter 7.

Chapter 2

Preliminaries

2.1 The Satisfiability Problem

The *satisfiability problem for propositional logic* or *SAT problem* for short, describes the problem of finding an assignment for the variables of a Boolean formula such that the formula evaluates to true. Even though the task seems simple, this problem is *NP-hard* according to Cook's theorem[1] [Coo71].

Definition 2.1.1. *A Boolean formula φ, comprised of Boolean variables $x_1, \dots, x_n$ with Boolean operators $(\wedge, \vee, \neg)$ is called* satisfiable *if there exists an assignment α for its variables such that the formula evaluates to true. Such an assignment is called a* model *of the formula. If no satisfying assignment exists, the formula φ is called* unsatisfiable.

An obvious way to decide whether a formula φ is satisfiable is to calculate all possible assignments for all variables in φ in a truth table. This brute-force method is not practical for more complex formulas as the number of possible assignments that have to be checked is 2^n in the worst case, with n being the number of variables in a given formula. SAT-solvers nowadays use an algorithm called the *Davis-Putnam-Loveland-Logemann algorithm* or short *DPLL algorithm* [DLL62] or rather its improvement *conflict driven clause learning (CDCL)*. While the algorithm still has to check all 2^n possible assignments in a worst case, it performs very well in practice. The algorithm requires its input formula to be in *conjunctive normal form* or CNF, which means that the formula may only consist of a conjunction of disjunctions.

Definition 2.1.2. *A formula φ is in conjunctive normal form or CNF if*

$$\varphi \textit{ has the form } \varphi = \bigwedge^{i} (\bigvee^{j} l_{ij}) \tag{2.1}$$

where l_{ij} is the j-th literal in the i-th clause (a disjunction of literals).

Every Boolean formula can be transformed into an equivalent formula in CNF by applying basic Boolean transformation laws. This may result in an exponential increase of the size of the formula and may take exponential time. An alternative

[1]also known as the Cook-Levin theorem

procedure is called *Tseitin transformation* [Tse68], in which a formula φ is transformed into an equisatisfiable formula ψ in CNF. This means that the new formula ψ is satisfiable if, and only if, the original formula φ is satisfiable. This is done by introducing new variables and can be done in polynomial time. Also, for any satisfiable assignment for ψ, the respective assignment for φ can be easily derived.
The CDCL algorithm can be divided into three main operations which are combined in a certain way until, either a satisfying assignment for the formula was computed or the formula is found to be unsatisfiable:

Decision: The algorithm guesses the truth value for an unassigned literal l_i according to some heuristic. Every decision marks a *decision level dl*.

Boolean constraint propagation (BCP): The algorithm checks if the latest decision entails any other assignments of variables in the formula. This is done by checking the clauses where all literals evaluate to false, except for one literal l_j, which is unassigned. These clauses are called *unit clauses* and force the assignment of the literal l_j. This is because, for a formula in CNF to be satisfied by an assignment, all its clauses must evaluate to true. Therefore, as clauses are disjunctions of literals, at least one literal of each clause must evaluate to true. This is continued for all literals assigned in this phase. Should the algorithm at one point detect a conflict, it continues with conflict analysis and otherwise makes the next decision.

Conflict analysis: If the algorithm detected a conflicting assignment during BCP, it determines the cause of the conflict by backtracking to the last relevant decision. It then adds a new *conflict clause*, which fixes the variable assignment that caused the conflict to the opposite assignment. The newly added conflict clause reduces the search space by creating more restrictions on the assignments of the literals. Should the algorithm at one point determine that the conflicting assignment was made on decision level zero, then the formula is unsatisfiable, because the algorithm found a conflict without any literals freely assigned by itself.

2.2 Satisfiability Modulo Theories

The *Satisfiability Modulo Theories Problem (SMT)* expands the satisfiability problem from propositional formulas to formulas over some theory $\mathcal{T}$. To that end, we define some basic notations used throughout the thesis. Notice though, that we are only interested in the *quantifier-free* fragment of first-order logic, which means that there are no quantifier-symbols and all variables are implicitly existential-quantified. Therefore any definitions regarding quantifiers will be omitted.

In the context of SMT, a theory $\mathcal{T}$ consists of a domain D and interpretations for the used function and predicate symbols, mapping all function symbols to functions $f : D^n \to D$ and all predicate symbols to functions $P : D^n \to \{0,1\}$. As constant symbols can be viewed as 0-ary functions, they are also mapped to the used domain.

- *term*: A variable or a constant is a term. Furthermore if $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ are terms, with f being a n-ary function symbol.
- *constraint*: If $t_1, \dots, t_n$ are terms, then $P(t_1, \dots, t_n)$ is a constraint, with $P \in \Sigma$ being a n-ary predicate symbol.

- *formula*: A formula consists of constraints with Boolean connectives ($\wedge, \vee, \neg$).

The syntax of the terms, constraints and formulas is defined by some abstract grammar. A more detailed definition of theories in the context of SMT can be found in [BHvMW09]. Given a first-order logic formula φ, the goal is to find an assignment $\alpha(x) \in D$ for every variable x in φ, such that $\alpha \models_{\mathcal{T}} \varphi$ (φ is $\mathcal{T}$*-satisfiable*).

An SMT-solver normally consists of two separate solvers, a SAT-solver and a theory solver or $\mathcal{T}$-solver. There are two main approaches to the interaction between SAT-solver and theory solver. In the first approach, called *eager* SMT-solving, the formula is transformed into an equisatisfiable formula in propositional logic and then solved for satisfiability by a SAT-solver.

The second approach, called *lazy* SMT-solving switches the order of the operations. This time the SAT-solver works on an abstraction of the original formula called the *Boolean skeleton*. The Boolean skeleton is constructed by encoding every distinct constraint in the formula into a unique propositional variable. The SAT-solver tries to compute a satisfying assignment for the Boolean skeleton and, if successful, passes it to the $\mathcal{T}$-solver. The theory solver receives the set C of constraints which were represented by the Boolean variables assigned to true and returns whether C poses a conflict under the theory $\mathcal{T}$. Should the theory solver detect a conflict, it returns an *infeasible subset* of the given set of constraints. The infeasible subset is the set of constraints that caused the conflict and serves as an explanation to the SAT-solver for making changes to the current conflicting assignment. Lazy SMT-solvers can be divided into two subclasses, depending on when the theory solver is called. In a *full-lazy* SMT-solver, the SAT-solver tries to compute a satisfying full assignment for the Boolean skeleton before calling the theory solver. In a *less-lazy* SMT-solver (Figure 2.1) on the other hand, the theory solver is called upon after every BCP operation. Either way, the theory solver has to fulfill a number of conditions to work alongside the SAT-solver effectively:

incrementality: The theory solver may be called upon by the SAT-solver to check for conflicts under the theory for several assignments for one problem. These assignments may differ only slightly from one another and therefore the $\mathcal{T}$-solver should be able to use its prior computations, instead of starting from scratch every time it is invoked. This is especially important for less-lazy SMT-solving, where the theory solver is called upon for partial assignments.

backtracking: After the theory solver returns that it found a conflict, the SAT-solver will backtrack to change the conflicting assignments. The theory solver should be able to backtrack alongside the SAT-solver, only reverting changes when necessary.

theory propagation: The theory solver should pass all information it can extract from the given set of constraints back to the SAT-solver. This includes, but is not limited to, the infeasible subset.

2.3 SMT-RAT

The framework expanded in this thesis is the *SMT Real Arithmetic Toolbox (SMT-RAT)* [CKJ+15]. SMT-RAT does not consist of a single solver for SMT, but rather

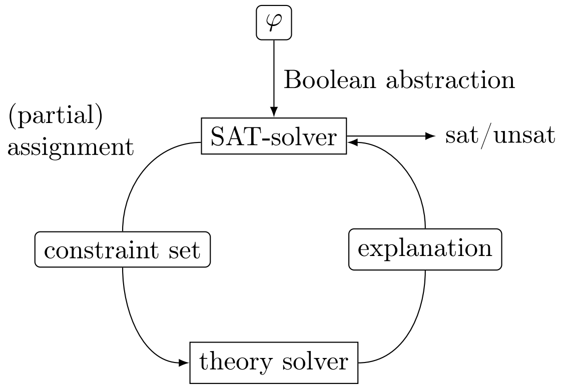


Figure 2.1: Diagram of a less-lazy SMT-solver

a collection of different preprocessors, $\mathcal{T}$-solvers and a SAT-solver. It uses the C++ library CArL [KCJ+], which consists of data structures and functions allowing to work directly on, for example, constraints and formulas. This allows for a high level of abstraction when implementing new solvers for SMT-RAT. Its architecture can be divided into the three following parts:

module: A module m_i consists of a single preprocessing or solving method encapsulated in a common interface. This interface allows for adding formulas to the considered set of formulas F and removing formulas from it. The main function of the interface calls for the module to check F for satisfiability and to return the appropriate answer. Furthermore, the module returns an infeasible subset of the set of formulas if it was unsatisfiable. The module may invoke another module m_j as its *backend*. The module m_i passes a formula F to the backend m_j and gets either a model, if F is satisfiable, or otherwise an infeasible subset back. Lastly, a module may define additional *lemmas*, valid formulas, which allow it to pass additional information to other modules.

strategy: A strategy is a user-defined composition of modules. It allows a user to define which modules are invoked and under which conditions. Figure 2.2 shows a simple strategy consisting of a module m_1, which runs module m_2 as its backend, if condition c_1 is met and m_3, if condition c_2 is met.

manager: The manager coordinates the overall solving of an input formula F. Should a module m_i, currently solving a formula $F_i \subseteq F$, call another module m_j as its backend on a subformula F', the manager passes F' to the called module. After m_j reached a result, the manager passes this result back to m_i, which uses this information for solving the formula F.

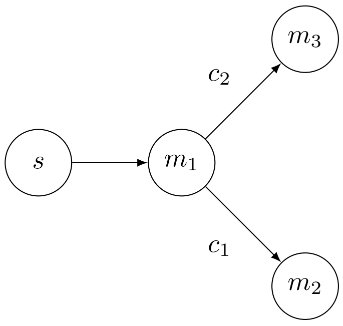

$$\begin{tikzcd} & & m_3 \\ s \arrow[r] & m_1 \arrow[ur, "c_2"] \arrow[dr, "c_1"'] & \\ & & m_2 \end{tikzcd}$$

Figure 2.2: Graphic representation of a simple strategy

2.4 Difference Logic

This thesis aims to implement theory solvers for the quantifier-free fragment of difference logic over reals $\mathcal{DL}(\mathbb{R})$ and integers $\mathcal{DL}(\mathbb{Z})$. The syntax of difference logic can be defined as follows:

$$\begin{aligned} \mathit{term} &: \mathit{variable} - \mathit{variable} \\ \mathit{constraint} &: \mathit{term} \leq \mathit{constant} \\ formula &: \neg formula \mid formula \wedge formula \mid \mathit{constraint} \end{aligned}$$

The SMT-LIB standard [BFT16] allows for a larger set of predicates, thus we have to consider a more general form of constraints $x - y \bowtie c$ with $\bowtie \in \{<\,, >\,, \leq\,, \geq\,, =\,, \neq\}$. Furthermore, the SMT-LIB standard allows for terms containing only a single variable. Nonetheless, we can focus on constraints of the form $x - y \leq c$, because all other constraints allowed by the SMT-LIB standard can be transformed into that form. The transformations are as follows:

$$x - y = c \iff x - y \leq c \wedge x - y \geq c \tag{2.2}$$
$$x - y \neq c \iff ((x - y < c \vee x - y > c) \wedge \neg(x - y < c \wedge x - y > c)) \tag{2.3}$$
$$x - y > c \iff y - x < -c \tag{2.4}$$
$$x - y < c \iff x - y \leq c - \varepsilon,\ \varepsilon = 1, \text{ if } D = \mathbb{Z};\ \varepsilon \text{ sufficiently small, if } D = \mathbb{R} \tag{2.5}$$
$$x \leq c \implies x - z \leq c,\ z \overset{!}{=} 0 \tag{2.6}$$

In the case of $\mathcal{DL}(\mathbb{Z})$, strict inequalities are transformed directly by SMT-RAT's SAT-solver by applying $\varepsilon = 1$. For $\mathcal{DL}(\mathbb{R})$ the theory solvers keep an adjust value $a(x) \in \mathbb{Z}$ for every variable x. This allows the solvers to use ε symbolically during its computations. This is described in Section 3.1.

Constraints of the form $x \leq c$ are transformed into $x - z \leq c$ by introducing a fresh variable z that is not part of the considered set of constraints. Any satisfying assignment is then shifted by $-z$ to ensure $z = 0$.

Theorem 2.4.1. *Shifting the variable assignments of a model α of a set of difference constraints C by a constant factor also yields a satisfying assignment.*

Proof. Assume a set of difference constraints C of the form $x_i - x_j \leq b_{ij}$. Furthermore let α be a model of C, with variable assignments $\alpha(x_i)$ for all variables in C.

Then

$$\alpha(x_i) - \alpha(x_j) \leq b_{ij} \tag{2.7}$$

has to hold for all constraints.

$$\tag{2.8}$$

Now assume a variable assignment β with $\beta(x_i) = \alpha(x_i) + d$, where d is a value from the domain. Then

$$\begin{aligned} &(\alpha(x_i) + d) - (\alpha(x_j) + d) \leq b_{ij} \\ \equiv &\alpha(x_i) + d - \alpha(x_j) - d \qquad \leq b_{ij} \\ \equiv &\underbrace{\alpha(x_i) - \alpha(x_j) \qquad\qquad \leq b_{ij}}_{\text{true, because of Eq. 2.7}} \end{aligned}$$

Thus, β is a satisfying assignment for C. □

2.5 The Graph-based Approach

The following basic graph notations and proofs are taken from [CLRS09].

We will use a graph-based approach to check systems of difference constraints in both theory solvers. To that end, we construct a graph that will represent the system of difference constraints by creating a vertex for every variable occurring in the given set of constraints and connect them with directed edges from the vertex representing the subtrahend to the minuend vertex for every constraint. For an edge (u,v), we will call u the *source* and v the *target* of the edge. Furthermore, an additional vertex is added, that does not represent a variable from the set of constraints and has outgoing edges to all other vertices with weight zero. This vertex serves two purposes: First, it poses as a distinct start node for the Bellman-Ford algorithm and second, it is used to determine a satisfying assignment, if there exists one.

Definition 2.5.1. *Given a set C of difference constraints of the form $x_i - x_j \leq b_k$ over variables $x_0, \ldots, x_n$, the corresponding constraint graph is a weighted, directed graph $G = (V,E)$ with a weight function $w : E \to D$, where*

$$V = \{s, v_0, v_1, \ldots, v_n\}$$

and

$$E = \{(v_i,v_j) : x_j - x_i \leq b_k \text{ is a constraint in } C\} \cup \{(s,v_0),(s,v_1),\ldots,(s,v_n)\}$$

with $w(v_i,v_j) = b_k$ and $w(s,v_i) = 0$.

$$\begin{aligned} x_2 - x_1 &\le 2 \\ x_3 - x_1 &\le -1 \\ x_4 - x_3 &\le 0 \\ x_1 - x_4 &\le 4 \\ x_4 &\le 3 \end{aligned}$$

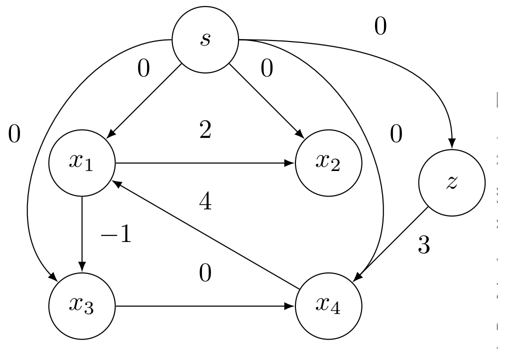


Figure 2.3: Example of a constraint graph

Figure 2.3 shows an example of a simple set of difference constraints represented in a constraint graph. Notice that the vertices are directly labeled with the names of the variables, as every vertex represents exactly one variable from the system of difference constraints. Therefore we may refer to vertices with the names of the variables and the other way around if the meaning is clear.

To check for the satisfiability of a system of difference constraints with a graph-based approach, we need a condition that the constraint graph has to fulfill to determine the satisfiability of said system. This condition is the absence of negative cycles in the graph.

Definition 2.5.2. *A negative cycle $v_0 \overset{c_0}{\to} v_1 \overset{c_1}{\to} v_2 \overset{c_2}{\to} \cdots \overset{c_{n-1}}{\to} v_n \overset{c_n}{\to} v_0$ is a path from a vertex to itself with a negative path weight*

$$\sum_{i=0}^{n} w(c_i) < 0 \tag{2.9}$$

Theorem 2.5.1. *A set of difference constraints C is unsatisfiable if the corresponding constraint graph has a negative cycle.*

Proof. Assume C is a set of difference constraints and $G = (V,E)$ its constraint graph. Furthermore let $c = \langle v_0, \dots, v_k \rangle$ be a negative cycle in G. The edges (v_{i-1},v_i), $i = 1,2,\dots,k$ with weight $w(v_{i-1},v_i) = b_{i,i-1}$ represent the set of constraints $x_i - x_{i-1} \le b_{i,i-1}$. The cycle has negative weight, thus

$$\sum_{i=1}^{k} w(v_{i-1},v_i) < 0 \text{ and therefore } \sum_{i=1}^{k} b_{i,i-1} < 0 \tag{2.10}$$

hold. Since every constraint represented by an edge in the graph must be satisfied, the sum of the constraints in the negative cycle has to be satisfiable. This leads to

$$\sum_{i=1}^{k} x_i - \sum_{i=1}^{k} x_{i-1} \le \sum_{i=1}^{k} b_{i,i-1} \tag{2.11}$$

The two sums on the left-hand side are equal because $v_0 = v_k$ and therefore $x_0 = x_k$

(every vertex represents exactly one variable).

$$0 \leq \underbrace{\sum_{i=1}^{k} b_{i,i-1}}_{<0,\text{ because of Eq. 2.10}} \tag{2.12}$$

This is false no matter the chosen values for x_i, $i = 1,2,\ldots,k$ and therefore unsatisfiable overall. □

To check for negative cycles we use shortest-paths algorithms. A shortest-paths algorithm is given a weighted, directed graph G and determines the shortest paths in accordance with the considered subproblem. The Bellman-Ford algorithm finds shortest paths for a prior determined source vertex to all other vertices and the Floyd-Warshall algorithm finds shortest paths for all pairs of vertices in the graph. There are other variations of the problem, but they are of no interest to the implemented solvers.

Definition 2.5.3. *The shortest path weight from u to v is defined as*

$$\delta(u,v) = \begin{cases} \min(w(p) : u \overset{p}{\rightsquigarrow} v), \textit{ if there exists a path from } u \textit{ to } v \\ \infty, \textit{ otherwise} \end{cases}$$

with $w(p) = \sum_{e \in p} w(e)$.

We exploit the fact that shortest-paths algorithms abort if they encounter a negative cycle. This is because a shortest path from a vertex v to another vertex u that contains a negative cycle could always be improved by traversing the negative cycle one more time, thus leading to a path with a weight tending towards negative infinity.

Lemma 2.5.2. *Given a weighted, directed graph G, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then p_{ij} is a shortest path from v_i to v_j.*

Proof. The proof is by contradiction. If any subpath p_{ij} of a shortest path p could be improved then the path p could be improved. Therefore p would not have been a shortest path, which poses a contradiction to the initial statement. □

If the theory solver is passed a satisfiable set of difference constraints, SMT-RAT requires it to return a satisfying assignment. Theorem 2.5.3 shows that this directly relates to the computed shortest path weights from the start node s to the vertices of the constraint graph representing the variables.

Theorem 2.5.3. *If a constraint graph contains no negative cycle, then the represented set of difference constraints is satisfiable and the computed shortest path weights form a satisfiable assignment.*

Proof. Assume C is a set of difference constraints and $G = (V,E)$ its constraint graph. G contains no negative cycle and thus has shortest paths $s \overset{p}{\rightsquigarrow} v$ for all $v \in V$ with weight $\delta(s,v)$. For any edge $(v_i,v_j) \in E$ with weight $w(v_i,v_j) = b_{ij}$ representing a constraint $x_j - x_i \leq b_{ij}$ the triangle inequality $\delta(s,v_j) \leq \delta(s,v_i) + w(v_i,v_j)$ holds. Transformation leads to $\delta(s,v_j) - \delta(s,v_i) \leq w(v_i,v_j)$. If we replace each part with the component from the constraint it represents, we get $x_j - x_i \leq b_{ij}$. Since this is true for all edges in G all constraints have a satisfying assignment. □

Chapter 3

Bellman-Ford-based Solver

The aforementioned Bellman-Ford algorithm is a single-source shortest-paths algorithm that uses the principle of *dynamic programming.* This means, that the algorithm recursively solves subproblems of the examined problem, and uses the gained information to find a solution to the original problem. The Bellman-Ford algorithm does this by finding shortest paths of increasing lengths in each iteration. This is possible because, as Lemma 2.5.2 states, all subpaths of a shortest path are again shortest paths.

Definition 3.0.1. *For every vertex v in the constraint graph $G = (V,E)$, we define a predecessor $\pi[v]$. The predecessor is either another vertex or* NULL.

To construct the infeasible subset, the solver has to be able to construct a negative cycle, if one was found. Therefore, we store the direct predecessor on the path for every vertex. This allows us to construct the path by going from one vertex to its predecessor until we reach the source node s because, as Lemma 2.5.2 states, all subpaths of a shortest path are in turn shortest paths. Thus, it is enough to store a single predecessor for every vertex, as a changed predecessor indicates a better estimate for the shortest path p, and therefore all other paths with p as a subpath will be adapted correctly.

Definition 3.0.2. *For every vertex v, we keep a shortest-path estimate $d(v)$, which serves as an upper bound on the weight of the shortest path $s \overset{p}{\leadsto} v$, where s is the start node of the Bellman-Ford algorithm.*

The shortest-path estimates are initialized with $d(v) = \infty$ for all vertices in the constraint graph. The Bellman-Ford algorithm iteratively lowers that bound until $d(v) = \delta(s,v), \forall v \in V$ holds or a negative cycle was found. To that end the process of relaxing edges is used. The RELAX-function (Algorithm 1) gets an edge $(u,v) \in E$ as its input and checks if the current distance approximation $d(v)$ of the target is larger than the sum of the edge weight and the distance approximation of the source $w(u,v) + d(u)$. If that is the case, a path through (u,v) would be an improvement of the shortest path from s to v. The distance approximation of v is then changed to $d(v) = w(u,v) + d(u)$ and u is set as the predecessor of v. This means the new shortest path for v is the same as for u with addition of the relaxed edge (u,v). If an edge cannot be relaxed any more, it is called *stable.*

Should a received formula be satisfiable, a model can be constructed by using the distance approximation of every vertex as the assignment for its represented variable (see Theorem 2.5.3).

```
function RELAX($u$,$v$)
    if $d(u) + w(u,v) < d(v)$ then
        $d(v) \leftarrow d(u) + w(u,v)$
        $\pi(v) \leftarrow u$
    end if
end function
```

Algorithm 1: Edge relaxation

The Bellman-Ford algorithm repeats the relaxation $((|V|-1) \cdot |E|)$-times to ensure that the shortest paths are found. The algorithm starts with paths of length one in the first iteration and checks longer paths in each iteration. This is sufficient because the shortest paths have to be *simple*. This means that no vertex can be visited more than once for each path and thus there cannot be a cycle in a shortest path. It follows that such a path contains maximal $|V| - 1$ edges, else there would be a cycle. If the sum of the weights in that cycle were positive, RELAX would not have chosen it as part of the path. A negative-weight cycle, on the other hand, would have been included in the path by RELAX. In Section 2.5 we chose shortest-paths algorithms especially for their property to abort upon finding a negative cycle. Bellman-Ford does that by iterating over all edges one last time and checking if RELAX would find a better path. If that is the case, Bellman-Ford returns false, because it found a negative cycle.

```
function BELLMAN-FORD($G$,$s$)
    for all $v$ in $V$ do
        $d(v) \leftarrow \infty$
        $\pi[v] \leftarrow$ NULL
    end for
    $n \leftarrow |V|$
    $d(s) \leftarrow 0$
    for $i \leftarrow 0$ to $n - 1$ do
        for all $(u,v)$ in $E$ do
            RELAX($u$,$v$)
        end for
    end for
    for all $(u,v)$ in $E$ do
        if $d(u) + w(u,v) < d(v)$ then
            return False
        end if
    end for
    return True
end function
```

Algorithm 2: Bellman-Ford

The most serious drawback of using Bellman-Ford to find negative cycles is that it only checks for negative cycles in the end. Furthermore, it keeps trying to relax edges, even when all edges are stable. It therefore always has a complexity of $\mathcal{O}(|V| \cdot |E|)$ as the initialization takes $\Theta(|V|)$, the $|V| - 1$ iterations over all edges take $\mathcal{O}(|E|)$ each and the last check for negative cycles takes $\mathcal{O}(|E|)$ as well. As the SAT-solver might only apply minor changes to the passed system of difference constraints, compared to the last invocation of the theory-solver, an incremental algorithm could reduce the running time significantly. The first step to achieve that is to ensure that Bellman-Ford is still sound and complete with an arbitrary set of initial node values, because the incremental version will use the existing constraint graph and also make use of the prior computed distance approximations of the vertices, to reduce the number of relaxations needed to achieve a stable graph if possible. The following two theorems are taken from [WIGG05].

Theorem 3.0.1. *For a constraint graph representing a set of difference logic constraints, Bellman-Ford with arbitrary initial node values assigns suitable distance approximations, such that every edge satisfies its representing constraint if possible.*

Proof. Let $G = (V,E)$ be a constraint graph and $(u,v) \in E$ an arbitrary edge with weight $w(u,v) = c$ representing the constraint $x - y \leq c$. There are a two different cases to acknowledge:

Case: RELAX(u,v) does not change $d(v)$
If the relaxation operation does not change the distance approximation of v, the inequality $d(u) + c < d(v)$ does not hold and therefore $d(u) + c \geq d(v)$ holds.

$$\begin{aligned} d(u) + c &\geq d(v) \\ \equiv \qquad c &\geq d(v) - d(u) \end{aligned}$$

This exactly matches the represented constraint $x - y \leq c$, as v represents x and u represents y.

Case: RELAX(u,v) changes $d(v)$
The proof is done by induction over the number of consecutive relaxed edges k.

Base case $k = 1$:
After the relaxation operation, $d(v) = d(u) + c$ holds. If we input the distance approximations into the constraint as values for their respective variables, we get

$$\begin{aligned} d(u) + c - d(u) &\leq c \\ \equiv \qquad c &\leq c \end{aligned}$$

Induction hypothesis:
Suppose the Bellman-Ford algorithm computed correct distance approximations in n consecutive relaxation operations.

Induction step $k = n + 1$:
For every relaxed edge (v_i,v_{i+1}), the predecessor $\pi(v_{i+1})$ is set to v_i. Therefore a path $p = \langle v_1, v_2, \ldots, v_n \rangle$ exists in the graph before the $(k + 1)$-th relaxation. As every edge (v_{i-1},v_i) in p was relaxed

$$d(v_i) = d(v_{i-1}) + w(v_{i-1},v_i), \quad \forall (v_{i-1},v_i) \in p \tag{3.1}$$

holds. Substituting the equalities along the path yields

$$d(v_n) = d(v_1) + \sum_{i=2}^{n} w(v_{i-1},v_i) \tag{3.2}$$

Next, the edge (v_n,v_{n+1}) is relaxed, thus

$$d(v_{n+1}) = d(v_n) + w(v_n,v_{n+1})$$

Substituting $d(v_n)$

$$d(v_{n+1}) = d(v_1) + \left(\sum_{i=2}^{n} w(v_{i-1},v_i)\right) + w(v_n,v_{n+1}) \tag{3.3}$$

Inserting into the represented constraint

$$d(v_{n+1}) - d(v_n) \leq w(v_n,v_{n+1})$$

Substituting $d(v_{n+1})$ by Eq. 3.3 and $d(v_n)$ by Eq. 3.2

$$\begin{aligned} & d(v_1) + \left(\sum_{i=2}^{n} w(v_{i-1},v_i)\right) + w(v_n,v_{n+1}) - d(v_1) + \sum_{i=2}^{n} w(v_{i-1},v_i) \leq w(v_n,v_{n+1}) \\ \equiv \quad & w(v_n,v_{n+1}) \leq w(v_n,v_{n+1}) \\ \equiv \quad & 0 \leq 0 \end{aligned}$$

Therefore the distance approximations assigned by Bellman-Ford form a satisfying model for the set of difference constraints if there is one. □

Theorem 3.0.2. *For the purpose of detecting negative weight cycles, Bellman-Ford is sound and complete by starting with an arbitrary set of initial node values (instead of initializing $d(v)$ to ∞).*

Proof. Let $G = (V,E)$ be a constraint graph and the final node value $d(v) \in D, \forall v \in V$. Furthermore let $c = \langle v_0,v_1,\dots,v_k \rangle$, $v_0 = v_k$ be a negative cycle. For the sake of contradiction, let us assume, that Bellman-Ford returned true, despite G containing a negative cycle c. Therefore, all edges have to be stable, meaning $d(u)+w(u,v) \geq d(v)$, $\forall (u,v) \in E$ holds. Otherwise, the algorithm would have returned false. Furthermore, because c is a negative cycle

$$\sum_{i=1}^{k} w(v_{i-1},v_i) < 0 \tag{3.4}$$

For all edges in c, the aforementioned inequality $d(v_{i-1}) + w(v_{i-1},v_i) \geq d(v_i)$, $i = 1,\dots,k$ holds. We can add up these inequalities along the negative cycle c, leading to the following inequality:

$$\sum_{i=1}^{k} d(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1},v_i) \geq \sum_{i=1}^{k} d(v_i) \tag{3.5}$$

The two sums over the distances are in fact equal, as both contain the shortest-path estimates of all vertices in c. The first sum adds $d(v_0) + \cdots + d(v_{k-1})$ and the second $d(v_1) + \cdots + d(v_k)$, with $v_0 = v_k$. Thus, they can be eliminated from the inequality leaving

$$\underbrace{\sum_{i=1}^{k} w(v_{i-1},v_i)}_{<0,\text{ because of Eq. 3.4}} \quad \geq 0 \tag{3.6}$$

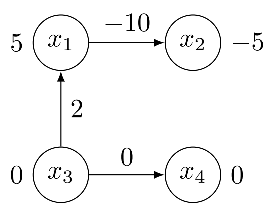
which poses a contradiction.

□

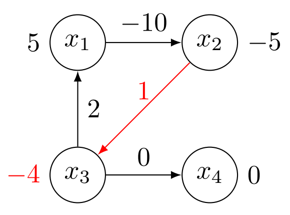
3.1 Incremental Negative Cycle Detection

As we have just proven, the Bellman-Ford algorithm is sound and complete for finding negative cycles and gives a satisfying model, if there is one, when initialized with arbitrary initial node values. The next step is to introduce the incremental algorithm. We have implemented the incremental negative cycle detection algorithm presented in [WIGG05] (see Algorithm 3). Assume that the Bellman-Ford algorithm does not find a negative cycle. Therefore, all edges in the current constraint graph $G = (V,E)$ have to be stable, and the theory solver returns satisfiable to the SAT-solver. The SAT-solver either returns that the system of difference constraints is satisfiable overall and finishes or gives additional constraints to the theory solver to check. Should Bellman-Ford now detect a negative cycle in the expanded constraint graph $G' = (V',E')$, at least one edge, representing a newly added constraint, has to be part of the cycle. If no new edge were part of the negative cycle, it would have already been present in the old graph G and therefore found by Bellman-Ford beforehand. This property is the main invariant of the incremental algorithm.

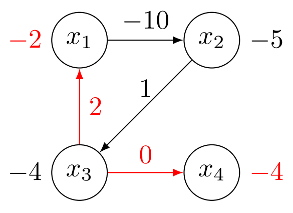
Instead of one full call to the Bellman-Ford algorithm, the incremental algorithm is invoked to check for negative cycles after each newly added edge (v_i,v_j) and tries to relax this edge. If the edge was relaxed, the node value for the target v_j is changed. This change is forwarded throughout the graph by relaxing all outgoing edges of the target $(v_j,v_k) \in E$. This process is repeated until all edges are stable again or the newly added edge is reached again. If that happens, (v_i,v_j) has to be part of a negative cycle. The proof is similar to Theorem 3.0.2. Figure 3.1 shows an invocation of the algorithm on a simple graph. The relaxation operations are highlighted in red. After the case (d) in Figure 3.1, the algorithm would have to relax the edge (x_2,x_3) again and aborts, returning that a negative cycle was found. Notice, that the constraint graph does not contain a start node s. This is because the algorithm is invoked for every edge when it is added. Therefore, it uses the source of the newly added edge as the start node for this invocation of the algorithm. Also, the graph does not have to be connected, as the proof of Theorem 3.0.1 shows, that a correct model is computed even for single edges. This reduces the size of the constraint graph by $|V|$ edges, reducing the running time of the theory solver.



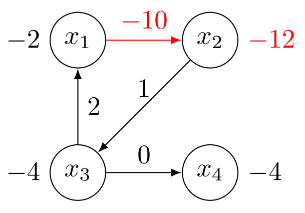
(a) before adding the new edge



(b) RELAX(x_2,x_3)



(c) RELAX($x3$,$x4$), RELAX(x_3,x_1)



(d) RELAX(x_1,x_2)

Figure 3.1: Example of incremental negative cycle checking

```
 1: function DETECTNEGATIVECYCLE($u$,$v$,$G$)
 2:     if $d(v) > d(u) + w(u,v)$ then
 3:         RELAX($u$,$v$)
 4:         ENQUEUE($v$)
 5:     end if
 6:     while $x$ = DEQUEUE do
 7:         for all $(x,y) \in E$ do                ▷ sequenced with priority queue
 8:             if $d(y) > d(x) + w(x,y)$ then
 9:                 if $x == u$ AND $y == v$ then
10:                     return True
11:                 else
12:                     RELAX($x$,$y$)
13:                     ENQUEUE($y$)
14:                 end if
15:             end if
16:         end for
17:     end while
18:     return False
19: end function
```

Algorithm 3: The incremental negative cycle detection algorithm

In line [7] of Algorithm 3, the outgoing edges of the currently considered vertex x are ordered in a priority queue. For every edge (x,y), the corresponding priority is computed as $p(x,y) = d(y) - (d(x) + w(x,y))$. A higher priority represents a larger negative change to the distance value of y when the edge (x,y) is relaxed. By relaxing these edges first, the algorithm generally finds a negative cycle with fewer checks compared to simply checking them in the order they are stored. The resulting running time of the incremental negative cycle detection algorithm is in $\mathcal{O}(|V| \log |V| + |E|)$

for every checked constraint.

We were able to improve the performance of the algorithm by adding an option for the theory solver to enforce a directed *simple graph*, which is a graph $G' = (V',E')$ without parallel edges, by only adding the edge with the lowest weight e_{ij} for every pair of vertices v_i,v_j. The other parallel edges are stored as a set of alternatives for each edge. When an edge is removed, the best alternative replaces it. Theorem 3.1.1 shows the correctness of this method in the context of shortest-paths problems. This reduces the running time of the algorithm from $\mathcal{O}(|V|\log|V| + |E|)$ to $\mathcal{O}(|V|\log|V| + |E'|)$. Depending on the structure of the examined problem, E' may be significantly smaller than E.

Theorem 3.1.1. *When searching for shortest paths in a graph $G = (V,E)$ with parallel edges, it is enough to search through a simple graph $G' = (V',E')$, $V' = V$, $E' \subset E$, with E' only containing the edge with the smallest weight for every set of parallel edges.*

Proof. Let $G = (V,E)$ be a graph with parallel edges and let E_{ij} denote the set of parallel edges between two vertices $v_i,v_j \in V$. Assume that the shortest path p from v_i to v_j only contains a single edge $e_{ijk} \in E_{ij}$. When p is a shortest path, then $w(e_{ijk}) = \min_{e_{ijl} \in E_{ij}} w(e_{ijl})$ must hold and because of Lemma 2.5.2 it must hold for paths of arbitrary length. $\square$

For $\mathcal{DL}(\mathbb{R})$ the solver has to be able to work with strict inequalities. This is done by using an adjust value. For every vertex in the graph, and thus for every variable in the set of constraints, an adjust value $a(v) \in \mathbb{Z}, v \in V$ is kept. Every time an edge $(u,v) \in E$ is relaxed, the adjust value is changed to $a(v) = a(u) + a(u,v)$, where $a(u,v) = 1$, if the constraint represented by the edge has the relation ">" and $a(u,v) = -1$, if the relation is "<". For all other relations the adjust value of the edge is $a(u,v) = 0$. When returning a model for the current system of difference constraints, every variable x_i represented by a vertex v_i is assigned $x_i = d(v_i) + \frac{a(v_i)}{\#\text{strict constraints}+1}$.

3.2 Conflict Resolution and Backtracking

After finding a negative cycle in the graph, it is no longer stable after exiting the algorithm. This violates the invariant mentioned at the beginning of the last section. Thus, the algorithm records all changes made during its current invocation by storing the distances of vertices before they are changed for the first time. Upon finding a negative cycle, all changes are reverted. The theory solver then removes the last added edge e from the graph, restoring it to a stable state, and schedules the edge to be added again. The predecessors are not restored, as they are used afterwards to determine the infeasible subset. This poses no problem because the predecessors are only used to reconstruct a negative cycle in the graph. When the algorithm finds a negative cycle, it traversed all its edges beforehand and therefore assigned appropriate predecessors.

Should the preceding module not determine unsatisfiability for the complete formula, the theory solver is tasked to remove a subset C' of its currently considered set of constraints C. This is done by simply removing the corresponding edges from the graph or, if the constraints were not added to the graph, by removing the constraints from the set of scheduled constraints. The distances of the vertices are not reverted to the values before the constraints in C' were added. This is possible because every constraint imposes bounds on the values of its variables, and by removing constraints, these bounds are loosened. A satisfying assignment for C must therefore also satisfy the set of constraints $C \setminus C'$.

Chapter 4

Floyd-Warshall-based Solver

The incremental algorithm described in the previous chapter has a running time that scales with both the number of edges and the number of vertices in the constraint graph. This directly relates to the number of constraints and variables in the system of difference constraints. Another well-known shortest-paths algorithm is the Floyd-Warshall algorithm, which solves the all-pairs shortest-paths problem. Its running time scales only with the number of vertices in the graph with a running time in $\mathcal{O}(|V|^3)$. Thus, for problems resulting in a highly dense constraint graph, it might prove advantageous to use this algorithm instead. A simple graph $G = (V,E)$ is called *dense* when the number of its edges approaches the maximum number of possible edges. A simple graph contains the maximum number of edges when every vertex is connected to every other vertex by an edge. Therefore it may only contain a maximum of $|V| \cdot (|V| - 1)$ edges. There is no similar definition for a graph with parallel edges because a graph $G = (V,E)$ with only two vertices $u,v \in V$ may already contain an infinite number of parallel edges $(u,v) \in E$.

The basic idea of the Floyd-Warshall algorithm is to check for every possible path from a vertex i to another vertex j in the constraint graph if traversing a third vertex k along the path leads to a better estimation of the shortest-path weight. The shortest path $i \overset{p}{\rightsquigarrow} j$ is then changed to the concatenation of the two shortest paths $i \overset{p_{ik}}{\rightsquigarrow} k$ and $k \overset{p_{kj}}{\rightsquigarrow} j$. This is repeated for every vertex k and every path $i \overset{p_{ij}}{\rightsquigarrow} j$.
The algorithm uses a matrix with dimensions $|V| \times |V|$, called the *distance matrix* D, which is initialized with a distance of infinity for all pairs of vertices without an edge in the graph and the edge weight for those connected by an edge. Furthermore, the diagonal of the matrix contains the paths from any vertex i to itself. These are initialized with a distance of zero.

$$d[i,j] = \begin{cases} w(i,j) & ,(i,j) \in E \\ \infty & ,(i,j) \notin E \\ 0 & ,i = j \end{cases}$$

$d[i,j]$ is the entry in the i-th row and j-th column of D

Every time an improvement of a possible path is found, the distance value in the matrix is updated. In the end, the matrix contains all shortest paths in the graph with the entry in the i-th row and j-th column denoting the shortest-path weight of the path $i \overset{p}{\rightsquigarrow} j$. This allows us to construct a model for a satisfying formula by using the shortest-path weights from s to all other vertices as the assignment for the represented variables in accordance to Theorem 2.5.3. This is done by iterating over the row for s in the distance matrix. A negative cycle is found if an entry on the diagonal of the matrix is set to a negative value, as this would indicate a shortest path with negative weight from the vertex to itself. The algorithm is shown in Algorithm 4.

This would be enough to determine the satisfiability of a formula and, if satisfiable, a model. As SMT-RAT implements a lazy SMT-solver, it also requires the theory solver to return an infeasible subset if the passed formula was unsatisfiable. Therefore, the solver has to be able to reconstruct the negative cycle. To achieve this, a second matrix is implemented, called the *path matrix* P, which is initialized to

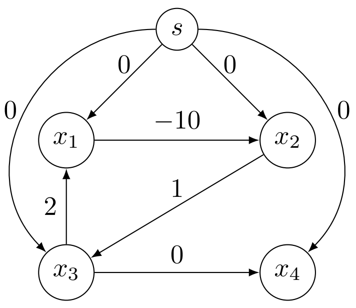
$$p[i,j] = \begin{cases} j & ,(i,j) \in E \\ \perp & ,\text{else} \end{cases}$$

where $p[i,j]$ is the entry in the i-th row and j-th column of P

If a path $i \overset{p}{\rightsquigarrow} j$ is improved by traversing the vertex k, the entry in the path matrix is updated to $p[i,j] = k$. This allows us to reconstruct the shortest-path tree for any path by recursively going through the entries in the path matrix and decomposing the path into its single edges in a similar fashion, as they were built. For example, if for a vertex i, there was an entry in the path matrix $p[i,i] = j$, then the path from i to itself could be decomposed into the path from i to j and the path from j to i. This is repeated recursively until the paths are composed of only singular edges (see Algorithm 5). An example for that is shown in Figure 4.1(d).

```
1: function FLOYD-WARSHALL($D$,$P$,$G$)
2:     INITIALIZEMATRIX($D$,$P$,$G$)
3:     for $k \leftarrow 1$ to $|V|$ do
4:         for $i \leftarrow 1$ to $|V|$ do
5:             for $j \leftarrow 1$ to $|V|$ do
6:                 if $d[i,j] > d[i,k] + d[k,j]$ then
7:                     $d[i,j] \leftarrow d[i,k] + d[k,j]$
8:                     $p[i,j] \leftarrow k$
9:                 end if
10:                if $d[i,i] < 0$ then
11:                    return False
12:                end if
13:            end for
14:        end for
15:    end for
16:    return True
17: end function
```

Algorithm 4: Floyd-Warshall



$$\begin{array}{c|ccccc|ccccc|}
 & s & x_1 & x_2 & x_3 & x_4 & r & x_1 & x_2 & x_3 & x_4 \\
\hline
s & 0 & 0 & 0 & 0 & 0 & \perp & x_1 & x_2 & x_3 & x_4 \\
x_1 & \infty & 0 & -10 & \infty & \infty & \perp & \perp & x_2 & \perp & \perp \\
x_2 & \infty & \infty & 0 & 1 & \infty & \perp & \perp & \perp & x_3 & \perp \\
x_3 & \infty & 2 & \infty & 0 & 0 & \perp & x_1 & \perp & \perp & x_4 \\
x_4 & \infty & \infty & \infty & \infty & 0 & \perp & \perp & \perp & \perp & \perp
\end{array}$$

(a) after initialization

$$\begin{array}{c|ccccc|ccccc|}
 & s & x_1 & x_2 & x_3 & x_4 & r & x_1 & x_2 & x_3 & x_4 \\
\hline
s & 0 & 0 & -10 & 0 & 0 & \perp & x_1 & x_1 & x_3 & x_4 \\
x_1 & \infty & 0 & -10 & \infty & \infty & \perp & \perp & x_2 & \perp & \perp \\
x_2 & \infty & \infty & 0 & 1 & \infty & \perp & \perp & \perp & x_3 & \perp \\
x_3 & \infty & 2 & -8 & 0 & 0 & \perp & x_1 & x_1 & \perp & x_4 \\
x_4 & \infty & \infty & \infty & \infty & 0 & \perp & \perp & \perp & \perp & \perp
\end{array}$$

(b) after check for $k = 1$

$$\begin{array}{c|ccccc|ccccc|}
 & s & x_1 & x_2 & x_3 & x_4 & r & x_1 & x_2 & x_3 & x_4 \\
\hline
s & 0 & 0 & -10 & -9 & 0 & \perp & x_1 & x_1 & x_2 & x_4 \\
x_1 & \infty & 0 & -10 & -9 & \infty & \perp & \perp & x_2 & x_2 & \perp \\
x_2 & \infty & \infty & 0 & 1 & \infty & \perp & \perp & \perp & x_3 & \perp \\
x_3 & \infty & 2 & -8 & -7 & 0 & \perp & x_1 & x_1 & x_2 & x_4 \\
x_4 & \infty & \infty & \infty & \infty & 0 & \perp & \perp & \perp & \perp & \perp
\end{array}$$

(c) after check for $k = 2$, found negative cycle through x_3

$$\begin{tikzcd}
 & & x_3,x_3 \arrow[dl] \arrow[dr] & \\
 & x_3,x_2 \arrow[dl] \arrow[dr] & & x_2,x_3 \\
x_3,x_1 & & x_1,x_2 &
\end{tikzcd}$$

(d) reconstruction of the negative cycle $x_3 \to x_1 \to x_2 \to x_3$

Figure 4.1: Example of the Floyd-Warshall algorithm with path reconstruction

```
1: function reconstructPath($i$,$j$, $P$)
2:     if $p[i,j] = j$ then
3:         addToInfeasableSubset($i$,$j$)
4:     else
5:         reconstructPath($i$,$p[i,j]$,$P$)
6:         reconstructPath($p[i,j]$,$j$,$P$)
7:     end if
8: end function
```

Algorithm 5: Reconstruction of a shortest path

4.1 Incremental Floyd-Warshall

Again an incremental version of the algorithm would be advantageous, as often only minor changes are made to the constraint graph between theory calls. To achieve this, we developed an incremental version of the Floyd-Warshall algorithm based on [RHK15] and [HRK17]. The basic idea is that instead of checking for every vertex in the graph if traversing it would lead to an improvement for any path, the incremental algorithm checks if a newly added edge leads to an improvement of the shortest-path estimate. Thus, the algorithm is called for every edge that is added to the graph, instead of once at the end. The core comparison of the Floyd-Warshall algorithm to decide whether traversing a vertex k would lead to an improvement for the path $i \overset{p}{\leadsto} j$ is $d[i,j] > d[i,k] + d[k,j]$. The incremental algorithm performs a similar comparison $d[i,j] > d[i,s] + w(s,t) + d[t,j]$. It checks if using the shortest path from the vertex i to the source of the new edge s, plus traversing the new edge (s,t) with weight $w(s,t)$ and going from the target of the edge t to j leads to a better estimate of the shortest-path weight. This is repeated for all possible paths $i \overset{p}{\leadsto} j,\ i,j \in V$. This changes the running time of the algorithm to $\mathcal{O}(|V|^2)$ for every newly added edge because the outermost loop is removed. The non-incremental Floyd-Warshall algorithm has a time complexity of $\mathcal{O}(|V|^3)$ for computing the shortest paths for the whole graph. In practice often only a small subset of the set of constraints is changed and therefore, the incremental algorithm performs better. Given that not all constraints are exchanged for every call to the theory solver, the incremental algorithm should perform better. The incremental Floyd-Warshall algorithm uses the same data structures as the non-incremental version. A distance matrix is kept, recording the current distances of the paths in the graph and a path matrix to reconstruct any negative cycles. The initialization procedure has to be changed to accommodate for new edges after a previous check. Instead of initializing the complete distance matrix at once, it is initialized as a 1×1-matrix, only containing the start node s. The same goes for the path matrix. Every time an edge is added, it is checked whether the source and target vertex are already part of the graph. Should that not be the case, then a row and a column are added for the missing vertices to both matrices and initialized with infinity. Then the distance matrix is checked whether the weight of the new edge is smaller than the current entry in the distance matrix. If that is the case, the corresponding entry is updated, and the incremental Floyd-Warshall algorithm is executed for that edge. Furthermore, the entry in the path matrix is set to the target of the new edge. If the entry in the distance matrix is smaller than the weight of the new edge, there already exists a path in the graph with a smaller weight. This means that the new edge would not be part of the shortest path between the two vertices it connects. Thus, it would not be part of any shortest path (Lemma 2.5.2) in the graph, and the algorithm does not have to be executed, as no changes were made to the matrices.

```
function addEdge($G$,$D$,$P$,$i$,$j$)
    if $i \notin V$ then
        addToMatrix($D$,$i$)        ▷ add row and column, initialize with infinity
        $d[i,i] \leftarrow 0$
        addToMatrix($P$,$i$)
    end if
    if $j \notin V$ then
        addToMatrix($D$,$j$)
        $d[j,j] \leftarrow 0$
        addToMatrix($P$,$j$)
    end if
    if $d[i,j] > w(i,j)$ then
        $d[i,j] \leftarrow w(i,j)$
        $p[i,j] \leftarrow j$
        incrementalFloydWarshall($D$,$P$,$i$,$j$,$w(i,j)$,$G$)
    end if
end function
```

Algorithm 6: Add an edge to the graph

The satisfying model is obtained the same way as for the non-incremental Floyd-Warshall algorithm, but reconstructing the negative cycle requires some extra care. The non-incremental algorithm stops as soon as an entry on the diagonal of the distance matrix is set to a negative value. The negative cycle can then be reconstructed from the path matrix. When the incremental algorithm detects a negative cycle, it may have found it before the path matrix was changed appropriately, thus leading to an incomplete reconstruction of the negative cycle. This problem does not arise with the non-incremental algorithm, as all edges are present from the beginning. To avoid this problem, the diagonal of the distance matrix is checked before any changes are made to the matrix. If adding a new edge (v_s,v_t) would lead to the presence of a negative cycle, then all other edges of the negative cycle have to be present beforehand. For any vertex, that is part of the negative cycle, the entry on the diagonal of the distance matrix would be set to a negative value. Therefore it is enough to check the entries on the diagonal at the beginning of the algorithm. If an entry $d[v_i,v_i]$ would be set to a negative value, the negative cycle can be obtained by reconstructing the paths $v_i \overset{p_{is}}{\rightsquigarrow} v_s$ and $v_t \overset{p_{ti}}{\rightsquigarrow} v_i$ in addition with the new edge (v_s,v_t). This is shown in Figures 4.2 and 4.3.

$$\begin{tikzcd} x_1 \arrow[dr, "0"] & \\ & x_2 \end{tikzcd}$$

$$\begin{array}{c|cc|cc} & x_1 & x_2 & x_1 & x_2 \\ \hline x_1 & 0 & 0 & \bot & x_2 \\ x_2 & \infty & 0 & \bot & \bot \end{array}$$

(a) after checking (x_1,x_2)

$$\begin{tikzcd} & x_1 \arrow[dr, "0"] & \\ x_3 \arrow[ur, "-2"] & & x_2 \end{tikzcd}$$

$$\begin{array}{c|ccc|ccc} & x_1 & x_2 & x_3 & x_1 & x_2 & x_3 \\ \hline x_1 & 0 & 0 & \infty & \bot & x_2 & \bot \\ x_2 & \infty & 0 & \infty & \bot & \bot & \bot \\ x_3 & -2 & -2 & 0 & x_1 & x_1 & \bot \end{array}$$

(b) after checking (x_3,x_1)

$$\begin{tikzcd} & x_1 \arrow[dr, "0"] & \\ x_3 \arrow[ur, "-2"] & & x_2 \arrow[ll, "1"'] \end{tikzcd}$$

$$\begin{array}{c|ccc|ccc} & x_1 & x_2 & x_3 & x_1 & x_2 & x_3 \\ \hline x_1 & -1 & 0 & \infty & x_3 & x_2 & \bot \\ x_2 & \infty & 0 & \infty & \bot & \bot & \bot \\ x_3 & -2 & -2 & 0 & x_1 & x_1 & \bot \end{array}$$

(c) found negative cycle after checking (x_2,x_3)

$$\begin{tikzcd} & x_1,x_1 \arrow[dl] \arrow[dr] & \\ \lightning\ x_1,x_3 & & x_3,x_1 \end{tikzcd}$$

(d) edge $x_2 \to x_3$ not yet in the path matrix

$$\begin{tikzcd} & x_1,x_1 \arrow[dl] \arrow[d] \arrow[dr] & \\ x_1,x_2 & x_2,x_3 & x_3,x_1 \end{tikzcd}$$

(e) solution: split path reconstruction around new edge

Figure 4.2: Example for failing to reconstruct a negative cycle

$$\begin{tikzcd}
 & x_1 \arrow[dr, "0"] & \\
x_3 \arrow[ur, "-2"] & & x_2 \arrow[ll, "1"']
\end{tikzcd}$$

$$\begin{array}{c|ccc|ccc}
 & x_1 & x_2 & x_3 & x_1 & x_2 & x_3 \\
\hline
x_1 & 0 & 0 & \infty & \bot & x_2 & \bot \\
x_2 & \infty & 0 & 1 & \bot & \bot & x_3 \\
x_3 & -2 & \infty & 0 & x_1 & \bot & \bot
\end{array}$$

(a) after initialization

$$\begin{array}{c|ccc|ccc}
 & x_1 & x_2 & x_3 & x_1 & x_2 & x_3 \\
\hline
x_1 & 0 & 0 & \infty & \bot & x_2 & \bot \\
x_2 & \infty & 0 & 1 & \bot & \bot & x_3 \\
x_3 & -2 & -2 & 0 & x_1 & x_1 & \bot
\end{array}$$

(b) after check for $k = 1$

$$\begin{array}{c|ccc|ccc}
 & x_1 & x_2 & x_3 & x_1 & x_2 & x_3 \\
\hline
x_1 & 0 & 0 & 1 & \bot & x_2 & x_2 \\
x_2 & \infty & 0 & 1 & \bot & \bot & x_3 \\
x_3 & -2 & -2 & -1 & x_1 & x_1 & x_2
\end{array}$$

(c) negative cycle found after check for $k = 2$

$$\begin{tikzcd}
 & & x_3{,}x_3 \arrow[dl] \arrow[dr] & \\
 & x_3{,}x_2 \arrow[dl] \arrow[dr] & & x_2{,}x_3 \\
x_3{,}x_1 & & x_1{,}x_2 &
\end{tikzcd}$$

Figure 4.3: The non-incremental Floyd-Warshall can reconstruct the cycle

Theorem 4.1.1. *Executing the incremental Floyd-Warshall algorithm for every newly added edge is a sound and complete procedure for finding shortest paths in a graph.*

Proof. Let $G_k = (V_k,E_k)$ be the constraint graph present in the k-th invocation of the algorithm. The incremental Floyd-Warshall algorithm is executed after each added edge, thus $|E_k| = k$. Furthermore, let $e_k \in E_k$ be the last edge added before the k-th execution of the algorithm. The proof of correctness is done by induction over the number of edges in G_k.

Base case $k = 1$:
The constraint graph $G_1 = (V_1,E_1)$ contains only two vertices $v_1,v_2 \in V_1$ and the edge $e_1 = (v_1,v_2)$ connecting them. When e_1 was added to the graph, $d[v_1,v_2]$ was set to $w(v_1,v_2)$. This is already the correct shortest path.
Induction hypothesis:
Suppose the incremental Floyd-Warshall algorithm computed a correct distance matrix for a graph G_n with $|E_n| = n$.

```
function INCREMENTALFLOYD-WARSHALL($D$,$P$,$s$,$t$,$w(s,t)$,$G$)
    for $i \leftarrow 1$ to $|V|$ do
        if $d[i,i] > d[i,s] + w(s,t) + d[t,i]$ then
            RECONSTRUCTPATH($i$,$s$,$P$)
            RECONSTRUCTPATH($t$,$i$,$P$)
            ADDTOINFEASABLESUBSET($s$,$t$)
            return False
        end if
    end for
    for $i \leftarrow 1$ to $|V|$ do
        for $j \leftarrow 1$ to $|V|$ do
            if $d[i,j] > d[i,s] + w(s,t) + d[t,j]$ then
                $d[i,j] \leftarrow d[i,s] + w(s,t) + d[t,j]$
                if $j = t$ AND $i \neq s$ then
                    $p[i,j] = s$
                else
                    $p[i,j] = t$
                end if
            end if
        end for
    end for
    return True
end function
```

Algorithm 7: Incremental Floyd-Warshall

Induction step $k = n + 1$:
For every pair of vertices v_i,v_j, the shortest paths, that were computed in the prior invocations of the algorithm, are a combination of edges from the set of edges E_n, or there is no path $v_i \overset{p_{ij}}{\rightsquigarrow} v_j$. A new edge $e_{n+1} \in E_{n+1}$ is added afterwards. The algorithm is then executed again, and for all pairs of vertices, it is checked, if traversing e_{n+1} as an intermediate edge yields an improvement. If a change was made to a shortest path $p_{i,j}$, the new path can be decomposed into $p_{ik} + (v_k,v_l) + p_{lj}$, where (v_k,v_l) is the new edge e_{n+1}. The paths p_{ik} and p_{lj} both consist of an arbitrary number of edges from the set E_n. These subpaths were either computed during a prior invocation of the algorithm or directly represent edges in the graph. Otherwise, their entries in the distance matrix would still be set to the initial infinite value and no change would have been made. If no change was made, the inequality $d[i,j] > d[i,k] + w(k,l) + d[k,j]$ does not hold, either because a path over the new edge e_{n+1} would yield no improvement or because there is no such path in the graph. □

4.2 Backtracking

Suppose the theory solver is tasked with removing the constraint $x_i - x_j \leq c$ from its system of considered constraints. In contrast to the incremental Bellman-Ford algorithm, which could remove the respective edge from the graph directly, the incremental Floyd-Warshall algorithm has to restore a previous state completely. The reason is that the Floyd-Warshall algorithm uses the distance matrix not only to store

the computed shortest-path weights but also to determine if a path from a vertex to another exists at all. Therefore setting the entry $d[v_i,v_j]$ in the distance matrix to infinity would not only indicate that there is no direct edge from v_i to v_j in the graph but no path at all. The naive solution would be to reconstruct the distance matrix from the graph after the edge has been removed. This would result in an enormous overhead because all prior computations made by the algorithm would be lost and therefore had to be done again. The result would be, that after every series of removing operations, the theory solver practically would have to run the non-incremental Floyd-Warshall algorithm on the whole graph. In the worst case, only a single entry was changed in the distance matrix, but all other entries would have to be recomputed nonetheless. This would result in a solver that is only incremental for adding new constraints and non-incremental once after every call to remove a subset of its constraints.
Instead, chronological backtracking can be achieved by keeping records on all changes made to the matrix in combination with the responsible formulas. This allows the solver to backtrack to the point before a certain formula was added, only reverting the changes made afterwards. To that end, the solver keeps a history stack H of *history elements* H_i. Every history element H_i is made up of a set of constraints F_i and a stack of *changes* C_i. A change is of a tuple $c_{ij} = (k,l,d[k,l]^c,p[k,l]^c)$ consisting of the row and column of the entry that was changed, as well as the content of the distance matrix and path matrix before the change. Instead, every history element could store a copy of the whole matrix and simply copy it back when needed. In practice, reapplying changes made to the matrix one after another has proven to be faster, despite the overhead of keeping track of all made changes. In an extreme case, the theory solver would have to store the whole matrix in the history stack for a single additionally added constraint, that may not cause any changes to the matrix at all.
When the solver is tasked with removing a constraint φ, the corresponding edge is removed from the graph, and the top element H_t of the history stack H is examined. All constraints from the set F_t are removed from the graph, and all changes stored in C_t are reverted. H_t is then removed from the stack. This is repeated until $\varphi \in F_t$ holds, in which case the solver stops after removing the currently checked history element H_t. Instead of storing a set of constraints, we could have a history element for every single constraint, which would allow the solver to avoid backtracking too far. This poses a trade-off between storing fewer data structures in the history stack and backtracking more exactly. In practice storing a set of constraints for every history element has proven to be more efficient.

```
function RESTOREHISTORY($D$,$P$,$G$,$H$,$f$)
    $found \leftarrow$ False
    while SIZE($H$) $> 0$ AND $\neg found$ do
        $H_t = (F_t, C_t) \leftarrow$ POP($H$)
        for all $g$ in $F_t$ do
            REMOVEEDGE($g$)
            if $g == f$ then
                $found \leftarrow$ True
            else
                SCHEDULETOADD($g$)
            end if
        end for
        while SIZE($C_t$) $> 0$ do
            $c = (i,j,d[i,j]^c,p[i,j]^c) \leftarrow$ POP($C_t$)
            $d[i,j] \leftarrow d[i,j]^c$
            $p[i,j] \leftarrow p[i,j]^c$
        end while
    end while
end function
```

Algorithm 8: Restore the state before the formula was added

Chapter 5

Preprocessing

In the context of SMT-solving, preprocessing describes a series of methods and algorithms that transform a formula φ into an equisatisfiable formula φ'. The goal is to simplify the original formula, such that the following SAT-solver and $\mathcal{T}$-solver may find a solution faster. A simple preprocessing was implemented, that splits equalities and disequalities as described in Section 2.4, by iterating over the input formula once. Additionally, the preprocessing module has an option to record the number of unique constraints and the number of non-Boolean variables to determine the expected density of the resulting constraint graph. The preprocessing module can then manually select which of the two implemented theory solvers to use. This is useful because SMT-RAT provides no equivalent condition for its strategies.

It was noticed, however, that in practice there are many cases were the theory solver is only called on a small subset of constraints from the input formula. Given that some of the problems, the solver was tested on, contain more than two million constraints, iterating over the whole formula may introduce a significant overhead. Therefore a second module was implemented, that splits the constraints after the SAT-solver and thus avoids unnecessary splitting of constraints. The procedure is as follows:

- Iterate over the received set of constraints and find all equalities and disequalities.
- For every disequality $x - y \neq c$, that was not already checked, add the lemma $x - y \neq c \implies ((x - y < c \lor x - y > c) \land \neg(x - y < c \land x - y > c))$.
- If a new lemma was added or the set of constraints contains a disequality without at least one constraint from its lemma, return the answer unknown.
- Split any equality $x - y = c$ into two constraints $x - y \leq c$ and $y - x \leq -c$.

Splitting constraints this way takes more time per constraint than splitting in the preprocessor. This is because the module has to keep track of all already split disequalities, and the SAT-solver has to process the passed lemmas. Depending on the problem, the total number of splits is reduced enough to provide an improved running time, nonetheless.

SMT-RAT already includes a number of preprocessing modules. Two of them were selected to compare their influence on the performance of SMT-RAT on difference logic formulas. The other preprocessing modules were excluded because their simplifications did not apply to difference logic or yielded no improvement for any benchmark. The first preprocessing module uses the method of symmetry breaking in the spirit of [DFMWP11], which consists of excluding subformulas that are only permutations of other subformulas. The second preprocessing module checks for top-level equality constraints and substitutes the respective variables in the complete formula.

Chapter 6

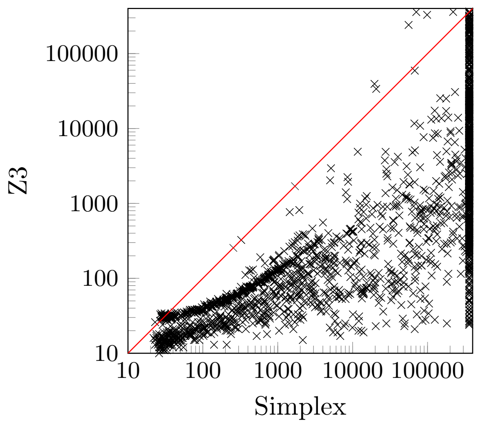
Experimental Results

The newly implemented solvers were tested against SMT-RAT's existing simplex solver for linear arithmetic, which is also able to solve difference logic problems. Notice, that both implemented solvers split equalities and disequalities before the SAT-solver if not explicitly stated otherwise. Furthermore, we compared the performance of the simplex solver against Z3, a state-of-the-art solver that has performed well in prior SMT-competitions, to provide a baseline on the current performance of SMT-RAT. All solvers were tested on the following set of benchmarks:

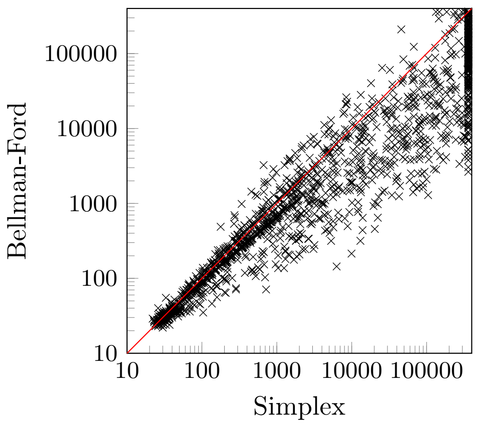
- 255 problems from the `QF_RDL` set of benchmarks
- 1732 problems from the `QF_IDL` set of benchmarks
- 325 DFT problems
- 325 incremental DFT problems

The `QF_RDL` and `QF_IDL` set of benchmarks are provided by SMT-LIB [BFT16]. The `asp` benchmark suite from the `QF_IDL` set of benchmarks had to be excluded, because of an issue with the SAT-solver that could not be solved in time. All benchmarks were run on an AMD Opteron 6172 with a memory limit of 4GB and a timeout of six minutes using SMT-RAT's `benchmax` tool. All times are given in milliseconds, if not stated otherwise.

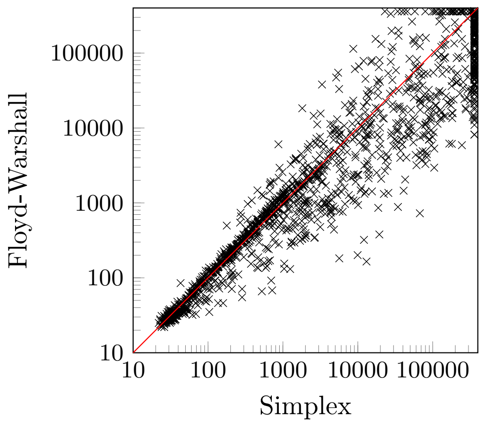
The results show that Z3 performs better than the simplex solver on nearly all benchmarks by a wide margin. Especially noticeable is the amount of problems that Z3 solves in a trivial amount of time, that cause the simplex solver to time out after six minutes. While the Bellman-Ford solver performs better than the simplex solver on all tested sets of benchmarks, especially for longer runtimes, it is still outperformed by Z3. The Floyd-Warshall solver performs worse than the Bellman-Ford solver on nearly all benchmarks, with the exception of a few select benchmarks from `QF_IDL` and the incremental DFT problems. On the one hand, this is caused by the fact, that Floyd-Warshall performs bad on problems containing a lot of variables, being implemented as a specialised solver for problems resulting in a dense constraint graph. On the other hand, Floyd-Warshall currently only supports chronological backtracking, which causes a significant overhead, depending on the problem. The Floyd-Warshall solver performs worse than the simplex solver on the non-incremental DFT problems and especially on `QF_RDL`, for the same reasons.



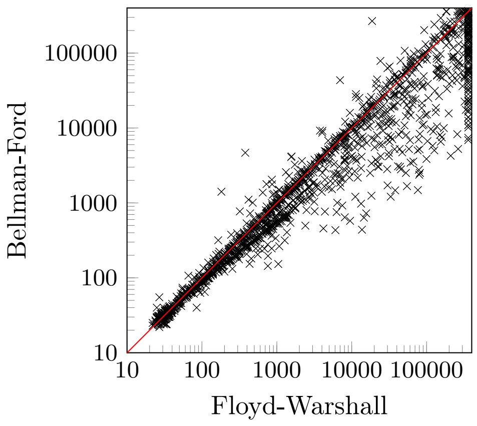
(a) Z3 vs. simplex



(b) Simplex vs. Bellman-Ford



(c) Simplex vs. Floyd-Warshall

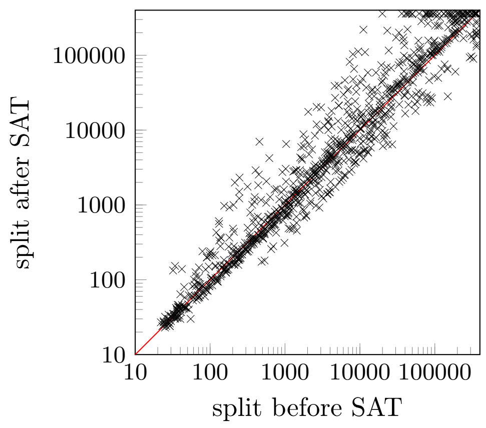


(d) Floyd-Warshall vs. Bellman-Ford

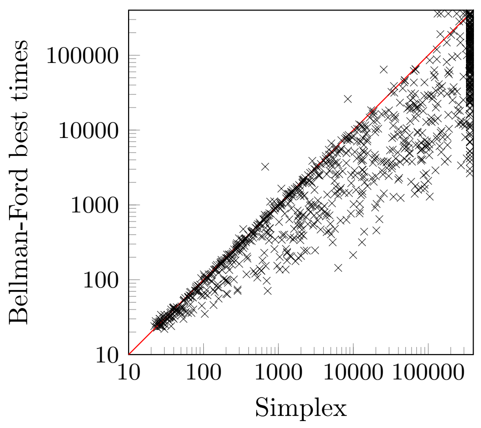
Figure 6.1: Performance comparison of the used solvers. The performance comparisons for the individual benchmark sets can be found in the appendix.

Benchmark Set	Solver	Runtime	SAT	UNSAT	Solved
QF_RDL	Bellman-Ford	96m14.809s	75	81	61.18%
	Floyd-Warshall	75m58.211s	59	49	42.35%
	Simplex	101m10.654s	71	55	49.41%
	Z3	68m52.274s	100	108	81.57%
QF_IDL	Bellman-Ford	503m50.627s	439	487	54.53%
	Floyd-Warshall	494m46.815s	410	453	50.82%
	Simplex	428m45.952s	313	432	43.88%
	Z3	321m37.227s	899	597	88.10%
DFT	Bellman-Ford	1m58.403s	10	315	100.00%
	Floyd-Warshall	3m23.833s	10	315	100.00%
	Simplex	3m1.476s	10	315	100.00%
	Z3	0m27.128s	10	315	100.00%
incremental DFT	Bellman-Ford	60m27.955s	11	115	38.77%
	Floyd-Warshall	67m19.48s	13	112	38.46%
	Simplex	47m48.091s	6	91	29.85%
	Z3	72m36.073s	0	169	52.00%

Table 6.1: Running times of the tested solvers. The complete tables can be found in the appendix.



(a) Splitting before the SAT-solver vs. splitting afterwards.



(b) Simplex vs. the best times achieved by both tactics of splitting constraints.

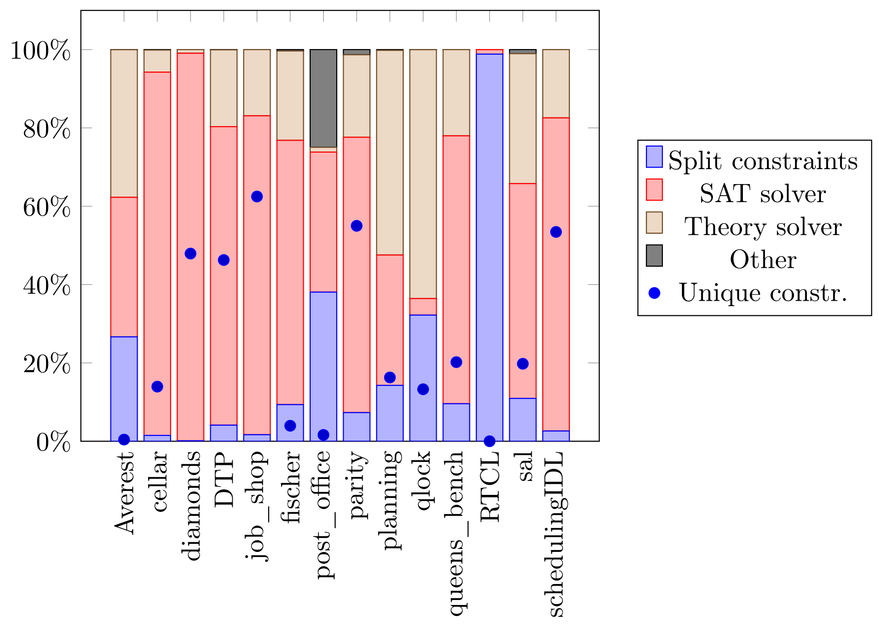
Figure 6.2: Performance comparison of the two strategies to either split constraints before the SAT-solver or afterwards, as well as, a comparison of the simplex solver against the best achieved times of both strategies.

Even though the implemented solvers generally improve the performance of SMT-RAT on difference logic problems, there are a significant number of benchmarks from QF_IDL where they perform worse than the simplex solver. Most of these benchmarks stem from Averest, RTCL and post_office, which is part of the mathsat suite. Figure 6.3(a) shows that for these sets of benchmarks a large percentage of the running time is spent on splitting equalities and disequalities, up to 99% for RTCL. Furthermore, it shows a correlation between the percentage of unique constraints in the formula and the percentage of time spent splitting constraints (see also Table 6.2). This can be explained by the fact that the solver splits every occurrence of every equality and disequality in the input formula. This may cause the solver to split the same constraints multiple times, especially if only a small percentage of the constraints in the input formula are unique. We can avoid this overhead by splitting the constraints after the SAT-solver as described in the previous chapter. Figure 6.2(a) shows a comparison between splitting constraints before and after the SAT-solver using the Bellman-Ford solver on QF_IDL. While splitting constraints before the SAT-solver performs better overall, there are a significant number of benchmarks where splitting afterwards provides an improved running time. The general increase in running time for a lot of benchmarks, when splitting after the SAT-solver is caused by a number of factors. First, the solver has to keep track of all disequalities that were already passed in a prior call by the SAT-solver to determine whether a lemma has to be passed back or not. Second, the SAT-solver has to process any lemmas and adjust its passed set of constraints accordingly. This is shown in Figure 6.3(b), where the amount of time spent in the SAT-solver increases drastically compared to the time spent in the theory solver as was seen in Figure 6.3(a). Third, by splitting the constraints beforehand, the SAT-solver may determine an answer faster, given that the resulting constraints are already part of other subformulas.

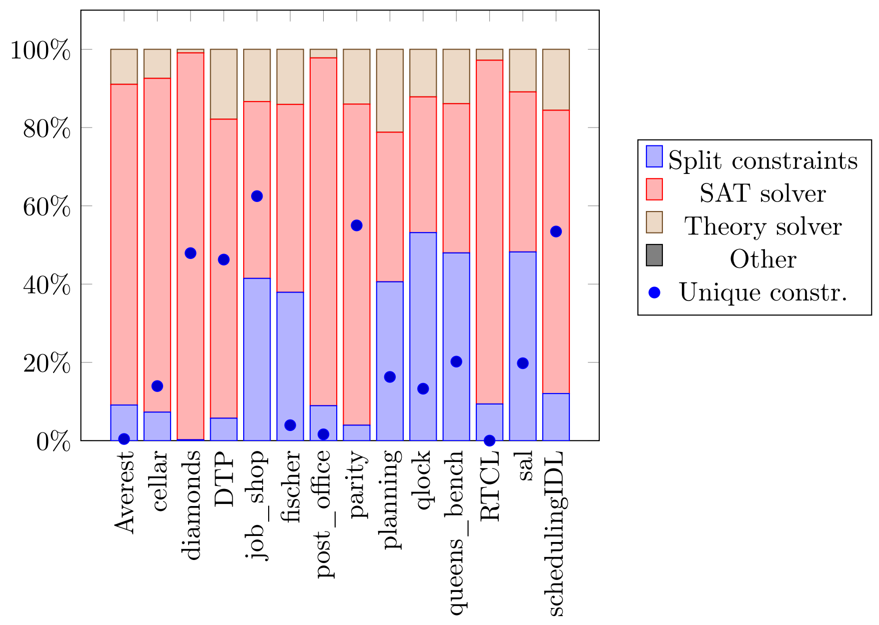
Benchmark Suite	Avg # Constr.	% unique Constr.	% Equalities	% Disequalities
Averest	271,770	0.38%	0.60%	0.00%
cellar	12,280	12.68%	1.35%	0.00%
diamonds	390	43.56%	0.00%	0.00%
DTP	840	42.06%	0.00%	0.00%
job_shop	9,040	56.80%	0.00%	13.60%
fischer	7,530	3.59%	15.47%	12.15%
post_office	6,300	1.47%	22.89%	4.59%
parity	111,290	50.00%	0.00%	0.00%
planning	6,030	14.80%	26.36%	0.00%
qlock	19,630	12.06%	30.98%	0.11%
queens_bench	18,960	18.36%	0.00%	32.97%
RTCL	2,379,070	<0.01%	13.39%	13.74%
sal	990	17.98%	28.24%	0.00%
schedulingIDL	14,710	48.58%	0.00%	0.00%
sep	450	28.05%	13.21%	0.00%

Table 6.2: Average number of constraints and the percentages of unique constraints, as well as, equalities and disequalities for QF_IDL

Another thing to note is that both graph-based solvers perform much better than the simplex solver on the incremental DFT problems compared to the non-incremental problems, even though the encoding and general structure of the problems stay the same. Furthermore, notice that both implemented solvers, as well as the simplex solver, return wrong answers for some of the incremental DFT problems (Table 6.1). This is caused by the `benchmax` tool because when executed directly on the incremental DFT problems, all solvers return the correct intermediate and final results. Therefore, the achieved runtimes should still provide correct information on the performance of the solvers.
As mentioned in the last chapter, SMT-RAT already comes with a number of implemented preprocessing modules. Figure 6.4 shows a performance comparison of the Bellman-Ford solver with additional preprocessing in the form of symmetry breaking and equality substitution and without. While using additional preprocessing generally increases the running time on the `QF_RDL` and non-incremental DFT problems, it proves to be beneficial for a large number of benchmarks from the other two sets.
Finally, Figure 6.5 shows that the percentage of the total running time spent with solving SMT differs greatly between the benchmark suites from SMT-LIB, making up less than one percent for `post_office`.

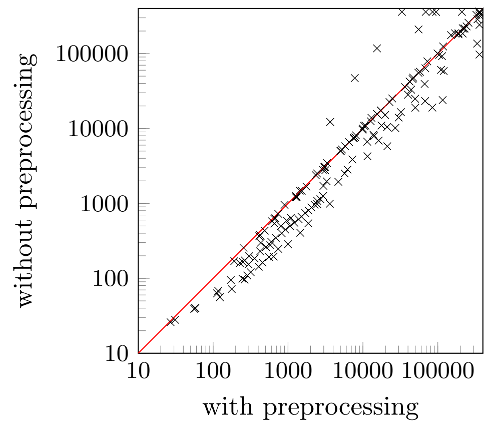


(a) Splitting before the SAT-solver

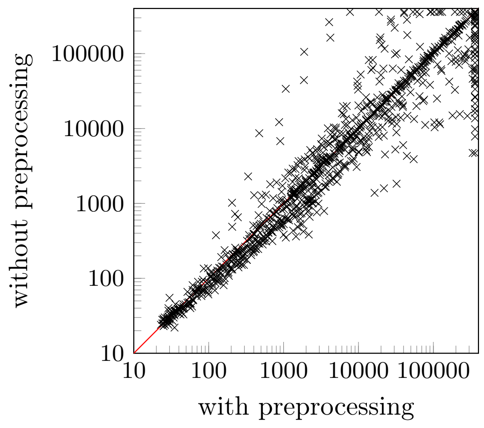


(b) Splitting after the SAT-solver

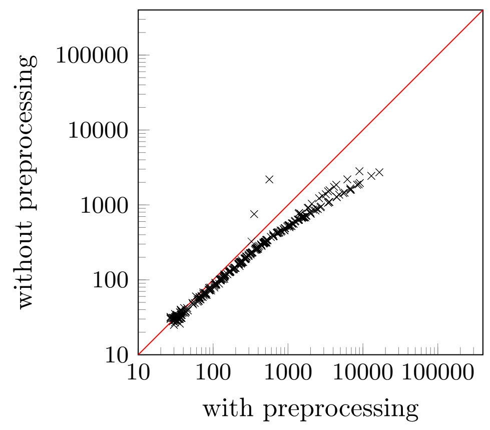
Figure 6.3: Percentage of time spent in the different phases of SMT-solving for `QF_IDL` using the Bellman-Ford solver with different splitting tactics. Additionally the percentage of unique constraints in the input formula is displayed.



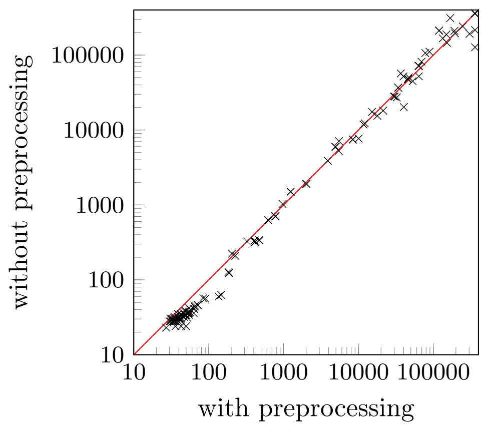
(a) QF_IDL



(b) QF_IDL

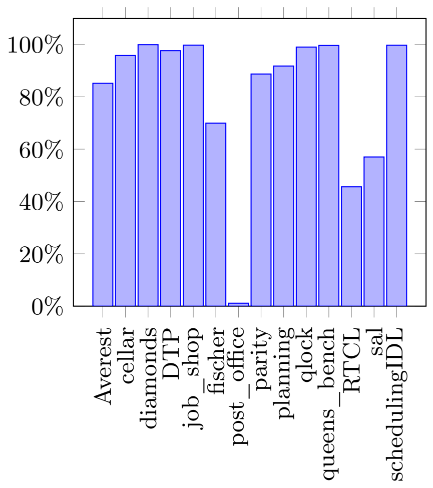


(c) DFT

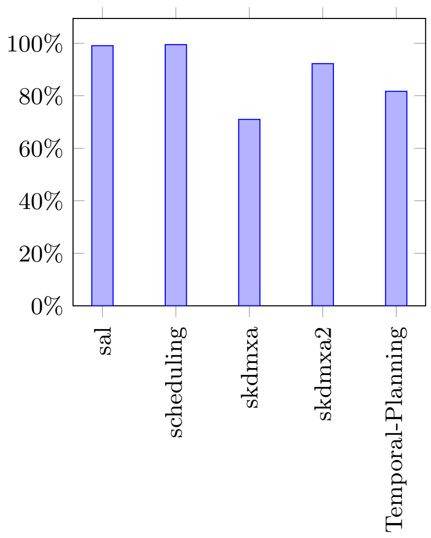


(d) incremental DFT

Figure 6.4: Performance comparison of the Bellman-Ford solver with additional preprocessing and without



(a) `QF_IDL`



(b) `QF_RDL`

Figure 6.5: Percentage of the total running time spent on SMT-solving. Only benchmarks with a runtime over one second were considered

Chapter 7

Conclusion

The goal of this thesis was to expand the SMT-RAT framework with two distinguished theory solvers for difference logic, thus improving SMT-RAT's performance on these problems.

7.1 Summary

We began by describing the general approach to SAT and its expansion SMT, followed by a presentation of the theory of difference logic. We continued by describing how a set of difference constraints can be represented as a graph, followed by some basic definitions on shortest-paths problems.

In Chapter 3, we described the Bellman-Ford algorithm and its application in the context of SMT-solving. We then introduced the implemented incremental algorithm taken from [WIGG05] by detailing the changes made from the Bellman-Ford algorithm as well as the needed conflict resolution and backtracking procedures. In the next chapter, we introduced the Floyd-Warshall algorithm as a specialised shortest-paths algorithm for problems resulting in dense constraint graphs, followed by its incremental version based on [RHK15] and [HRK17].

Following the implementation of both theory solvers, we described two implementations of the necessary procedure of splitting the equalities and disequalities in the input formula, detailing their respective advantages and disadvantages.

Finally, we compared the newly implemented solvers against the prior existing simplex solver and the state-of-the-art Z3 solver. Based on the obtained results, we examined the percentage of the total running time spent in the different modules for the difference logic solver using the incremental Bellman-Ford algorithm, and compared the influence of both implemented splitting procedures on the performance.

7.2 Discussion and Future Work

The experimental results show that the newly implemented solvers provide a substantial improvement to SMT-RAT's performance on difference logic formulas. However, Z3 still vastly outperforms SMT-RAT, especially for the `QF_IDL` set of benchmarks. The more detailed analysis of SMT-RAT's running time on `QF_IDL` revealed that for a large number of benchmarks only a small percentage of the total running time is spent in the theory solver. For these benchmarks, the majority of the running time is spent in the SAT-solver and in the constraint-splitting module. Moving the splitting operations behind the SAT-solver improved the performance on some of the benchmarks, even though the overall running time increased. A solution for that, would be to include the splitting of equalities and disequalities as an extra setting into the SAT-solver, and therefore removing the need to perform a large number of back-and-forth calls between the SAT-solver and the splitting module.

The amount of time spent in the SAT-solver can be reduced by providing additional information through advanced theory propagation. Currently both implemented theory solvers do not try to derive any additional information from their computations, returning only a model or infeasible subset when called. The returned infeasible subset is not necessarily minimal because the algorithms do not return the shortest negative cycle but the cycle with the highest negative weight. The solvers could use the last added edge as a starting point for a search for additional negative cycles, with the length of the originally found cycle serving as an upper bound. A naive version of this procedure using a breadth-first search was included temporarily in the Bellman-Ford solver but it caused a large overhead in both time and space. However, with further development, this procedure could improve the performance of SMT-RAT on difference logic problems.

Furthermore, the experimental results show that SMT-RAT's existing preprocessing modules already improve the running time for a large number of benchmarks. Specialized preprocessing procedures could provide further improvements to the performance of SMT-RAT on difference logic formulas.

Finally, the results show that for a considerable number of benchmarks, a large percentage of SMT-RAT's running time is not spent with SMT-solving. While this overhead is likely caused by SMT-RAT's parser, further analysis is needed to determine the exact cause.

In any case, the newly implemented solvers should provide a solid basis for further development.

Bibliography

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BHvMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, 2009.

[CKJ$^+$15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 360–368, Cham, 2015. Springer International Publishing.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, USA, 1971. ACM.

[DFMWP11] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 222–236, Berlin, Heidelberg, 2011. Springer.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[HRK17] Jacob M. Howe, Ed Robbins, and Andy King. Theory learning with symmetry breaking. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, pages 85–96, New York, USA, 2017. ACM.

[KCJ$^+$] Gereon Kremer, Florian Corzilius, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. Carl: Computer arithmetic and logic library. `www.smtrat.github.io/carl`.

[NMA$^+$02] Peter Niebert, Moez Mahfoudh, Eugene Asarin, Marius Bozga, Oded Maler, and Navendu Jain. Verification of timed automata via satisfiability checking. In Werner Damm and Ernst Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 225–243, Berlin, Heidelberg, 2002. Springer.

[RHK15] Ed Robbins, Jacob M. Howe, and Andy King. Theory propagation and reification. *Science of Computer Programming*, 111:3 – 22, 2015. Special Issue on Principles and Practice of Declarative Programming (PPDP 2013).

[Tse68] Grigori Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

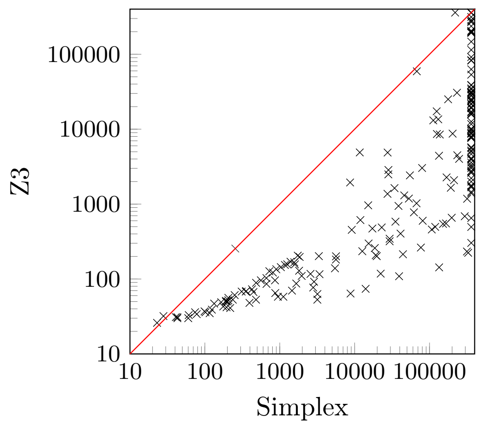
[VJK18] Matthias Volk, Sebastian Junges, and Joost-Pieter Katoen. Fast dynamic fault tree analysis by model checking techniques. *IEEE Transactions on Industrial Informatics*, 14(1):370–379, January 2018.

[WIGG05] Chao Wang, Franjo Ivančić, Malay Ganai, and Aarti Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 322–336, Berlin, Heidelberg, 2005. Springer.

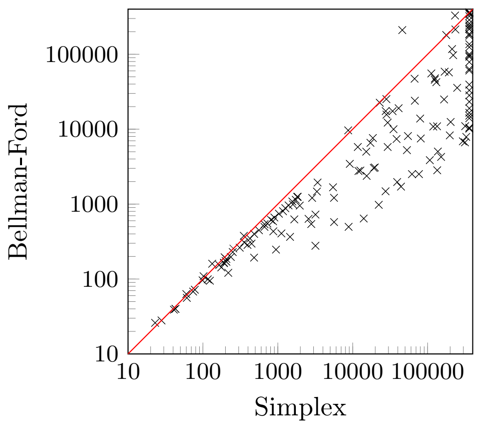
Appendix A

Runtimes

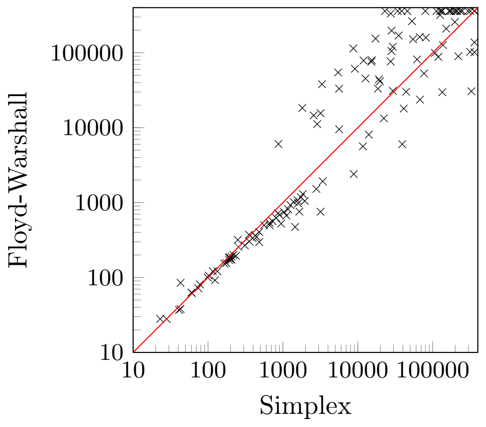
A.1 QF_RDL



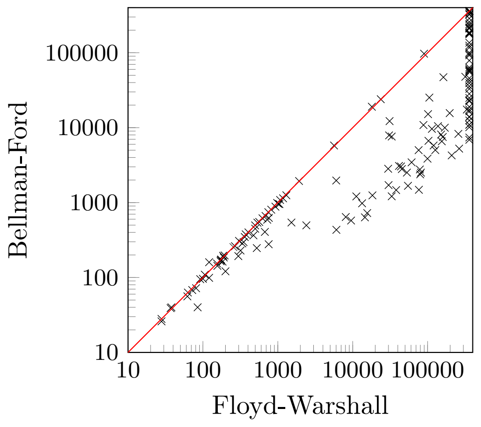
(a) Z3 vs. simplex



(b) Simplex vs. Bellman-Ford



(c) Simplex vs. Floyd-Warshall



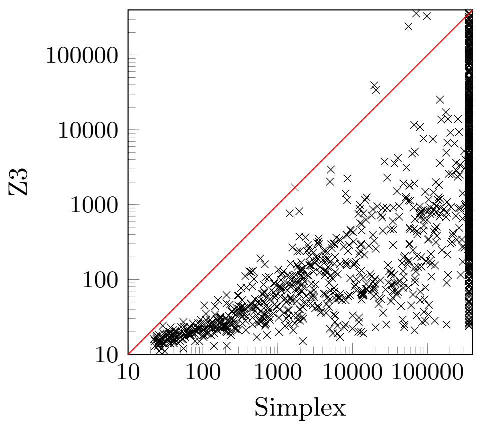
(d) Floyd-Warshall vs. Bellman-Ford

Figure A.1: Runtime comparison for `QF_RDL`

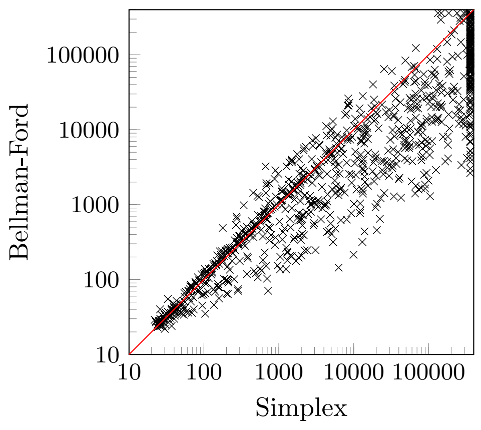
Solver	Benchmarks	Runtime	SAT	UNSAT	Solved
Bellman-Ford	SMT-Temporal-Planning	5m41.616s	43	0	84.31%
	check	0m0.054s	1	1	100.00%
	sal	33m53.224s	0	50	83.33%
	scheduling	46m59.025s	31	26	53.77%
	skdmxa	0m24.934s	0	1	25.00%
	skdmxa2	9m15.956s	0	3	9.38%
	total	**96m14.809s**	**75**	**81**	**61.18%**
Floyd-Warshall	SMT-Temporal-Planning	5m50.199s	42	0	82.35%
	check	0m0.056s	1	1	100.00%
	sal	18m6.46s	0	30	50.00%
	scheduling	52m1.496s	16	18	32.08%
	skdmxa	0m0.0s	0	0	0.00%
	skdmxa2	0m0.0s	0	0	0.00%
	total	**75m58.211s**	**59**	**49**	**42.35%**
Simplex	SMT-Temporal-Planning	11m16.264s	43	0	84.31%
	check	0m0.051s	1	1	100.00%
	sal	34m21.282s	0	30	50.00%
	scheduling	49m12.33s	27	21	45.28%
	skdmxa	2m46.0s	0	1	25.00%
	skdmxa2	3m34.727s	0	2	6.25%
	total	**101m10.654s**	**71**	**55**	**49.41%**
Z3	SMT-Temporal-Planning	2m21.387s	47	0	92.16%
	check	0m0.058s	1	1	100.00%
	sal	15m39.961s	0	59	98.33%
	scheduling	23m40.351s	36	28	60.38%
	skdmxa	3m58.355s	0	4	100.00%
	skdmxa2	23m12.162s	16	16	100.00%
	total	**68m52.274s**	**100**	**108**	**81.57%**

Table A.1: `QF_RDL`

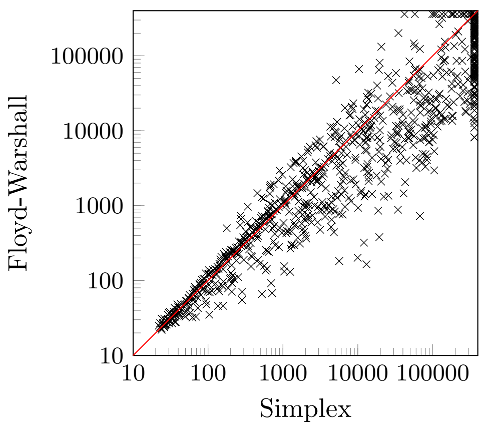
A.2 QF_IDL



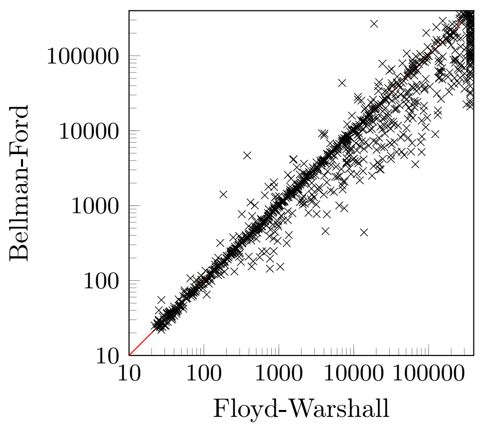
(a) Z3 vs. simplex



(b) Simplex vs. Bellman-Ford



(c) Simplex vs. Floyd-Warshall



(d) Floyd-Warshall vs. Bellman-Ford

Figure A.2: Runtime comparison for `QF_IDL`

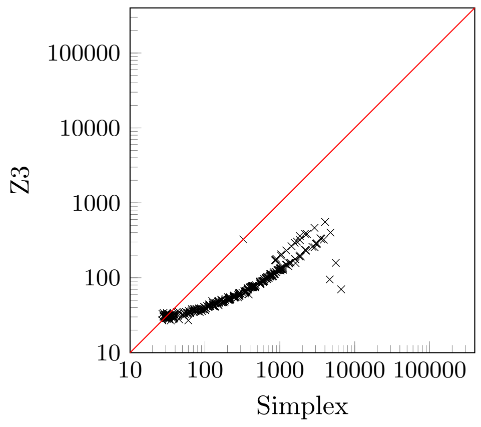
Solver	Benchmarks	Runtime	SAT	UNSAT	Solved
Bellman-Ford	Averest	75m26.251s	149	94	96.43%
	DTP	3m13.634s	32	28	100.00%
	RTCL	4m28.293s	2	4	18.18%
	bcnscheduling	0m0.0s	0	0	0.00%
	cellar	1m40.046s	0	9	69.23%
	check	0m0.049s	0	2	100.00%
	diamonds	17m21.997s	0	15	41.67%
	fuzzy-matrix	0m0.0s	0	0	0.00%
	job_shop	20m34.83s	30	8	31.67%
	mathsat	28m3.041s	12	118	89.04%
	parity	125m14.146s	55	68	49.60%
	planning	4m43.548s	2	43	100.00%
	qlock	28m41.977s	12	13	34.72%
	queens_bench	58m36.134s	66	17	28.04%
	sal	0m46.676s	10	40	100.00%
	schedulingIDL	134m57.819s	60	20	28.57%
	sep	0m2.186s	9	8	100.00%
	total	**503m50.627s**	**439**	**487**	**54.53%**
Floyd-Warshall	Averest	74m50.172s	150	94	96.83%
	DTP	5m38.047s	32	28	100.00%
	RTCL	4m28.824s	2	4	18.18%
	bcnscheduling	0m0.0s	0	0	0.00%
	cellar	4m29.771s	0	9	69.23%
	check	0m0.046s	0	2	100.00%
	diamonds	17m28.396s	0	15	41.67%
	fuzzy-matrix	0m0.0s	0	0	0.00%
	job_shop	27m6.218s	31	8	32.50%
	mathsat	31m37.119s	13	116	88.36%
	parity	99m23.62s	48	50	39.52%
	planning	22m19.139s	2	43	100.00%
	qlock	18m5.891s	7	9	22.22%
	queens_bench	70m47.65s	62	16	26.35%
	sal	1m3.091s	10	40	100.00%
	schedulingIDL	117m26.52s	44	11	19.64%
	sep	0m2.311s	9	8	100.00%
	total	**494m46.815s**	**410**	**453**	**50.82%**
Simplex	Averest	45m26.412s	122	94	85.71%
	DTP	31m46.773s	32	28	100.00%
	RTCL	2m29.003s	2	4	18.18%
	bcnscheduling	0m0.0s	0	0	0.00%
	cellar	2m49.688s	0	9	69.23%
	check	0m0.05s	0	2	100.00%
	diamonds	18m27.449s	0	15	41.67%
	fuzzy-matrix	0m0.0s	0	0	0.00%
	job_shop	15m21.142s	21	8	24.17%
	mathsat	52m27.167s	12	119	89.73%
	parity	96m50.793s	34	47	32.66%
	planning	30m0.953s	0	32	71.11%
	qlock	6m51.387s	2	4	8.33%
	queens_bench	35m1.909s	39	16	18.58%
	sal	4m10.598s	10	40	100.00%
	schedulingIDL	86m59.13s	30	6	12.86%
	sep	0m3.498s	9	8	100.00%
	total	**428m45.952s**	**313**	**432**	**43.88%**

Table A.2: `QF_IDL`

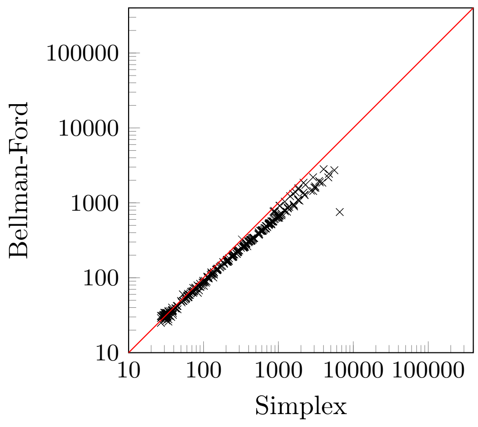
Solver	Benchmarks	Runtime	SAT	UNSAT	Solved
Z3	Averest	0m36.512s	157	95	100.00%
	DTP	0m5.711s	32	28	100.00%
	RTCL	0m2.533s	4	29	100.00%
	bcnscheduling	3m23.917s	4	3	53.85%
	cellar	0m7.455s	0	13	100.00%
	check	0m0.034s	0	2	100.00%
	diamonds	13m5.946s	0	34	94.44%
	fuzzy-matrix	0m0.0s	0	0	0.00%
	job_shop	24m58.555s	72	11	69.17%
	mathsat	14m10.065s	16	127	97.95%
	parity	178m26.919s	112	93	82.66%
	planning	0m37.096s	2	43	100.00%
	qlock	15m41.025s	36	20	77.78%
	queens_bench	14m1.743s	199	20	73.99%
	sal	0m9.177s	10	40	100.00%
	schedulingIDL	56m9.976s	246	31	98.93%
	sep	0m0.563s	9	8	100.00%
	total	**321m37.227s**	**899**	**597**	**88.10%**

Table A.3: `QF_IDL` continued

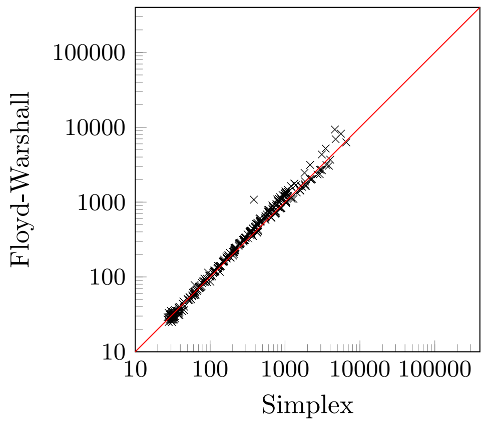
A.3 DFT



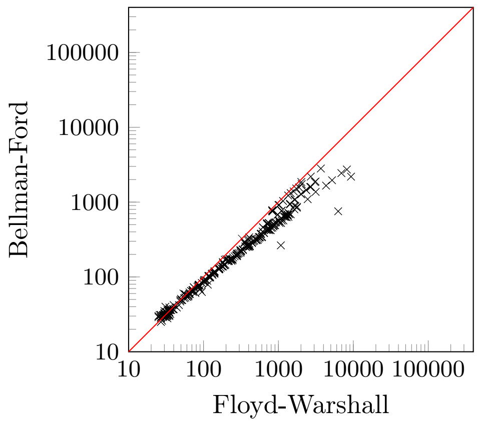
(a) Z3 vs. simplex



(b) Simplex vs. Bellman-Ford



(c) Simplex vs. Floyd-Warshall



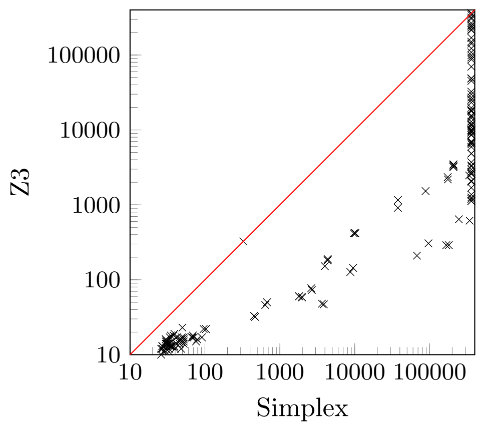
(d) Simplex vs. Floyd-Warshall

Figure A.3: Runtime comparison for DFT problems.

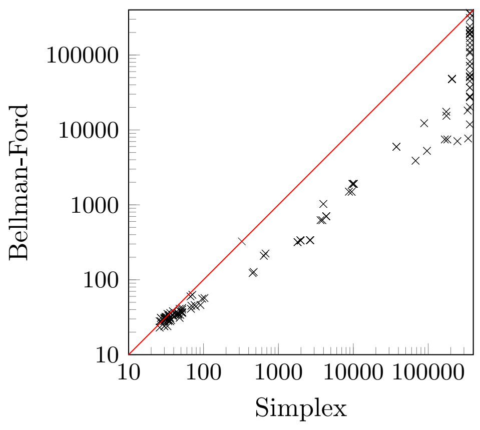
Solver	Benchmarks	Runtime	SAT	UNSAT	Solved
Bellman-Ford	hecs	0m38.458s	0	72	100.00%
	iso26262	0m21.537s	0	16	100.00%
	mcs	0m27.333s	0	96	100.00%
	rc	0m25.24s	0	68	100.00%
	sap	0m0.176s	1	3	100.00%
	toy	0m5.659s	9	60	100.00%
	total	**1m58.403s**	**10**	**315**	**100.00%**
Floyd-Warshall	hecs	1m0.023s	0	72	100.00%
	iso26262	0m24.987s	0	16	100.00%
	mcs	0m45.551s	0	96	100.00%
	rc	0m53.887s	0	68	100.00%
	sap	0m0.176s	1	3	100.00%
	toy	0m19.209s	9	60	100.00%
	total	**3m23.833s**	**10**	**315**	**100.00%**
Simplex	hecs	1m2.553s	0	72	100.00%
	iso26262	0m26.586s	0	16	100.00%
	mcs	0m37.681s	0	96	100.00%
	rc	0m40.268s	0	68	100.00%
	sap	0m0.182s	1	3	100.00%
	toy	0m14.206s	9	60	100.00%
	total	**3m1.476s**	**10**	**315**	**100.00%**
Z3	hecs	0m7.938s	0	72	100.00%
	iso26262	0m4.659s	0	16	100.00%
	mcs	0m6.722s	0	96	100.00%
	rc	0m5.366s	0	68	100.00%
	sap	0m0.128s	1	3	100.00%
	toy	0m2.315s	9	60	100.00%
	total	**0m27.128s**	**10**	**315**	**100.00%**

Table A.4: DFT problems

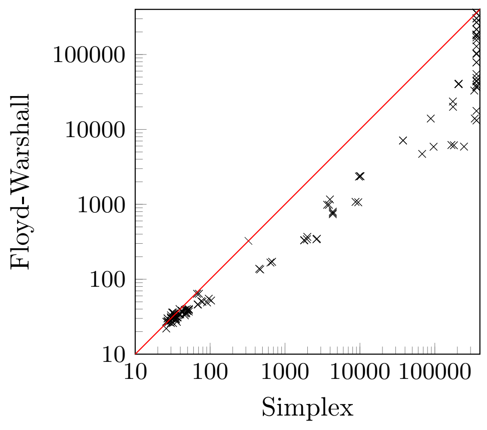
A.4 DFT Incremental



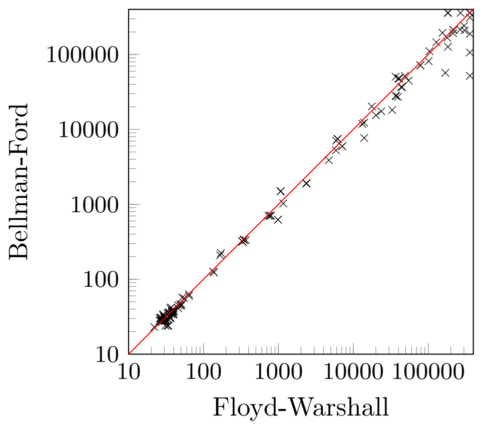
(a) Z3 vs. simplex



(b) Simplex vs. Bellman-Ford



(c) Simplex vs. Floyd-Warshall



(d) Simplex vs. Floyd-Warshall

Figure A.4: Runtime comparison for incremental DFT problems.

Solver	Benchmarks	Runtime	SAT	UNSAT	Solved
Bellman-Ford	hecs	6m31.808s	1	6	9.72%
	iso26262	0m0.0s	0	0	0.00%
	mcs	45m16.421s	5	34	40.62%
	rc	6m28.61s	0	11	16.18%
	sap	0m0.18s	0	4	100.00%
	toy	2m10.936s	5	60	94.20%
	total	**60m27.955s**	**11**	**115**	**38.77%**
Floyd-Warshall	hecs	12m10.789s	2	6	11.11%
	iso26262	0m0.0s	0	0	0.00%
	mcs	43m10.22s	4	32	37.50%
	rc	11m26.291s	2	11	19.12%
	sap	0m0.187s	0	4	100.00%
	toy	0m31.993s	5	59	92.75%
	total	**67m19.48s**	**13**	**112**	**38.46%**
Simplex	hecs	3m9.94s	1	3	5.56%
	iso26262	0m0.0s	0	0	0.00%
	mcs	27m10.091s	0	15	15.62%
	rc	14m4.958s	0	10	14.71%
	sap	0m0.206s	0	4	100.00%
	toy	3m22.896s	5	59	92.75%
	total	**47m48.091s**	**6**	**91**	**29.85%**
Z3	hecs	24m13.276s	0	20	27.78%
	iso26262	0m0.0s	0	0	0.00%
	mcs	21m28.307s	0	62	64.58%
	rc	25m39.406s	0	17	25.00%
	sap	0m0.061s	0	4	100.00%
	toy	1m15.023s	0	66	95.65%
	total	**72m36.073s**	**0**	**169**	**52.00%**

Table A.5: Incremental DFT problems