Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

MASTER OF SCIENCE THESIS

CONNECTING SIMPLEX AND FOURIER-MOTZKIN INTO A NOVEL QUANTIFIER ELIMINATION METHOD FOR LINEAR REAL ALGEBRA

Paul Kobialka

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Arie Koster

Additional Advisor:
Jasper Nalbach

Aachen, 16.07.2021

Abstract

In order to decide the satisfiability of a *quantifier-free linear real arithmetic* (*QF_LRA*) formula either the *Simplex* or the variable elimination algorithm by *Fourier-Motzkin* is used. This thesis presents a novel approach *FMPlex* with similarities to Simplex and Fourier-Motzkin. Compared to Fourier-Motzkin as variable elimination procedure, FMPlex has singly exponential complexity instead of doubly exponential at the cost of introducing disjunctions in the projection result. Compared to Simplex, multiple branches with each short computations instead of a long pivoting chain are constructed. Additionally, an advanced not-equal constraint handling for FMPlex and Simplex is presented. This method can decide the satisfiability of a QF_LRA formula with linear not-equal constraints without combinatorial blow-up.

Acknowledgements

I want to thank Prof. Dr. Erika Ábrahám for the possibility to write this thesis and the continuous, flawless support. It was an interesting and exciting time working on a thesis like this. Also, I would like to thank her for her support throughout my studies and my academical path. I also want to thank Prof. Dr. Arie Koster for being the additional examiner.

I want to thank Jasper Nalbach for his relentless support and the many meetings we had.

Thanks to my family for the whole support and patience I received until now.

Contents

Chapter 1

Introduction

At the heart of computer science lies the drive to discover general and fast algorithms. One does not aim for tailor-made algorithms, but advanced general ones. Linear programs are used in optimization for a long time. They gained a lot of attention as many combinatorial problems can be stated as linear programs. For many of the combinatorial problems occurring in practice, the existence of a solution is already sufficient. Meaning, the solution not always needs to be optimal. A feasible solution is already sufficient. In the *Satisfiability* problem a satisfying assignment for a Boolean formula is searched. It was the first problem proven to be *NP-complete*, [Coo71, Lev73]. The field of *Satisfiability Modulo Theories* extends the boolean formulas in SAT by further arithmetics. This thesis considers the *linear real arithmetic*. This means, Boolean combinations over linear constraints are considered.

The first algorithm to decide whether a solution to a conjunction of linear constraints exists was proposed by Fourier in 1827 [Fou24, Dan72]. As it was rediscovered in 1936 by Motzkin in [Mot36], the algorithm is called *Fourier-Motzkin*. At the time it was the only way to solve such kind of problems. However, it has a huge drawback which rendered it computably infeasible for real-world problems. It eagerly combines all lower and upper bounds until all variables are eliminated or a conflict is found. Through this, the size of the constraint set increases doubly exponential within the execution. Even for small problems, this blow-up is computably not feasible.

A huge milestone in this development marks the discovery of *Simplex* algorithm by Dantzig in 1947, [Dan90]. Simplex itself was based on the idea to traverse the extreme points of the underlying polyhedron until an optimal solution is found. How to traverse those extreme points is decided by a *pivoting* rule. The selection of the pivoting rule has a huge impact on the running time of Simplex. However, so far no pivot rule with polynomial time is found, [AZ96]. Nevertheless, Simplex performs quite well in practical applications. This was for a long time subject of research. It showed that Simplex is highly probable to have efficient running time. The chances of hitting an exponential input instance are quite low, [ST04].

In this thesis, a new approach to solve a set of linear constraints is proposed. It uses the combinatorial aspect from Fourier-Motzkin and the extreme-point traversal from Simplex. By doing so it achieves a singly exponential running time.

This work is structured as follows. In Chapter 2 the preliminaries are presented. Firstly, the theoretical setting is defined. Additionally, linear programs and SAT modulo theories are introduced. Secondly, the Fourier-Motzkin algorithm is pre-

sented. Thirdly, it is extended by the *Imbert Acceleration Theorems* discovered in the last century. Those improvements reduce the number of constraints the algorithm considers. Afterwards is the Simplex algorithm presented. Every presented algorithm is illustrated by a running example.

The adapted approach, called *FMPlex*, is presented in Chapter 3. It is defined and its properties are proven. Additionally, its functionality is extended to strict constraints and not-equal constraints.

The way FMPlex works can be transferred to Simplex by a new pivoting heuristic. The heuristic and a novel not-equality handling for Simplex is presented in Chapter 4. All presented algorithms are evaluated and their practical behaviour is analyzed in Chapter 5. The chapter also gives some expectations for future work.

Followed by Chapter 6 which concludes this thesis.

Chapter 2

Preliminaries

2.1 Satisfiability Modulo Theories

In the *SAT* problem it is to decide whether for a given Boolean formula φ there exists an assignment for its variables such that the whole formula evaluates to true. It was the first problem shown to be *NP complete* independently by Cook in [Coo71] and Levin in [Lev73]. Due to that, it is from great theoretical interest. *Satisfiability Modulo Theories* (*SMT*) are a logical extension to the SAT problem. Hereby, the *existential fragment of first order logic* with respect to given theories is considered. This means that single constraints from one or more theories are connected by logical operators. It is then checked whether constraints can be satisfied such that the formula evaluates to true. A detailed definition can be found in [BHvM09].

It showed that many practical problems do not require an optimized solution with respect to some target function but the sole existence of a solution is of interest. The decision problem and no optimization problem needs to be solved. In those cases, one wants to prove the existence of a solution that exceeds Boolean algebra. One example are scheduling problems in which are modelled by linear constraints. The interest lies in the existence of the solution, not in an optimization problem. This thesis considers *linear real arithmetic* which is presented in Section 2.2. In SMT solving many different arithmetics are of interest: e.g. linear real, linear integer, non-linear, uninterpreted functions and many more. An overview is given in the summary over the last years of the SMT competition in [WCD+19]. This thesis focuses only on linear real arithmetic.

2.2 Linear Real Arithmetic

Definition 2.2.1 (Linear Real Arithmetic)**.** *Linear real arithmetic is the first-order theory with signature {0,1,+,<} and the domain being the reals* $\mathbb{R}$*.*

Definition 2.2.2 (Linear Constraint)**.** *Let* $x_1, \ldots, x_n$ *be real valued variables,* $a_1, \ldots, a_n \in \mathbb{Q}$ *coefficients,* $b \in \mathbb{Q}$ *a constant and* $\bowtie \in \{<, >, =, \leq, \geq\}$ *a relation symbol. Then* $a_1x_1 + \ldots + a_nx_n \bowtie b$ *is a linear constraint.*

Negation and conjunction are given, other logical operators as $\vee, \oplus$ and $\rightarrow$ can be constructed.

As later parts of this thesis use results from *linear optimization*, linear programs and their notation are introduced. In addition, the connection between QF_LRA formulas and linear programs is given.

A *linear program* is an optimization problem where one wants to maximize the gain given by a vector $c \in \mathbb{Q}^n$ such that m constraints $a_1, \ldots, a_m \in \mathbb{Q}^n$ are satisfied by the solution $x \in \mathbb{R}^n$:

$$\max \sum_{i}^{n} c_i \cdot x_i$$
$$\text{subject to } \langle a_j, x \rangle \leq b_j \text{ for all } j \in \{1, \ldots, m\}$$

The normal form of a linear program gathers the single constraints as rows in a matrix $A \in \mathbb{Q}^{m \times n}$ and the constants b_j in a vector $b \in \mathbb{Q}^m$. The linear program is then given as:

$$\max \sum_{i}^{n} c_i \cdot x_i$$
$$\text{subject to } Ax \leq b.$$

Let φ be a QF_LRA formula with only conjunctions of linear constraints $a_1, \ldots a_n$. Then φ can be written as:

$$\bigwedge_{i}^{n} a_i.$$

Thus, φ can also be stated the constraints of a linear program, $Ax \leq b$, given as system (A,b). This lies in the fact that one is not interested in a maximization but rather the pure existence of a solution. In this utilization the maximization vector c is not needed.

Let a be a linear constraint over variables $x_1, \ldots x_n$ and $\mathcal{X} = \{x_1, \ldots, x_n\}$. A function $\alpha : \mathcal{X} \to \mathbb{R}$ is called *assignment for variables in* $\mathcal{X}$. If the assignment α satisfies a, we write $\alpha \models a$. In case the assignment does not satisfy a, we write $\alpha \not\models a$. Within this thesis refers *partial assignment* to a function $\alpha : \mathcal{X}' \to \mathbb{R}$ with $\mathcal{X}' \subset X$. Thus, not all variables are assigned.

2.2.1 Geometric Interpretation

This section lays out the foundation of the geometric interpretation of this work. As we will later see, many of the used algorithms have very nice geometric interpretations. To be able to conduct the necessary analysis, a few definitions are needed. The definitions are according to [WN99] and [Axl97].

Definition 2.2.3 (Linear Combination)**.** *Given a set $S \subseteq \mathbb{R}^n$, the point $x \in \mathbb{R}^n$ is a linear combination over S if there exists points $v_1, \ldots, v_n$ in S with coefficient $\lambda_1, \ldots, \lambda_n$ such that $x = \sum \lambda_i v_i$.*

If no such linear combination over points in S exists to produce a point x, x is called *linearly independent* to the points in S.

Definition 2.2.4 (Convex Combination)**.** *Given a set of points $S \subseteq \mathbb{R}^n$, a point $x \in \mathbb{R}^n$ is a convex combination of points in S if there exists a linear combination $x = \sum \lambda_i v_i$ with $\sum_i \lambda_i = 1$.*

This definition means, that a set is convex iff for all two points within the set, the line between those points is fully contained in the set. With this, the empty set $\emptyset$ is a convex set and the whole space $\mathbb{R}^n$ is also a convex set. A nontrivial convex set is depicted in Figure 2.1b.

Definition 2.2.5 (Convex Hull)**.** *Given a set of points S, the convex hull of S, denoted as $conv(S)$ is the set of all convex combinations of points in S.*

The next definition builds the geometric interpretation for linear programs/constraints.

Definition 2.2.6 (Polyhedron)**.** *A polyhedron $P \subseteq \mathbb{R}^n$ is the set of points satisfying a finite number of linear inequations, thus $P := \{x \in \mathbb{R}^n : Ax \leq b\}$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.*

Theorem 2.2.1. *The cut of n convex sets $S_1, \ldots, S_n$ is convex.*

Theorem 2.2.2. *A polyhedron is a convex set.*

By now, it is shown that a set of linear constraints can be visualized as convex polyhedron. It is important to note that the *extreme points* play a special role in later algorithms.

Definition 2.2.7 (Extreme Points)**.** *Let P be a polyhedron. Then $x \in P$ is an extreme point of P if there are no $x_1, x_2 \in P$ with $x_1 \neq x_2$ such that $x = \frac{x_1 + x_2}{2}$. Thus, x is not part of a convex combination besides the one that only contains itself.*

In Figure 2.1b the extreme points are marked as blue points numbered from A to E. The running example of this thesis is given by the following QF_LRA formula. The formula is then also displayed in its matrix form. This problem is revisited multiple times throughout this thesis for every presented algorithm.

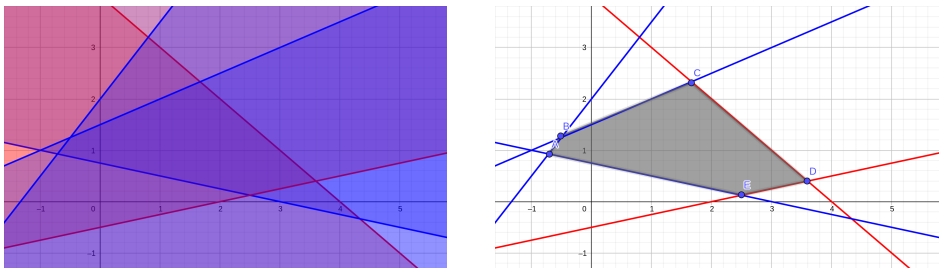
Example 2.2.1 (QF_LRA Formula)**.** *Let φ be a QF_LRA formula defined by*

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

The corresponding matrix representation is given by

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

A visualization of the constraints and the corresponding polyhedron is given in Figure 2.1. On the left side in Figure 2.1a the visualization of the constraints is given. Hereby are constraints with a positive coefficient on x_1 colored red and with a negative one colored blue. For both colours the valid region is shaded in a darker tone. On the right side, in 2.1b, the visualization of the convex polyhedron spanned by the constraints is given. The points A to E spanning the convex hull are marked. As the polyhedron is not empty, we know that there exists a solution satisfying all constraints. The first algorithm to find this solution is given in Section 2.3.



(a) Constraints (b) Polyhedron

Figure 2.1: Constraints and Polyhedron

2.3 Fourier-Motzkin Variable Elimination

The *Fourier-Motzkin* (FM) elimination method was firstly published by Joseph Fourier in 1827 in [Fou24]. It was independently rediscovered by Motzkin in 1936 in [Mot36] and published in modern English by Dantzig in 1972 [Dan72]. Given is a system (A,b) of linear inequalities over variables $x = (x_1, \ldots, x_n)$. Hereby is A a rational $m \times n$ matrix, $A \in \mathbb{Q}^{m \times n}$, and b an m dimensional vector, $b \in \mathbb{Q}^m$. Let $x = (x_1, \ldots, x_n)$ be a sequence of real variables. This sequence is used as variable ordering on the variables $x_1, \ldots, x_n$, thus $x_1 < x_2 < \cdots < x_n$. Now to deduce the satisfiability of the system (A,b), one can eliminate all variables and check if a conflict occurs. For the first variable in the ordering, x_1, A is divided in three sets of inequalities: U, L and N. U is the set of upper-bound constraints, i.e. x_1 has a positive coefficient in constraints in U. L is the set of lower-bound constraints, i.e. x_1 has a negative coefficient. The remaining constraints in which x_1 does not appear, i.e. its coefficient is zero, are in N, the "no bound" constraints. Assume $u \in U$ corresponds to the j-th row in (A,b), $u := (a_j x \leq b_j)$. As this inequality is an upper bound on x_1, the row can be rewritten to:

$$a_{j1} x_1 \leq \sum_{k=2}^{n} -a_{jk} x_k + b_j.$$

In this notation, the coefficient a_{ji} is the entry in the j-th row and i-th column of A. Equivalently, in case $l \in L$ corresponds to the j-th row of (A,b), it can be written as

$$\sum_{k=2}^{n} a_{jk} x_k - b_j \leq a_{j1} x_1.$$

This thesis makes often use of the comparison between inequalities. To make those comparisons easier to read, the connection between constraints in U, L, N and rows of A is dropped. Constraint $l \in L$ is written as $l_1 x_1 + \cdots + l_n x_n \leq b_l$, corresponding to any row in (A,b). In the same way upper bounds $u \in U$ are written as $u_1 x_1 + \cdots + u_n x_n \leq b_u$. For the coefficients holds $l_1, \ldots, l_n, u_1, \ldots, u_n, b_l, u_l \in \mathbb{Q}$.

Let $l = (l_1 x_1 + \cdots + l_n x_n \leq b_l)$ be a lower bound and $u = (u_1 x_1 + \cdots + u_n x_n \leq b_u)$ an upper bound for variable x_i over variables $x_1, \ldots, x_n$. This means, the coefficient of x_i in l, i.e. i-th element l_i, is negative and in u positive. Thus, l can be written as $\sum_{0 \leq j, j \neq i}^{n} l_j x_j - b_l \leq l_i x_i$ and u as $u_i x_i \leq \sum_{0 \leq j, j \neq i}^{n} -u_j x_j + b_u$. Combining both

inequalities by multiplying l with u_i and u with l_i gives us:

$$|u_i| \cdot (\sum_{0 \leq j, j \neq i}^{n} l_j x_j - b_l) \leq |l_i| \cdot (\sum_{0 \leq j, j \neq i}^{n} -u_j x_j + b_u).$$

Or equivalently, dividing l by l_i and u by u_i gives:

$$\frac{1}{|l_i|} \cdot (\sum_{0 \leq j, j \neq i}^{n} l_j x_j - b_l) \leq \frac{1}{|u_i|} \cdot (\sum_{0 \leq j, j \neq i}^{n} -u_j x_j + b_u).$$

In terms of vector operations resembles this comparison the elimination of x_i by combining the vector representing u with the vector representing l. This operation is not restricted to the previously suggested factors. Other multiplicities like the greatest common divisor are also possible as long as x_i has the same coefficient in both vectors. The comparison between two bounds $l = (\sum_i^n l_i x_i \bowtie_l b_l)$ and $u = (\sum_i^n u_i x_i \bowtie_u b_u)$,$l \bowtie u$ with $\bowtie \in \{<, \leq\}$, is defined as

$$|u_i| \cdot (\sum_{0 \leq j, j \neq i}^{n} l_j x_j - b_l) \bowtie |l_i| \cdot (\sum_{0 \leq j, j \neq i}^{n} -u_j x_j + b_u).$$

Hereby depends $\bowtie$ on $\bowtie_l$ and $\bowtie_u$. This comparison is used in the later to illustrate coherences between constraints.

Throughout this thesis refers the operation COMBINE(l,u,x) to the elimination of the common variable x. Hereby are l,u constraints over n variables. The variable x is part of the input in order to determine what kind of bounds l and u on x are. In later parts becomes this determination more important. The operation returns a constraint where x is eliminated, as previously demonstrated. The pseudocode for the Fourier-Motzkin elimination is given in Algorithm 1.

Algorithm 1 Fourier-Motzkin Algorithm

```
1:  procedure FOURIER-MOTZKIN ALGORITHM(inequation system S = (A,b) and
    variable ordering x̄ = (x_1, ..., x_n))
2:      for Var x_i := x̄_i do
3:          Partition S in L,U,N
4:          S := COMBINESETS(L,U,x) ∪ N
5:          if conflict in S then
6:              return UNSAT
7:      return SAT
8:  procedure COMBINESETS(L,U,x)
9:      Constraint set S = ∅
10:     for l ∈ L do
11:         for u ∈ U do
12:             S := S ∪ {COMBINE(l,u,x)}
13:     return S
```

For later proofs, one insight about the nature of created inequations is important. The following theorem states the structure of all intermediate inequality.

Theorem 2.3.1 (Intermediate Inequalities)**.** *Every intermediate inequality is a linear combination with positive coefficients of original constraints.*

Proof. The proof is done by induction on n, the steps in the FM algorithm.

For $n = 0$: Constraints form their own positive linear combination with coefficient 1.

Induction step: $n \to n+1$. In the $n+1$-th step two equations with positive linear combinations are combined to a new inequation. To do so, both linear combinations are multiplied each with a positive coefficient and then added up. This leads to another positive linear combination over elements in the original constraints. □

The Fourier-Motzkin algorithm also gives another point of view to solve the system (A,b). Let A again be divided into U, L and N. Instead of solving the whole system, one can also solve the *reduced* system (A',b'). In the reduced system are the variables in $x_2, \ldots, x_n$ and (A', b') computed by comparing $(l \leq u) \cup N$ for $l \in L, u \in U$. This results in solving the system $(L \leq U) \cup N$:

$$\begin{aligned} l &\leq u \text{ for } l \in L, u \in U \\ n &\leq 0 \text{ for } n \in N \end{aligned} \tag{2.1}$$

Afterwards an assignment for x_1 is found which satisfies

$$\max_{l \in L} l \leq x_1 \leq \min_{u \in U} u.$$

Such an assignment for x_1 can always be found iff there exist an assignment for $x_2, \ldots, x_n$ satisfying (A',b'). The proof for the Fourier-Motzkin Algorithm and the reduced system can be found in [Dan72]. As all variables can be iteratively eliminated, a further statement about the satisfiability of the original system can be made. This theorem will prove helpful in later parts of this work.

Theorem 2.3.2 (Feasibility Theorem)**.** *The system (A,b) is solvable iff there are no non-negative weights $\gamma_1, \ldots, \gamma_m$ such that*

$$\sum_{i=1}^{m} \gamma_i \cdot b_i > 0 \text{ and } \sum_{i=1}^{m} \gamma_i \cdot a_{ij} = 0, \text{ for } j = 1, \ldots, n.$$

This means, that the system is solvable if and only if there is no linear combination with non-negative coefficients leading to a contradiction, $0 \leq d$ with $d > 0$. The proof was firstly proposed by Kuhn in 1965 in [Kuh56].

Proof. This proof is done by contradiction. Assume that α is an assignment for $x = (x_1, \ldots, x_n)$ satisfying the system (A,b). Additionally, assume that coefficients $\gamma_1, \ldots, \gamma_n \geq 0$ exist which satisfy:

$$\sum_{i=1}^{m} \gamma_i \cdot b_i > 0 \text{ and } \sum_{i=1}^{m} \gamma_i \cdot a_{ij} = 0, \text{ for } j = (1, \ldots, n).$$

From the system (A,b) can now a contradiction be constructed.

$$\sum_{j=1}^{n} (\sum_{i=1}^{m} \gamma_i \cdot a_{ij}) \cdot x_j \geq \sum_{i=1}^{m} \gamma_i \cdot b_i.$$

When inserting the conditions $\sum_{i=1}^{m} \gamma_i \cdot a_{ij} = 0$ and $\sum_{i=1}^{m} \gamma_i \cdot b_i > 0$, the term evaluates to $\sum_{j=1}^{n} 0 \cdot x_j \geq \sum_{i=1}^{m} \gamma_i \cdot b_i > 0$. Thus, a contradiction to the satisfiability of the system could be constructed by combining elements from the original system.

Hence, the condition is necessary. Now, assume that no solution for the system exists, thus it is unsatisfiable. As previously described: every new constraint is a linear combination with positive coefficients of the previous system. Leading to the fact, that every inequation in an intermediate system is a linear combination with positive coefficients over elements of the initial system. Meaning that the reason for unsatisfiability can be deduced in form of $\gamma_i < 0$ by non-negative linear combinations of the initial system. This proves the theorem. □

From Theorem 2.3.2 one can fastly conclude the famous Lemma from Farkas for linear optimization. It will also be used in the later presented novel algorithm in Chapter 3. Farkas presented his lemma firstly in [Far02], written in German, an overview over its variants and the in the following used variant can be found in [Sch98].

Lemma 2.3.3 (Farkas Lemma)**.** *Let (A,b) be a system of linear inequalities. Then there exists a solution $x \in \mathbb{R}^n$ if and only if $\langle \gamma, b \rangle \geq 0$ for every $\gamma \in \mathbb{R}^n$ with $\gamma A = 0$.*

Due to the similarity with Theorem 2.3.2 the proof is omitted. One of the reasons why the FM algorithm is rarely used in practice is rooted in the following theorem.

Theorem 2.3.4 (Fourier-Motzkin Running Time)**.** *The Fourier-Motzin algorithm has a doubly exponential running time.*

Proof. Let (A,b) with $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$ be the system solved by the FM algorithm. It is possible that U and L completely cover A, i.e. N is empty, and U,L have size $\frac{m}{2}$. In this case, the FM algorithm produces in every iteration $(\frac{m}{4})^2$ new constraints. As this behavior might occur for all n variables, the algorithm creates in the worst case $\mathcal{O}((\frac{m}{4})^{2^n})$ many constraints. Thus, it has a doubly exponential running time. □

Due to its blow up caused by its doubly exponential complexity, FM is infeasible on many practical instances. Thus, the Simplex algorithm presented in Section 2.4 is mainly used in practice.

Example 2.3.1 (Fourier-Motzkin Algorithm)**.** *This example demonstrates the Fourier-Motzkin algorithm by solving the previous example from Section 2.2. The constraints are given by*

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

The previous example already visualized that the example is SAT*, i.e. there exists an assignment for x and y such that all constraints are satisfied. Let the variable ordering be $x < y$, this means that firstly x is eliminated and secondly y. To do so, the constraints are divided in sets U, L and N. Due to their positive coefficients, the first and second constraint are in U. The lower bounds, the constraints with negative*

x coefficient, are in L. Thus, the remaining constraints are in L and N is empty. The following division results:

$$U := \begin{pmatrix} x+y & \leq & 4 \\ x-4y & \leq & 2 \end{pmatrix} \begin{matrix} (U_1) \\ (U_2) \end{matrix}$$

$$L := \begin{pmatrix} -x-4y & \leq & -3 \\ -x+2y & \leq & 3 \\ -3x+2y & \leq & 4 \end{pmatrix} \begin{matrix} (L_1) \\ (L_2) \\ (L_3) \end{matrix}$$

In order to eliminate x, all upper bounds need to be combined with all lower bounds :

$$\forall u \in U, \forall l \in L : l \leq u.$$

The first new equation is formed by combining U_1:

$$\begin{array}{rrcl} & x+y & \leq & 4 \\ \iff & x & \leq & 4-y \end{array}$$

with L_1:

$$\begin{array}{rrcl} & -x-4y & \leq & -3 \\ \iff & -4y+3 & \leq & x \end{array}$$

to

$$\begin{array}{rrcl} & -x-4y & \leq & 4-y \\ \iff & -3y & \leq & 1 \end{array}$$

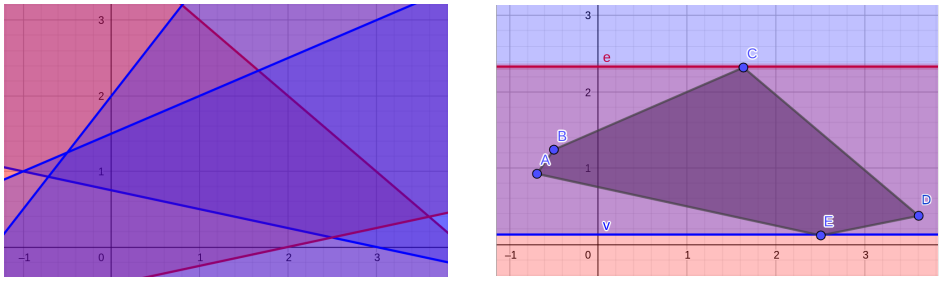
This procedure is then repeated for all remaining combinations between U_1, U_2 and L_1, L_2 and L_3. The following 6 new equations are created and form the new system:

$$\begin{pmatrix} -3y & \leq & 1 \\ 3y & \leq & 7 \\ 5y & \leq & 16 \\ -8y & \leq & -1 \\ -2y & \leq & 5 \\ -10y & \leq & 10 \end{pmatrix} \begin{matrix} (\text{COMBINE}(L_1, U_1, x)) \\ (\text{COMBINE}(L_2, U_1, x)) \\ (\text{COMBINE}(L_3, U_1, x)) \\ (\text{COMBINE}(L_1, U_2, x)) \\ (\text{COMBINE}(L_2, U_2, x)) \\ (\text{COMBINE}(L_3, U_2, x)) \end{matrix}$$

The COMBINE *operation on the right side states how the corresponding constraint was created. No equation from the previous system is copied as N is empty. This example also gives a nice visualization for the Fourier-Motzkin algorithm. The elimination of a variable can be seen as a projection of the polyhedron on the remaining variables. This connection is depicted in Figure 2.2. Meaning, the polyhedron spanned by the constraints, Figure 2.2a, is projected on its Y-axis. The projection is shown in Figure 2.2b. For a better overview only the dominating upper and lower bound is depicted. One can see that the new constraints always go through the cut of their generating constraints. The combination of U_1 with L_2 forms the smallest upper bound, $3y \leq 7$ and U_2 with L_1 generate the largest lower bound, $-8y \leq -1$. In the next step the remaining variable y is eliminated. To do so, the equations are again divided into U and L.*

$$U' := \begin{pmatrix} 3y & \leq & 7 \\ 5y & \leq & 16 \end{pmatrix} \begin{matrix} (U_1') \\ (U_2') \end{matrix}$$

$$L' := \begin{pmatrix} -3y & \leq & 1 \\ -8y & \leq & -1 \\ -2y & \leq & 5 \\ -10y & \leq & 10 \end{pmatrix} \begin{matrix} (L_1') \\ (L_2') \\ (L_3') \\ (L_4') \end{matrix}$$



(a) Polyhedron (b) Projection

Figure 2.2: Polyhedron and Projection

Again, all possible combinations between upper and lower bounds are formed:

$$\begin{pmatrix} 0 & \leq & \frac{8}{3} \\ 0 & \leq & \frac{53}{24} \\ 0 & \leq & 4.83 \\ 0 & \leq & \frac{10}{3} \\ 0 & \leq & \frac{53}{15} \\ 0 & \leq & \frac{123}{40} \\ 0 & \leq & 5.7 \\ 0 & \leq & 4.2 \end{pmatrix} \begin{matrix} (\text{COMBINE}(L_1', U_1', y)) \\ (\text{COMBINE}(L_2', U_1', y)) \\ (\text{COMBINE}(L_3', U_1', y)) \\ (\text{COMBINE}(L_4', U_1', y)) \\ (\text{COMBINE}(L_1', U_2', y)) \\ (\text{COMBINE}(L_2', U_2', y)) \\ (\text{COMBINE}(L_3', U_2', y)) \\ (\text{COMBINE}(L_4', U_2', y)) \end{matrix}$$

All variables were eliminated and no conflict was found, all constraints in the final system are satisfied. Thus, the system is indeed satisfiable. In a backward pass, one can now construct a solution. Firstly, a solution for y is constructed. As depicted in Figure 2.2 is y upper bounded by $\frac{7}{3}$ and lower bounded by $\frac{1}{8}$. A value can now be chosen arbitrarily from $[\frac{1}{8}, \frac{7}{3}]$. For simplicity $y = 1$ is chosen. Inserting this partial solution into the original system results in the following constraints on x:

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

The variable x is required to be in the interval $[-\frac{2}{3}, 3]$. Let x be set to 0. When inserting this assignment, $x = 0$ and $y = 1$ into the initial system, all constraints are satisfied and thus a valid solution is found. Herewith, we did not solely graphically show that the problem is satisfiable but also algorithmically with the Fourier-Motzkin Algorithm.

As the previous example shows, the number of constraints in the intermediate systems grows rapidly. To delay this growth as long as possible, improvements were made.

2.3.1 Strict Inequations

The Fourier-Motzkin algorithm can be extended to strict constraints, i.e. $\sum_i^n a_i x_i < b$. Let constraint u be an upper bound on x_i:

$$x_i \bowtie_u \frac{1}{|u_i|} \cdot (\sum_{0 \leq j, j \neq i}^{n} -u_j x_j + b_u)$$

and l be a lower bound on x_i:

$$\frac{1}{|l_i|} \cdot (\sum_{0 \leq j, j \neq i}^{n} l_j x_j - b_l) \bowtie_l x_i.$$

Hereby is $\bowtie_l, \bowtie_u \in \{<, \leq\}$. The definition of the comparison between the two bounds needs to be updated:

$$\text{COMBINE}(l,u,x) := \begin{cases} l \leq u, & \text{if } \bowtie_l \text{ is } \leq, \bowtie_u \text{ is } \leq \\ l < u, & \text{otherwise} \end{cases}$$

By doing so, the equal part of the strict constraint is also excluded in the combined constraint.

2.3.2 Imbert Acceleration

As the Fourier-Motzkin algorithm has a doubly exponential running time, it is of great interest to discover unneeded/redundant inequalities as early as possible. Through this, one tries to shift the blow-up of the constraints as far as possible. In the late 1990's several algorithms to detect such redundant inequalities were proposed. To do so, different approaches were chosen. While Kohler proposed in [Koh67] an algorithm based on the rank of the constraint system, Chernikov showed in [Che63] a criterion based on comparisons of histories.

In the end, Imbert proposed a local criterion in [Imb90] and showed in [Imb93] that the proposals by Kohler and Chernikov are indeed equivalent. The big advantage of Imbert's so-called *Acceleration Theorems* is that they are local criteria. To decide the redundancy by Kohler or Chernikov, the new inequation is compared to the already existing inequations. Considering the doubly exponential blow-up, both operations become rather costly. However, Imbert's redundancy criterion can locally be decided. This means, that the redundancy of a given inequation is decided by only considering the information of this one inequation and not taking the whole system into consideration. For Imbert's acceleration theorems, some more notation is needed. A more detailed explanation can be found in [Imb90] and [Imb93].

In addition to the inequation system, the improvements make use of more information than only the inequation. Not only the inequation itself but also its origin, the way it was produced, is used. This origin is called *historical subset* or *history*, it contains the indices of the elements in linear combination forming the new constraint. For every inequation c in the system, H_c denotes its *history*. Let i_c be the index of constraint c. For all of the original constraints is H_c defined as $\{i_c\}$, the set of its own index. As original constraints, they do not have a history besides themselves. When combining inequations l_1 and u_1 to inequation c', their histories are merged, $H_{c'} := H_{l_1} \cup H_{u_1}$.

Furthermore, the eliminated variables within the algorithm are also further divided. Let $x_1, \ldots, x_k$ be the variables eliminated from the original system. It is important to note that the algorithm might need less than k steps to eliminate those variables. They are called *officially eliminated* and form the set of officially eliminated variables, O_k. This set is now further divided. Let c be a intermediate constraint. We can now divide O_k in three disjoint sets: *effectively* eliminated, *implicitly* eliminated and *other* variables. The set of *effectively* eliminated variables for constraint c, E_c, contains variables who's official elimination produces initial or intermediate inequations used in the combination for c. A variable is in the set of *implicitly* eliminated variables for c, I_c, if it satisfies three conditions: it occurs in at least one inequation of H_c, it does not occur in c and it is not effectively eliminated for c. Those conditions ensure that (1) the variable was once an element of an ancestor of c, (2), was then removed, (3), but not by the elimination process. This last distinction is important, as the three sets need to be disjoint. It is important to note, that in practice there might be variables in I_c who are not in O_k. This is the case when one combination eliminates a variable but this variable occurs in other constraints in which it is not eliminated. All remaining variables who are not in $E_c \cup I_c$ are called *other* variables. As they are not needed in the following, their set remains unnamed.

Example 2.3.2. *To demonstrate the different classes of eliminated variables consider the following modified example.*

$$\begin{pmatrix} x+y & \leq & 4 \\ -1x-4y & \leq & -3 \\ -1x-y & \leq & 0 \end{pmatrix} \quad \begin{matrix} (1; & ; &) \\ (2; & ; &) \\ (3; & ; &) \end{matrix}$$

After eliminating variable x, the resulting system is:

$$\begin{pmatrix} -3y & \leq & 4 \\ 0 & \leq & 4 \end{pmatrix} \quad \begin{matrix} (1,2; & x; &) \\ (1,3; & x; & y) \end{matrix}$$

The vector on the right side of the equations denotes in the first element the history of the equations. The second element are the effectively eliminated variables, E. In the last position the implicit eliminated variables I are given. After the first operation contains the set of officially eliminated variables O only x. However, in the second equation, we see an example where a variable, y, is implicitly eliminated but not officially eliminated.

Let c be a constraint in the inequation system and H_c its historical subset. We call the historical subset *minimal*, if there is no other constraint c' with $c' \neq c$ such that $H_{c'} \subset H_c$. Based on those divisions, Imbert builds two acceleration theorems.

Theorem 2.3.5 (First Acceleration Theorem)**.** *If H_c is a minimal subset, then the following relation is satisfied:*

$$1 + |E_c| \leq |H_c| \leq 1 + |E_c \cup (I_c \cap O_k)|$$

Theorem 2.3.6 (Second Acceleration Theorem)**.** *Let c be an inequation such that $1 + |E_c| = |H_c|$, then H_c is minimal.*

Both proofs can be found in [Imb90].

The first acceleration theorem detects non-minimal subsets. This means inequations that do not fulfil it, do not need to be checked for minimality. In addition, the

second theorem detects minimal inequations. If an inequation fulfils the criterion for the second acceleration theorem, it is guaranteed to be minimal.

There are equations that fulfil the first but not the second theorem. Meaning, there are inequations that are not detected to be redundant but are also not detected to be minimal. For those constraints, Imbert proposes to use the redundancy check given by Kohler or Chernikov. The main focus of this work is researching a new method, which is described later in Chapter 3. For that reason, not both algorithms are described but only the redundancy check by Chernikov. Imbert states in [Imb93] an English translation of Chernikovs method. Chernikovs method is based on his *Minimal Subset* theorem. In this he states that the Fourier-Motzkin algorithm requires only constraints with minimal historical subsets.

Theorem 2.3.7 (Minimal Subsets)**.** *The Fourier-Motzkin algorithm needs only constraints with minimal historical subsets in order to work correctly.*

This means, if the history of an inequation c is contained in the history of inequation c', c' is not needed in the final system. The proof can be found in [Koh67]. The drawback of this decision is the quadratic running time. Every newly created inequation has to be compared to all other histories on that level. To do so, all new histories need to be computed first and then eventually discarded. This makes Imbert's accelerations way more powerful from a practical viewpoint. For them, not all inequalities need to be created. Their minimality can be decided by a local criterion. Important to note, to apply Chernikovs improvement the histories are needed. The new constraints can be computed in a second step after not discarding the inequality. However, Imbert's acceleration theorems are not capable of discovering all minimal subsets. They are very fast in deciding minimality or non-minimality but still rely on other decision procedures. It is important to note that the minimality decision by Kohler is equivalent to the one by Chernikov, [Imb93]. It uses the rank of matrix of the inequation-systems to decide minimality. As it would not provide further insights, it is not described in this work. The improved Fourier-Motzkin algorithm is described in Algorithm 2.

Example 2.3.3. *In this extended example, the modified Fourier-Motzkin algorithm is presented. Both of the Imbert accelerations and the comparison method are demonstrated. As the previous examples demonstrated the conversion from a QF_LRA formula to a constraint set S, the following systems are directly given in their symbolic form. Let the problem be given by:*

$$\begin{pmatrix} x+y & \leq & 4 \\ x-4y & \leq & 2 \\ -x-4y & \leq & -3 \\ -x+2y & \leq & 3 \\ -3x+2y & \leq & 4 \\ -x-y & \leq & 4 \end{pmatrix} \quad \begin{matrix} (1; & ; &) \\ (2; & ; &) \\ (3; & ; &) \\ (4; & ; &) \\ (5; & ; &) \\ (6; & ; &) \end{matrix}$$

The variable ordering is again $x < y$, thus firstly x is eliminated and secondly y.

Initially, the set of officially eliminated variables is empty, $O_1 = \emptyset$. After eliminating x, the following system results. The vector on the right side of the equations

Algorithm 2 FM+Imbert Algorithm

procedure FOURIER-MOTZKIN ALGORITHM(inequation-system $S = (A,b)$ and variable ordering $\bar{x} = (x_1, \ldots, x_n)$)
  Let $O := \emptyset$ ▷ Set of officially eliminated variables
  for Var $x_i := \bar{x}_i$ **do**
    Partition S in L,U,N
    if $L \neq \emptyset \vee U \neq \emptyset$ **then**
      $O := O \cup \{x_i\}$
    else
      $S := N$
    $H :=$ HISTORIES(L,U) ▷ All possible new histories
    for all pair $(l,u) \in L \times U$ **do**
      Let the sets H_l, E_l and I_l be connected with l
      and the sets H_u, E_u, I_u with u
      $H_c := H_l \cup H_u$, $E_c := E_l \cup E_u$ and $I_c := I_l \cup I_u$
      if $|H_c| \leq 1 + |E_c \cup (I_c \cap O)|$ **then** ▷ First acceleration theorem
        if $|E_c| = |H_c|$ **then** ▷ Second acceleration theorem
          $S := S \cup \{$COMBINE(l,u,x)$\}$
        else
          if Exists no history $H_{c'} \in H$ with $H_{c'} \subsetneq H_c$ **then**
            $S := S \cup \{$COMBINE(l,u,x)$\}$ ▷ Minimal by Chernikov criterion
      else
        continue
    $S := S \cup N$
    if conflict in S **then**
      return UNSAT
  return SAT
procedure HISTORIES(L,U)
  Histories $H' := \emptyset$
  for $l \in L$ **do**
    for $u \in U$ **do**
      $H' := H' \cup \{H_l \cup H_u\}$
  return H'

shows the history, effectively eliminated variables and implicitly eliminated variables.

$$\begin{pmatrix} -3y & \leq & 1 \\ 3y & \leq & 7 \\ 5y & \leq & 16 \\ 0 & \leq & 8 \\ -8y & \leq & -1 \\ -2y & \leq & 5 \\ -10y & \leq & 10 \\ -5y & \leq & 6 \end{pmatrix} \quad \begin{matrix} (1,3; & x; &) \\ (1,4; & x; &) \\ (1,5; & x; &) \\ (1,6; & x; & y\ ) \\ (2,3; & x; &) \\ (2,4; & x; &) \\ (2,5; & x; &) \\ (2,6; & x; &) \end{matrix}$$

In this step, no equation could be omitted. The first acceleration theorem states that all constraints minimal, $1 + |E_c| = |H_c|$.

Now, in the second step the set of officially eliminated variables contains x, $O_1 = \{x\}$. *However, in this step, not all equations need to be formed. For demonstration*

purposes shows the following system all equations.

$$\begin{pmatrix} 0 & \leq & 8 \\ 0 & \leq & \frac{8}{3} \\ 0 & \leq & \frac{53}{24} \\ 0 & \leq & 4.83 \\ 0 & \leq & \frac{10}{3} \\ 0 & \leq & \frac{53}{15} \\ 0 & \leq & \frac{53}{15} \\ 0 & \leq & \frac{123}{40} \\ 0 & \leq & 5.7 \\ 0 & \leq & 4.2 \\ 0 & \leq & \frac{22}{6} \end{pmatrix} \quad \begin{pmatrix} (1,6; & x; & y &) \\ (1,3,4; & x,y; & &) \\ (1,2,3,4; & x,y; & &) \\ (1,2,4; & x,y; & &) \\ (1,2,4,5; & x,y; & &) \\ (1,2,4,6; & x,y; & &) \\ (1,3,5; & x,y; & &) \\ (1,2,3,5; & x,y; & &) \\ (1,2,4,5; & x,y; & &) \\ (1,2,5; & x,y; & &) \\ (1,2,5,6; & x,y; & &) \end{pmatrix}$$

In practice, this is not necessary. The Imbert acceleration and the comparison method can be applied without computing the resulting constraint. Only the sets of eliminated variables are needed. The next system shows only the minimal constraints, omitting all non-minimal constraints.

$$\begin{pmatrix} 0 & \leq & 8 \\ 0 & \leq & \frac{8}{3} \\ 0 & \leq & 4.83 \\ 0 & \leq & \frac{53}{15} \\ 0 & \leq & 4.2 \end{pmatrix} \quad \begin{pmatrix} (1,6; & x; & y &) \\ (1,3,4; & x,y; & &) \\ (1,2,4; & x,y; & &) \\ (1,3,5; & x,y; & &) \\ (1,2,5; & x,y; & &) \end{pmatrix}$$

All constraints with a history of length 4 are omitted due to the first acceleration theorem. With only two eliminated variables, constraints with a history of length 4 can never be minimal. The constraints with a history of length 3 are always minimal due to the second acceleration theorem. In addition, all constraints whose histories contain $\{1,6\}$ *can be omitted due to the Chernikov minimality criterion. Finally, one can also deduce* SAT *for the system.*

This small example already demonstrates rather powerfully the impact of the acceleration operations. The final system is only half the size of the unreduced one. In case we would apply further rounds of the Fourier-Motzking algorithm the impact would be considerably large. None of the eliminated constraints could produce succeeding constraints.

Recently, a new approach for the Chernikov method was proposed. In 2015 a method was proposed which transfers the Chernikov decision to a graph problem, [BZ15].

2.4 Simplex

The Simplex algorithm was initially published by Dantzig in 1947 in [Dan90]. In the last 80 years, it was steadily improved and further heuristics were proposed. A summary of the main improvements can be found in [Bix02]. A rather comprehensive description can be found in [CLRS90]. Simplex for SMT solving is described in [DDM06].

2.4.1 Overview

The Simplex algorithm was originally developed for linear optimization. This means, given linear constraints a linear target function needs to be optimized. In this setting, *linear programs* are used. The Simplex method operates in two phases. In the first phase, a feasible solution to the relaxed problem is searched. Meaning, the problem is reformulated and checked whether there exists a start position within the constraint polyhedron. If there is no feasible solution, i.e. the polyhedron is empty and the problem unsatisfiable, the algorithm is able to detect this case and return UNSAT. With a found feasible solution the first phase ends. This feasible solution is then optimized according to the target function in the second phase. Afterwards, the optimal solution is found. In this search, the Simplex algorithm traverses the edges of the polyhedron and searches for the optimal solution. As this thesis deals with SMT-solving, the second phase is not from interest and thus not further described. Its description can be found in the previously mentioned sources.

2.4.2 Slack Form

Let $Ax \leq b$ be a system with m linear constraints and n variables, thus $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. To apply the Simplex algorithm it firstly needs to be transformed into its slack form. In this form, every inequality is transformed into an equality and slack variables are introduced. Slack variables are added variables that account for the lost solution space when transforming an inequality constraint to an equality constraint. To transform the system into slack form a slack variable s_i is added for every constraint $\langle a_i, x \rangle \leq b_i$ in (A,b). Additionally, a new constraint restricting the slack variable is added. The new system has then the form:

$$\langle a_i, x \rangle = s_i \wedge s_i \leq b_i \text{ for } i = 1, \ldots, m$$

The set of variables is then $\mathcal{X} := \{x_1, \ldots, x_n, s_1, \ldots, s_m\}$.

Example 2.4.1. *To illustrate the slack form, the previous example is revisited. The problem is given by:*

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

Now, every constraint is set equal to a slack variable s_i:

$$\begin{pmatrix} 1 & 1 \\ 1 & -4 \\ -1 & -4 \\ -1 & 2 \\ -3 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_4 \end{pmatrix}$$

and the slack variables s_i are constrained by the previous bounds:

$$\begin{pmatrix} s_1 & \leq & 4 \\ s_2 & \leq & 2 \\ s_3 & \leq & -3 \\ s_4 & \leq & 3 \\ s_4 & \leq & 4 \end{pmatrix}$$

With this, the initial system is transformed into the slack form.

2.4.3 Simplex Tableau

All operations performed by the Simplex algorithm are performed on the *Simplex tableau*. Previously, $x := (x_1, \dots x_n)$ was defined to be the vector of real valued variables $x_1, \dots x_n$. In the following denotes $\mathcal{X} := \{x_1, \dots, x_n\}$ the set of variables in vector x. The Simplex algorithm distinguishes between *nonbasic* variables $\mathcal{N} \subseteq \mathcal{X}$ and *basic* variables $\mathcal{B} = \mathcal{X} \setminus \mathcal{N}$. The basic variables are dependent on the nonbasic variables. For every $x_j \in \mathcal{B}$ the Simplex tableau T contains a row encoding an equation $x_j = \sum_{x_i \in N} a_{ji} x_i$. In addition, two mappings u and l are taken into account, which maps each variable $x \in \mathcal{X}$ to its upper/lower bound. The mapping $\alpha : \mathcal{X} \to \mathbb{R}$ assigns each variable $x \in \mathcal{X}$ to its current, real assignment. Now, given a set of linear constraints, an initial tableau and assignment are constructed. Initially, all original variables are non-basic variables and the slack variables form the set of basic variables. The equations $\sum_{x_i \in \mathcal{N}} a_{ji} x_i = s_j$ form the initial tableau and b_j is added as bound for s_j for $j = 1, \dots, m$. Let T_{ji} denote the coefficient in row j and column i in the Simplex tableau. The initial assignment α is $\forall x \in \mathcal{X} : \alpha(x) = 0$. All equations hold, but the bounds of basic variables might be violated. It is important to note that it is even an invariant of the Simplex algorithm that nonbasic variables always satisfy their bounds. This invariant is satisfied from the beginning as the nonbasic variables do not have any bounds. The Simplex tableau T is now constructed as depicted below:

$$T = \left[\begin{array}{cccc|cccc} x_1 & x_2 & \cdots & x_n & s_1 & s_2 & \cdots & s_m \\ a_{11} & a_{12} & \cdots & a_{1n} & -1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & -1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & 0 \\ a_{m1} & a_{m2} & \cdots & a_{mn} & 0 & \cdots & 0 & -1 \end{array}\right]$$

In the following, the tableau is displayed as a $m \times (m + n)$ matrix with m being the number of basic variables, $|\mathcal{B}|$, and n the amount of non-basic variables, $|\mathcal{N}|$. Each row in the matrix describes the basic variable as a linear combination of nonbasic variables. The basic variables are always the columns with a negative unity vector entry. As previously defined, in the beginning, the slack variables $s_1, \dots, s_m$ are the initial nonbasic variables. The equation in the i-th row of the Simplex tableau can be read as

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - s_i = 0.$$

The tableau expresses every nonbasic variable in dependency to the basic variables. Thus, the slack variable is defined as the weighted sum over the variables. As said before, the nonbasic variables always satisfy their bounds as they do not have any bounds, the bounds were transferred on the slack variables which are basic variables. However, the basic variables might violate their bounds arbitrarily.

The tableau is now modified by applying the rules depicted in Figure 2.3. This description can be found in [KBD+17]. To apply a rule, all of the premises in the numerator need to be fulfilled. The denominator states the effects on the components of the problem when applying the rule. To fix the bound of a basic variable, the variable is first moved into the set of nonbasic variables, by applying the `Pivot` rule, and then secondly updated to satisfy its bounds, `Update` rule. A basic variable

$$\texttt{Pivot}_1 \quad \frac{x_i \in \mathcal{B} \quad \alpha(x_i) < l(x_i) \quad x_j \in \text{slack}^+(x_i)}{T := pivot(T,i,j) \wedge \mathcal{B} := \mathcal{B} \cup x_j \setminus \{x_i\}}$$

$$\texttt{Pivot}_2 \quad \frac{x_i \in \mathcal{B} \quad \alpha(x_i) > u(x_i) \quad x_j \in \text{slack}^-(x_i)}{T := pivot(T,i,j) \wedge \mathcal{B} := \mathcal{B} \cup x_j \setminus \{x_i\}}$$

$$\texttt{Update} \quad \frac{x_j \in \mathcal{N} \quad \alpha(x_j) < l(x_j) \vee \alpha(x_j) > u(x_j) \quad l(x_j) = \alpha(x_j) + \delta \vee u(x_j) = \alpha(x_j) + \delta}{\alpha := update(\alpha, x_j, \delta)}$$

$$\texttt{Failure} \quad \frac{x_i \in \mathcal{B} \quad (\alpha(x_i) < l(x_i) \wedge \text{slack}^+(x_i) = \emptyset) \vee (\alpha(x_i) > u(x_i) \wedge \text{slack}^-(x_i) = \emptyset)}{\text{UNSAT}}$$

$$\texttt{Success} \quad \frac{\forall x_i \in \mathcal{X} \quad l(x_i) \leq \alpha(x_i) \leq u(x_i)}{\text{SAT}}$$

Figure 2.3: Standard Simplex Rules

x_i is switched with a non-basic variable x_j by using the $\texttt{Pivot}_1$ or $\texttt{Pivot}_2$ rule, depending on which bound is broken. Not every variable is suitable to be pivoted with x_i. The update of x_i also affects the now basic, former non-basic, variable x_j. Before performing the pivoting step it is ensured, that x_j is at least a suitable pivoting candidate. This means that x_j is not guaranteed to violate its bounds after the update of x_i. For this, x_j has to be in the *slack* of x_i:

$$\text{slack}^+(x_i) = \{x_j \in \mathcal{N} \mid (T_{i,j} > 0 \wedge \alpha(x_j) < u(x_j)) \vee (T_{i,j} < 0 \wedge \alpha(x_j) > l(x_j))\}$$

$$\text{slack}^-(x_i) = \{x_j \in \mathcal{N} \mid (T_{i,j} < 0 \wedge \alpha(x_j) < u(x_j)) \vee (T_{i,j} > 0 \wedge \alpha(x_j) > l(x_j))\}$$

Though, even with the pivoted variable being in the slack, it is not ensured that the now new basic variable does not violate its bounds. It is possible that the new basic variable has to be fixed again. After a pivot operation $\texttt{Pivot}(T,i,j)$ is the basic variable x_i replaced by the entering variable x_j and the equation in row i is changed according to the linear conversion. Also, to keep the tableau consistent, every occurrence of x_i is also adapted to the new linear representation of x_j. The assignment α is changed by the `Update` operation. For a non-basic variable $x_j \in \mathcal{N}$, $update(\alpha,x_j,\delta)$ returns an assignment α' with modification $\alpha'(x_j) = \alpha'(x_j)+\delta$ and the values of all basic variables $x_i \in \mathcal{B}$ are updated according to their linear representation $T_{i,j}$ of x_j. This means the update rule pays attention to the influence of x_j on the basic variables. The assignment of the variable switched into the nonbasis is always set to one of its bounds, e.g. either the new assignment equals the upper bound or the lower bound. This property is used again in the next section where the geometric properties of the simplex algorithm are discussed. The `Failure` rule applies, when there is a basic variable violating its bound but it can not be pivoted with any nonbasic variable. Then the problem is unsatisfiable. On the contrary, when all variables in $\mathcal{X}$, i.e. the variables in $\mathcal{B}$ satisfy their bounds, the algorithm terminates as it found a valid assignment. In this case the `Success` rule can be applied and SAT can be deduced.

Example 2.4.2. *Firstly, the slack form from Example 2.4.1 is transformed into the*

corresponding Simplex tableau:

$$T = \begin{array}{c} \begin{matrix} x & y & s_1 & s_2 & s_3 & s_4 & s_5 \end{matrix} \\ \left[\begin{array}{cc|ccccc} 1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & -4 & 0 & -1 & 0 & 0 & 0 \\ -1 & -4 & 0 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & -1 & 0 \\ -3 & 2 & 0 & 0 & 0 & 0 & -1 \end{array}\right] \end{array} \qquad \begin{pmatrix} s_1 & \leq & 4 \\ s_2 & \leq & 2 \\ s_3 & \leq & -3 \\ s_4 & \leq & 3 \\ s_4 & \leq & 4 \end{pmatrix}$$

The set of variables is $\mathcal{X} = \{x,y,s_1,s_2,\ldots,s_5\}$. *The assignment* α *is given by* $\alpha(x) = 0$ *for* $x \in \mathcal{X}$. *As the problem is given in standard form the slack variables have only upper bounds. Those upper bounds are set to the right side of the constraints, e.g.* $u(s_1) = 4$. *Thus, the only violating slack variable is* s_3, $\alpha(s_3) = 0 \not\leq -3$. *Without loss of generality, let the variable order be given by the position of the variables in the tableau, e.g.* $x < y < s_1 < \cdots < s_5$. *Initially, the basis consists of slack variables* $\mathcal{B} = \{s_1,\ldots,s_5\}$ *and the nonbasis of the original variables* $\mathcal{N} = \{x,y\}$. *In the first pivoting and update step is* x *updated to 3, in order that* s_1 *is at its lower bound. The resulting tableau is:*

$$T' = \begin{array}{c} \begin{matrix} x & y & s_1 & s_2 & s_3 & s_4 & s_5 \end{matrix} \\ \left[\begin{array}{cc|ccccc} 0 & -3 & -1 & 0 & -1 & 0 & 0 \\ 0 & -8 & 0 & -1 & -1 & 0 & 0 \\ -1 & -4 & 0 & 0 & -1 & 0 & 0 \\ 0 & 6 & 0 & 0 & 1 & -1 & 0 \\ 0 & 14 & 0 & 0 & 3 & 0 & -1 \end{array}\right] \end{array}$$

The updated assignment is then: $\alpha(x) = 3, \alpha(y) = 0, \alpha(s_1) = \alpha(s_2) = \alpha(s_3) = 3, \alpha(s_4) = -3$ *and* $\alpha(s_5) = -9$. *Thus, only* s_2 *breaks its bound. To do so,* y *is updated so that* s_2 *fulfils its assignment and is then pivoted with it.*

$$T'' = \begin{array}{c} \begin{matrix} x & y & s_1 & s_2 & s_3 & s_4 & s_5 \end{matrix} \\ \left[\begin{array}{cc|ccccc} 0 & 0 & -1 & \frac{3}{8} & -\frac{5}{8} & 0 & 0 \\ 0 & -1 & 0 & -\frac{1}{8} & -\frac{1}{8} & 0 & 0 \\ -1 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & -\frac{6}{8} & \frac{2}{8} & -1 & 0 \\ 0 & 0 & 0 & -\frac{14}{8} & \frac{10}{8} & 0 & -1 \end{array}\right] \end{array}$$

Finally, the assignment is $\alpha(x) = 2.5, \alpha(y) = \frac{1}{8}, \alpha(s_1) = \frac{21}{8}, \alpha(s_2) = 2, \alpha(s_3) = -3, \alpha(s_4) = -\frac{18}{8}, \alpha(s_5) = -\frac{58}{8}$. *As all bounds are satisfied, the* SUCCESS *rule is applied and* SAT *is deduced. The corresponding basis is* $\mathcal{B} = \{x,y,s_1,s_4,s_5\}$.

2.4.4 Properties of the Simplex Algorithm

Whether or not the Simplex algorithm terminates highly depends on the selection of pivoting rule. Meaning which variables are chosen to leave the basis and which is chosen to enter the basis. The most famous and easiest rule to ensure termination was proposed by Bland in [Bla77]. Bland's termination rule needs an ordering on the variables. When selecting the entering and leaving variables, those with the smallest index are chosen.

Theorem 2.4.1. *Bland's Rule ensures termination for the Simplex algorithm.*

The proof can be found in [Bla77]. To estimate the running time of the Simplex algorithm, a few theorems are necessary. As shown before, there exists a connection between the basis variables and the vertices of the Simplex polyhedron. It shows, that in the worst-case scenario the Simplex algorithm traverses over all possible vertices. The following theorems can be found in [CLRS90]. Firstly, we need to prove that the basis always determines a slack form uniquely.

Lemma 2.4.2. *The solution set of the slack form of a linear program is uniquely determined by the set of basis variables, $\mathcal{B}$.*

Proof. In order to draw a contradiction, we assume that there are two different slack forms with the same set of basis variables. The first slack form is given by:

$$x_i = b_i - \sum_{j \in \mathcal{N}} a_{ij} x_j \text{ for i } \in \mathcal{B}$$

and the second by:

$$x_i = b'_i - \sum_{j \in \mathcal{N}} a'_{ij} x_j \text{ for i } \in \mathcal{B}.$$

When subtracting both equations for a x_i, the resulting system is:

$$\begin{aligned} 0 &= b_i - b'_i - \sum_{j \in \mathcal{N}} (a_{ij} - a'_{ij}) x_i \text{ for i } \in \mathcal{B} \\ \iff \sum_{j \in \mathcal{N}} a_{ij} x_j &= (b_i - b'_i) + \sum_{j \in \mathcal{N}} a'_{ij} x_j \text{ for i } \in \mathcal{B} \end{aligned}$$

What remains to prove is, that for a set of indices of variables, $I \subset \{1, \ldots, |\mathcal{X}|\}$,

$$\sum_{i \in I} \alpha_i x_i = \gamma + \sum_{i \in I} \beta_i x_i.$$

implies that $\alpha_i = \beta_i \ \forall i \in I$ and $\gamma = 0$. Hereby are the x_i any assignment for the variables. Since the statement holds for all possible assignments, we can use specific ones to prove the claim. If we let $x_i = 0 \ \forall i \in I$ we can conclude $\gamma = 0$. Now, we let one $x_i \neq 0$ and all other $x_k = 0$ for $k \in I \wedge k \neq i$ to conclude $\alpha_i = \beta_i$ for all $i \in I$. This proves the lemma. □

Theorem 2.4.3. *If the Simplex algorithm does not terminate in $\binom{n+m}{m}$ iterations, it cycles. Hereby is n the number of variables and m the number of constraints.*

Proof. By Lemma 2.4.2 we know that every basis of the Simplex algorithm has a unique slack form. By construction, we know that the basis has the same size as the number of constraints, $|\mathcal{B}| = m$. To the initially n variables are m slack-variables added. Thus, there is a total of $n + m$ variables to select the m basis variables from. Leaving us with a total of $\binom{n+m}{m}$ possible basis combinations. If the Simplex algorithm visits a basis twice, it is forced to cycle. Otherwise, it terminates in less or equal to $\binom{n+m}{m}$ steps. □

By now, it is proven that the Simplex algorithm terminates in at most exponentially many steps or starts cycling. In worst-case examples visits the Simplex algorithm all of the vertices of the polyhedron. Such a worst-case example is for many pivoting rules the *Klee-Minty* cube [KM72]. To prove that Bland's rule can actually take exponentially many steps, a modified version of the Klee-Minty cube is constructed. This is done in the next theorem after defining Klee-Minty cubes. The definition and proofs can be found in [AZ96].

Definition 2.4.1 (Klee-Minty Cubes)**.** *A Klee-Minty cube of dimension d, for some* $\epsilon \in [0, \frac{1}{2}]$ *is given by the inequations:*

$$\begin{array}{rcccll} 0 & \leq & x_1 & \leq & 1 & \\ \epsilon x_{j-1} & \leq & x_j & \leq & 1 - \epsilon x_{j-1} & \text{for } 2 \leq j \leq d \end{array}$$

Theorem 2.4.4 (Simplex Running time)**.** *The Simplex algorithm with Bland's pivoting rule visits all* 2^d *vertices of the d-dimension Klee-Minty cube.*

The proof can be found in [AZ96]. Thus, also Bland's rule has exponentially running time.

2.4.5 Further Remarks Geometric Interpretation

As previously mentioned traverses the Simplex algorithm in the optimization phase the edges of the spanned polyhedron. This does not exactly hold for the first phase though. In this, the algorithm traverses not along the edges but jumps from extreme point, a corner of the polyhedron, to extreme point.

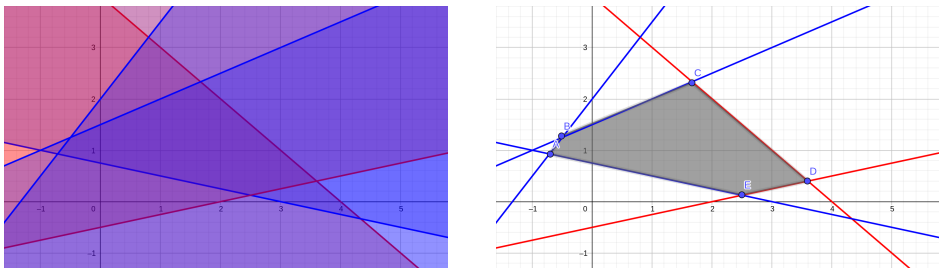
But, the "current position" of the algorithm can be read from the nonbasis of the Simplex tableau. This means the nonbasis assembles an extreme point for systems of full rank, i.e. a point on the convex hull of the Simplex polyhedron. Systems with not full rank are *extreme facets*.

Example 2.4.3. *This visualization is shortly shown by the last example. Here, the final basis was* $\mathcal{B} = \{x, y, s_1, s_4, s_5\}$ *and thus the nonbasis* $\mathcal{N} = \{s_2, s_3\}$ *with the following assignment:* $\alpha(x) = 2.5, \alpha(y) = \frac{1}{8}, \alpha(s_1) = \frac{21}{8}, \alpha(s_2) = 2, \alpha(s_3) = -3, \alpha(s_4) = -\frac{18}{8}, \alpha(s_5) = -\frac{58}{8}$*. When visualizing the current point, one considers the point created by the original variables,* $(2.5, \frac{1}{8})$*, it lays exactly on the intersection of the corresponding inequations to* s_2 *and* s_3*. Figure 2.4 displays on the left side the constraints and on the right side the Simplex polyhedron. One can observe that the intersection of* s_2 *and* s_3 *is point E of the polyhedron.*

One more nice property of the Simplex algorithm appears on UNSAT examples. As previously defined can the UNSAT rule only be applied when a basis variable violates its bound but has no possible pivoting partner. Thus, its slack-set is empty. To deduce the reason for this conflict, only the non-zero coefficient nonbasic variables and the violating basis variable are collected. The infeasible subset is now generated by the constraints producing those variables. In this way, more than one conflict, if existent, can be read from the Simplex tableau.

2.4.6 Simplex in Modern Applications

Even in current times, nearly 70 years after its first publication, is the Simplex algorithm still widely used and further improved. Though there exist polynomial algorithms, e.g. the *Ellipsoid* method in [AS80], Simplex is still used besides its possibly



(a) Constraints (b) Polyhedron

Figure 2.4: Constraints and Polyhedron

exponential running time. Detailed analysis shows that the Simplex algorithm performs well in practice as it is very unlikely to ”hit” an exponential running time input, [ST04]. The Simplex algorithm is used in linear optimization to solve linear programs, in SMT solving for linear real arithmetic and in operations research. Additionally, many variants of the Simplex algorithm are in use today. They utilize a wide variety of heuristics to select the entering and leaving variables, e.g. [KBD13] or [PSS03]. There are even approaches to verify neuronal networks by a modified Simplex algorithm [KBD+17].

Chapter 3

FMPlex

3.1 Splitting the Polyhedron

As previously seen, every step in the FM algorithm can be seen as a projection of the solution polyhedron on the remaining variables. Now, instead of considering the whole polyhedron in every step, we could split it into multiple parts. The idea is to split the polyhedron on intersections of bounds in the same set. This means, instead of projecting the whole polyhedron on the remaining variables $x_d, \dots x_n$, we could only project parts of the polyhedron. The single parts are determined by the intersections of lower or upper bounds. In FM, those intersections are not considered any further. This is not necessary, as all possible combinations between upper and lower bounds are considered.

However, we would like to reduce the number of constraints within the system. In the new approach, the bounds are still split into sets of upper bounds U, lower bounds L and non-bounds, N. Now, not all combinations from lower and upper bounds are considered, but every bound from L is compared to all bounds in U. Additionally, one has to ensure that the selected lower bound is indeed the largest lower bound. To do so, the lower bound is also compared to all other lower bounds. This resembles the computation of

$$\max_{l \in L} l \leq x_1 \leq u, \forall u \in U.$$

The maximum over the lower bounds is determined in a case distinction. Every lower bound is checked once to be the largest lower bound by comparing it to the other lower bounds. Special attention needs to be paid to the case that the bound assumed to be the largest lower bounds conflicts with other lower bounds. In this case, the wrong lower bound was selected. However, this is no conflict on which we can conclude the unsatisfiability of the system. We have to backtrack to the faulty decision and check another lower bound. This is repeated until either a SAT branch was found and we can conclude that the system is satisfiable or UNSAT can be deduced. Previously was argued that the distinction is made to find the largest lower bound. The same distinction can be done in order to find the smallest upper bound. Then the computation resembles

$$l \leq x_1 \leq \min_{u \in U} u, \forall l \in L.$$

The minimum over the upper bounds is determined in a case distinction. The decision

on which bounds to split is arbitrary. From now on, the COMBINE operation is not only called with a lower and an upper bound as input. It might be called with two lower or two upper bounds as input. The operation COMBINE has still its old domain, it maps two constraints c_1 and c_2 over n variables to a constraint c over $n-1$ variables. Firstly, only non-strict constraints are considered. For constraints c_1, c_2, variable x and partitions L,U holds:

$$\text{COMBINE}(c_1,c_2,x) := \begin{cases} c_1 \leq c_2, & \text{if } c_1 \in L, c_2 \in U \\ c_1 \leq (-1) \cdot c_2, & \text{if } c_1 \in L, c_2 \in L \\ (-1) \cdot c_1 \leq c_2, & \text{if } c_1 \in U, c_2 \in U \end{cases}$$

However, we chose to minimize the number of branches in order to keep the number of backtracking operations small. The amount of constraints within the single branches stays roughly the same. The only difference is whether the constraint is created by a comparison of lower and upper bounds or by bounds from the same set. In Figure 3.1 the split of the polyhedron is depicted. Hereby denote the red lines upper bounds and the blue lines lower bounds. In this example, the upper bounds are selected for splitting as there are fewer upper bounds ($|U| = 2$) than lower bounds ($|L| = 3$). The black line separating the polyhedron corresponds to the cut via the intersection of the upper bounds (point D). Every single part of the polyhedron is

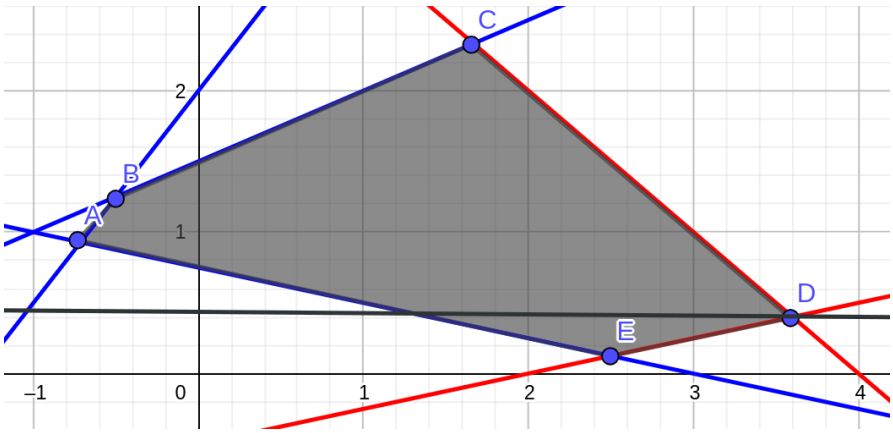


Figure 3.1: Polyhedron from the Running Example Splitted on Upper Bounds

processed by a separate *branch*. Hereby refer branches not to parallelized processes but to different execution branches within the algorithm. Occasionally, a branch is also called *child*. This name refers to the fact that splitting the parent polyhedron spans a tree. Every split of the polyhedron is handled by another child in the computation tree. Let $C \in \{L,U\}$ be the constraints the polyhedron is split on. For every constraint $c \in C$ a separate system of constraints S_c is constructed. A branch is identified by its constraint system. Without loss of generality, let the constraints selected for the split be the lower constraint, $C = L$. Then S_c is formed by combining c with all upper constraints in U and all other constraints in L. By this, it is ensured that c is not covered by another lower bound and it is not conflicting with any upper bound. Leading to:

$$S_c := \bigcup_{u \in U} c \leq u \cup \bigcup_{l \in U, l \neq c} l \leq c.$$

It is important to note that the comparison of an upper and lower bound is equivalent to multiplying the term of c with -1. This multiplication transforms c to an upper bound and it can be compared to $l_i \in L$. However, the linear combination of intermediate constraint, thus constraints in S_c, are not exclusively positive linear combinations of original constraints. Now, also linear combinations with negative coefficients are present. Special attention is needed when handling constraints with such linear combinations. To do so, a CONFLICT-LEVEL, cl, counter is introduced for every constraint. The conflict-level of constraint c is denoted by $cl(c)$. For a given constraint, the conflict-level counter names the number of levels since one of its parents was part in a same-bound combination, i.e. the comparison between two lower bounds. This means a constraint with a conflict-counter of level 1 has a parent whose parent was compared to a same-bound equation. Constraints constructed by constraints in the same set, i.e. both constraints are in U or both are in L, are called *same-bound combination*. The regular combination with lower and upper bounds is called *FM combination* or *lower-upper combination*. It would have also been created in the FM algorithm. A constraint with a conflict-counter of 0 is the product of a same-bound comparison. It would not have been created in the FM algorithm. This leads to the following computation of the conflict-level:

- FM combination: If those constraints were present on the FM algorithm, the same combination would have been formed. Thus, if a conflict can be formed with this combination, it can be formed in every sibling node on the same level. The child counter is the minimum over the parent counters increased by one. $c := l \leq u \rightarrow cl(c) = min(cl(l), cl(u)) + 1$ for $l \in L$ and $u \in U$.
- Same-bound combination: If the resulting constraint is part of a conflict, it is possible that just the wrong bound was select as the largest lower bound or smallest upper bound. In this case, one of its siblings is the right choice and another branch should have proceeded. Thus, those constraints receive a conflict-level counter of 0. $c := c' \leq c'' \rightarrow cl(c) = 0$ for c',$c'' \in C$.

The linear combinations with not only positive elements play a special role in conflict detection, as explained now. After creating S_c, one of the following cases occurs:

- **One or more conflict is found:** Finding a conflict in the original FM algorithm enabled one to deduce UNSAT. A positive linear combination over original constraints, which leads to a conflict, was found. Thus, the whole system is unsatisfiable, see Farkas Lemma, Lemma 2.3.3.

  However, in our new approach, the linear combination behind a conflict might contain a negative element. In this case, it is not a witness to the unsatisfiability of the whole system. A conflict e in system S_c is defined as $e := (e_1, \ldots, e_m) \in R^m$. Every element e_i of e represents the coefficient of a constraint c_i from the set of original constraints S'. Recall that every constraint c_i is defined as $c_i := (c_{i_1} x_1 + \ldots + c_{i_n} x_n \bowtie b_{c_i})$ with $\bowtie_{c_i} \in \{<, \leq\}$. The sum $\sum_{i=1}^{m} e_i \cdot (c_{i_1} + \ldots + c_{i_n}) \bowtie \sum_{i=1}^{m} e_i \cdot b_{c_i}$ evaluates to a contradiction. Strict constraints in FMPlex are formally introduced in Section 3.2. We write, $e \geq 0$ if all coefficients γ_i are greater or equal to 0. For conflict e is the conflict level $cl(e)$ defined as the conflict level of its representing constraint. This conflict is only a witness that one of the previous "minimal" lower bounds was chosen wrongly. The conflict is not guaranteed to occur in every branch. In case of a conflict, one can backtrack *conflict-level* many of the decisions within the computation tree.

However, it may occur that negative linear combinations cancel each other out. When a constraint with a negative linear combination is again chosen to be the largest lower or smallest upper bound, the resulting constraint might have a positive linear combination. In this case, $e \geq 0$, one can conclude UNSAT for the whole system and does not need to backtrack. This decision is again based on Farkas Lemma, Lemma 2.3.3.

- **The system S_c is SAT:** In this case, SAT can be deduced for the original problem. As there occurs no conflict on the branch, the largest lower bound was selected in every previous decision. Additionally, no upper bound conflicts with any lower bound. Thus, SAT can be deduced.
- **S_c is neither SAT nor UNSAT:** This case is identical to the FM algorithm. In order to create S_c a variable was eliminated. The same algorithm can now be recursively applied to S_c. Meaning, the subsystem is now split along its lower or upper bounds and the remaining variables are eliminated. Its call either returns SAT or UNSAT, which is then propagated as described before.

The whole algorithm is given in pseudocode in Algorithm 3. It is called *FMPlex* due to a similarity to Simplex which will be handled in Section 3.3.

Example 3.1.1 (FMPlex Algorithm)**.** *This example demonstrates the FMPlex algorithm by resolving our running example. The initial problem is again given by:*

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

Similar to Example 2.3.1, constraints are divided into upper and lower bounds.

$$U := \begin{pmatrix} x + y & \leq & 4 \\ x - 4y & \leq & 2 \end{pmatrix} \begin{matrix} (U_1) \\ (U_2) \end{matrix}$$

$$L := \begin{pmatrix} -x - 4y & \leq & -3 \\ -x + 2y & \leq & 3 \\ -3x + 2y & \leq & 4 \end{pmatrix} \begin{matrix} (L_1) \\ (L_2) \\ (L_3) \end{matrix}$$

As the set of upper bounds is smaller than the set of lower bounds, the upper bounds are selected for splitting the polyhedron. Thus, two branches are created, S_{U_1} and S_{U_2}. In S_{U_1} is U_1 and in S_{U_2} is U_2 the chosen smallest upper bound. The system S_{U_1} consists of lower-upper combinations $L_1 \leq U_1 \wedge L_2 \leq U_1 \wedge L_3 \leq U_1$ and the same-bound combination $U_1 \leq U_2$. The system S_{U_2} is created analogous. Thus, the two branches are given by:

$$S_{U_1} := \begin{pmatrix} -3y & \leq & 1 \\ 3y & \leq & 7 \\ 5y & \leq & 16 \\ -5y & \leq & -2 \end{pmatrix} \begin{matrix} (\text{COMBINE}(L_1, U_1, x)) \\ (\text{COMBINE}(L_2, U_1, x)) \\ (\text{COMBINE}(L_3, U_1, x)) \\ (\text{COMBINE}(U_1, U_2, x)) \end{matrix} \quad S_{U_2} := \begin{pmatrix} -8y & \leq & -1 \\ -2y & \leq & 5 \\ -10y & \leq & 10 \\ 5y & \leq & 2 \end{pmatrix} \begin{matrix} (\text{COMBINE}(L_1, U_2, x)) \\ (\text{COMBINE}(L_2, U_2, x)) \\ (\text{COMBINE}(L_3, U_2, x)) \\ (\text{COMBINE}(U_2, U_1, x)) \end{matrix}$$

The conflict-counter of every constraint resulting from a lower-upper combination is set to 1*. The conflict resulting from comparing $U1$ and $U2$ with each other is set to* 0*. This split on the upper bounds is visualized in Figure 3.1. As none of the two*

Algorithm 3 FMPlex Algorithm

procedure FMPLEX(constraints S and variable ordering $x := (x_1, \ldots, x_n)$)
  $e := (0, \ldots, 0)$
  Partition S in L,U,N
  Let $C \in \{L,U\}$ be the set to split on
  for $c_i \in C$ **do**
    S_{c_i} := COMBINECONSTRAINTS(c_i,L,U,x) $\cup N$
    $cl(n) := cl(n) + 1$ for $n \in N$
    if S_{c_i} is SAT **then**
      return SAT
    if S_{c_i} is UNSAT **then**
      Conflicts $e_1, \ldots, e_n \in \mathbb{R}^m$ ▷ Linear combinations over S_{c_i}
      if $\exists\ e_i \geq 0$ **then**
        return UNSAT
      return $\arg\max_{e_i}(cl(e_i))$
  $\bar{x} := (x_2, \ldots, x_n)$
  for System S_{c_i} **do**
    if FMPLEX(S_{c_i},$\bar{x}$) is SAT **then**
      return SAT
    if FMPLEX(S_{c_i},$\bar{x}$) returns conflict e' **then**
      if $e' \geq 0$ **then**
        return UNSAT
      else if $cl(e') > 0$ **then**
        $cl(e') := cl(e') - 1$
        return e
      else
        $e := e + e'$ ▷ Save conflict
  return e ▷ Return merged conflict
procedure COMBINECONSTRAINTS(c_i,L,U,x)
  System $S := \emptyset$
  if Split on lower bounds **then**
    for $u_j \in U$ **do**
      Constraint $c' :=$ COMBINE(c_i,u_j,x)
      $S := S \cup \{c'\}$
      $cl(c') := min(cl(c_i), cl(u_j)) + 1$
    for $l_j \in L \setminus \{c_i\}$ **do**
      Constraint $c' :=$ COMBINE(l_j, c_i,x)
      $S := S \cup \{c'\}$
      $cl(c') := 0$
  else
    for $u_j \in U \setminus \{c_i\}$ **do**
      Constraint $c' :=$ COMBINE(c_i,u_j,x)
      $S := S \cup \{c'\}$
      $cl(c') := 0$
    for $l_j \in L \setminus \{c_i\}$ **do**
      Constraint $c' :=$ COMBINE(l_j,c_i,x)
      $S := S \cup \{c'\}$
      $cl(c') := min(cl(c_i), cl(l_j)) + 1$
  return S

branches contains a conflict or is satisfied, the algorithm continues its execution with S_{U1}*. The four constraints are again divided into* U *and* L*:*

$$U' := \begin{pmatrix} 3y & \leq & 7 \\ 5y & \leq & 16 \end{pmatrix} \begin{matrix} (U_1') \\ (U_2') \end{matrix}$$

$$L' := \begin{pmatrix} -3y & \leq & 1 \\ -5y & \leq & -2 \end{pmatrix} \begin{matrix} (L_1') \\ (L_2') \end{matrix}$$

This time, the constraints are split on L' *in two new branches with systems* $S_{L_1'}, S_{L_2'}$*:*

$$S_{L_1'} := \begin{pmatrix} 0 & \leq & 8 \\ 0 & \leq & 53 \\ 0 & \leq & -11 \end{pmatrix} \begin{matrix} (\text{COMBINE}(L_1', U_1', y)) \\ (\text{COMBINE}(L_1', U_2', y)) \\ (\text{COMBINE}(L_2', L_1', y)) \end{matrix}$$

In this system occurs a conflict, $0 \nleq -11$*. But checking the linear combination reveals that the constraint contains a negative element:*

$$\{0 \leq -11\} \rightarrow (-5) \cdot \Big[L_1 + U_1\Big] + (3) \cdot \Big[-U_1 + U_2\Big] = -8 \cdot U_1 + 3 \cdot U_2 - 5 \cdot L_1.$$

The returned error is $e = (-8, 3, -5, 0, 0)$*. Thus, one has to backtrack* $cl(e)$ *many steps and continue the execution. The counter of* $\{0 \leq -11\}$ *was set to 0 as is formed by combining two lower bounds. With this, the branches' sibling* $S_{L_2'}$ *is processed. It is created in the same way as* $S_{L_1'}$ *was created and contains the following constraints:*

$$S_{L_2'} := \begin{pmatrix} 0 & \leq & 29 \\ 0 & \leq & 14 \\ 0 & \leq & 11 \end{pmatrix}$$

As all its constraints evaluate to true and no conflict is found, is $S_{L_2'}$ *a* SAT *branch. Through this, one can conclude that the whole system is satisfiable.*

3.1.1 Properties

This subsection aims to prove the properties of the FMPlex algorithm. Firstly, it is to prove that FMPlex is indeed a correct algorithm. Thus, it needs to be *sound* and *complete*. Secondly, the advantage of FMPlex over FM is shown by comparing the amount of created constraints. Thirdly, the relation between FMPlex and the Imbert accelerations is shown.

An algorithm is called *sound* iff every formula φ on which the algorithm outputs SAT is actually satisfiable. As previously defined is an QF_LRA formula called satisfiable iff there exists an assignment α such that every single constraint is satisfied under α.

Theorem 3.1.1. *The FMPlex algorithm is sound.*

Sketch of Proof. To start with the satisfiable case. Let φ be an QF_LRA formula on which FMPlex outputs SAT. It is now to prove, that an assignment satisfying all constraints can be constructed. As FMPlex returns SAT, the algorithm is in a branch where repetitively the correct, i.e. largest lower or smallest upper, bound was selected. Additionally, the lower and upper bounds are not conflicting. Now, an assignment α for all original variables x_1 to x_n can be constructed by backtracking

through the computation tree. As no conflict was found, this is possible on every level. The assignment can be chosen just like in FM. Thus, the formula φ is indeed satisfiable.

To prove the unsatisfiable case it suffices to show that FMPlex does not return UNSAT if φ is satisfiable. Though some inner calls of the FMPlex algorithm might return UNSAT, it will never discover a conflict leading to return UNSAT. As φ is satisfiable, no constraints with a backtrack level of the depth of the tree can be conflicting. This follows from the soundness of FM. Moreover, no conflict with a positive linear combination can be found. If it would exist, φ would not be satisfiable. In the worst case, FMPlex traverses through the whole tree and discovers SAT branch at last. □

An algorithm is called *complete* iff for every formula φ the algorithm either outputs SAT or UNSAT in finite time.

Theorem 3.1.2. *The FMPlex algorithm is complete.*

As it is already proven to be sound, the found answer is correct. Now it is to prove that such an answer can always be found within finite time.

Sketch of Proof. To prove completeness it is sufficient to prove that FMPlex terminates in finite time. As FM eliminates in every call a variable x_i, so does FMPlex. FMPlex however might backtrack to the same level and eliminates the variable again differently via projecting a different part of the polyhedron. Thus, every branch system S_c with variables $x_1, \dots x_n$ spans at most a subtree of depth n. Let m be the number of constraints in S_c. In the worst case produces the system $m-1$ children. However, every of the $m-1$ many children uses only $n-1$ many variables. Thus the depth and the width of the computation tree is finite. The algorithm can traverse the whole tree and terminates within a finite amount of time. □

The next theorems give bounds on the size of the inner systems and the total amount of constraints considered.

Lemma 3.1.3. *The number of constraints in the child system is at least one smaller than the number of constraints in the parent system.*

Proof. The parent system S is split into U, L and N for a given variable x. All the child systems S_c are formed by copying the constraint from N to S_c and combining one constraint c with all constraints in L and U, excluding the combination with itself. Let $C \in \{L,U\}$ be the set to split on. S_c replaces all constraints in U and L with a combined constraint, except for the constraint c. Thus, $|S_c| \leq |S| - 1$ for all $c \in C$. This bound is not tight as more constraints might evaluate to true or become redundant as multiples from other constraints. □

Theorem 3.1.4 (Singly Exponential Growth)**.** *The total amount of constraints within FMPlex grows only single exponentially in the number of original constraints.*

Proof. Let the initial formula contain m constraints and n variables. In the first elimination step, we create at most m branches with each at most $m-1$ many constraints. Thus, in total $(m) \cdot (m-1)$ many constraints and $n-1$ remaining variables. Repeating this for n steps, the final system has a total of

$$(m) \cdot (m-1) \cdot (m-2) \cdot \ldots \cdot (m-(n+1)) < m^{n+1}$$

constraints. Thus, only singly exponential many constraints are produced. □

Theorem 3.1.4 suggests that FMPlex should be way faster on larger problems than the original FM algorithm. The reduction from doubly exponential growth to only singly exponential growth is considerable. However, the open question is whether FMPlex could be improved with the Imbert acceleration. The next theorem answers this question.

Theorem 3.1.5 (FMPlex Minimality)**.** *Every constraint in a FMPlex branch fulfils the criterion from the second acceleration theorem, thus is minimal.*

To prove this theorem, an important difference between the classical FM algorithm and FMPlex needs to be highlighted. While FMPlex eliminates variables in different ways, i.e. the different combinations of lower and upper bounds eliminate FMPlex a constraint only on one way. It uses only one constraint to eliminate a variable in a branch. This connection is formalized in the following lemma.

Lemma 3.1.6. *Let $S' \subset S$ be the set of minimal constraints in the FMPlex branch. For a constraint $c \in S'$ let H_c be its history and E_c the set of explicitly eliminated variables. Then there exists a bijection $f : \mathcal{X} \to \mathbb{N}$ mapping explicitly eliminated variables to indices of original constraints. Thus, the index of the constraint used to eliminate x is the same for all constraints in S.*

Proof for Lemma 3.1.6. FMPlex creates multiple branches on every split. In branch S_c is constraint c used to eliminate a variable x. The variable x can only be eliminated by a combination with c. In the history H_c of c, exists an index $i \in H_c$ which was not added through an elimination operation. In case $|H_c| = 1$, i is the only element in H_c. In this case is c an original constraint. If $|H_c| > 1$, then is the historical subset H_c the result of merging two histories in an elimination operation. There exists one index i in H_c which was never element of the histories of constraints used to eliminate variables in other constraints. Due to this, the index i is not contained in other histories within the branch.

Every constraint containing x is now combined with this one selected constraint c. Thus, in order to explicitly eliminate x, a combination with c is formed. In terms of the bijection, f maps the variable x to the index i, $f(x) = i$. This mapping can be repeated for all eliminated variables, thus the lemma is proven. As there is always only one constraint used in the branch to eliminate a variable, f is indeed a bijection. □

Proof for Theorem 3.1.5. It is to prove that for every constraint c holds

$$|H_c| = 1 + |E_c|.$$

Remember that H_c is a set of indices and E_c a set of constraints. This proof can be done by induction on the depth of the computation tree.

For the 0-th level of the tree, i.e. for the initial constraints holds

$$|H_c| = 1 = 1 + 0 = 1 + |E_c|.$$

Let the level now be $i + 1$. In the induction step it is to show that the statement also holds for constraints on the $i+1$-th level. Let c be now a constraint on the $i+1$-th level. It is formed by combining the two constraints l and u. Let l be formed on level $i_l \leq i$ and u on level $i_u \leq i$. By the induction hypothesis, those two were a minimal constraint. Now, Lemma 3.1.6 states that there is a bijection f mapping eliminated variables to indices of original constraints. Thus, if x' is contained in E_l and E_u,

it is mapped to the same index in H_l and H_u. The history of the newly combined constraint c, H_c is then uniquely determined. It is determined by the indices $f(x)$ for $x \in E_l \cup E_u$. By doing so, the set of histories is also determined to be the distinct indices of variables in E_c. As the history H_c additionally contains the initial index, it is proven that $|H_c| = 1 + |E_c|$ □

The previous theorem is pretty powerful. It states that neither the Imbert Accelerations nor the Chernikov method needs to be applied on FMPlex. FMPlex does not generate constraints that could be eliminated by either of those methods at all.

3.2 Extension to Strict Inequalities

The algorithm presented previously is not able to handle strict constraints. However, the extension to strict constraints is similar to the FM extension. In FMPlex one has to consider the comparisons between lower-upper combinations and same-bound combinations. In the case of lower-upper combinations the same handling as in FM can be applied. If one of the parent constraints is strict, the resulting constraint is also strict. In the case of same-bound combinations, a further distinction is needed. Let a same-bound combination be formed from l_1 and l_2 and the comparison operator of l_i be denoted by $Op(l_i)$ for $i \in \{1,2\}$. Let l_1 be the bound chosen for the split, e.g. the largest lower bound. Now the following cases exist:

- $Op(l_1)$ is $<$ and $Op(l_2)$ is $<$: In this case, is a strict bound is supposed to be larger or equal to a strict bound. Both of the constraints exclude the equality case. Thus, the resulting constraint can be chosen non-strict.
- $Op(l_1)$ is $<$ and $Op(l_2)$ is $\leq$: A strict bound is set to be larger than a non-strict bound. Meaning the smaller bound can be equal to the larger lower bound. Thus, the resulting constraint is non-strict.
- $Op(l_1)$ is $\leq$ and $Op(l_2)$ is $\leq$: When both parents are non-strict, the combined constraint is also non-strict. The cut does not need to be excluded.
- $Op(l_1)$ is $\leq$ and $Op(l_2)$ is $<$: This case is the only one producing a strict constraint. The smaller of the two constraints, l_2, is strict and thus not allowed to actually reach the upper-lower bound l_1 as it is not strict. To ensure this, the resulting constraint is strict.

According to the previous discussion is the definition of the COMBINE operation extended on strict and non-strict constraints:

$$\text{COMBINE}(c_1,c_2,x) := \begin{cases} c_1 \leq c_2, & \text{if } c_1 \in L, c_2 \in U, \bowtie_{c_1} \text{ and } \bowtie_{c_2} \text{ are } \leq \\ c_1 < c_2, & \text{else if } c_1 \in L, c_2 \in U \\ c_1 < (-1) \cdot c_2, & \text{if } c_1 \in L \wedge c_2 \in L, \bowtie_{c_1} \text{ is } < \text{ and } \bowtie_{c_2} \text{ is } \leq \\ c_1 \leq (-1) \cdot c_2, & \text{else if } c_1 \in L \wedge c_2 \in L \\ (-1) \cdot c_1 \leq c_2, & \text{if } c_1 \in U \wedge c_2 \in U, \bowtie_{c_1} \text{is} \leq \text{ and } \bowtie_{c_2} \text{is} < \\ (-1) \cdot c_1 \leq c_2, & \text{else if } c_1 \in U \wedge c_2 \in U \end{cases}$$

To embed this distinction in FMPlex, the CombineConstraints function needs only to use the updated COMBINE operation.

3.3 Similarity to Simplex

In this section, the similarity between Simplex and FMPlex is explained. As previously anticipated there is a connection between FMPlex and Simplex. The main property of the Simplex algorithm lies within the basis and nonbasis selection. It was shown that every basis is unique and distinct for a vertex of the polyhedron. In the following, a relationship between the PIVOT operation in Simplex and the constraint selected for a split is established. To do so, the running example is reconsidered. For every line in the initial Simplex tableau exists a constraint in the initial FMPlex system. So far either L or U was considered for splits. Though there is no practical use in splitting on lower and upper bounds, it is theoretically valid. Meaning, one could build all splits over the lower and upper bounds. To demonstrate the similarity between the two algorithms, this is an important factor.

Example 3.3.1. *In Example 2.4.2 one firstly pivots s_3 with x. The constraint producing the slack variable s_3 is $-x - 4y \leq -3$, which is a lower bound (L_1) in the initial FMPlex system. If we split now on L_1 the resulting system S_{L_1} is:*

$$S_{L_1} := \begin{pmatrix} -3y & \leq & 1 \\ -8y & \leq & -1 \\ 6y & \leq & 1 \\ 14y & \leq & 13 \end{pmatrix} \begin{matrix} (\text{COMBINE}(L_1, U_1, x)) \\ (\text{COMBINE}(L_1, U_2, x)) \\ (\text{COMBINE}(L_2, L_1, x)) \\ (\text{COMBINE}(L_3, L_1, x)) \end{matrix}$$

By now, the coefficients of the original variables ($\{y\}$) between the combined constraints and their counterparts in the Simplex tableau match. When one now inserts the bounds of the slack variables into the tableau constraints and updates the equality according to the relation in the bounds, the resulting constants match the constants in the FMPlex system. Take for example the first line of the tableau: $-3y = s_1 + s_3$ with $s_1 \leq 4$ and $s_3 \leq -3$. After inserting the bounds, the resulting constraint is $-3y \leq 1$, equivalent to the first constraint within the FMPlex system.

The constraints in the system might differ by a multiple from the constraint in the tableau. However, this depends only on the representation in the FMPlex system and the tableau. The only constraint which is in the Simplex tableau but not in FMPlex is the pivoted constraint. The reason is that FMPlex eliminates at least one constraint in each combination step. Simplex, however, might revert a pivoted variable multiple times. When repeating this for multiple levels, the same correspondence can be observed.

On a more technical level lies the correspondence between the two algorithms in the type of operation they perform. The pivot operation performs technically also a variable elimination. For a given nonbasic variable are all occurrences replaced by a basic variable. To do so, the equations are rewritten in dependence of the basic variable and then inserted into the remaining constraints. Thus, after the operation, the previously violated basic variable is now satisfied and nonbasic. While Simplex only considers broken constraints for pivoting, the eager FMPlex algorithm considers every constraint for pivoting. This connection gives an additional interpretation of the FMPlex algorithm.

Through its eager splitting on constraints, it tries to construct a satisfying basis, i.e. searches for an extreme point of the polyhedron that satisfies all constraints. Its backtracking operation resembles the correction of basis variables. Once it can conclude that a selection for the basis is wrong, it backtracks and tries to fix this

selection. Moreover, in the case of a SAT input, the final branch can be seen as a linear pivoting order. When the Simplex algorithm would follow this pivoting ordering, it could find within n pivot operation a satisfying assignment. As discussed earlier, the Simplex algorithm might take exponentially long to find an answer. In case of a wrong basis element, it re-pivots in and brings another variable into the basis.

Concluding, the name FMPlex comes from the typical FM-style of constraint combinations and the Simplex-style of computing a valid basis.

The following theorem concludes the similarity.

Theorem 3.3.1 (Equivalent Nonbasis)**.** *For every nonbasis of the Simplex algorithm exists a FMPlex node with the same nonbasis.*

Proof. Let an arbitrary Simplex nonbasis be given by $\mathcal{N}$ and let $s_1, \ldots, s_k \in \mathcal{N}$ be the slack-variables pivoted into the nonbasis. Note that $k \leq n$ whereby n denotes the number of variables. The nonbasis consists of k slack variables and $n-k$ original variables. Let $\mathcal{S}_i$ be the set of slack-variables s_i in $\mathcal{N}$ which are still in the basis of FMPlex in the i-th level. In the beginning holds $\mathcal{S}_0 = \{s_i, \ldots, s_k\}$. It is now proven that $|\mathcal{S}_i| = |\mathcal{S}_{i+1}| - 1$, thus the size of $\mathcal{S}_i$ can be reduced on every level by 1. In the full version of FMPlex, one can pivot/select all of the remaining constraints. In every step a constraint c can be selected whose slack-variable s is in $\mathcal{S}_i$. As previously shown, selecting c for splitting is equivalent to pivoting s into the nonbasis. The variable leaving the nonbasis is an original variable. Thus, after this application increases the size of the cut between the Simplex nonbasis and the FMPlex nonbasis by one. This procedure can now be repeated for the next $k-1$ many levels. Finally, FMPlex contains a constraint system with the same nonbasis as Simplex. □

The benefit of Bland's rule is the prevention of cycles. In Simplex occurs a cycle as soon as a nonbasis is visited twice. Due to the similarity between Simplex and FMPlex, it is now of interest, whether FMPlex might visit the same nonbasis twice. For this proof, another version of the FMPlex algorithm is presented.

This version reduces the size of the constraint systems between siblings. Assume, that a conflict is detected in the first child which lets the FMPlex algorithm backtrack and visit its sibling. In this neighbour, however, the inverted modified constraint from its sibling is dropped. For example, the first child has the constraint $l_2 \leq l_1$, meaning l_1 is the largest lower bound and detects a conflict with it. Its neighbour would have the constraint $l_1 \leq l_2$, meaning l_2 is the largest lower bound. As we know by visiting the neighbour that its sibling detected a problem, we can drop the constraint. The idea behind this is to prevent FMPlex from visiting the same nonbasis twice. The algorithm remains correct, because all the omitted branches were already visited in the conflicting neighbour. By considering this adaptation, the next theorem can be proven.

Theorem 3.3.2 (Repetition of Nonbasis)**.** *The adaptation of FMPlex never visits the same nonbasis twice.*

Sketch of Proof. Assume that the same nonbasis N was visited twice. Then there existed two pivoting operations that moved a slack-variable from the basis into the nonbasis. In FMPlex, this operation corresponds to the split on the same, eventually modified, constraint. However, the adapted FMPlex can never pivot the same basis variable twice, as the corresponding constraint does simply not exists in other branch systems. Thus, the sibling branch would have to split on a constraint that is not part of its system. This is a contradiction and the statement is proven. □

It is important to note, that it is still an open question whether the previous proofs generalize on the FMPlex algorithm presented in Algorithm 3. Currently it is assumed that this is not the case. Due to the high variety of splits, it seems highly unlikely that no nonbasis is repeated. In order to repeat a nonbasis, FMPlex has to split on a constraint again it already selected in a neighbouring branch before. While it is preferable FMPlex would not visit a nonbasis twice, it is assumed that it can.

3.4 Considering Further Constraints

As the FMPlex algorithm is supposed to be used in SMT-solving, it is required to support equalities, i.e. $x = 5$, and not-equalities, i.e. $x \neq 5$. While for both types of constraints naive options exist, they need to be optimized. In the following for both types of constraints, the naive approach and the improved approach is presented.

3.4.1 Equalities

Equality constraints, i.e. $x = 5$, could be naively handled by splitting the equality into a lower and an upper bound:

$$x = 5 \equiv x \leq 5 \wedge x \geq 5.$$

However, given the number of equality constraints in a typical SMT formula, it is not recommended to simulate every equality by additional constraints which are then processed by the FMPlex algorithm. The increase of constraints is too large.

The better approach is to use *Gauss's* variable elimination, [Grc11], to eliminate one variable for every equal-constraint. For every constraint, a variable is chosen. This variable is then replaced in all other constraints and equalities by the given equality. The same procedure is repeated for every equality. By doing so, not only the total amount of variables is reduced but also some conflicts can be detected before FMPlex even started. The number of active constraints is not increased. As the Gauss elimination is widely known, the pseudocode is omitted.

3.4.2 Disequalities

The underlying SAT-solver may negate constraints. In case the negated constraint is an equal constraint, a not-equal constraint is passed to the FMPlex algorithm, i.e. $\neg(x = 5) \equiv x \neq 5$. Not-equal constraints are in the following also called *disequations*. From a theoretical point of view have not-equations wide implications. The considered polyhedron is no longer convex. To the running example from the previous Sections the not-equal constraint

$$x \neq 2$$

is added. It is visualized by the dark-line crossing the polyhedron in Figure 3.2. The points lying on the dark-red line are no valid solutions anymore. Thus, the polyhedron is no longer convex, it is not even connected anymore. In a naive approach, one can divide not-equal constraints into the single valid regions, ensuring that the points lying on the hyperplane are never taken:

$$S \cup \{x \neq 2\} \rightarrow S \cup \{x < 2\} \vee S \cup \{x > 2\}.$$

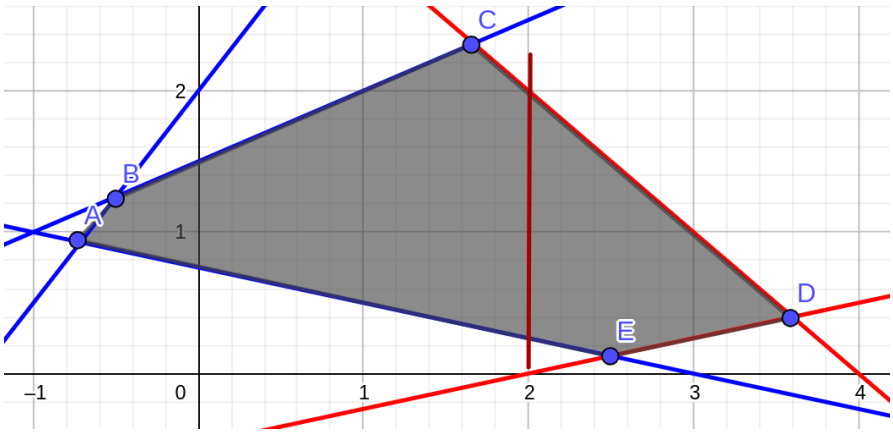


Figure 3.2: Polyhedron from the Running Example with Not-Equality $x \neq 2$

This means one can split the not-equal constraint in two systems containing '<' and '>'. Both resulting systems are then checked for their satisfiability. If one of them is satisfiable, the original one is too. In case both are unsatisfiable, then there exists no solution for the original system. This idea is formalized in the following theorem.

Theorem 3.4.1. *Let S be a system of linear constraints and $d = \sum_i a_i x_i \neq 0$ a not-equal constraint. Then holds*

$$S \cup d \text{ satisfiable} \iff S \cup \left\{ \sum_i a_i x_i < 0 \right\} \text{ satisfiable or } S \cup \left\{ \sum_i a_i x_i > 0 \right\} \text{ satisfiable.}$$

Proof. The not-equal constraint splits the solution space into multiple parts. Each of the single parts is again a convex polyhedron. Assume that $S \cup d$ is satisfiable. The single polyhedrons exactly describe the different cases in the union. This means every part of the solution space is exactly described by single parts of the union. No part of the initial solution space is discarded. Assume now that $S \cup d$ is not satisfiable. In this case are also all of the subsets unsatisfiable. If one was satisfiable, the whole system would be satisfiable. Thus, the merged system is satisfiable-equivalent. □

Through the previous theorem, one can already observe the problem for not-equal constraints. Splitting one not-equality into two new sub-systems leads to an exponentially growth in the size of not-equalities. Every not-equal constraint doubles the number of possible systems. In order to deduce UNSAT needs every system to be solved independently.

However, a not-equal constraint is not hard to satisfy. Though it technically splits the polyhedron into a non-convex solution set, the excluded parts are rather trivial. As soon as there is an ϵ environment around any feasible solution to the constraints in S, every not-equality can be fixed. In case a found solution does not satisfy a not-equal constraint, one just has to find that ϵ environment. The idea for this approach is given by the following theorem from [Gre96]. It shows that an exponential blow-up might not be necessary.

Theorem 3.4.2 (Not-Equal Constraints)**.** *Let S be a system of linear inequations and D a set of not-equal constraints. Then $S \cup D$ is satisfiable iff $S \cup d_i$ is satisfiable for all $d_i \in D$.*

In other words, if a satisfying solution for the constraints in S and one not-equal constraint can be found, then there exists a solution for S and all not-equations in G. Thus, every not-equation can be considered independently. The proof for Theorem 3.4.2 can be found in [Gre96].

The procedure to solve not-equal constraints in FMPlex is now described in the following. The FMPlex algorithm is now called to solve a set of linear constraints and not-equal constraints S. In the following denotes $D \subset S$ the set of not-equal constraints. Thus, S is partitioned in L, U, N and D. To the usual combinations of L and U, the variables in D also need to be eliminated. To do so, in every branch is the chosen constraint c additionally combined with the not-equal constraints containing the eliminated variable. Hereby, the constraint c is treated as equality to eliminate the variable in $d \in D$. Now, the disequality may be trivially satisfied by being covered by another constraint. For example, in $\{x \neq 5 \wedge x \leq 4\}$, $x \neq 5$ is trivially satisfied by $x \leq 4$.

In the execution of FMPlex has every branch now a set of not-equal constraints which might differ from the not-equal constraints in its parent. Furthermore, the not-equal constraints are only of interest when a SAT branch was found. If a conflict with a positive linear combination was already discovered, the not-equal constraints are irrelevant. They might only change a SAT instance to an UNSAT instance.

Assume now, that a SAT branch was found. In the original FM algorithm and the previously defined FMPlex algorithm, backtracking from the SAT leaf was only used to build an assignment α. When handling not-equal constraints, this backtracking is needed to unveil conflicting not-equal constraints or to verify that every not-equal constraint is satisfied. Initially, all variables are unassigned. With every backtracked level in the tree, an assignment for the eliminated variable is set. In case the system S is satisfiable, the resulting assignment α satisfied all constraints in S. The suitable values for the variables are computed while backtracking. Within the backtracking, one differentiates between two cases: is the variable *tight* or not. Meaning is there only a point-interval $[x,x], x \in \mathbb{R}$ to choose the variable from or the variable can be chosen from in interval with more than one element. Note that due to implicit eliminated variables, a point interval might also be symbolic. This means variables are implicitly eliminated in a way such that variable y is restricted to be equal to variable x. In case of a point interval, there might be not-equal constraints that can not be satisfied. In the second case, however, all of the not-equal constraints containing the assigned variable can be satisfied.

One can then find an assignment that satisfies all those constraints and propagate it with the knowledge that those not-equal constraints are satisfied. However, if the eliminated variable is tight and breaks any disequation, another form of conflict handling is needed. The conflict might be caused by three reasons:

- The conflict is caused by an unfortunate split. The created constraint forces with another bound the variable on a fixed assignment. This conflict is solved by choosing another branch. In this branch, the variable is then not tight and the not-equation can be solved.

- The conflict can be solved by checking the variables prior to the current level in the computation tree. Take for example a not-equation with initially two variables. After eliminating the first variable, the not-equation has only one remaining variable. If this variable is now tight by accident, the constraint can be solved on a higher level via adjusting the other variable.

- The formula is unsatisfiable. There is no conflict with only inequations but inequations and not-equations form the conflict, e.g. $x \leq 5 \wedge x \geq 5 \wedge x \neq 5$.

In a first step, one has to detect whether the conflict might be resolved, i.e. one of the first two cases occurred. To do so, one considers the constraints which restrict the variable to a point interval. If none of the constraints for either lower or upper bound have a conflict level smaller than the depth of the tree, the conflict might be resolved by backtracking into another branch. However, if that is not the case, the conflict might be resolved by adjusting eliminated variables which are not considered so far. Thus, the variables in the current and the original constraint are compared. In case there is a variable that was not considered so far, one tries to fix the conflict on a higher level. If all these approaches fail, the UNSAT witness needs to be detected. The infeasible subset can be build by taking the constraints forcing the variables on their fixed values and the disequation. It is important to note, that once a not-equation was solved, it does not need to be rechecked on higher levels. It suffices to find one level on which the not-equation is solved. The pseudocode of this algorithm is presented in Algorithm 4.

Algorithm 4 Solving Not-Equal Constraints

procedure CHECKNEQCONSTRAINTS(variable x with partitions L, U, partial assignment α for variables on already resolved levels and disequalities D)
  Insert α in L and U to construct L' and U'
  Interval $B := [max(L'), min(U')]$ ▷ Interval for x
  Assign $\alpha(x) := z$ for $z \in B$
  if $D = \emptyset$ **then**
    return (SAT, α)
  else
    $D' :=$ TOUCHEDDISEQUALITIES(D,x)
    $solved :=$ LOCALSOLVEDDISEQUALITIES(D',x,B,α)
    $T := D' \setminus solved$
    if $T \neq \emptyset$ **then**
      if $max_{l \in L}(cl(l)) = max_{u \in U}(cl(u)) = cl(d_i)$ **then**
        if $\exists$ disequation $d_i \in D'$ with $\alpha \not\models d_i$ **then**
          return Infeasible Subset $c \subset (L' \cup U') \cup \{d_i\}$
      else
        $L'' := argmax_{l \in L'}(cl(l))$
        $U'' := argmin_{u \in U'}(cl(u))$
        Backtrack $min(min_{l \in L''}(cl(l)), min_{u \in U''}(cl(u)))$ levels
    else
      return SAT
procedure TOUCHEDDISEQUALITIES(D,x)
  System $D' := \emptyset$
  for $d_i \in D$ **do**
    if x is var in d_i **then**
      $D' = D' \cup \{d_i\}$
  return D'
procedure LOCALSOLVEDDISEQUALITIES(touched disequalities D', var x, interval B, assignment α)
  $solved := \emptyset$ ▷ Initialize list for solved disequalities
  for $d_i \in D'$ such that x in d_i **do**
    if d_i solved on previous level **then**
      $solved = solved \cup \{d_i\}$
    if Interval B is point-interval **then**
      if $\alpha \models d_i$ **then**
        $solved = solved \cup \{d_i\}$
    else
      $solved = solved \cup \{d_i\}$
  return $solved$

Chapter 4

Improving Simplex

4.1 New Heuristic

As previously discussed exists a close connection between Simplex and FMPlex. Section 2.4 explained that the performance of Simplex depends highly on the heuristic. A natural question is, whether there exists a heuristic that mimics FMPlex. Practically it shows to be beneficial to prefer columns with small amount of entries for pivot operations, [KBD13]. This means, when deciding between two columns for pivoting, it is beneficial to choose the one with fewer entries. Though this heuristic is quite similar to FMPlex, an important difference is that FMPlex does not simply count the number of entries in the column. Furthermore, it divides between positive and negative entries and chooses the smallest column over those.

The part not mimicked by the heuristic is the backtracking property. Simplex does not need to revert pivoting steps, it can directly pivot another basis variable into the nonbasis. Let the number of positive entries in the column of a nonbasis variable $x \in \mathcal{N}$ be p_x and the number of negative entries be n_x. Let $B' \subseteq B$ be the set of basis variables with broken bounds. The *FMPlex*-heuristic orders now the nonbasis variables $\mathcal{N}$ increasing by $\min\{p_x, n_x\}$ for $x \in \mathcal{N}$. Then, a basis variable in B' is searched that is pivotable with the first nonbasis variable in the ordering. In case there is none, a basis variable in B' for the second nonbasis variable is searched. This continues until a pivoting pair is found. By doing so, the pivoted nonbasic variable has a minimal number of sign invariant dependent variables.

This heuristic is applied for the first n steps. Afterwards, another heuristic can be chosen or one continues with Bland's rule. The performance of this heuristic is compared in Chapter 5.

4.2 Improved Disequation Handling

As previously seen can the not-equation handling for FMPlex drastically be improved. The naive approach doubled the systems potentially to be solved with every not-equation. In contrast, the improved algorithm could decide all not-equations in one backwards pass.

The naive approach can be used in the Simplex algorithm in the same way as in FMPlex. Thus, for every not-equation two possible systems are considered.

However, one would now like to transfer the improved algorithm from FMPlex to Simplex. The idea to search for a possibility to handle not-equations without exponential blow-up is rooted in Theorem 3.4.2. It states, that if the system became unsatisfiable by adding not-equal constraints, then the reason for the unsatisfiability lies within one not-equal constraint and not the whole set. Equivalently, it suffices to solve every not-equation independently. If this is possible, there exists an assignment satisfying all constraints simultaneously. In the following the idea to find such an assignment is described. To do so, let D be the set of disequations $d_1, \ldots, d_k$ and α be the current assignment. For every disequation $d \in D$ one checks firstly if it is satisfied. In case it is, there is nothing to do. If it is broken, one has to check whether the assignment can be fixed. It suffices to find one variable in the not-equal constraint which can be altered slightly. Let d' be computed from d by replacing all basis variables with their nonbasis representation, $d' := \sum_{x_i \in \mathcal{N}} \gamma_i x_i$. To find such an update candidate, one iterates through all nonbasic variables x in d'. As soon as a suitable variable with $\alpha(x) < u(x)$ or $l(x) < \alpha(x)$ is found, it is checked whether it can be de- or increased without a dependent basic variable violating its bound. Hereby denotes $u(x)$ the upper bound of varible x and $l(x)$ the lower bound. If such a variable can be found, its assignment $\alpha(x)$ can directly be updated such that d is satisfied. While updating x special attention needs to be given to the other not-equal constraints in D. One has to make sure that the update of x does not violate another not-equal constraint that was satisfied previously. This can be ensured by reducing the update value until no additional not-equation is violated. If this is the case, the update of x can plainly be reduced until no further constraint is conflicting. As the dependent basic variables were checked before, this operation is always possible.

In case no such possible update value is found, an unsatisfiable subset can be generated. A UNSAT instance was detected. Assume now that all variables in d' can not be updated in any direction. Thus, every variable x in d' is either restricted by its own bounds or the bounds of the basic variables depending on it. Meaning, for all variables x in d' holds that its decrement is prevented by either its own bound, $\alpha(x) = l(x)$, or the bound of a basic variable. This happens for dependent basic variable x' when its coefficient γ in d' is positive and $\alpha(x') = l(x)$ holds or the coefficient is negative and $\alpha(x') = u(x)$. The same argumentation holds for the incremental case, i.e. $\alpha(x) < u(x)$. As shown in the explanation of the Simplex tableau, only slack variables have bounds. Original variables however are always unbounded. Thus, the restricting variables are always slack variables. Every slack variable was created for exactly one constraint, which is then collected. It suffices to collect one constraint for the increment and one for the decrement. The pseudo-code for the previous Algorithm is described in 5.

Theorem 4.2.1. *Let S be a system of linear constraint and D be the set of not-equal constraints. Algorithm 5 returns* SAT *iff $S \cup D$ is satisfiable. Otherwise, it constructs an infeasible subset.*

Sketch of Proof. Starting with the first part of the theorem. It is to show that the algorithm returns only SAT iff $S \cup D$ is satisfiable. Note that the algorithm does not violate the invariants of the Simplex algorithm. Thus, in every step of the algorithm is α a valid assignment and the tableau itself is consistent. The algorithm is only called when a satisfying assignment for the constraints in S was found. The condition in lines 12-14 and 24-26 ensure that the updated assignment also satisfied all constraints in S. Implying that the algorithm only returns SAT when it indeed found a satisfying

Algorithm 5 Solving Not-Equal Constraints in Simplex

```
1: procedure CheckNEQConstraints(disequations D, assignment α)
2:     for Disequation d_i ∈ D do
3:         if α ⊭ d_i then
4:             Conflict C = FixDisequtions(d_i, D, α)
5:             if C ≠ ∅ then return C ∪ {d_i}
       return SAT
6: procedure FixDisequation(d_i, D, α)
7:     Let d' be d_i expressed in nonbasis variables x_1, ..., x_k
8:     Let C := ∅                          ▷ Set of constraints forming the conflict
9:     for Variable x_i in d' do
10:        B_+ := slack^+(x_i)
11:        B_- := slack^-(x_i)
12:        if α(x_i) < u(x_i) then
13:            if ∀x' ∈ B_+ : α(x') < u(x') and ∀x' ∈ B_- : α(x') > l(x') then
14:                Let δ be a feasible increase for x_i considering D
15:                update(α, x_i, δ)                          ▷ Update assignment
16:                return ∅
17:            else
18:                Let constraint b' restrict x' ∈ B_+ to l(x') or u(x')
19:                C := C ∪ {b'}
20:        else
21:            Let constraint b' restrict x_i to u(x_i)
22:            C := C ∪ {b'}
23:        if α(x_i) > l(x_i) then
24:            if ∀x' ∈ B_+ : α(x') > l(x') and ∀x' ∈ B_- : α(x') < u(x') then
25:                Let δ be a feasible decrease for x_i considering D
26:                update(α, x_i, −δ)                         ▷ Update assignment
27:                return ∅
28:            else
29:                Let constraint b' restrict x' ∈ B_- to l(x') or u(x')
30:                C := C ∪ {b'}
31:        else
32:            Let constraint b' restrict x_i to l(x_i)
33:            C := C ∪ {b'}
34:    return C
```

assignment. Now it is still to prove that for every satisfiable $S \cup D$ a satisfying assignment is found. This property also follows from the Simplex algorithm. If $S \cup D$ is satisfiable, there exists a satisfying assignment α'. With the help of Theorem 3.4.2, all not-equal constraints can be considered independently. Let $d \in D$ be not satisfied by the current assignment α. This assignment differentiates from α' for at least one of the nonbasis variables in d. Thus, the bounds of this variable form no point-interval and it can either be incremented or decremented. From the consistency of the Simplex tableau follows that this change can be found by Algorithm 5. It is important to note that the other not-equal constraints are considered in the update. In case another one would be violated by the update operation, the update value is reduced until no other constraint is violated. Thus, the equivalence is proven.

Assume now that $S \cup D$ is not satisfiable. There exists a $d \in D$ for which holds $S \cup \{d\}$ is unsatisfiable. Let $x_1, \ldots, x_k$ be the nonbasis variables d depends on. The algorithm collects in set C, line 8, for every variable $x_i \in \{x_1, \ldots, x_k\}$ constraints. In case x_i is bounded to its upper or lower bound, that constraint is collected. In case $l(x_i) < \alpha(x_i) < u(x_i)$ holds, constraints of basis variables restricting the nonbasis variable are collected. As every bounded variable is by construction a slack variable, the corresponding constraints are unique. Again one constraint for the upper bound and one for the lower bound is collected. At most $2 \cdot k$ constraints are collected. It might be less due to equality constraints. These collected constraints in C witness the impossibility to update any of the nonbasic variables in d. Thus, $c \cup \{d\}$ form an infeasible subset. □

Chapter 5

Evaluation

This chapter gives the technical background of this thesis. Firstly, the implementation of the presented algorithms is described. The algorithms are tested on the quantifier-free linear real arithmetic, QF_LRA, benchmarks in the *SMT-LIB* dataset `http://smtlib.cs.uiowa.edu`, [WCD+19].

5.1 Implementation

Multiple algorithms were implemented while writing this thesis. All implementations are integrated into the SMT-RAT project, `https://smtrat.github.io` [CKJ+15]. The structure of the SMT-RAT project is not described in detail as it was not the focus of this work. Roughly, SMT-RAT consists of multiple modules which can be exchanged and connected to build strategies to solve SMT formulas. To do so, modules need to provide special functions. One function to add formulas to the module, AddCore, one function to remove formulas from the module RemoveCore and a function to check the status of the current active formula, CheckCore. To check the benchmarks, the benchmarking program benchmax from the SMT-RAT project is used. When calling a single instance, multiple return values are possible.

- *SAT:* The instance was correctly identified as SAT.
- *UNSAT:* The instance was correctly identified as UNSAT.
- *UNKNOWN:* The instance could not be successfully identified.
- *WRONG:* The instance was wrongly identified as SAT or UNSAT.
- *TIMEOUT:* The instance could not be identified in the given run-time.
- *MEMOUT:* The instance could not be identified with the given memory.
- *SEGFAULT:* In the execution occurred an error, e.g. an assertion failed.

For a detailed definition of modules and SMT-RAT, it is referred to the documentation. All benchmarks are executed on 2.1 *GHz* AMD Opteron Processor 6172 with a timeout of 5 minutes and 10 *GB* memory.

The existing LRA-Module implements several versions of the Simplex algorithm. Though there existed an implementation of Fourier-Motzkin, it had to be rewritten

from scratch due to too many bugs. The Fourier-Motzkin algorithm and the FMPlex adaptation are both implemented in the FouMou-Module. The exact implementations are described in the following. Afterwards, a comparison between the single run-times and some statistics are given.

5.1.1 Fourier-Motzkin

The Fourier-Motzkin algorithm was implemented rather straightforward. When selecting the next variable to eliminate, we aim for a small following constraint set. To do so, the variable with the smallest upper or lower set is chosen by evaluating $\min_x(|L|, |U|)$ for all variables x. This is done to delay the doubly exponential blow-up as far as possible. To produce the next system, all combinations between lower and upper bounds are formed. In contrast to the original algorithm, they are not directly inserted into the system. For every new constraint, it is checked whether it is worth to be inserted. This means, if either the constraint itself or a stronger parallel version is already contained, it is not inserted. However, if it is stronger than an already present constraint, the weaker constraint is removed from the system. There could be a more eager approach, e.g. detect linear dependency on constraints. As the systems to be checked become very large, a fast criterion was chosen. Additionally, with improvements like Imbert's acceleration, a more eager approach is not expected to pay off the increased run-time.

Afterwards, the FM module was extended by the two Imbert accelerations and Chernikovs criterion, see Algorithm 2. To apply Imbert's accelerations the three additional sets, history, effectively eliminated variables and implicit eliminated variables, needed to be considered. Those sets are created simultaneously with the new constraint. The previously discussed disequation handling for FMPlex was not implemented in the FM algorithm. When FM discovers a SAT instance and has disequations, it returns UNKNOWN. Most of the instances can still be solved with help of the underlying SAT solver, but some terminate with an UNKNOWN response. The improved disequation handling was not implemented, as it did not seem to be promising.

5.1.2 FMPlex

Furthermore, several versions of the FMPlex algorithm were implemented. To detect every conflict, multiple structures are needed. Every branch needs to maintain its set of constraints and their origins. Furthermore, the linear combination of every constraint is stored. One could also recompute the linear combination for every constraint, but this solution is more time-efficient. To be able to apply the improved not-equation handling, every branch in FMPlex needs to maintain its not-equations. Thus, those are also stored in a structure and updated on every level. For every disequation, its current, resolved form and its original constraint are preserved. The original constraint is needed to detect as early as possible when a constraint can not be fixed anymore. As the linear combination and the new constraint can be formed in similar operations, their computation is homogenized.

After implementing the classical version, Algorithm 3 which is stated above, several practical improvements were made. The original algorithm backtracks every time a conflict is found. The solver is denoted as FMPlex. In case a global conflict is found it backtracks to the root and returns UNSAT. Otherwise, it continues its exe-

cution. This way no conflicts are missed. The approach is called *eager FMPlex* and its solver is referred to as EAGER FMPLEX. In many of the branches, where already a local conflict was found, one can still find a global conflict and deduce UNSAT. If the branch can not deduced to be globally UNSAT, it is a dead-end. However, the point between discovering a local conflict lies mostly behind the peak of constraints. Thus, the most expensive operations are already executed. The cost of finishing the branch is then rather small with the large benefit of being able to conclude UNSAT rather early in the execution.

5.1.3 Simplex

To be able to compare the new Simplex heuristic correctly, a Simplex heuristic only consisting of Bland's rule was created. Based on this, the minimal sign heuristic was also implemented. Additionally, a hybrid approach consisting of FMPlex and Simplex was implemented. Due to limitations of time, it was not possible to implement the improved not-equality handling for Simplex. An experimental version of this algorithm was implemented and proved to be way more efficient than the naive approach to split the not-equations. Sadly, it was not possible in the given time to additionally implement it completely.

5.2 Comparisons

In this section the single algorithms are compared and evaluated. To do so, several aspects and statistics of the algorihthms are considered. In order to keep the single plots clear, every algorithm is abbreviated:

- *FMPlex*: Original FMPlex, Alg. 3
- *Eager FMPlex*: FMPlex without backtracking on local conflicts
- *FM*: The classical Fourier-Motzkin algorithm as presented before, Alg. 1
- *FM+Imbert*: Fourier-Motzkin extended with Imberts accelerations, Alg. 2
- *Simplex*: Simplex algorithm with only Blands rule
- *Min. Sign*: Simplex Algorithm with FMPlex heuristic
- *Z3*: Open source theorem prover by Microsoft Research

It is important to note that most of the incrementality of Simplex was deactivated. This was done to make Simplex comparable to FMPlex which was not implemented incrementally.

The following table gives an overview of the single algorithms and their outputs. The solver kills the execution of an instance as soon as it exceeds the memory threshold of 10 *GB* or the time limit of 5 *m*. To set the algorithms into relation with a commercial, state of the art solver, *Z3* is included, [dMB08]. Z3 is a theorem prover developed by *Microsoft Research* which is used in the following table as comparison.

A few insights can already be gained from Table 5.1 and Figure 5.1. Firstly, not surprising outperforms the Imbert Acceleration the classical Fourier-Motzkin algorithm in every part. Both, SAT and UNSAT instances are improved when choosing the Imbert Acceleration. Furthermore, as expected due to the reduced constraints

	Fourier-Motzkin	FM+Imbert	FMPlex
SAT	369	370	410
UNSAT	365	377	366
UNKNOWN	21	21	0
TIMEOUT	864	851	843
MEMOUT	29	29	29
SEGFAULT	0	0	0
WRONG	0	0	0

	Eager FMPlex	Bland	Min. Sign	Z3
SAT	413	521	489	852
UNSAT	368	381	361	592
UNKNOWN	0	0	0	0
TIMEOUT	838	717	769	204
MEMOUT	29	29	29	0
SEGFAULT	0	0	0	0
WRONG	0	0	0	0

Table 5.1: Overview of Performances for all Implemented Algorithms

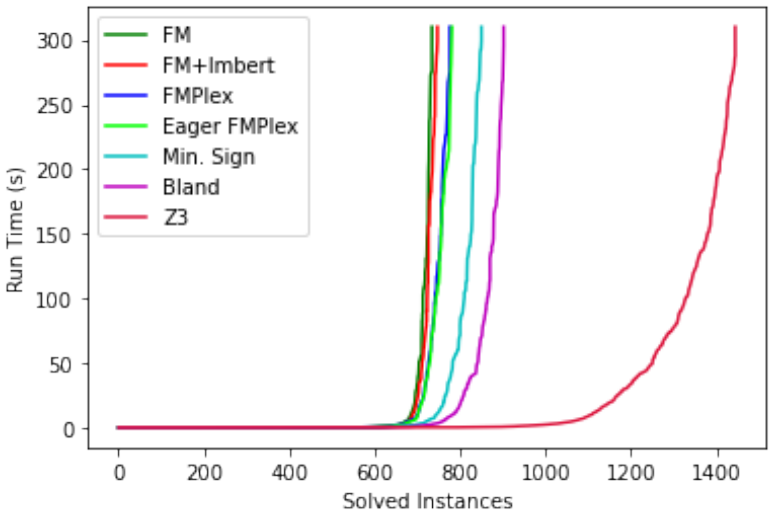


Figure 5.1: Progress of Solved Instances by Running Time and Algorithm

seems FMPlex is in advantage of FM+Imbert. The difference between FMPlex and Simplex is quite large on the first sight. The difference in solved SAT instances is way larger than in UNSAT instances.

In the further plots is Z3 not included to make a detailed analysis easier. The plots are more detailed on the solver implemented in SMT-RAT.

5.2.1 Evaluating FM

Figure 5.2 compares the number of created constraints between FM+Imbert and FM. It shows a difference between easy instances, which are solved rather fast and harder instances. The easy instances are depicted in the bottom left corner. Surprisingly, FM+Imbert creates on some of the instances more constraints than FM. A reason for this might be that FM can detect conflicts faster than FM+Imbert. Additionally, FM creates more redundant constraints which are detected by the trivial redundancy check. FM+Imbert does not create as many redundant constraints. The figure also demonstrates the expected long-time behaviour of FM+Imbert. In the long run, FM+Imbert creates fewer constraints than the classical version. As soon as the one million constraints are exceeded, FM+Imbert creates fewer constraints.

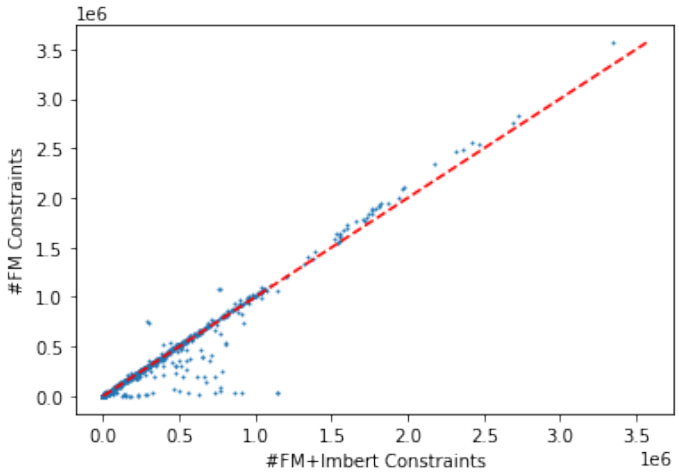
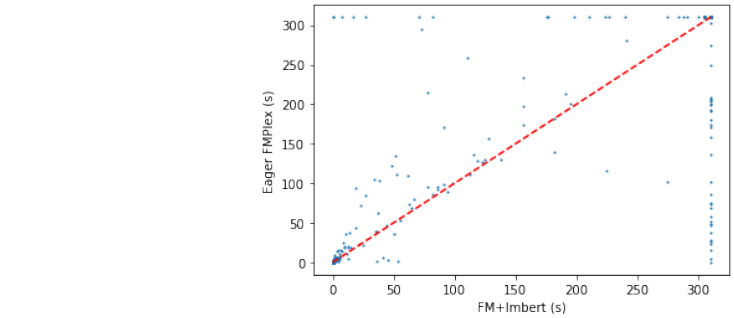


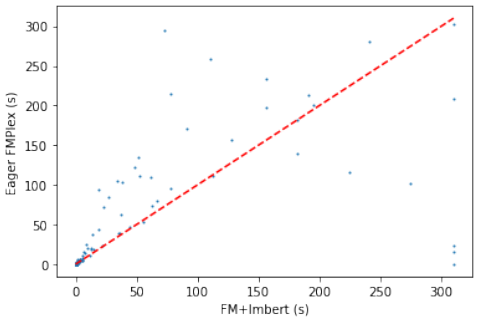
Figure 5.2: Comparison Between Added Constraints in FM+Imbert and FM

When comparing FM+Imbert with FMPlex, it already shows that FMPlex solves less UNSAT instances than FM+Imbert but more than FM. However, FMPlex solves 40 more SAT instances than FM+Imbert against 11 less UNSAT instances. One reason could be that the doubly exponential blowup in FM+Imbert makes it impossible for it to find the SAT instances in time. The benefit of the doubly exponential blow up is that it can discover conflicts earlier than FMPlex. The difference in solved UNSAT instances is smaller for the eager version. The eager version solves 2 UNSAT instances more than FMPlex. Even the eager FMPlex algorithm can run into multiple deadens, which lead to a timeout before the conflict is found. This behaviour is analyzed in the following.

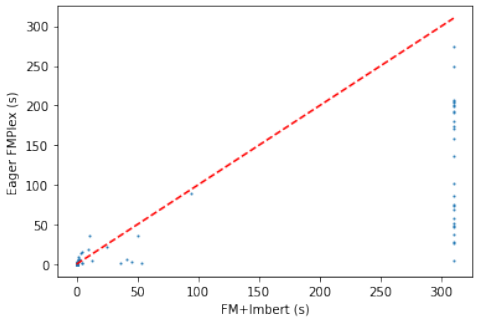
In Figure 5.3a the running time on all instances is compared. Here it seems that FMPlex runs slower than FM+Imbert on instances which both can solve.



(a) On All Instances



(b) On UNSAT Instances



(c) On sAT Instances

Figure 5.3: Run Time Comparison Between FM+Imbert and Eager FMPlex in Seconds

Considering the number of instances which FMPlex solves, FM+Imbert times out on many of them. When splitting the instances solved by eager FMPlex into satisfiable and unsatisfiable instances, a more diverse image arises. On the left side, in Figure 5.3b the UNSAT instances solved by eager FMPlex are shown. On the right side, in Figure 5.3c the SAT instances are shown. As previously described, FMPlex needs longer than FM+Imbert to find some conflicts. However, on SAT instances it is generally in advantage. As Figure 5.4 shows, constructs the eager FMPlex way less constraints than FM+Imbert. Thus, it is way faster in deciding SAT instances.

5.2.2 Evaluating Eager FMPlex

Furthermore, FMPlex and eager FMPlex are compared. In Figure 5.5 the running times of FMPlex and eager FMPlex are compared. It shows that the eager version is in general faster than the original one. One explanation for the superiority of the eager approach might lie within the size of the produced computation trees. In Figure 5.6 the histogram over the maximum number of branches within the whole instance execution is shown. The proportion of solved instances is coloured blue and timeouts are coloured red. It shows, that the eager approach can avoid some of the wide trees of width 15 - 20 and shifts the number of splits mainly to a tree-width of 1-5. FMPlex has the same peaks in the histogram as eager FMPlex. A big difference

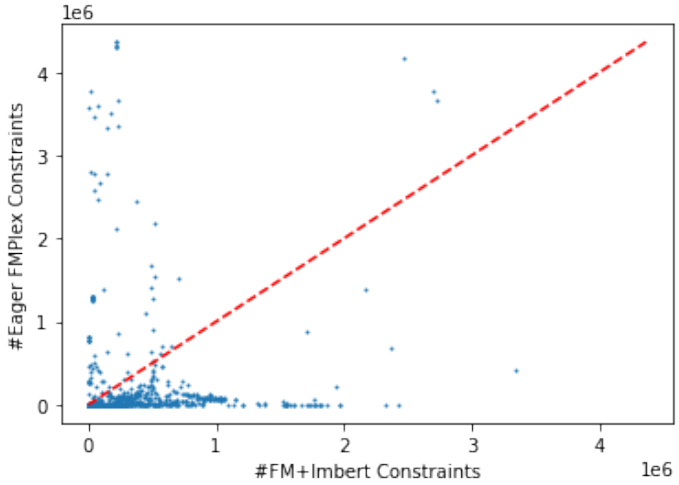


Figure 5.4: Comparison Between Added Constraints in FM+Imbert and FM

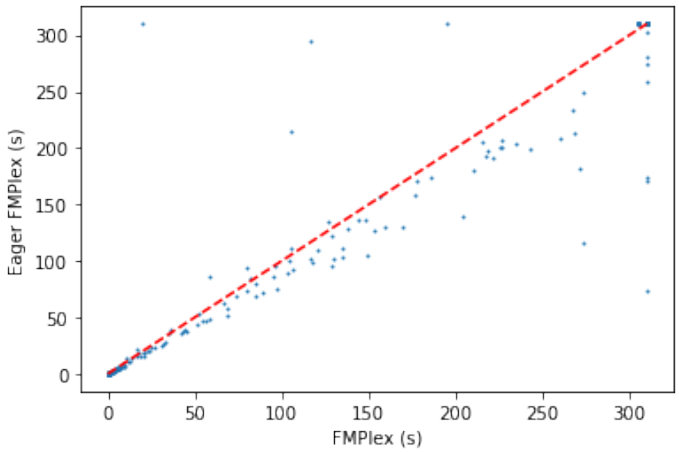


Figure 5.5: Run Time Comparison Between FMPlex and eager FMPlex in Seconds

is the size of the outliers. Through its depth traversal, the eager approach encounters some trees with a width of 35 and above. FMPlex does not even encounter them. Though the difference is rather small, it is significant enough to explain the few more instances solved. The figure also shows that through the exponentially growth of the tree, FMPlex generally times out on instances with a split number of 6 and above. Figure 5.7 depicts the difference between the two plots in Figure 5.6. The number of instances for a given width of FMPlex are subtracted from the number of instances in eager FMPlex. It shows that FMPlex has more instances with a maximum width of 6 and eager FMPlex more with a width of 5. While FMPlex can solve only a few with width 6, eager FMPlex can solve many of width 5. These two widths explain the difference in solved instances between the two approaches. For higher widths exist differences in the number of instances, but algorithms time out on

them. One can also observe that the difference is small. The peak is at width 6 with a difference of 22 instances. There exist some exceptions, but keeping the number of splits small is necessary for both versions to terminate. Lastly, the amount of created

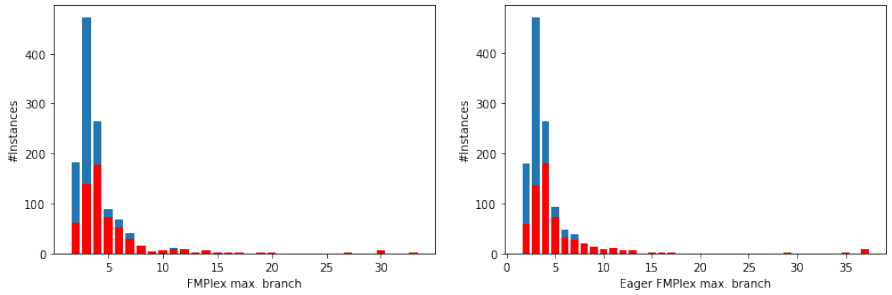


Figure 5.6: Comparison of Maximum Number of Branches Between FMPlex (l.) and Eager FMPlex (r.)

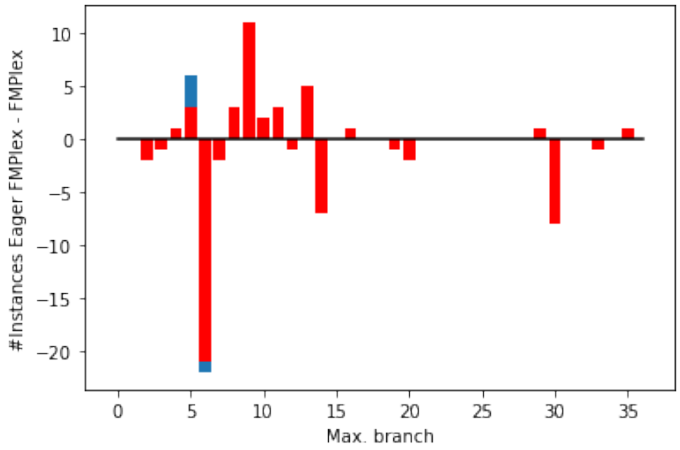


Figure 5.7: Difference Between Eager FMPlex and FMPlex in Instances per Maximal Width

constraints between FMPlex and the eager approach is analyzed, see Figure 5.8. It is important to note that this comparison does not only include the newly constructed constraints but also the copied ones. This is done to accredit that FMPlex backtracks more often than eager FMPlex but also to accredit that eager FMPlex considers more constraints in the depth of branches. Still excluded are constraints which are trivially detected to be redundant. It shows that both version consider roughly the same amount of constraints. Though the variance is not small around the equalizing line, none of both algorithms is in advantage.

The comparison of solved instances reveals that in our test scenario eager FMPlex performs slightly better than FMPlex. The detailed analysis however revealed only very small differences in the size of computation trees in favour of eager FMPlex.

Now, eager FMPlex is compared with the min. Sign heuristik. As Figure 5.1

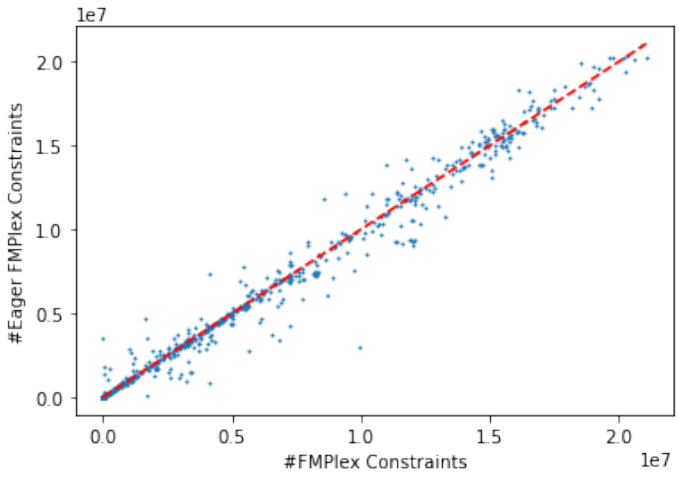


Figure 5.8: Comparison Between Constraints in FMPlex and Eager FMPlex

shows, solves the standard Simplex algorithm with the MIN. SIGN heuristic more instances than FMPLEX. This can be observed on SAT and UNSAT instances. Figure 5.9a shows that SIMPLEX is in general faster. When considering only instances decided by EAGER FMPLEX to be SAT, Simplex clearly outperforms FMPLEX, Figure 5.9c. When considering the UNSAT instances, a different picture emerges, Figure 5.9b. FMPLEX is capable of solving many instances Simplex can not solve. Additionally, on even more it is faster than Simplex. Nevertheless, Simplex is significantly faster on many of the instances.

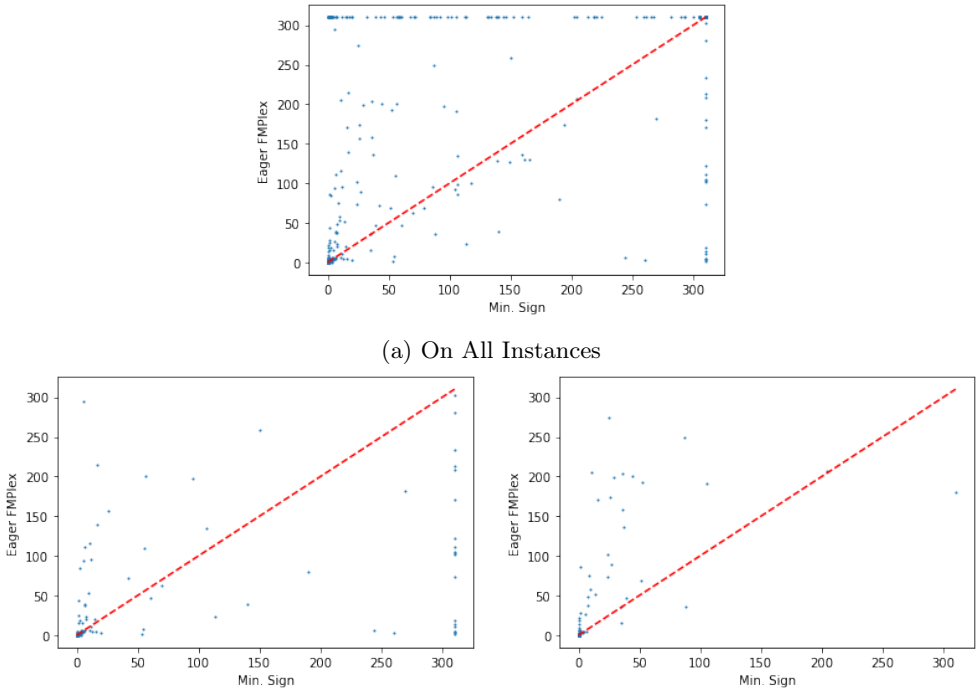
5.3 Hybrid Method

The previous analysis reveals that EAGER FMPLEX has in some instances an advantage over Simplex. Nevertheless, on most it can not compete with it. However, as variable elimination method it showed to be superior to Fourier-Motzkin.

It seemed to be promising to combine Simplex with FMPlex. We tried to reduce the constraint system as long as the increase of constraints were not too big. FMPlex was executed until a certain level was reached. As soon as a split larger than an arbitrary constant was reached, Simplex is called on the remaining equations. By doing so, the number of variables was reduced and the number of branches was still small.

In case Simplex would return UNSAT, the conflicting subset is checked. If it is a global conflict, the instance is unsatisfiable. The Simplex model is capable of returning multiple conflicts within the tableau. Every line of the tableau needs to be checked for a conflict. Through this, the EAGER FMPLEX approach would not be needed, as Simplex can directly find all of the present conflicts. If all the returned conflicts are local, FMPlex would backtrack and pursue another branch and then call Simplex again.

In case Simplex returns SAT, the whole system is satisfiable. A technical problem emerged rather fast. The current implementation is not capable of deciding trivial not-equations as soon as the assignment does not fulfil them by default. We encoun-



(a) On All Instances

(b) On UNSAT Instances

(c) On SAT Instances

Figure 5.9: Comparison Between Eager FMPlex and Min. Sign Heuristic in Seconds

tered examples in which Simplex was not able to decide that e.g. $x \neq 0$ was satisfiable even with x not occurring in any other constraint. Simplex assigns by default every variable to 0 and did not have any slack variable to pivot it with. To solve the technical limitations of the current implementation is one of the future goals presented in Section 5.5. Thus, Simplex returned on most of the instances UNSAT and FMPlex was pursued further. This leads to an increased running time for both of the algorithms. For FMPlex because it has to wait for Simplex to return UNKNOWN. For Simplex because it could not split the not-equation into two parts and was called on systems it could not solve.

In practical applications is FMPlex most times far away from its theoretical worst-case behaviour. This can also be seen from the number of splits in the previous comparisons. The reason is that the input instances are sparse. Their matrix representation is not nearly filled and a vast majority has 0 entries. The hardness for many instances lies within *backdoor* variables. Those variables connect multiple constraints such that the instance can not be solved easily, see [WGS03]. For this reason is the hybrid method appealing. It is expected to work quite well as it can reduce the number of variables quite fast and runs then a sophisticated algorithm to solve the instance thoroughly. However, this approach requires the Simplex algorithm to find answers on all instances without splitting the input.

5.4 Summary

As the previous analysis showed is there no single conclusion. The presented approach gives a whole new view on variable eliminations. It can additionally be used after applying the Gauss elimination to reduce the number of variables even further. The analysis proved it to be superior to FM in practical applications as well as in theoretical properties. Additionally, FMPlex is even superior in terms of solved instances. It was shown that even the advanced Imbert acceleration produces more constraints than FMPlex.

When comparing FMPlex to Simplex, it performs rather poorly. Nevertheless, there are instances on which FMPlex outperforms Simplex. On many instances, a mixed review emerges. Simplex can undercut many of the FMPlex running times, but not all of them. Especially on unsatisfiable instances is FMPlex able to defeat Simplex. When comparing those instances one has to keep in mind that Simplex still uses the naive not-equation handling.

Furthermore, due to FMPlex defeating Simplex on a couple of instances the expectations towards a hybrid method rise. The idea to apply FMPlex as variable elimination method on instances with many variables is still expected to pay off. On this part of the computation tree performs FMPlex very well. Simplex would profit from a reduced input system. This approach should be revisited after Simplex is indeed capable of handling not-equations.

Moreover, it is to conclude that an advanced not-equation handling for Simplex was discovered through FMPlex. The backtracking behaviour from FMPlex was transferred to Simplex. Due to its theoretical properties, it is expected to perform way better than the current splitting method. So far, we observed that Simplex is generally not capable of handling not-equations. Due to reasons of time and limitations of the current implementation was the not-equation handling for Simplex not finalized.

5.5 Future Work

5.5.1 Implementing Not-Equation Handling in Simplex

FMPlex gained a huge boost in solved instances as soon as not-equal constraints could be handled successfully. The same impact is well-founded expected on Simplex. The hybrid model showed that Simplex was not able to find a solution on many instances. It would rather start to split the solution polyhedron. Those splits are also for Simplex quite expensive operations. Though Simplex is implemented incrementally, those splits are then processed throughout the whole instance execution and cause and exponential growth in systems to solve. Implementing the advanced not-equation handling in Simplex would prevent the splitting of instances. Thereby, the exponential growth in the SAT solver is eliminated. Simplex would solve way harder instances than now. Currently, unnecessary constraints are learned. Concluding, implementing the not-equation handling is expected to give a huge boost in solved instances.

5.5.2 Incrementallity of FMPlex

FMPlex can be implemented incrementally. Until now, every addition of constraints triggers a whole new execution. Incrementality is a key concept in lazy SMT-solving. To make FMPlex incremental, the computation branch with its whole history could

be stored. When a new constraint is added, one has to backtrack to the highest present variable in the new constraint and retrigger all computations from before. This step can not be shortcutted, as the implied constraint from the new constraint is needed on later levels. While incrementality gives in Simplex a large boost, its impact on FMPlex is rather limited. It might accelerate the computation, but no significant speedup is expected due to several reasons.

Firstly, it's highly unlikely for an added constraint to be completely independent from the eliminated variables. Probably it is connected to the main working constraints. In this case, major parts of the tree need to be backtracked. Especially in the middle of the tree.

Secondly, those backtracked parts are the expensive operations in FMPlex. Comparable to FM, the middle part of the computation is the most expensive one. In this part, the exponential growth happens. Due to the reduction in constraints, it is not as expensive as in FM.

However, if this computation needs to be redone, the saved computations from the beginning are negligible given the computations in the middle of the tree.

5.5.3 Nonbasis Singularity

Some theoretical aspects of FMPlex which are believed to hold should also be proven. An important aspect of Simplex is, that if a nonbasis is visited twice, it starts to cycle. To prevent this from happening, elaborate heuristics are used in nowadays solver. As described in the Simplex section, there is a wide variety of heuristics. Many of them work in combination with Bland's rule to guarantee termination. Due to the similarity between FMPlex and Simplex, it is of interest whether FMPlex visits any nonbasis more than once. This was so far only proven for a modified version. Nevertheless, it is an important proof to defend FMPlex from Simplex. In case FMPlex would visit some basis twice, one could reduce the number of constraints again easily. As this thesis also presented several modifications of FMPlex it is from interest whether there is a common property that prevents this from happening. Currently, it is assumed that this property only holds for the modified version, but not for the original one.

5.5.4 Global Conflicts

Correlated to the previous point, one would like to prove that FMPlex never fails to reveal a global conflict on UNSAT instances. Thus, a conflict which enables FMPlex to backtrack to the root. The pseudocode considers the case that all branches of FMPlex find a conflict but none finds a global one. This means the case that no branch contains a global conflict, but all contain a local one with only little to no backtracking. We would like to prove that FMPlex always discovers a global conflict and never runs into the case that the conflict needs to be created from a set of conflicting branches. This behaviour can also be observed in practice. Probably, the proof needs to distinguish between the eager approach and the classical one. The current claim is, that the eager version always discovers the global conflict. For the classical FMPlex algorithm, it is not clear whether the global conflict could be missed due to backtracking. This means, that the branch with the global conflict also contains a local conflict and this local conflict leads to backtracking and thus missing the global conflict. In future work, we aim to prove both of the previous claims.

5.5.5 Hybrid Method

Previously, the first approach on a hybrid method between FMPlex and Simplex was presented. Due to the not-equation handling in Simplex, the method did not achieve the expected results. It was forced to execute FMPlex to large parts to avoid the UNKNOWN answers from Simplex. In the future, we aim to rework this approach. FMPlex is still a pretty fast quantifier elimination procedure, superior to Fourier-Motzkin. Additionally, it can be very fast applied in the first few instances without a large increase in constraints. FMPlex can be used as advanced preprocessing for Simplex. Afterwards, Simplex is then called on a system with fewer variables and, depending on when it is called, fewer constraints.

Chapter 6

Conclusion

In this thesis, a novel variable elimination procedure is presented, called FMPlex. Additionally, several modifications with special properties were constructed. The eager version, which follows every branch until its end, proved to perform best on the QF_LRA benchmarks in SMT-LIB. It showed to outperform the Fourier-Motzkin variable elimination procedure even with the Imbert accelerations by far. It was proven to be *sound* and *complete* and to have strong connections with Simplex and Fourier-Motzkin, see Theorems 3.3.1 and 3.3.2. Furthermore, Theorem 3.1.5 shows that it produces no constraints which could be removed by the Imbert accelerations. To perform well in SMT solving, the FMPlex algorithm was extended on strict, equal and not-equal constraints. While the extension to strict and equal constraints was quite similar to Fourier-Motzkin, a novel not-equal constraint handling was developed.

After discovering the similarity between FMPlex and Simplex, we transferred the not-equal handling from FMPlex to Simplex. We achieved to remove an exponential overhead for solving not-equal constraints. To the best of our knowledge, this was not done before. Additionally to the not-equal handling, a FMPlex-Simplex heuristic was designed. It showed to be a specialization from the widely used shortest-column heuristic.

Afterwards, the single algorithms were compared on the QF_LRA benchmarks in the SMT-LIB library. To compare all algorithms, the classical Fourier-Motzkin algorithm, Imbert accelerations, FMPlex in several versions and the Simplex heuristic were implemented.

The comparison revealed that FMPlex is powerful and superior to Fourier-Motzkin, but it does not manage to defeat Simplex. In the detailed comparison, we saw FMPlex defeating Simplex on some instances. This led to the idea to use a hybrid method where FMPlex takes the part of an advanced prepossessing and Simplex afterwards solves the instance.

However, the performance difference between FMPlex and the min. Sign Simplex heuristic was larger than expected. As the future work section mentions, some parts can still be improved in FMPlex. However, most of them are theoretical and the practical ones are expected to have a rather small impact.

Nevertheless, the ideas in FMPlex led to novel not-equation handling within Simplex. Furthermore, FMPlex proved to be an efficient variable elimination tool. Either FMPlex or the ideas developed while constructing FMPlex can certainly be reused in other tools.

Bibliography

[AS80] Bengt Aspvall and Richard E Stone. Khachiyan's linear programming algorithm. *Journal of Algorithms*, 1, 1980.

[Axl97] Sheldon Axler. *Linear algebra done right*. Springer Science & Business Media, 1997.

[AZ96] Nina Amenta and Günter Ziegler. Deformed products and maximal shadows of polytopes. In *Advances in Discrete and Computational Geometry*, 1996.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. Handbook of satisfiability. 185, 2009.

[Bix02] Robert E Bixby. Solving real-world linear programs: A decade and more of progress. *Operations research*, 50, 2002.

[Bla77] R. Bland. New finite pivoting rules for the simplex method. *Math. Oper. Res.*, 2, 1977.

[BZ15] Sergey Bastrakov and N Zolotykh. Fast method for verifying chernikov rules in fourier-motzkin elimination. *Computational Mathematics and Mathematical Physics*, 55, 2015.

[Che63] S. N. Chernikov. Contraction of finite systems of linear inequalities. *Dokl. Akad. Nauk SSSR*, 152, 1963.

[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*. Springer, 2015.

[CLRS90] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms. *Cambridge MA*, 1990.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 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*. Association for Computing Machinery, 1990.

[DDM06] Bruno Dutertre and Leonardo De Moura. Integrating simplex with dpll (t). *Computer Science Laboratory, SRI International*, 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and algorithms for the construction and analysis of systems*, 2008.

[Far02] Julius Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1902, 1902.

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

[Grc11] Joseph F Grcar. Mathematicians of gaussian elimination. *Notices of the AMS*, 58, 2011.

[Gre96] Harvey J Greenberg. Consistency, redundancy, and implied equalities in linear systems. *Annals of Mathematics and Artificial Intelligence*, 17, 1996.

[Imb90] Jean-Louis Imbert. About redundant inequalities generated by fourier's algorithm. In *Artificial Intelligence IV*. 1990.

[Imb93] Jean-Louis Imbert. Fourier's elimination: Which to choose? In *PPCP*, volume 1, 1993.

[KBD13] Tim King, Clark Barrett, and Bruno Dutertre. Simplex with sum of infeasibilities for smt. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013.

[KBD+17] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks, 2017.

[KM72] Victor Klee and George J Minty. How good is the simplex algorithm. *Inequalities*, 3, 1972.

[Koh67] David A Kohler. Projections of convex polyhedral sets. Technical report, Berkley University, 1967.

[Kuh56] Harold W Kuhn. Solvability and consistency for linear equations and inequalities. *The American Mathematical Monthly*, 63, 1956.

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

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

[PSS03] Konstantinos Paparrizos, Nikolaos Samaras, and George Stephanides. A new efficient primal dual simplex algorithm. *Computers Operations Research*, 30, 2003.

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. 1998.

[ST04] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51, 2004.

[WCD+19] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11, 2019.

[WGS03] Ryan Williams, Carla P Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI*, volume 3, 2003.

[WN99] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*, volume 55. John Wiley & Sons, 1999.