Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

UTILIZING HEURISTICS AND DATA STRUCTURES TO OPTIMIZE INCREMENTAL FMPLEX

Tim Luca Reimers

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Christina Büsing

Additional Advisor:
Valentin Promies

Aachen, 20th March 2023

Abstract

Satisfiability Modulo Theories (SMT) solving is utilized for determining the satisfiability of *quantifier-free first order logic* formulas over some theory. A method of solving the theory of *linear real arithmetic* (LRA) is given by the *FMplex* method. This method progressively eliminates variables by combining constraints with assumed greatest lower bounds or smallest upper bounds. In recent works the method has been extended to support incrementality. However, surprisingly testing has shown that enabling incrementality does not yield a benefit in performance.

Based on this we present ideas on how to optimize FMplex in general as well as its incremental adaptation. For this we look at various aspects covering the treatment of equalities, data structures and infeasible subsets. Furthermore, we discuss how to change the order in which variables are eliminated in an incremental setting. Testing reveals that the incremental version actually outperforms the non-incremental version and that the method profits from smaller infeasible subsets.

Acknowledgements

I would like to express my gratitude towards Prof. Dr. Erika Ábrahám and my advisor Valentin Promies. Their support for the theoretical as well as the technical parts is what enabled me to write this thesis. Furthermore, I would like to thank Prof. Dr. Christina Büsing for joining this thesis as a second examiner.

Lastly, I would also like to thank my family for providing their support during my Bachelor studies.

Contents

Chapter 1

Introduction

Determining the satisfiability of constraints in the form of equalities and inequalities has become an important topic in a lot of different research areas. The problems of the real word can often be abstracted to mathematical formulas, whose satisfiability can then be confirmed or disproved. For example, this can be used in program verification [SG09], circuit validation [LV11] or even the safety analysis of neural networks [KBD+17]. The study of *Satisfiability Modulo Theories (SMT)* solvers concerns itself with algorithms suited to check for the satisfiability of problems expressed in the existential fragment of first order logic over some theories.

To solve such formulas, most SMT solvers use a SAT solver and theory solver. The SAT solver tries to find a set of constraints that satisfies the Boolean structure of the formula. Meanwhile, the theory solver checks, whether this set of constraints is satisfiable in the chosen theory. Should the theory solver determine that this set is unsatisfiable, the SAT solver tries to find another set that satisfies the Boolean structure.

The focus of this work are theory solvers, in particular over the theory of *linear real arithmetic*, for checking the satisfiability of sets of constraints which are linear combinations of real-valued variables compared to rational constants. One of the first algorithms capable of deciding the satisfiability of such constraint sets, is the *Fourier-Motzkin variable elimination* method, which was first developed in the 1820s by Fourier and later rediscovered by Motzkin [Fou24, Mot36]. It aims to eliminate each variable in succession by transforming the constraints into lower and upper bounds respective to a variable and then combining the bounds to eliminate the variable. However, due to the doubly exponential runtime of this approach its practical relevance is restricted. Instead, a popular approach is provided in the *simplex* algorithm, which has an exponential runtime but works quite well in practice [Dan90].

The ideas of both of these approaches have been combined into the *FMplex* method with a singly exponential worst case runtime [Kob21]. This is achieved using a similar approach as the Fourier-Motzkin variable elimination method but instead of combining every upper and lower bound, an assumed greatest lower bound or smallest upper bound, a so called eliminator, is chosen to minimize the number of resulting constraints. To make the FMplex method more efficient, an incremental adaptation was developed [Ste22], which tries to minimize the number of redundant computations by using information gained from checking the satisfiability of previous constraint sets. Furthermore, heuristics for choosing variables and eliminators were added. Sur-

prisingly, experimental evaluation revealed that the incremental version of FMplex performs worse (solves less instances in reasonable time) than the non-incremental version. This thesis aims to investigate the reason behind this difference in performance and to make further optimizations.

To begin with, in Chapter 2 we establish needed preliminaries. In particular, we introduce satisfiability checking as well as a definition of linear real arithmetic. Furthermore, we give a small rundown of the Fourier-Motzkin variable elimination, before we explain the incremental FMplex method. Then, in Chapter 3 a novel adaptation of the FMplex method is presented, which aims to improve the performance of the incremental FMplex by putting more focus on the heuristics and introducing new data structures. Additionally, a more efficient treatment of equations, an extension to not-equal constraints and a change in the generation of infeasible subsets is proposed. Afterwards, in Chapter 4 an implementation of this variant is discussed and in Chapter 5 the thesis is concluded.

Chapter 2

Preliminaries

2.1 Satisfiability Checking

One of the most famous problems in computer science is the *Boolean satisfiability problem (SAT)*, where the satisfiability of a formula in *propositional logic* is examined. In fact, in the 1970s the SAT problem was the first to be discovered to be NP-complete [Coo71, Lev73]. However, propositional logic is not very versatile.

Therefore, it is often more practical to encode problems in quantifier-free first order logic formulas over some theory. Such formulas can be solved using an SMT solver. Many SMT solvers consist of a SAT solver and a theory solver for the relevant theory. This approach is also called *DPLL(T)*, where T stands for some theory [GHN$^+$04]. The problem is first transformed into a Boolean abstraction, where all constraints over some theory are substituted by Boolean variables. Then, the SAT solver constructs either a partial or full assignment of truth values to the Boolean variables in such a way that the Boolean abstraction is satisfied.

In this approach the SAT solving algorithm is the *Davis-Putnam-Logemann--Loveland (DPLL)* algorithm [DP60, DLL62]. It works in three steps which are repeatedly done, until a satisfying assignment is found or unsatisfiability can be concluded: *decide, propagate* and *backtrack*. In the *decide* step an unassigned variable is chosen and assigned a truth value, with which the current partial assignment is extended. Then, in the *propagate* step additional truth values are assigned to variables based on the current partial assignment according to some deduction rules. Should the algorithm run into a situation where the partial assignment cannot be extended in such a way that the formula stays satisfiable, the *backtrack* step is executed. We call this situation a *conflict*. This means that the algorithm backtracks and reverses a decision made in prior steps as this assignment cannot satisfy the formula.

When the SAT solver has constructed a partial or full assignment, we still need to check if the theory constraints, whose abstracting Boolean variables are true and the negation of those whose abstraction variables are false, are actually satisfiable. To do this, the SAT solver passes all those theory constraints to the theory solver. The theory solver checks, whether the conjunction of these constraints is satisfiable. In case that these constraints are unsatisfiable, the theory solver generates a preferably minimal subset of constraints that is already unsatisfiable and passes it back to the SAT solver. Then, the SAT solver extends the formula with a clause that is the negation of the conjunction of the Boolean variables corresponding to constraints that

were passed from the theory solver. Through this the SAT solver needs to backtrack as the current assignment cannot satisfy that clause.

There are two important approaches to SMT solving, that is *full lazy* and *less lazy* SMT solving [ÁK17]. In full lazy SMT solving the SAT solver generates a full assignment, which already satisfies the Boolean abstraction, and only then the theory solver is consulted. Meanwhile, in the less lazy approach the SAT solver consults the theory solver after every *decide* and *propagate* step, if there is no conflict. Therefore, possible conflicts between constraints may be revealed earlier and the SAT solver can already backtrack.

In the less lazy scenario it is useful, if the theory solver can work *incrementally*. This means that the theory solver can make use of the previous satisfiability check for the check of the extended set passed by the SAT solver. Therein lies the fact that motivated the development of an incremental adaptation of FMplex.

2.2 Quantifier-Free Linear Real Arithmetic

The FMplex method is a theory solver for *linear real arithmetic* (LRA).

Definition 2.2.1 (Linear Constraints)**.** *A* linear constraint *has the form*

$$a_1x_1 + \cdots + a_nx_n \bowtie b$$

where $x_1, \ldots, x_n$ *are real-valued variables,* $a_1, \ldots, a_n, b \in \mathbb{Q}$ *and* $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$

Definition 2.2.2 (Trivial Constraints)**.** *We call a linear constraint* trivial*, if it has the form:*

$$0 \leq b$$

with some $b \in \mathbb{Q}$*. If* b *is negative, then we call it trivially false. Otherwise we call it trivially true.*

Definition 2.2.3 (QFLRA formula)**.** *A* quantifier-free linear real arithmetic *(QF-LRA) formula is a Boolean combination of linear constraints evaluated over the reals:*

$$\varphi := c \mid \neg\varphi \mid \varphi \wedge \varphi$$

Note that since we have $\neg$ and $\wedge$, other logical operators such as $\vee$, $\rightarrow$ or $\oplus$ can be used as syntactical sugar. For the explanation of Fourier-Motzkin variable elimination as well as FMplex we will for now assume that equalities and not-equal constraints are transformed into inequalities in the following way:

$$a_1x_1 + \cdots + a_nx_n = b \Leftrightarrow (a_1x_1 + \cdots + a_nx_n \leq b) \wedge (a_1x_1 + \cdots + a_nx_n \geq b)$$

$$a_1x_1 + \cdots + a_nx_n \neq b \Leftrightarrow (a_1x_1 + \cdots + a_nx_n < b) \vee (a_1x_1 + \cdots + a_nx_n > b)$$

Since this approach leads to a blowup in the number of constraints, we will discuss another treatment of equalities and not-equal constraints in Section 3.3 and Section 3.4.

Furthermore, in the following we will assume that all inequalities will be using an operator $\bowtie \in \{<, \leq\}$, as the inequalities with operators $\{>, \geq\}$ are equivalent to the former when multiplying both sides of the inequality by -1. For now we will only look at non-strict inequalities and will later discuss how FMplex can be extended to strict inequalities.

Example 2.2.1 (QFLRA formula)**.** *A syntactically correct QFLRA formula φ is given through*

$$\varphi := (x - y \leq 4) \wedge (-x - 2y \leq 6) \wedge (2x + y \leq -1) \wedge (-3x + 4y \leq 4)$$

Now the role of the SMT solver is to decide whether there is an assignment that satisfies such a formula.

Definition 2.2.4 (Assignment)**.** *Let $\mathcal{X} := \{x_1, \ldots, x_n\}$ be the set of variables in a QFLRA formula φ. Then, we call a function $\mathcal{V} : \mathcal{X} \to \mathbb{R}$ an assignment. A partial assignment refers to functions $\mathcal{V} : \mathcal{X}' \to \mathbb{R}$ with $\mathcal{X}' \subset \mathcal{X}$, so a mapping that does not necessarily cover all variables.*

Let φ be a formula and $\mathcal{V}$ be an assignment. We can evaluate φ under the assignment $\mathcal{V}$ by replacing every variable in φ by the value assigned from $\mathcal{V}$. In case $\mathcal{V}$ is only a partial assignment, the evaluation results again in a formula, although with fewer variables. When every variable in φ is substituted, we obtain a formula with only trivial constraints, from which the truth value can easily be deduced.

If a formula evaluates to true under an assignment, we call that assignment a model of the formula. We call a formula satisfiable, if there exists such a model. Otherwise we call it unsatisfiable. Finding a model of a formula corresponds to finding a solution to the *system of linear inequalities* presented by the constraints.

Definition 2.2.5 (System of Linear Inequalities)**.** *A* system of linear inequalities *corresponding to a set of constraints $\mathcal{C} = \{c_1 \ldots c_m\}$ with variables $\mathcal{X} = \{x_1, \ldots x_n\}$ can be represented using matrices with each constraint c_i being $a_{i1}x_1 + \cdots + a_{in}x_n \leq b_i$:*

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} = b$$

The system can then be referred to as (A,b).

We call such a system *solvable*, if the conjunction of the constraints is satisfiable.

Example 2.2.2 (System of Linear Inequalities)**.** *Looking again at Example 2.2.1 we can now view the formula as the following system of linear inequalities:*

$$\begin{pmatrix} 1 & -1 \\ -1 & -2 \\ 2 & 1 \\ -3 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 4 \\ 6 \\ -1 \\ 4 \end{pmatrix}$$

This system of constraints is visualized in Figure 2.1. The polyhedron where the solution spaces of all constraints overlap encompasses all models of the formula.

Definition 2.2.6 (Upper and Lower Bounds)**.** *Let $c : (a_1x_1 + \cdots + a_nx_n \leq b)$ be a linear constraint with variables $\mathcal{X} = \{x_1, \ldots, x_n\}$. We say that c is an* upper bound *with respect to variable x_j, if the coefficient $a_j > 0$. Analogously, we say that c is a* lower bound *with respect to a variable x_j, if the coefficient $a_j < 0$. If the coefficient $a_j = 0$, we say that c is* non-bound *with respect to x_j.*

That means that for every variable in a formula we can partition the constraints of the formula to a set of upper bounds U, a set of lower bounds L and a set of non-bounds N. For convenience we will write upper bounds $u \in U$ as $u_1x_1 + \cdots + u_nx_n \leq b_u$ and lower bounds $l \in L$ as $l_1x_1 + \cdots + l_nx_n \leq b_l$.

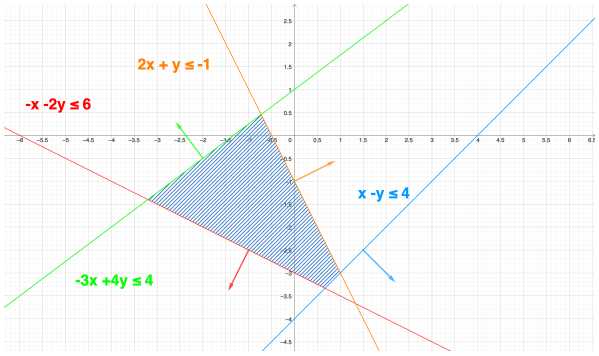


Figure 2.1: System of linear constraints from Example 2.2.2. The inequalities are true in the halfspace opposite of their normal vector. The solutions to the formula form the shaded area.

2.3 Fourier-Motzkin Variable Elimination

The Fourier-Motzkin variable elimination method is one of the earliest methods for solving systems of linear inequalities, created by Fourier and later rediscovered by Motzkin [Fou24, Mot36]. It solves the system (A,b) by iteratively eliminating variables and thereby reducing the system until it arrives at a linear inequality system (A',b'), which has no variables left. That means that only trivial constraints are left. The truth value of those can easily be determined, as the constraints are only composed of constant terms, and therefore the inequality can be checked for correctness.

The variables are removed one after another, based on an ordering of the variables, e.g. $x_1 < \cdots < x_n$. When removing the variable x_k, the inequalities are partitioned into upper bounds, lower bounds and non-bounds according to the Definition 2.2.6. The lower bounds $l : (l_1x_1 + \cdots + l_nx_n \leq b_l)$ can be rewritten as

$$x_k \geq \underbrace{\frac{1}{l_k}\left(b_l - \left(\sum_{j=1,j\neq k}^{n} l_j x_j\right)\right)}_{lb(l)}$$

and similarly the upper bounds $u : (u_1x_1 + \cdots + u_nx_n \leq b_u)$ can be rewritten as

$$x_k \leq \underbrace{\frac{1}{u_k}\left(b_u - \left(\sum_{j=1,j\neq k}^{n} u_j x_j\right)\right)}_{ub(u)}.$$

Through this a new constraint

$$lb(l) \leq ub(u)$$

can be created. This constraint corresponds to a linear combination of the constraints

l and u with scalars $\frac{1}{|l_k|}$ and $\frac{1}{|u_k|}$:

$$\begin{aligned}
&\frac{1}{|l_k|} \cdot \left(\sum_{j=1}^{n} l_j x_j \right) + \frac{1}{|u_k|} \cdot \left(\sum_{j=1}^{n} u_j x_j \right) \leq \frac{1}{|l_k|} b_l + \frac{1}{|u_k|} b_u \\
\Leftrightarrow &\frac{1}{|l_k|} \cdot \left(\sum_{j=1, j\neq k}^{n} l_j x_j \right) + \frac{1}{|u_k|} \cdot \left(\sum_{j=1, j\neq k}^{n} u_j x_j \right) \leq \frac{1}{|l_k|} b_l + \frac{1}{|u_k|} b_u \\
\Leftrightarrow &\frac{1}{|l_k|} \cdot \left(-b_l + \sum_{j=1, j\neq k}^{n} l_j x_j \right) \leq \frac{1}{|u_k|} \cdot \left(b_u - \sum_{j=1, j\neq k}^{n} u_j x_j \right) \\
\Leftrightarrow &\frac{1}{l_k} \cdot \left(b_l - \sum_{j=1, j\neq k}^{n} l_j x_j \right) \leq \frac{1}{u_k} \cdot \left(b_u - \sum_{j=1, j\neq k}^{n} u_j x_j \right)
\end{aligned}$$

In the following we will simply write such a combination as $\frac{1}{|l_k|} l + \frac{1}{|u_k|} u$.

Let L denote the set of all lower bounds and U the set of all upper bounds with respect to x_k. Then, let $C = \{ \frac{1}{|l_k|} l + \frac{1}{|u_k|} u \mid l \in L \wedge u \in U \}$ be the set of constraints, which is created by combining every upper and lower bound. Together with the set of non-bounds N with respect to x_k we arrive at a new set of constraints $M = N \cup C$. The constraints in M do not have the variable x_k, since the combination of bounds eliminated any occurrence of x_k. Therefore, a new system of linear inequalities (A', b') with at least one variable less can be created from this set. It can be shown that this reduced system is solvable if and only if the original system is solvable [Dan72].

The Fourier-Motzkin method can furthermore be used to induce a useful theorem about the solvability of linear inequality systems, which we will utilize in the FMplex method.

Theorem 2.3.1 (Feasibility Theorem [Dan72])**.** *The linear inequality system (A, b) is solvable iff there are no non-negative weights $y_1 > 0, \ldots, y_m > 0$ such that*

$$\sum_{i=1}^{m} y_i \cdot b_i < 0 \text{ and } \sum_{i=1}^{m} y_i \cdot a_{ij} = 0 \text{ for every } j = 1, \ldots, n \tag{2.1}$$

Proof. Assume a solution $x_1, \ldots, x_n$ exists to the system (A,b) and there are non-negative weights $y_1, \ldots, y_m$ that satisfy the conditions of the Feasibility Theorem. Then, this implies

$$\sum_{j=1}^{n} (\sum_{i=1}^{m} y_i a_{ij}) \cdot x_j \leq \sum_{i=1}^{m} y_i b_i$$

which together with the conditions from (2.1) implies $0 \cdot x \leq \sum_{i=1}^{m} y_i b_i < 0$ - a contradiction.

The other direction can be shown using the Fourier-Motzkin method. Assume that the system (A,b) has no solution. Then, using the Fourier-Motzkin variable elimination we can arrive at a trivially false inequality. Since this inequality is a linear combination of the original inequalities with only non-negative scalars, we can directly derive the weights $y_1, \ldots, y_n$ that satisfy the conditions (2.1) from the theorem. □

Unfortunately, the Fourier-Motzkin method has the problem of a doubly exponential worst case runtime as the combination of upper and lower bounds can lead

to an extreme blowup in the number of constraints. To reduce the number of generated constraints a new method FMplex was developed [Kob21], which combines Fourier-Motzkin with ideas from the simplex method.

2.4 FMplex

The following section will discuss the incremental adaptation of the FMplex method as presented by Svenja Stein [Ste22].

Similarly to Fourier-Motzkin, the FMplex method eliminates each variable from a set of constraints one after the other. To do so, lower bounds and upper bounds are again combined. However, instead of combining all lower bounds with all upper bounds, either a *greatest lower bound (GLB)* or a *smallest upper bound (SUB)* is selected. In the case of the GLB that means when eliminating a variable x_k we assume that a lower bound $l \in L$ is a greatest among all lower bounds. Therefore, we only need to compare this GLB l to all upper bounds. But to ensure that the GLB is actually a greatest lower bound, we need additional constraints comparing the GLB to all other lower bounds. The original set of constraints is satisfiable, if for any GLB the resulting set of constraints is satisfiable. Therefore, reminiscent of depth-first search we try out different lower bounds as GLB until we can conclude unsatisfiability or find a solution.

In this thesis we will focus on GLBs as the SUBs can be handled analogously. But in the implementation both GLBs and SUBs are used.

2.4.1 Notation

For constraints we will use a similar notation as Stein [Ste22]. A constraint c will now be written as

$$c = \langle a_1, \ldots, a_n, b \mid d_1, \ldots, d_m \rangle.$$

This representation is composed from the first section $a_1, \ldots, a_n, b$ and the second section $d_1, \ldots, d_m$. The first section stores the coefficients and the constant term of the constraint. For simplicity we assume that constraints are now always compared to 0, with the constant part b added to the linear combination.

$$\sum_{i=1}^{n} a_i x_i + b \leq 0.$$

Linear constraints according to Definition 2.2.1 can be easily transformed to this representation.

The second section is composed from the *derivation coefficients*, which represent the constraint as a linear combination of the original constraints. Let $c_1^*, \ldots, c_m^*$ be the original constraints passed from the SAT solver with $c_i^* : \left(\sum_{j=1}^{n} a_{ij}^* x_j + b_i^* \leq 0\right)$. Then, the derivation coefficient d_i is the scalar for the original constraints c_i^*. For the original constraint c_i^* that means that all derivation coefficients are zero except for $d_i = 1$.

When combining two constraints these derivation coefficients are scaled and added just like the constraints. Therefore, for every constraint c it holds that c is equal to:

$$\sum_{i=1}^{m} d_i \left(\sum_{j=1}^{n} a_{ij}^* x_j + b_i^* \right) \leq 0.$$

2.4.2 Combining Constraints

Similarly as in the Fourier-Motzkin method we eliminate variables one after another according to some dynamic ordering on the variables. When eliminating x_k we first need to choose a GLB $l \in L$. With this l we can compute the *upper-lower combinations* just as in the Fourier-Motzkin method, by combining l with all upper bounds. In a next step we need to make sure that l is actually the GLB by computing the *same-bound combinations*. These same-bound combinations state that l must be greater or equal to all other lower bounds. However, combining two bounds of the same type can not be done by simply scaling with a positive factor as the sign of x_k is the same in both bounds. Therefore, to combine l with another lower bound it first needs to be made into an upper bound by multiplying the coefficients and the constant term of l with -1.

Due to the assumptions made by the choice of GLBs, deriving a trivial false constraint no longer necessarily implies the unsatisfiability of the formula. Until now we could conclude this unsatisfiability through Theorem 2.3.1 but by multiplying with -1 the new constraint is no longer a linear combination of the original constraints with non-negative scalars. Therefore, we now need to distinguish between *local conflicts* and *global conflicts*.

Definition 2.4.1 (Global Conflict and Local Conflict)**.**
Let $c = \langle a_1, \ldots, a_n, b \mid d_1, \ldots, d_m \rangle$ be a trivially false constraint. Then, we call c a global conflict *iff for all $i \in \{1, ..., m\}$ it holds that $d_i \geq 0$. We call c a* local conflict *iff there exists an $i \in \{1, ..., m\}$ such that $d_i < 0$.*

A global conflict still implies unsatisfiability according to Theorem 2.3.1. However, if we run into a local conflict we can no longer conclude that the original set of constraints is unsatisfiable. Instead, the conflict could also have been created by choosing the wrong GLB. Therefore, we need to backtrack and choose another GLB.

The computation of the FMplex method can also be viewed as tree, where each node represents the elimination of a variable and has a child for every possible choice of GLB. When backtracking we want to go back to the last level, where the local conflict was involved in a same-bound combination, as this is the only cause for the derivation coefficients to become negative. To keep track of that level each constraint c has an associated *conflict level cl(c)*. Initially, this value is set to 0. In an upper-lower combination the new constraint gets the maximum conflict level of both parent constraints, while in a same-bound combination the conflict level of the new constraint is set to the level of the node in the tree.

In total, a local conflict causes the algorithm to backtrack to its conflict level and to choose a new GLB. The GLB is set to $\perp$ in case we are eliminating a variable with only bounds in one direction. Then, we do not need to add any new constraints as the variable is unbounded in one direction and can be assigned a value in such a way that all inequalities are satisfied.

If all GLBs were already tried out, we instead backtrack one level higher. Should we already be at the root level and have no other GLB to pick, we can conclude the unsatisfiability of the formula, since every choice led to a conflict.

The procedure behind combining constraints is presented in Algorithm 1. In the following we will also call the constraint that was chosen as the GLB the *eliminator*. Constraints that are combined with this eliminator are referred to as *eliminees*.

Algorithm 1 FMplexCombine as in [Ste22]

Input: The set of lower bounds L, the assumed GLB c (which was removed from L), the set of upper bounds U, the variable to be eliminated x_k, the current level lvl
Output: The set of constraints resulting from the elimination of x_k with c as GLB

function FMplexCombine(L,c,U,x_k,lvl)
  if $c = \bot$ **then**
    return $\emptyset$
  else ▷ $c : (c_1x_1 + \cdots + c_nx_n + b_c \leq 0)$
    Constraint set $R = \emptyset$
    for $u : (u_1x_1 + \cdots + u_nx_n + b_u \leq 0) \in U$ **do** ▷ Upper-lower
      Constraint $c_{new} = \frac{1}{|u_k|} \cdot u + \frac{1}{|c_k|} \cdot c$
      $cl(c_{new}) \leftarrow max\{cl(c), cl(u)\}$
      $R \leftarrow R \cup \{c_{new}\}$
    end for
    for $l : (l_1x_1 + \cdots + l_nx_n + b_l \leq 0) \in L$ **do** ▷ Same-bound
      Constraint $c_{new} = \frac{1}{|l_k|} \cdot l - \frac{1}{|c_k|} \cdot c$
      $cl(c_{new}) \leftarrow lvl$
      $R \leftarrow R \cup \{c_{new}\}$
    end for
    return R
  end if
end function

2.4.3 Incrementality

Now that we looked at the combining of constraints in FMplex, we continue with the main algorithm. As said before we are in an incremental setting, where the theory solver receives a subset of the original constraints $c_1^*, \ldots, c_m^*$ over the variables $x_1, \ldots, x_n$ from the SAT solver. When the SAT Solver passes the next set of constraints, the theory solver should be able to use the generated constraints from the prior subset to reduce redundant computations. We again think of the computation as a tree structure, where each level eliminates a variable and the children are the different choices for a GLB. If we want to reuse generated constraints from previous runs, we need to remember the branch that led us to the result. To keep track of the constraints generated on all levels $1 \leq i \leq n+1$, every level possesses state variables:

- C_i saves the constraints which have so far not been considered, when combining the constraints.
- x_{k_i} refers to the variable, which is to be eliminated on this level with $k_i \in \{1, \ldots, n, \bot\}$. The variable $x_\bot$ is also just written as $\bot$ and is used when the variable is not chosen yet or when we only have trivial constraints.
- c_i refers to the chosen GLB. In case the GLB is not chosen yet or we only have either upper or lower bounds this gets the value $\bot$.
- L_i^{todo} is the set of lower bounds with respect to x_{k_i} that we have not yet tried as the GLB.
- L_i^{done} is the set of lower bounds with respect to x_{k_i} that we have already tried as the GLB and each of them has led to a local conflict.

- U_i, L_i and N_i are again the set of constraints partitioned in upper bounds, lower bounds and non-bounds for x_{k_i}.

Additionally, the variable C keeps track of all original constraints submitted by the SAT solver and the variable C^{new} tracks, which received constraints are new compared to the last call of FMplex. In case that FMplex returned UNSAT in the previous run or that the SAT solver removed one or more constraints compared to the previous run, we need to reset everything and consider all constraints in C as new again $C^{new} \leftarrow C$. Furthermore, the maximal level that was reached is remembered in the global variable $maxLvl$.

The incremental FMplex method works by iterating over the tree associated with the constraints C. We start at the level 1. When arriving on a level lvl, all new constraints are stored in C_{lvl}. The constraints in C_{lvl} were either generated by the previous level or in case of C_1 passed from the set of newly received constraints C^{new}. The first thing to do on a new level is to check, if C_{lvl} contains conflicts. Should that be the case the function `AnalyzeAndBacktrack(`C_{lvl}`)` is called to determine the conflict level. If there is a global conflict the special value 0 is returned, as to indicate that the conflict was introduced by the original constraints and the algorithm returns UNSAT. Otherwise, the method backtracks to the determined level, resets everything below that level and chooses a new GLB from L_{lvl}^{todo}.

In case C_{lvl} has no trivially false constraints, it is checked whether we reached a completely new level, in which a new variable $x_{k_{lvl}}$ needs to be chosen. If the level was already visited, the previous variable choice stays valid.

Now the not yet considered constraints from C_{lvl} are partitioned in such a way that the lower bounds with respect to $x_{k_{lvl}}$ are stored in L^{comb}, the upper bounds in U^{comb} and the non-bounds stay in C_{lvl}. These *combination sets* L^{comb} and U^{comb} contain the constraints, which have not yet been combined with the GLB. Therefore, they are the only sets passed to the FMplexCombine method. These constraints are then furthermore added to the L_{lvl}, U_{lvl} and N_{lvl} sets, which save all lower/upper/non-bounds (also those already combined with the GLB in prior iterations). Moreover, the new lower bounds in L^{comb} also need to be added to L_{lvl}^{todo} as they have not been tried out as a GLB.

Should no GLB have been chosen yet, a next constraint needs to be chosen from L_{lvl}^{todo}. A newly chosen GLB requires the reset flag r to be set to true. This flag causes all constraints from L_{lvl}, U_{lvl} and N_{lvl} to be passed to respectively L^{comb}, U^{comb} and C_{lvl}. After all, now that a new GLB has been chosen no constraint has yet been combined with it.

Finally, we call the FMplexCombine method with the combination sets, c_{lvl} and $x_{k_{lvl}}$. These newly created constraints are then passed onto the next level together with the non-bounds contained in C_{lvl}. Afterwards, C_{lvl} needs to be emptied as all constraints have been considered and the algorithm moves onto the next level. The algorithm returns SAT, if a level is entered where all constraints in C_{lvl} are trivially true. This main part of the algorithm is presented in Algorithm 2. For more readability we look at the partitioning of constraints in a second function DistributeConstraints.

Algorithm 2 The FMplex algorithm for checking satisfiability as in [Ste22]

Input: The set of all constraints C, the newly added constraints C^{new}
Output: The algorithm returns SAT, if the constraints in C are satisfiable and UNSAT otherwise

function CheckSAT
  $lvl \leftarrow 1$
  $C_1 \leftarrow C_1 \cup C^{new}$
  while C_{lvl} contains at least one not trivially true constraint **do**
    if C_{lvl} contains at least one trivially false constraint **then**
      $lvl \leftarrow$ AnalyzeAndBacktrack(C_{lvl})
      if $lvl = 0$ **then**
        ResetBelow(0)
        $C^{new} \leftarrow C$
        return UNSAT
      else
        $c_{lvl} \leftarrow$ ChooseNextConstraint(lvl)
        ResetBelow(lvl)
        $r \leftarrow true$
      end if
    else if $lvl > maxLvl$ **then**
      $x_{k_{lvl}} \leftarrow$ ChooseNextVariable(C_{lvl})
      $maxLvl \leftarrow maxLvl + 1$
    end if
    DistributeConstraints ▷ Sort constraints based on their bound type
    $C_{lvl+1} \leftarrow C_{lvl+1} \cup C_{lvl} \cup$ FMplexCombine($L^{comb}, c_{lvl}, U^{comb}, x_{k_{lvl}}$)
    $C_{lvl} \leftarrow \emptyset$
    $lvl++$
  end while
  $C^{new} \leftarrow \emptyset$
  return SAT
end function

Algorithm 3 Choose a new eliminator

Input: The level on which a new constraint needs to be chosen lvl
Output: The constraint that was chosen as eliminator

function ChooseNextConstraint(lvl)
  if $c_{lvl} \neq \perp$ **then**
    $L^{done}_{lvl} \leftarrow L^{done}_{lvl} \cup c_{lvl}$
  end if
  Choose new constraint c from L^{todo}_{lvl} according to some heuristic.
  $L^{todo}_{lvl} \leftarrow L^{todo}_{lvl} \setminus c$
  return c
end function

Algorithm 4 Logic behind the distribution of the constraints to the different sets

Input: We assume that all state variables on the level are accessible
Output: The constraints are distributed to their respective sets

function DISTRIBUTECONSTRAINTS
  $L^{comb} \leftarrow \{c \in C_{lvl} \mid \text{c is a lower bound with respect to } x_{k_{lvl}}\}$
  $U^{comb} \leftarrow \{c \in C_{lvl} \mid \text{c is an upper bound with respect to } x_{k_{lvl}}\}$
  $C_{lvl} \leftarrow C_{lvl} \setminus (L^{comb} \cup U^{comb})$
  $N_{lvl} \leftarrow N_{lvl} \cup C_{lvl}$
  $L^{todo}_{lvl} \leftarrow L^{todo}_{lvl} \cup L^{comb}$
  $U_{lvl} \leftarrow U_{lvl} \cup U^{comb}$
  if $c_{lvl} = \bot \wedge L^{todo}_{lvl} \neq \emptyset$ **then** ▷ No eliminator was chosen yet
    $c_{lvl} \leftarrow$ CHOOSENEXTCONSTRAINT(lvl)
    $L^{comb} \leftarrow L^{comb} \setminus \{c_{lvl}\}$
    $r \leftarrow true$
  end if
  if $r = true$ **then** ▷ A new eliminator was chosen
    $L^{comb} \leftarrow L^{todo}_{lvl} \cup L^{done}_{lvl}$
    $U^{comb} \leftarrow U_{lvl}$
    $C_{lvl} \leftarrow N_{lvl}$
    $r \leftarrow false$
  end if
end function

Example 2.4.1 (Incremental FMplex Method)**.** *Assume we have the same constraints as in Example 2.2.2 over the variables x and y converted to the notation introduced in Subsection 2.4.1:*

$$\begin{array}{cccc|cccc} x & y & b & d_1 & d_2 & d_3 & d_4 & \\ \left(\begin{array}{ccc|cccc} 1 & -1 & -4 & 1 & 0 & 0 & 0 \\ -1 & -2 & -6 & 0 & 1 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 1 & 0 \\ -3 & 4 & -4 & 0 & 0 & 0 & 1 \end{array}\right) & \begin{array}{l} cl = 0 \\ cl = 0 \\ cl = 0 \\ cl = 0 \end{array} \end{array}$$

At the start, all these constraints are considered new and are therefore put into the set C^{new}, which is passed to C_1 on the first level of the algorithm. Assuming we now want to eliminate x, we first sort the constraints according to bound type into the sets L^{comb} and U^{comb}, with non-bounds staying in C_1.

$$L^{comb} = \begin{array}{cccc|cccc} x & y & b & d_1 & d_2 & d_3 & d_4 & \\ \left(\begin{array}{ccc|cccc} -1 & -2 & -6 & 0 & 1 & 0 & 0 \\ -3 & 4 & -4 & 0 & 0 & 0 & 1 \end{array}\right) & \begin{array}{l} cl = 0 \\ cl = 0 \end{array} \end{array}$$

$$U^{comb} = \begin{array}{cccc|cccc} x & y & b & d_1 & d_2 & d_3 & d_4 & \\ \left(\begin{array}{ccc|cccc} 1 & -1 & -4 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}\right) & \begin{array}{l} cl = 0 \\ cl = 0 \end{array} \end{array}$$

As all constraints are either upper or lower bounds with respect to x, the set C_1 becomes empty. The next step is to choose a GLB from L^{comb} and in this case we

just choose the first row.

$$c_1 = \begin{array}{c} \begin{matrix} x & y & b & d_1 & d_2 & d_3 & d_4 \end{matrix} \\ \left(\begin{array}{ccc|cccc} -1 & -2 & -6 & 0 & 1 & 0 & 0 \end{array}\right) \end{array} \; cl = 0$$

Then, we execute the FMplexCombine method to get the new constraints for the next level C_2. However, we scale the constraints with the least common multiple of the coefficients of x to get nicer numbers.

$$C_2 = \begin{array}{c} \begin{matrix} y & b & d_1 & d_2 & d_3 & d_4 \end{matrix} \\ \left(\begin{array}{cc|cccc} -3 & -10 & 1 & 1 & 0 & 0 \\ -3 & -11 & 0 & 2 & 1 & 0 \\ 10 & 14 & 0 & -3 & 0 & 1 \end{array}\right) \end{array} \begin{array}{l} cl = 0 \\ cl = 0 \\ cl = 1 \end{array}$$

Now on level 2 we need to eliminate y and again sort the constraints.

$$L^{comb} = \begin{array}{c} \begin{matrix} y & b & d_1 & d_2 & d_3 & d_4 \end{matrix} \\ \left(\begin{array}{cc|cccc} -3 & -10 & 1 & 1 & 0 & 0 \\ -3 & -11 & 0 & 2 & 1 & 0 \end{array}\right) \end{array} \begin{array}{l} cl = 0 \\ cl = 0 \end{array}$$

$$U^{comb} = \begin{array}{c} \begin{matrix} y & b & d_1 & d_2 & d_3 & d_4 \end{matrix} \\ \left(\begin{array}{cc|cccc} 10 & 14 & 0 & -3 & 0 & 1 \end{array}\right) \end{array} \; cl = 1$$

We again choose the first row as the GLB and apply the FMplexCombine method with the same scaling method as before.

$$c_2 = \begin{array}{c} \begin{matrix} y & b & d_1 & d_2 & d_3 & d_4 \end{matrix} \\ \left(\begin{array}{cc|cccc} -3 & -10 & 1 & 1 & 0 & 0 \end{array}\right) \end{array} \; cl = 0$$

Therefore, on level 3 we arrive at the following set of constraints.

$$C_3 = \begin{array}{c} \begin{matrix} b & d_1 & d_2 & d_3 & d_4 \end{matrix} \\ \left(\begin{array}{c|cccc} -58 & 10 & 1 & 0 & 3 \\ -1 & -1 & 1 & 1 & 0 \end{array}\right) \end{array} \begin{array}{l} cl = 1 \\ cl = 2 \end{array}$$

The constraints in C_3 are trivially true and therefore the algorithm returns SAT. In an incremental scenario the SAT solver would then continue solving the Boolean abstraction and consult the theory solver again after extending the assignment. Therefore, now assume the SAT solver passes the additional constraint c_5^* in the following iteration.

$$C_{new} = \begin{array}{c} \begin{matrix} x & y & b & d_1 & d_2 & d_3 & d_4 & d_5 \end{matrix} \\ \left(\begin{array}{ccc|ccccc} 2 & 0 & -4 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \end{array} \; cl = 0$$

The d_5 entry for the derivation coefficients is assumed to be 0 in all already existing constraints. As before, C_{new} is passed to C_1, where the constraints are split into upper, lower and non-bounds. Currently, the only constraint in C_1 is c_5^* as we emptied C_1 in the last iteration. Therefore, U^{comb} contains only c_5^* and L^{comb} and C_1 remain

empty. Now we only need to combine the previously chosen GLB c_1 on level 1 with c_5^* because all other combinations have been done in the previous run of FMplex and are already saved in the sets U_2, L_2 and N_2.

$$C_2 = \begin{array}{cccccccc} y & b & d_1 & d_2 & d_3 & d_4 & d_5 & \\ (-4 & -16 & \mid 0 & 2 & 0 & 0 & 1\,) & cl = 0 \end{array}$$

With this we continue as in level 1 and set L^{comb} to C_2 and let U^{comb} and C_2 be empty. Then, we again only need to combine the new constraint with the previously chosen GLB c_2 on level 2.

$$C_3 = \begin{array}{cccccccc} b & d_1 & d_2 & d_3 & d_4 & d_5 & & \\ (-8 & \mid -4 & 2 & 0 & 0 & 3\,) & & cl = 2 \end{array}$$

C_3 only contains trivially true constraints and therefore the algorithm again returns SAT.

2.4.4 Heuristics for Variable and Constraint Choice

In Example 2.4.1 the choice of which variable to eliminate and which GLB to use on every level was arbitrary. These choices can be done according to some heuristics to make the method more efficient. On every level C_i we need to choose the variable which is eliminated as well as the constraint, with which the elimination is done. In contrast to the example, the constraint does not need to be a GLB but could also be a SUB. Therefore, we need to not only choose a variable but also the *direction*, which decides whether we choose a GLB or a SUB on this level.

When selecting the variable on level C_i a reasonable objective is to minimize the number of branches the elimination creates, which is equivalent to minimizing the number of same-bound combinations. To do this, we determine for every variable x the number of upper bounds $\#u(x)$ and the number of lower bounds $\#l(x)$ on this level C_i. We can then order the variables according to the following criteria.

Let $min(x) = min\{\#u(x), \#l(x)\}$ and $max(x) = max\{\#u(x), \#l(x)\}$ for every variable x. Then, for two different variables x and x' it holds that $x < x'$, if $min(x) < min(x')$. Should $min(x) = min(x')$, we consider two different cases. In the first case $min(x) = min(x') = 0$, meaning we have only bounds in one direction, then $x < x'$, if $max(x) > max(x')$. In the second case $min(x) = min(x')$ with $min(x) \neq 0$, meaning we have bounds in both directions, then $x < x'$, if $max(x) < max(x')$. This distinction is made due to fact that, if a variable has no bounds in one direction, we want the other direction to have as much bounds as possible, because that means that we can drop more constraints. If for two variables x and x', it holds that $min(x) = min(x')$ and $max(x) = max(x')$ we consider them equal and just order them arbitrarily.

We then choose the smallest variable according to this order as the one that is eliminated on this level. Assume that we have chosen the variable x^* to be eliminated. Now we still need to choose the direction. If $min(x^*) = 0$, we only have bounds in one direction and do not need to choose a direction, as we simply set the eliminator to $\perp$. Should $min(x^*) \neq 0$, we use GLBs, if $min(x^*) = \#l(x^*)$ and otherwise use SUBs. On a level C_i a new GLB or SUB - depending on the direction - is chosen when we first enter that level or when we backtrack to this level due to a local conflict. When selecting the eliminator we prioritize constraints, which have the most zeros

among the derivation coefficients. That means constraints whose linear combination uses as few of the original constraints as possible, as to keep the derivation coefficients compact.

2.4.5 Extension to Strict Inequalities

So far we have solely considered non-strict inequalities. Now we discuss how to handle strict inequalities.

First, we look at the treatment of the operator when combining constraints. This treatment is different for same-bound and upper-lower combinations. Let c_1 and c_2 be two constraints, which need to be combined. In case c_1 and c_2 correspond to an upper-lower combination the resulting constraint c_{new} has a strict operator, if and only if at least one of the constraints is a strict inequality and otherwise has a weak operator. In the other case c_1 and c_2 correspond to a same-bound combination. Assume c_1 is the constraint that was chosen as the eliminator on this level and that c_1 is a GLB (for a SUB it works analogously). Then, the only case where c_{new} has a strict operator, is when c_1 is non-strict and c_2 is strict. After all, if the bounds induced by c_1 and c_2 are equal, then c_1 would not actually be the GLB as c_2 is strict and c_1 not.

Secondly, we need to adjust the handling of conflicts. Through combining constraints it is possible to arrive at constraints, which possess a strict operator even though the derivation coefficients only reference non-strict constraints. That means in case of a global conflict it needs to be checked, if the conflict would still occur even with a non-strict operator, as the linear combination of the original constraints would only result in a non-strict operator. Should the conflict not hold with a non-strict operator the Feasibility Theorem can no longer be applied and the conflict needs to be treated as a local conflict.

2.4.6 Properties

The FMplex method is a theory solver that determines the satisfiability of a conjunction of constraints in finite time. This can be shown by proving the correctness and the completeness of the algorithm.

Theorem 2.4.1 (FMplex Correctness [Ste22])**.** *The FMplex method is correct i.e. if the method returns SAT/UNSAT, then the underlying constraint set is satisfiable/unsatisfiable.*

Sketch of Proof. Assume the method returns SAT. That means the method has found a branch where on the last level only trivially true constraints remain. This implies that all choices regarding the GLB or SUB were suitable and that these do not conflict with bounds of the other type. Therefore, no lower and upper bound is conflicting. With this a satisfying assignment can be built by backtracking through the branch. On each level all variables except the one that is eliminated on that level can be substituted by their assignment as we are iterating backwards. The variable that is eliminated on that level can then be assigned a value in the interval given between the GLB and SUB (now consisting only out of real numbers). This interval exists as no lower and upper bound conflicts. This assignment then satisfies the original constraint set.

Now assume that the method returns UNSAT. This can only happen when either a global conflict has been found or on level 1 every eliminator choice led to a local conflict. In case a global conflict has been found the Feasibility Theorem applies and we can conclude that the original constraint set is unsatisfiable. In the other case we return UNSAT because every choice for an eliminator led to a local conflict. W.l.o.g assume that the direction on the first level is lower bounds. That means every lower bound was tried out as a GLB, yet all of these led to a local conflict. This is a contradiction as we try out every subtree and therefore we can conclude unsatisfiability for the original constraint set. □

We show the finite termination of the algorithm by showing that in the worst case only a singly exponential number of constraints is generated. Therefore, the worst case runtime is also singly exponential. This also shows the superiority of the worst case runtime in comparison to the doubly exponential Fourier-Motzkin method.

Theorem 2.4.2 (FMplex complexity [Kob21])**.** *The FMplex method generates at most a singly exponential number of constraints in terms of the number of original constraints.*

Proof. Let m be the number of original constraints and n be the number of variables occurring in these constraints. At the beginning we can choose from at most m eliminators, which in turn leads to the next level with at most $m-1$ constraints. This is done until in the worst case all n variables are eliminated. Therefore, the number of generated constraints on the last level is limited to

$$m \cdot (m-1) \cdot (m-2) \cdot \dots \cdot (m-n)$$

in the worst case. Additionally, to the constraints in the leave nodes, we have the constraints in the inner nodes. We have a maximum of $n+1$ levels and on each level the total amount of possible generated constraints is smaller than the number of generated constraint on the last level. Therefore, in the worst case the number of all generated constraints is limited by

$$(n+1) \cdot (m \cdot (m-1) \cdot (m-2) \cdot \dots \cdot (m-n)) < (n+1) \cdot m^n$$

This means only a singly exponential number of constraints is generated in one run of the FMplex method. □

2.4.7 Motivation for Optimization

Stein [Ste22] developed and tested an implementation of incremental FMplex on top of the SMT-RAT project [CKJ+15]. The tests were run on the RWTH High Performance Cluster with a time limit of 5 minutes and a memory limit of 5 GB while using the quantifier-free linear real arithmetic benchmark set from SMT-LIB [BFT16]. Stein tested different variations of the presented algorithm. Among them were the *Standard* version, which uses incrementality as well as heuristics, the *No-Heu* version, which uses incrementality but no heuristics and the *No-Incr* version, which uses heuristics but no incrementality (branch is reset after every call of FMplex). An excerpt from the results is shown in Table 2.1

The results show that the heuristics for variable and constraint choice are very important for the performance of the algorithm as the No-Heu version only manages

Configuration	**Memory Out**	**Time Out**	**SAT**	**UNSAT**	**Total Unsolved**	**Total Solved**
Standard	358	682	426	287	1040	713
No-Heu	195	1417	56	85	1612	141
No-Incr	216	794	431	312	1010	743

Table 2.1: Performance of different variants from [Ste22]

to solve 141 instances from the total 1753 instances. What is surprising though is that the number of solved instances is lower for the Standard version that uses both incrementality and heuristics than for the No-Incr version that just uses heuristics. In fact, the No-Incr version solves 30 instances more than the Standard version. The reason that the No-Incr version seems to be more efficient may lie in the fact that with enabled incrementality we are bound by the variable and constraint choices made in the last run until we run into a conflict. This could be detrimental to the efficiency as with additional constraints a prior variable choice may induce a much higher number of same-bound combinations compared to before. Therefore, the old variable choice could create a much larger computation tree than a new variable choice would. This could also be the reason why the No-Incr version has a much smaller number of Memory Outs.

Chapter 3

Optimized Incremental FMplex

In this chapter we present an optimization of the incremental FMplex method, which aims to reduce the runtime as well as the memory usage. To do so, we first introduce new data structures for the constraints since the combining and storing of constraints is an important factor in the efficiency of the method. Furthermore, an approach to make more use of the heuristics in subsequent runs of algorithm is considered, as to enable the solver to undo variable decisions when confronted with new constraints. A further attempt to increase the performance by continuing the computation after encountering local conflicts is also given. Additionally, we make the infeasible subsets smaller and change the treatment of equalities and not-equal constraints to utilize their special characteristics.

3.1 Data Structures

The theory solver works on a set of constraints. As such it is very important that the constraints are stored in a suitable data structure. Currently, constraints are stored using the `BasicConstraint` data structure from the CArL library [CKJ+]. This structure is templated with a `MultivariatePolynomial` and is applicable to a variety of scenarios. Therefore, it can be used for constraints in linear real algebra, but also for example in nonlinear real algebra. However, there is no need to use such a powerful structure and it may be more efficient to use structures more tailored towards FMplex.

To fully capture a constraint $c = \langle a_1, \ldots, a_n, b \mid d_1, \ldots, d_m \rangle$ it is enough to store the following:

- Relation: The operator of the constraint needs to be stored, so whether it is strict or non-strict inequality.
- Coefficients: For every variable x_k the constraint has a rational coefficient a_k, which needs to be associated with the respective variable.
- Constant Term: The constant term b is a special case that is not associated with any variable.
- Derivation Coefficients: The derivation coefficients $d_1, \ldots, d_m$ need to be remembered, so that we can identify the type of conflicts.

- Conflict level: The conflict level cl needs to be associated with the constraint, such that the backtrack level can be determined.

To make the combination of constraints easier to implement, we use a template class `SimpleConstraint<PolyType>`, which keeps track of the derivation coefficients, the conflict level as well as the relation. This class is templated with a structure defining the linear term, that is the coefficients as well as the constant term. Constraints can then be combined by scaling and adding the linear terms, while the resulting relation, conflict level and derivation coefficients can be considered separately.

For the implementation of the linear term we use three different approaches.

- `MultivariatePolynomial`: For reference we use the MultivariatePolynomial type that is used to template the BasicConstraint data type. This implementation just uses the polynomial as defined in CArL and wraps it in a structure with the same signature as the other approaches.

- `Map`: Here we distinguish between the constant term and the coefficients. The constant term is just saved by itself as a `Rational` as defined in CArL. Meanwhile, the coefficients are stored utilizing the map data structure from the C++ standard library [Jos12]. Let $X = \{x_1, \dots x_n\}$ be the set of variables occurring in a constraint with the coefficients $a_1, \dots, a_n$. Then, the map is a function $f : X \to \mathbb{Q}$, which maps each variable to its coefficient: $x_1 \mapsto a_1, \dots, x_n \mapsto a_n$. This map only saves entries for variables with nonzero coefficients, as to be more memory efficient.

  The addition of linear terms p_1 and p_2 can then be done by copying the map of p_1 and iterating over all map entries of p_2. If the map of p_1 already has a value for a variable entry of the map of p_2, the values of the coefficients are added together. In case this addition causes the coefficient to become 0 the entry for the variable is removed. If the map of p_1 has no value for a variable entry of the map of p_2, a new entry is created with the coefficient value of p_2. Finally, the constant terms are added as well.

- `Vector`: Just as in the map approach the constant term is saved separately as a `Rational`. However, in this approach the coefficients are saved in the vector data structure from the C++ standard library. More specifically, we use a vector of 2-tuples, where the first component is a variable and the second component is the coefficient. Let $X = \{x_1, \dots x_n\}$ be the set of variables occurring in a constraint with the coefficients $a_1, \dots, a_n$. Then, the vector would look like this $\langle (x_1, a_1), \dots, (x_n, a_n) \rangle$. Again, as in the map approach only nonzero coefficients are actually stored as to keep the vector compact. Furthermore, the vector is sorted according to an ordering on the variables to make the searching and adding more efficient.

  The addition of two linear terms p_1 and p_2 utilizes the same idea as merge sort. We start iterating over both vectors at the same time. Then, we check if the current entry of p_1 references a smaller variable than p_2. Should that be the case, we append the current entry of p_1 to the result and increment the iterator of p_1. If the current entry of p_2 references a smaller variable than p_1, we do the same just for p_2. Otherwise, if both current entries reference the same variable, we append an entry with said variable and the addition of the coefficients to the result and increment the iterators of both p_1 and p_2. However, should that

addition result in a coefficient that equals 0, the entry is not appended and just the iterators are incremented. We do this until one iterator reaches the end of the vector and subsequently finish iterating over the other vector and just append every entry to the result.

3.2 Leveraging Heuristics

3.2.1 Reversing Decisions

To make the FMplex method as efficient as possible, it is important that only few branches need to be examined until a branch with a satisfying assignment or a global conflict is found. The ordering in which branches are visited heavily depends on the variable and eliminator choice. This is reflected in the massive effect of the heuristics on the number of solved instances as seen in the Table 2.1.

However, in an incremental setting the variable choices are carried over in subsequent runs of the algorithm. When the method finds a satisfying assignment on a set of constraints, it returns SAT to the SAT solver and remembers the branch that led to this assignment, including the variable and eliminator choices as well as the resulting constraints. The SAT solver may then extend the set of constraints and ask the FMplex method, if this set is still satisfiable. Since the theory solver remembers the previously computed branch, it is enough to combine the new constraints with the already chosen eliminators. While this approach makes sure that branches previously found to be conflicting are not visited again, it also introduces the problem that the method needs to adhere to previous variable choices. This is problematic since the new constraints may change the situation in such a way that the elimination of another variable may lead to a result in less time.

For example, let x_k be the variable that the heuristic picked to be eliminated on level 1 in the prior run of the method. Let us assume that the choice was made due to the fact that x_k had the smallest number of lower bounds in the remaining variables. Accordingly, a GLB c_k was chosen as the eliminator. Now in the worst case the SAT solver added a lot of lower bounds with respect to x_k to the set of constraints, which leads to the fact that x_k has now the highest number of lower bounds among the remaining variables. However, the method is bound to the choice of x_k, which in turn could lead to an increase in local conflicts until a non-conflicting GLB is found. Therefore, adhering to the choice of x_k may lead to a decrease in performance instead of an improvement. The fact that the non-incremental version solves more instances than the incremental one as shown in Table 2.1, may also be caused by this problem.

Since we want to make use of incrementality, while still leveraging heuristics in the variable choice, we propose an adaptation of the algorithm. This adaptation aims to make the best of both worlds, by using the incrementality like the usual FMplex method, while additionally checking, if a variable choice is still reasonable under new circumstances. Should the method decide that due to the influx of new constraints the variable choice is not optimal according to the heuristic, the choice is reverted and everything below the current level is reset. Afterwards, a new variable is selected and the algorithm continues as normal. This check is done on every level, which was visited before and now received new constraints.

Applying the heuristic for variable choice is expensive, as we need to iterate over every constraint. That being the case, we first want to check a cheaper criteria to determine, if a variable should be considered for resetting.

For that we use the function `ShouldResetVariable`$(x_{k_{lvl}}, L_{lvl}^{todo}, L_{lvl}^{done}, C_{lvl})$. Given a variable choice $x_{k_{lvl}}$, the not yet tested lower bounds L_{lvl}^{todo}, the already tested lower bounds L_{lvl}^{done} and the newly added constraints C_{lvl}, it returns `true`, if a variable should be considered for resetting and `false` otherwise. Here we consider only GLBs for the direction, but the implementation also works for SUBs. In the current heuristic, the variable which is to be eliminated, is chosen as the one, which has the least number of lower bounds - in case there is no variable that has only bounds in one direction. Considering this fact, the function counts the number of constraints in C_{lvl} that are lower bounds with respect to $x_{k_{lvl}}$. This number is added to the number of constraints that have yet to be tested as GLB $\#L_{lvl}^{todo}$. Should this addition result in a number that is higher than the total amount of lower bounds $\#L_{lvl}^{todo} + \#L_{lvl}^{done}$ that were present in the previous run of the method, we return `true`. This is because, the initial decision to choose $x_{k_{lvl}}$ was based on the fact that only $\#L_{lvl}^{todo} + \#L_{lvl}^{done}$ branches needed to be tried out at most. However, the new constraints changed the situation, such that the number of potential branches is higher than before.

Therefore, the heuristic for variable choice is applied again to check, which variable would be chosen with all constraints in consideration. If this newly chosen variable is different from $x_{k_{lvl}}$, we reset everything below the current level and continue on with the new variable. Otherwise, the heuristics would still choose $x_{k_{lvl}}$ as the variable and we can continue without resetting. Other approaches like checking how many levels a variable change would undo and then deciding whether to reverse the decision, are also possible, but are not further considered here.

Algorithm 5 Decide if a variable should be reset

Input: The currently selected variable $x_{k_{lvl}}$, the set of constraints that were no yet tried out as GLBs L_{lvl}^{todo}, the set of constraints that were already tried out as GLBs L_{lvl}^{done}, the set of newly added constraints C_{lvl}

Output: The algorithm returns true, if a variable should be considered for resetting and false otherwise

```
1: function ShouldResetVariable(x_{k_lvl}, L_lvl^todo, L_lvl^done, C_lvl)
2:     #newBounds ← #{c ∈ C_lvl | c is a lower bound with respect to x_{k_lvl}}
3:     #doneBounds ← #L_lvl^done
4:     #todoBounds ← #L_lvl^todo
5:     if #newBounds + #todoBounds ≥ #doneBounds + #todoBounds then
6:         return true
7:     else
8:         return false
9:     end if
10: end function
```

3.2.2 Continue after Conflict

To faster identify unsatisfiability we present another approach, which tries to reach a global conflict by continuing after local conflicts. That means after finding a local conflict, we do not directly backtrack but instead continue eliminating variables in the hope that the local conflict correlates with a global conflict. In particular, we look at two different ideas, which can be used at the same time.

- **Peek further:** The first approach is based on the idea that a local conflict might indicate that a global conflict appears in the next few levels. Therefore, after finding a local conflict, we peek a fixed number of levels further to check whether a global conflict is present. In particular, we introduce the program variable `pardon`, which defines the number of levels that are checked after a local conflict is found. For now we set `pardon` to 1, although different thresholds can be tested.

- **Continue until the end:** The second approach is formed on the idea that there is a higher chance for a global conflict, if all variables are eliminated. Based on this, we decide to continue the computation until all variables are eliminated, if there are only few variables remaining. That means that we ignore local conflicts, when only a certain number of variables remain. Then, only trivial constraints are left with hopefully a global conflict among them. The number of variables upon we decide to continue until the end is saved in the program variable `endThreshold` and is currently set to 3. Again, different thresholds can be tested.

The adapted algorithm can be seen in Algorithm 6, where the part for reversing decisions is marked in blue and the part for continuing after local conflicts is marked in red.

Algorithm 6 Optimized FMplex

Input: The set of all constraints C, the newly added constraints C^{new}
Output: The algorithm returns SAT, if the constraints in C are satisfiable and UNSAT otherwise

function CheckSAT
  $pardon \leftarrow 1$
  $endThreshold \leftarrow 3$
  $lvl \leftarrow 1$
  $C_1 \leftarrow C_1 \cup C^{new}$
  while C_{lvl} contains at least one not trivially true constraint **do**
    if C_{lvl} contains at least one trivially false constraint **then**
      $lvl \leftarrow$ AnalyzeAndBacktrack(C_{lvl})
      if $lvl = 0$ **then**
        ResetBelow(0)
        $C^{new} \leftarrow C$
        return UNSAT
      else if Number of variables in C_{lvl} is greater than *endThreshold* or
      equal to 0 **then**
        ConflictHandling
      end if
    else if $lvl > maxLvl$ **then**
      $x_{k_{lvl}} \leftarrow$ ChooseNextVariable(C_{lvl})
      $maxLvl \leftarrow maxLvl + 1$
    else if $C_{lvl} \neq \emptyset \wedge$ ShouldResetVariable($x_{k_{lvl}}, L_{lvl}^{todo}, L_{lvl}^{done}, C_{lvl}$) **then**
      ReverseDecision
    end if
    DistributeConstraints ▷ Sort constraints based on their bound type
    $C_{lvl+1} \leftarrow C_{lvl+1} \cup C_{lvl} \cup$ FMplexCombine($L^{comb}, c_{lvl}, U^{comb}, x_{k_{lvl}}$)
    $C_{lvl} \leftarrow \emptyset$
    $lvl++$
  end while
  $C^{new} \leftarrow \emptyset$
  return SAT
end function

Algorithm 7 Conflict handling

Input: We assume that all state variables on the level are accessible
Output: When pardon is greater 0, reduce pardon, otherwise backtrack

```
1: function ConflictHandling
2:     if pardon > 0 then
3:         pardon ← pardon − 1
4:     else
5:         pardon ← 1
6:         ChooseNextConstraint(lvl)
7:         ResetBelow(lvl)
8:         r ← true
9:     end if
10: end function
```

Algorithm 8 Reverse the variable decision

Input: We assume that all state variables on the level are accessible
Output: Applies the heuristic for variable choice again and accordingly resets the level

```
1: function ReverseDecision
2:     newVariable ← ChooseNextVariable(C_lvl ∪ U_lvl ∪ N_lvl ∪ L_lvl)
3:     if x_{k_lvl} ≠ newVariable then
4:         ResetBelow(lvl)
5:         x_{k_lvl} ← newVariable
6:         C_lvl ← C_lvl ∪ L_lvl ∪ U_lvl ∪ N_lvl
7:         L_lvl ← ∅
8:         U_lvl ← ∅
9:         N_lvl ← ∅
10:    end if
11: end function
```

3.3 Handling Equalities

Equalities are special constraints, because they can be seen as a GLB as well as SUB at the same time. However, in the current implementation of FMplex equalities are simply split up into two non-strict inequalities. For an equation e with variables $x_1, \ldots, x_n$ it is done in the following way:

$$\sum_{i=1}^{n} a_i x_i + b = 0 \Leftrightarrow \sum_{i=1}^{n} a_i x_i + b \leq 0 \wedge \sum_{i=1}^{n} -a_i x_i - b \leq 0$$

This approach is problematic since it increases the number of constraints, which in turn increases the number of choices for the eliminator. Therefore, the number of combinations as well as the number of possible branches is increased and the performance of the method worsens.

For that reason, we discuss a different handling of equalities that does not cause a splitting of constraints. If we want to use equalities without splitting them, we need a

way of combining them with other constraints. To do so, we need to look at the case, where the equality is the eliminator and the case, where the equality is eliminated using an inequality.

3.3.1 Equality as Eliminator

First, we look at the case, where an equality was chosen as the eliminator (GLB/SUB). Assume that we are on level k and the variable x_1 was chosen to be eliminated. The choice of x_1 is just for notational convenience, it is the same for any other variable. Furthermore, the equality e was chosen as an eliminator. Then, the variable x_1 can be eliminated using Gauß's variable elimination. Meaning in every other constraint on level k the variable x_1 is substituted according to the equality e. Let c be a constraint on level k with $c_1 \neq 0$ and $\bowtie \in \{<, \leq, =\}$ of the following form:

$$\sum_{i=1}^{n} c_i x_i + d \bowtie 0.$$

To eliminate x_1 in c using the equality $e : \sum_{i=1}^{n} a_i x_i + b = 0$, we multiply e by $-\frac{c_1}{a_1}$ and add it to c. This results in the constraint

$$\begin{aligned} c_{new} : \; -\frac{c_1}{a_1} \cdot (\sum_{i=1}^{n} a_i x_i + b) + \sum_{i=1}^{n} c_i x_i + d \bowtie 0 \\ \Leftrightarrow \sum_{i=2}^{n} (c_i - \frac{c_1}{a_1} a_i) x_i + d - \frac{c_1}{a_1} b \bowtie 0 \end{aligned}$$

without the variable x_1.

Using an equality as the eliminator has the benefit that we make no assumption about any GLB or SUB. Consequently, this elimination can not be the cause of any conflict. Therefore, the conflict level of the resulting constraint is always the maximum conflict level of the parent constraints. Hence, we never backtrack to this level to choose a new eliminator, which means a node in the computation tree with an equality as the eliminator can only ever have one child. The combining of constraints with an equality as the eliminator is shown in Algorithm 9.

Algorithm 9 Using an equation as the eliminator

Input: The set of constraints C, the equality used as eliminator $e : (e_1 x_1 + \ldots e_n x_n + b_e = 0) \notin C$, the variable to be eliminated x_k with $1 \leq k \leq n$, $e_k \neq 0$
Output: The set of constraints resulting from the elimination of x_k with e as eliminator

```
1: function APPLYEQ(C, e, x_k)
2:     Constraint set R = ∅
3:     for c ∈ C do
4:         Constraint c_new = c − (c_k / e_k) · e
5:         c_new.relation ← c.relation        ▷ The relation symbol of c is retained
6:         cl(c_new) ← max{cl(c), cl(e)}
7:         R ← R ∪ {c_new}
8:     end for
9: end function
```

3.3.2 Equality as Eliminee

In the second case the eliminator is an inequality and we want to eliminate a variable in e using this inequality. Assume that we are on level k and the variable x_1 was chosen to be eliminated, again only for notational convenience. Furthermore, the inequality c was chosen as an eliminator. For now we assume that c is a non-strict inequality. The handling of strict inequalities is explained later. For simplicity, assume that c is a lower bound with respect to x_1, so c is the assumed GLB. If c were an assumed SUB, it could be treated analogously. Now we want to eliminate x_1 in e using an assumed GLB c.

To do so, we first look at what would happen, if we were to split the equality. Then, we would have the following constraints:

$$c: \sum_{i=1}^{n} c_i x_i + d \leq 0$$
$$e_1: \sum_{i=1}^{n} a_i x_i + b \leq 0$$
$$e_2: \sum_{i=1}^{n} -a_i x_i - b \leq 0$$

If we combine c with e_1 and e_2, we get - depending on the sign of a_1 - two resulting constraints c_1 (Combination of c and e_1) and c_2 (Combination of c and e_2). The combination itself follows the principle used in the `FMplexCombine` method.

- $a_1 > 0$: That means e_1 is an upper bound and e_2 is a lower bound. Hence, the combination of e_1 and c is an upper-lower combination c_1, while the combination of e_2 and c is a same-bound combination c_2.

$$c_1: \frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) + \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) \leq 0$$
$$c_2: \frac{1}{|-a_1|}\left(\sum_{i=1}^{n} -a_i x_i - b\right) - \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) \leq 0$$
$$\Leftrightarrow -\left(\frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) + \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right)\right) \leq 0$$

  The resulting constraints c_1 and c_2 must both hold on level $k+1$, which is why we can treat them as being in a conjunction and combine them together to an equality.

$$c_1 \wedge c_2 \Leftrightarrow \frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) + \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) = 0$$

  Thus. we end up with an equality as the resulting constraint c_{new}.

- $a_1 < 0$: That means e_1 is a lower bound and e_2 is an upper bound. Therefore, this time the combination of e_1 with c is a same-bound combination c_1, while

that of e_2 and c is an upper-lower combination c_2.

$$c_1 : \frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) - \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) \leq 0$$
$$c_2 : \frac{1}{|-a_1|}\left(\sum_{i=1}^{n} -a_i x_i - b\right) + \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) \leq 0$$
$$\Leftrightarrow -\left(\frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) - \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right)\right) \leq 0$$

This causes a switch in signs, although c_1 and c_2 can still be combined together to an equality.

$$c_1 \wedge c_2 \Leftrightarrow \frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) - \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) = 0$$

Thus, we again end up with an equality as the resulting constraints c_{new}.

Overall, we always get an equality as the resulting constraint but in case that a_1 has the same sign as c_1 we need to additionally multiply c by -1, so that x_1 disappears. However, this creates the problem that the resulting equality is always based on one same-bound combination. Due to this, any conflict that is found using c_{new} may be based on the fact that c was wrongly assumed as the GLB. As such the conflict level of c_{new} is always the current level lvl. Furthermore, it is unclear what the derivation coefficients of such a constraint should be, as the equality is based on two combinations, one where c multiplied by $\frac{1}{|c_1|}$ and one where c is multiplied by $-\frac{1}{|c_1|}$. This is problematic because the type of conflict is determined by the derivation coefficients.

To make sure we do not wrongly conclude a global conflict, we add a flag called `involvedInEQ` to every constraint, which is initially set to `false`. If a constraint is created by eliminating an equality with an inequality, the flag is set to `true`. In every other combination the flag of the resulting constraint is `true` if and only if at least one of the parent constraints has the flag set to `true`. When analyzing a conflict, this flag tells us that we need to interpret the conflict as a local conflict due to a prior involvement in an equality elimination. Since the flag is set, the derivation coefficients of c_{new} do not matter anymore and we can just arbitrarily choose one of the possibilities.

So far we only discussed how to eliminate an equality with a non-strict inequality. In fact, the elimination of an equality with a strict inequality as an assumed GLB instantly leads to conflicting constraints. Again assume c is the assumed GLB, although this time as a strict inequality, and we want to eliminate x_1 in the equality e. Just like before we assume e is split into the two non-strict inequalities e_1 and e_2. Then, for $a_1 > 0$ (the other case behaves analogously) we get the following resulting

constraints:

$$
\begin{aligned}
c_1 : \ & \frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) + \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) < 0 \\
c_2 : \ & \frac{1}{|-a_1|}\left(\sum_{i=1}^{n} -a_i x_i - b\right) - \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) \leq 0 \\
\Leftrightarrow & -\left(\frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) + \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right)\right) \leq 0 \\
\Leftrightarrow & \frac{1}{|a_1|}\left(\sum_{i=1}^{n} a_i x_i + b\right) + \frac{1}{|c_1|}\left(\sum_{i=1}^{n} c_i x_i + d\right) \geq 0
\end{aligned}
$$

The strictness of c_1 is due to the fact that in a upper-lower combination the strict relation is dominant. As becomes visible, this is already a contradiction, which is why c cannot be the GLB. Therefore, we need to choose a different constraint as the eliminator. In short, a strict inequality cannot be the eliminator on a level in case the level also includes equalities.

3.3.3 Change in Heuristic and Conflicts

The different treatment of equalities also impacts the choices of the heuristics as well as the analysis of the conflicts.

Beginning with the heuristic for variable choice, we change it in the following way. The first choice prefers variables that only have bounds in one direction. However, if there are no such variables, we prefer variables that are involved in at least one equality. Should there also be no variable that is involved in equalities, we follow the heuristics as described before. In terms of eliminator choice, we now choose an equality, if possible, as this causes the least number of children branches. If there is no equality, we instead proceed as described before.

When analyzing the conflict, we need to be able to differentiate between local and global conflicts. The different treatment of equalities brings two changes to this distinction. Firstly, the previously introduced `involvedInEQ` flag takes priority, in such a way that all conflicts, where the flag is set need to be treated like local conflicts. Secondly, if the flag is set to `false` and we check the type of conflict using the derivation coefficients, coefficients corresponding to equalities can be treated differently. While a global conflict still requires that all coefficients belonging to inequalities are positive, the signs of the coefficients of equalities do not matter. This is due to fact that, when the flag is not set, we only ever used equalities to apply Gauß's variable elimination, which by itself cannot introduce conflicts that are based on the eliminator choice.

3.4 Handling Not-Equal Constraints

So far we assumed that any not-equal constraint is split into two strict inequalities, which are connected with a disjunction. Since FMplex assumes that all input formulas are implicitly in a conjunction, this had to a happen in preprocessing. This behavior is now slightly changed in such a way, that FMplex accepts not-equal constraints

but first it does not take them into account when determining satisfiability. Should the FMplex method conclude that all constraints beside the not-equal constraints are satisfiable, it tries to construct a model as described before. Afterwards, it is tested whether this model also satisfies all not-equal constraints. If that is the case, SAT can be returned. Otherwise, FMplex returns the value UNKNOWN and tells the SAT solver to split any not-equal constraint, which is not satisfied by the model. To delegate such splits to the SAT solver is a well known approach that is often used to avoid case splitting directly in the theory solver [BNOT06]. The SAT solver then splits the not-equal constraints and constructs a new problem for FMplex to solve.

3.5 Infeasible Subsets

In case the theory solver finds a set of constraints to be unsatisfiable, it needs to pass an infeasible subset to the SAT solver. This infeasible subset is a set of constraints, whose conjunction is unsatisfiable. With the help of this set the SAT solver can rule out assignments without checking with the theory solver again. To exclude as many assignments as possible, the infeasible subset should be as small as possible.

The current implementation of FMplex just returns all constraints as the infeasible subset. However, we can make this set smaller by using global conflicts. When we reach global conflicts, we conclude unsatisfiability. Meanwhile, the derivation coefficients include all constraints that were used to derive that conflict. Let $c = \langle a_1, \ldots, a_n, b \mid d_1, \ldots, d_m \rangle$ be a global conflict and $C^* = \{c_1^*, \ldots, c_m^*\}$ be the set of original constraints. Then, $I = \{c_i^* \in C^* \mid 1 \leq i \leq m \wedge d_i \neq 0\}$ is an infeasible subset. This set must be unsatisfiable as we can use the constraints to derive a global conflict.

In the case we conclude unsatisfiability due to the fact that all possible eliminator choices in the first level led to local conflicts, we again use all constraints as the infeasible subset. Another possibility would be to unite the infeasible subsets of all local conflicts. However, in practice this scenario is very rare and as such we do not use the memory to save this set.

Chapter 4

Benchmarks

In this chapter we will report on testing the aforementioned adaptations on a benchmark set and discuss their effects on the performance of the FMplex method. The implementation of these adaptations was done on top of the existing FMplex implementation of Stein [Ste22] in the *Satisfiability Modulo Theories Real Algebraic Toolbox* (SMT-RAT) [CKJ+15].

4.1 Setup

The benchmark set used for testing the implementation is the quantifier-free linear real arithmetic benchmark set from SMT-LIB [BFT16], containing 1753 instances as of February 2023. For solving the underlying Boolean structure of the instances the standard SAT solver implemented in SMT-RAT is used, while FMplex is used as the theory solver. To compare the effect of each of the adaptations, we test different configurations of the FMplex method with different optimizations enabled. All tests were run on the RWTH Aachen High Performance Computing Cluster. More specifically, the tests were run on a 2.1GHz *Intel Xeon Platinum 8160* CPU with a time limit of 5 minutes and a memory limit of 5GB.

Each instance produces an outcome from the following options:

- `SAT`: Correctly identified instance as satisfiable
- `UNSAT`: Correctly identified instance as unsatisfiable
- `Wrong`: Produced wrong result
- `Time Out`: Computation took longer than 5 minutes
- `Memory Out`: Computation exceeded 5GB of memory

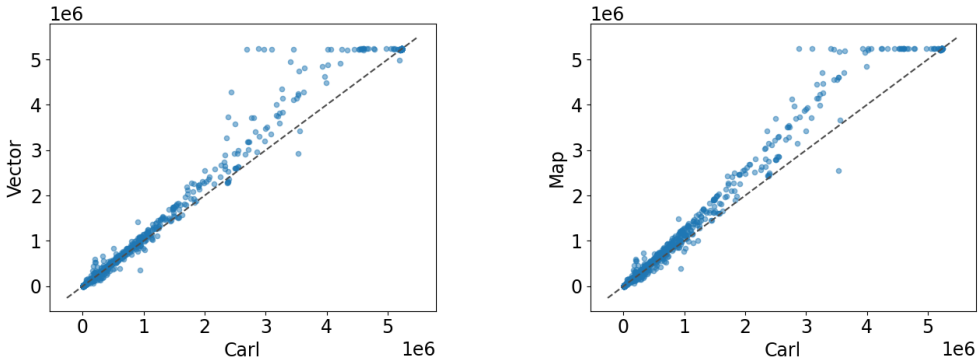
In our case none of the configurations produced a wrong result and as such this outcome is not considered any further.

4.2 Constraint Type

First of all, we want to evaluate, if the new constraint data types presented in Section 3.1 bring any improvement. To do so, we consider three configurations:

	Unsolved		Solved			
Configuration	**Memory Out**	**Time Out**	**SAT**	**UNSAT**	**Total Unsolved**	**Total Solved**
CArL	248	715	447	343	963	790
Map	281	686	443	343	967	786
Vector	278	687	445	343	965	788

Table 4.1: Outcomes for the different constraint types



(a) Comparison Vector and CArL

(b) Comparison Map and CArL

Figure 4.1: Comparison of peak memory usage in kilobyte on all instances

- *CArL*: Uses the `MultivariatePolynomial` from the CArL library [CKJ+]
- *Map*: Uses the Map data structure
- *Vector*: Uses the Vector data structure

Table 4.1 contains the outcomes of all instances on each configuration.

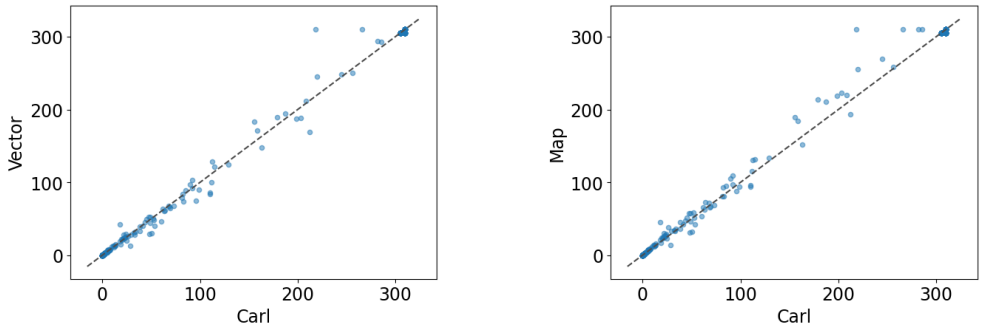
The number of solved instances is very similar for each constraint type, with the *CArL* type solving only a few more instances than the other types. However, the main difference can be found in the memory consumption. In fact, there is a high increase in memory outs in both the *Vector* and *Map* configuration. This indicates that the memory efficiency of both the *Map* and *Vector* constraint type is worse than this of the *CArL* constraint type. Indeed, when looking at the peak memory usage in kilobyte in Figure 4.1, it becomes obvious that the *CArL* constraint type is superior in terms of memory efficiency.

Looking at the comparison of the *CArL* and *Map* configurations 4.1b, we can see that the *Map* configuration also has a slightly worse peak memory usage than the *Vector* configuration.

The reason that both *Vector* and *Map* type seem so much worse could lie in the fact that many copies of the variables are created. It might be more efficient to only create one copy of every variable and then use pointers to reference to them. However, this might negatively impact the runtime as the variables in a constraint may then be stored on different pages, which would lead to more page faults.

Nonetheless, the runtime of both the *Map* as well as the *Vector* approach is similar to the runtime of the *CArL* approach as shown in Figure 4.2.

Therefore, in the future an optimization of memory usage for the *Vector* and *Map* constraints type can be considered. Even so, the *CArL* constraint type is the most



(a) Comparison Vector and CArL

(b) Comparison Map and CArL

Figure 4.2: Comparison of the runtime in seconds on all instances

efficient one. As such all following configurations will use this constraint type.

4.3 Experimental Results

Now that we decided on a constraint type, we will look at the effects of the other changes. That is the reversing of decision described in Section 3.2.1, the continuing after a local conflict described in Section 3.2.2, the different treatment of equalities described in Section 3.3 and the infeasible subsets described in Section 3.5. Each of these changes can be enabled or disabled. Moreover, the incrementality can also be disabled, meaning that the branch is reset after every run. However, with no incrementality the reversing of decisions is of no effect. Additionally, the different treatment of not-equal constraints described in Section 3.4 is also implemented, but likely does not affect the performance of the solver too much. This is why that treatment is always enabled and not discussed in more detail.

The Table 4.2 shows the different configurations based on which optimization is enabled.

Configuration	**Incrementality**	**Reversing Decisions**	**Continue after Conflict**	**Equality Handling**	**Infeasible Subset**
Full	×	×	×	×	×
Full NoIncr			×	×	×
Only Incr	×				
No infSub	×	×	×	×	
No infSub/EQ	×	×	×		
No conAfterCon	×	×		×	×
No revDec	×		×	×	×

Table 4.2: Settings for the configurations. The × symbol stands for enabled, while a blank field stands for disabled

For all of these configurations the outcomes are given in Table 4.3, while in Figure 4.3 a performance profile over the number of solved instances is shown.

	Unsolved		Solved			
Configuration	**Memory Out**	**Time Out**	**SAT**	**UNSAT**	**Total Unsolved**	**Total Solved**
Full	248	715	447	343	963	790
Full NoIncr	181	801	438	333	982	771
Only Incr	264	751	437	301	1015	738
No infSub	248	771	437	297	1019	734
No infSub/EQ	260	758	431	304	1018	735
No conAfterCon	248	713	446	346	961	792
No revDec	262	701	446	344	963	790

Table 4.3: Outcomes for the different configurations

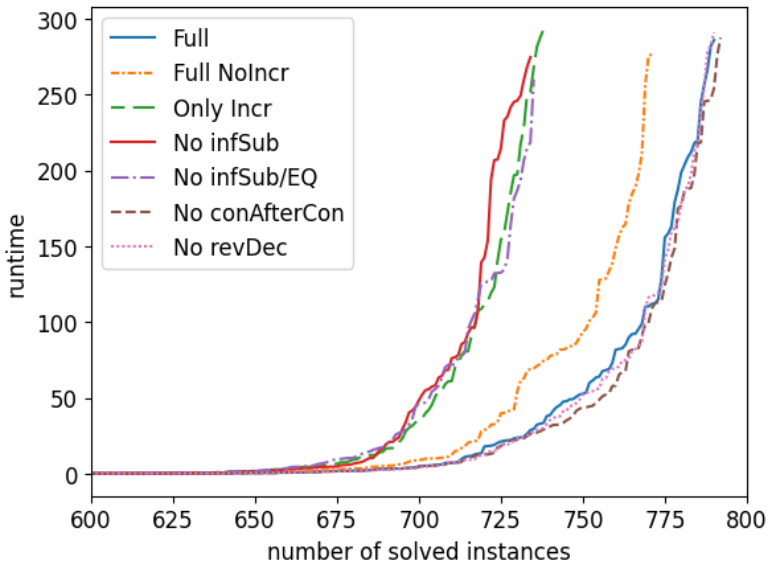


Figure 4.3: Performance profile for all configurations. Note that the x-scale starts at 600 as to make differences easier discernable. Around 600 instances are very easy to solve.

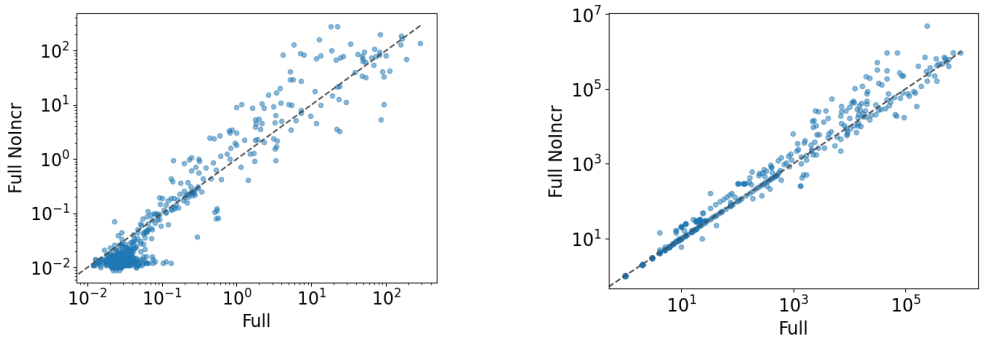
The comparison of the *Full* and the *Only Incr* configuration shows that with all optimizations enabled, FMplex manages to solve 52 more instances than without them. However, the only factor that seems to be responsible for the change in the number of solved instances is the presence of small infeasible subsets. This is especially noticeable in the number of solved UNSAT instances. All configuration that use small infeasible subsets and incrementality solve roughly 40 more UNSAT instances than those where no small infeasible subsets are used.

The only other factor that causes a significant change in the number of solved instances seems to be incrementality with the non-incremental configuration solving 19 less instances than the incremental one with the same settings. Meanwhile, the other optimizations introduced in this thesis do not seem to have much of an effect. Continuing after a local conflict even decreases the number of solved instances. The same is true for the different treatment of equalities and the reversing of decisions, as they barley change the number of solved instances.

4.3.1 Incremental vs. Non-incremental

In contrast to Stein's results from Table 2.1, our testing reveals that the incremental version performs better than the non-incremental version. However, this observation is not a result of the here introduced optimizations. In fact, the reason that Stein observed a decrease in performance for incrementality lies with a bug in the implementation. This bug caused a duplication of constraints in the incremental setting. Therefore, the incremental version had to deal with more constraints than the non-incremental version. Due to this, Stein [Ste22] observed a higher runtime and a higher number of generated constraints for incrementality.

This changes, when the bug is removed, as can be seen in Figure 4.4.



(a) Runtime in seconds

(b) Number of generated constraints

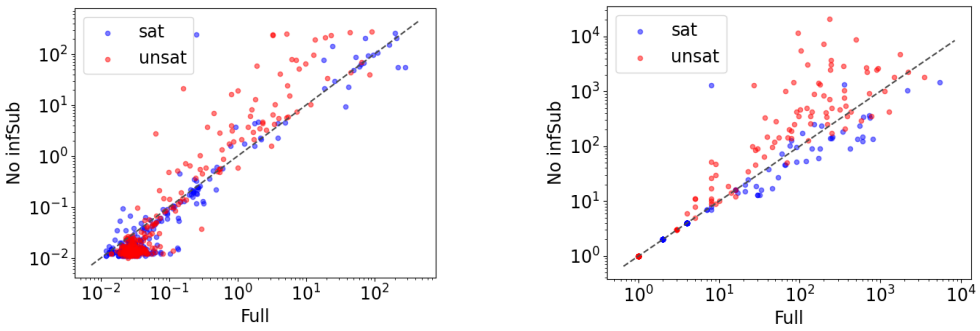
Figure 4.4: Comparison of *Full* and *Full NoIncr* configurations in terms of generated constraints and runtime in seconds on solved instances. Note that both plots use a logarithmic scale.

In reality the usage of incrementality decreases the runtime as well as the number of generated constraints. Intuitively, this makes more sense because previously generated constraints are saved after SAT is concluded. Hence, in subsequent runs of the algorithm they do not need to be generated again. The only disadvantage of the *Full* configuration, is that it produces more memory outs since the branch is saved.

However, the *Full NoIncr* version does not solve any instances that cause a memory out in the *Full* configuration.

4.3.2 Infeasible Subsets

The addition of smaller infeasible subsets makes the greatest improvement in terms of the number of solved instances. The reason for this increase in efficiency can also be seen in Figure 4.5b, which shows the number of times the FMplex solver needed to be called until satisfiability or unsatisfiability could be concluded.



(a) Runtime in seconds

(b) Number of theory calls

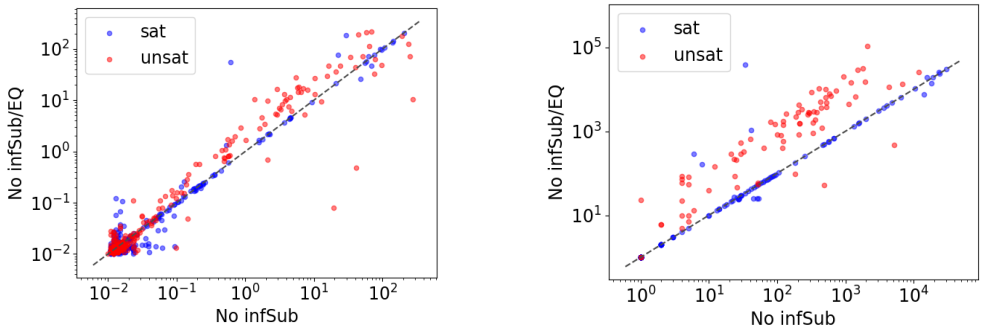
Figure 4.5: Comparison of *Full* and *No infSub* configurations in terms of runtime in seconds and the number of theory calls on solved instances. Note that both plots use a logarithmic scale.

Especially for instances, which are unsatisfiable the number of times the theory solver needed to be called is much lower, if infeasible subsets are enabled. This is due to the fact that with smaller infeasible subsets the SAT solver can exclude more assignments on the Boolean abstraction. Therefore, only fewer assignments need to be tested with the theory solver. The runtime on unsatisfiable instances, which can be seen in Figure 4.5a, is thereby also much lower.

Interestingly, there are quite a few cases, where satisfiable instances need less theory calls, when no smaller infeasible subsets are used. The reason for that could lie in the fact that due to the larger infeasible subsets the order in which the SAT solver checks the assignments is different. Through that, the SAT solver may reach a full satisfying assignment on the Boolean abstraction faster. However, while the *No infSub* configuration is able to find two satisfiable instances that time out in the *Full* configuration, the *Full* configuration is able to find 12 satisfiable instances that time out without infeasible subsets. Therefore, smaller infeasible subsets seem to be superior even for satisfiable instances.

4.3.3 Handling Equalities

The comparison of the *No infSub* configuration to the *No infSub/EQ* configuration provides the surprising conclusion that the different handling of equalities does not make much of a difference. In fact, the number of solved instances only changes by one, which is not significant since the runtime can always vary slightly depending on conditions during the execution. The only thing that changes is that the *No infSub*



(a) Runtime in seconds

(b) Number of local conflicts

Figure 4.6: Comparison of *No infSub* and *No infSub/EQ* configurations in terms of runtime in seconds and the number of local conflicts over solved instances. Note that both plots use a logarithmic scale.

configuration manages to identify 6 satisfiable instances more, while *No infSub/EQ* configuration manages to identify 7 unsatisfiable instances more. This is actually quite surprising, as the general trend shows that the *No infSub/EQ* configuration performs worse on unsatisfiable instances, as can be seen in Figure 4.6a.

There it can be observed, that the different treatment for equalities decreases the runtime for unsatisfiable instances. Meanwhile, satisfiable instances behave almost the same whether equalities are split or not. A possible reason for that is the fact that using equalities as eliminators removes the possibility of a conflict occurring on that level. Therefore, on that level only one eliminator needs to be tried out and less local conflicts are produced. This way the computation tree is smaller compared to the scenario where we split the equalities. For unsatisfiable instances this means that we might find a branch with a global conflict much faster than otherwise. This is supported by Figure 4.6b, which shows that the number of local conflicts occurring is much lower, if we do not split the equalities.

For satisfiable instances the number of local conflicts hardly changes. This may be the case, since for a satisfiable set of constraints the number of branches that need to be checked may be generally rather small. A reduction in the number of possible branches would then not lead to much of a speedup.

In general, we had hoped that a different treatment of equalities would lead to much more of an improvement. The main problem that stands against this is likely the fact that we can not conclude a global conflict, if an equality was eliminated with an inequality. Due to this, we only reach a global conflict, if we first eliminate the equalities and afterwards the inequalities. This likely prohibits the incrementality from working effectively because achieving any global conflict requires undoing the eliminator choice on the highest level, where an equality was eliminated with an inequality.

4.3.4 Continuing After a Local Conflict

The goal behind continuing after a local conflict was to find unsatisfiable instances faster by finding global conflicts sooner. However, this goal was not reached, instead this option causes the solver to solve even less unsatisfiable instances. In Figure 4.7

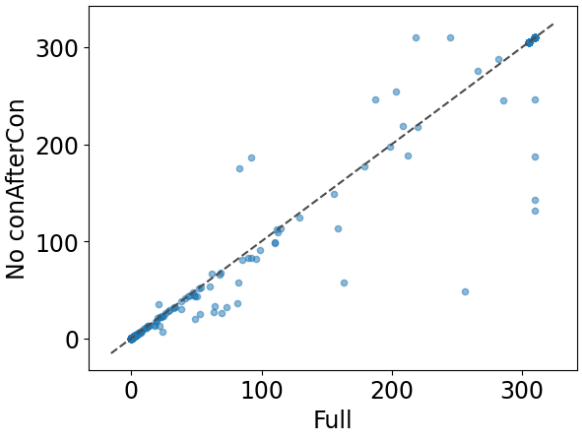


Figure 4.7: Comparison of *Full* and *No conAfterCon* configurations in terms of runtime in seconds over all instances.

the runtime for the *Full* and *No conAfterCon* configuration is shown.

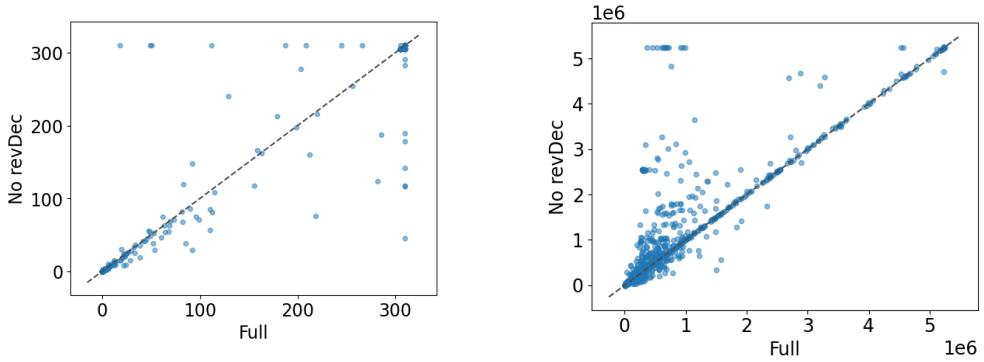
In almost all cases not continuing after a local conflict produces a better performance. There are only 6 instances, where the *Full* configuration is meaningfully faster. These are likely the only instances, where continuing after a conflict actually led to a global conflict. However, almost all instances are faster without this option. Therefore, we can conclude that following a branch that already has a local conflict almost never leads to a global conflict. Moreover, further testing has revealed that even, if we continue for more levels, the number of solved instances only goes down. Overall, it seems that continuing after a local conflict is not a worthwhile pursuit and there are only very few specific cases, where it is useful.

4.3.5 Reversing Decisions

Originally we assumed that the non-incremental version of FMplex outperforms the incremental version. Under that viewpoint the idea was to put more focus on the heuristic for variable choice and reverse variable decisions. However, the original assumption was caused by an implementation error. Thus, incrementality is much more helpful than previously thought and the heuristics role is not as important as it seemed. This leads to the fact that the reversing of decision has not much of an impact on the number of solved instances. In fact, the number of solved instances does not change at all. However, without reversing decisions the solver suffers from 14 more memory outs, although even in the *Full* version these instances time out.

Nonetheless, there are a few instances where the reversing of decisions has a significant impact on the memory consumption. This can be seen in Figure 4.8b.

There are 51 instances that use over 2GB more memory, if we do not reverse decisions. These instances all belong to the benchmark set `sc`, which is a subset of the SMT-LIB benchmark set. However, even with the reversing of decisions none of these instances were solved but instead timed out. Additionally, as can be seen in Figure 4.8a, this optimization does not cause a better runtime in general. On the contrary, the reversing of decisions seems to increase the runtime in more instances than it decreases them.



(a) Runtime in seconds

(b) Peak memory usage in kB

Figure 4.8: Comparison of *Full* and *No revDec* configurations in terms of runtime in seconds and peak memory usage in kilobyte over all instances.

It is likely that the large impact in the number of solved instances for the heuristics observed in Table 2.1 stems from the fact that all variables with only bounds in one direction are eliminated at the start. After all, this way a lot of constraints can be simply dropped. The variable choices after that may not strongly increase the efficiency by much. This could explain why the reversing of decisions has not much of a benefit, as we only reverse decisions, if new constraints are added in subsequent runs. Therefore, there is no variable that is unbounded in one direction, where this was not already the case in the previous run. Hence, by changing the variable, we only change the number of possible eliminators that need to be tried out. Combining that with the fact that we additionally lose everything below the level, where the variable has changed, it may no be worthwhile to do so. Consequently, there is no general benefit in reversing the decisions but specific instances profit from it, likely due to the fact that by chance a global conflict may be found faster than before. Furthermore, here only one fixed criteria for reversing was considered. Using different heuristics, when deciding to reverse a decision could lead to an improvement.

Chapter 5

Conclusion

5.1 Summary

In this thesis we presented various optimizations for the incremental FMplex method. This method chooses greatest lower bounds or smallest upper bounds to iteratively eliminate variables until either a satisfying assignment or a conflict that is not based on the eliminator choice is found. The computation can be thought of as a tree, where each node corresponds to the elimination of a variable with a child for every eliminator choice. In an incremental setting a branch that led to a satisfying assignment can be remembered and used in the computation for the next set of constraints. However, surprisingly past testing showed that leveraging the incrementality did not result in an increase in the number of solved instances.

Based on this fact we proposed a few ideas for optimization. We first introduced new data structures for the constraints to lower memory usage. Afterwards, an approach for reversing variable choices made in previous runs was presented, as to be able to change the variable order in an incremental setting. Furthermore, we considered continuing the computation on branches with conflicts to find indications for unsatisfiability faster. Moreover, an idea on how to combine equalities with inequalities and a different treatment of not-equal constraints was implemented. Finally, the generation of infeasible subsets was changed to create smaller sets and speed up the computation.

During the realization of these changes we were able to identify, that the incrementality had a problem in the previous implementation. We were able to fix that problem and conclude that incrementality actually outperforms a non-incremental approach. Meanwhile, the testing of the optimizations presented in this thesis produced rather poor results. The new constraint types needed more memory than the old type, due to missing optimizations in memory consumption. Additionally, only the generation of smaller infeasible subsets gave the method a significant boost in the number of solved instances. In comparison, the different treatment of equalities was able to reduce the runtime on unsatisfiable instances but not by a significant amount. The reversing of decisions did not provide much difference in the performance, which we traced back to the fact that the heuristic for variable choice has a lower impact than assumed. Furthermore, we judged that continuing on a conflicting branch does not lead to a proof for unsatisfiability as continuing after a conflict decreased the performance of the solver.

5.2 Future Work

While the new constraint types did not lead to an improvement, there is definitely the potential for further development. As explained in Section 3.1 a constraint can be remembered by only saving one coefficient for every variable that occurs in the constraint. If one is able to optimize the memory usage, while not increasing the complexity for combining the constraints, it should be possible to reduce the number of memory outs significantly.

Another chance to increase the performance of this method lies again with the equalities. Currently, eliminating an equality with an inequality instantly turns any resulting conflict into a local conflict. An alternative way to handle the resulting constraint, would be to use two different derivation coefficients. One for the same-bound combination and one for the upper-lower combination. Such a treatment may lead to a global conflict faster than treating everything like a local conflict.

Furthermore, the algorithm may profit more from using the reversing of decisions, if we use a stricter measure, when identifying possible candidates for reversing. This way we can minimize the losses made by resetting a branch due to a new variable choice. Another option would be to change the heuristics all together. Following the VSIDS idea from SAT solving, we could preferably choose variables and constraints that have been involved in conflicts recently.

Bibliography

[ÁK17] Erika Ábrahám and Gereon Kremer. SMT solving for arithmetic theories: theory and tool support. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 1–8. IEEE, 2017.

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

[BNOT06] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning: 13th International Conference, LPAR 2006*, pages 512–526. Springer, 2006.

[CKJ+] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika. Ábrahám. CArL: Computer ARithmetic and Logic Library. `https://github.com/smtrat/carl`.

[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 *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368. Springer, 2015.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[Dan72] George B Dantzig. Fourier-Motzkin elimination and its dual. Technical report, Standford University, 1972.

[Dan90] George B Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. ACM, 1990.

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

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[Fou24] Joseph Fourier. Histoire de l'académie, partie mathématique. *Mémoires de l'Académie des sciences de l'Institut de France*, 7, 1824.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL (T): Fast decision procedures. In *Computer Aided Verification: 16th International Conference, CAV 2004*, pages 175–188. Springer, 2004.

[Jos12] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 2nd edition, 2012.

[KBD+17] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[Kob21] Paul Kobialka. Connecting simplex and Fourier-Motzkin into a novel quantifier elimination method for linear real algebra. Master's thesis, RWTH Aachen University, 2021.

[Lev73] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[LV11] Lingyi Liu and Shobha Vasudevan. Efficient validation input generation in RTL by hybridized source code analysis. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[Mot36] Theodore Samuel Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. Azriel Press, 1936.

[SG09] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234. Association for Computing Machinery, 2009.

[Ste22] Svenja Stein. An incremental adaption of the FMPlex method for solving linear real algebraic formulas. Bachelor's thesis, RWTH Aachen University, 2022.