Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

BACHELOR OF SCIENCE THESIS

AN INCREMENTAL ADAPTION OF THE FMPLEX METHOD FOR SOLVING LINEAR REAL ALGEBRAIC FORMULAS

Svenja Stein

Examiners:
Prof. Dr. Erika Ábrahám
Jasper Nalbach

Additional Advisor:
Prof. Dr. Christina Büsing

Aachen, 24.05.2022

Abstract

Satisfiability modulo theories is the problem of deciding the satisfiability of a Boolean formula whose atoms come from a background theory. We present two procedures for solving SMT problems with sets of constraints from the background theory of linear real arithmetic: The Fourier-Motzkin variable elimination which repeatedly eliminates variables by combining all of their upper bounds with all of their lower bounds and the Simplex algorithm which operates by pivoting so-called basic and non-basic variables in a tableau in order to find an assignment for the variables that satisfies the given constraints. These methods have recently been combined into the *FMPlex* method which behaves similarly to the Fourier-Motzkin variable elimination, but chooses an assumed greatest lower bound or a smallest upper bound to compare to all opposite bounds and all other bounds of the same time. As such choices can be wrong, it may be necessary to backtrack to previous decisions and try another choice. In lazy SMT solving, we might have to check a set of constraints for satisfiability on one call, while the next requires us to check this very set again, only in conjunction with additional constraints. This is the incentive for presenting an incremental version of the algorithm which reuses its previous results and models. Apart from adding incrementality, we presented appropriate data structures and heuristics as well as different manners of backtracking. These aspects were then implemented, tested and evaluated.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Svenja Stein
Aachen, den 24. Mai 2022

Acknowledgements

I would like to thank Prof. Erika Abraham for the opportunity to write this thesis and her extensive, patient and inspiring support for the theoretical side of it, as well as Jasper Nalbach for the support in the technical aspects.
Additionally, I would like to thank my mother, my friends and my cat for keeping me sane during this wondrous journey.

Contents

1 Introduction **9**

2 Preliminaries **11**
2.1 Satisfiability Modulo Theories 11
2.2 Background Theory: Linear Real Arithmetic 12
2.3 Fourier-Motzkin Variable Elimination 13
2.4 Simplex 17

3 FMPlex **23**
3.1 Notation 23
3.2 Combining Constraints 24
3.3 Conflict Handling 25
3.4 Obtaining a Model 29
3.5 Extension to Strict Operators and Equalities 29
3.6 Properties 30

4 Incremental FMPlex **33**
4.1 Data Structures and Auxiliary Functions 33
4.2 Incrementality 35
4.3 Heuristics 38

5 Implementation and Testing **43**
5.1 Implementation 43
5.2 Testing Results and Discussion 44

6 Conclusion **49**
6.1 Summary 49
6.2 Future Work 49

Bibliography **51**

Chapter 1

Introduction

One of the most fundamental satisfaction problems is that of Boolean satisfiability, often referred to as SAT. SAT solving is one of the state-of-the-art technologies to check the satisfiability of propositional logic formulas. A SAT solver receives a formula over Boolean variables connected via operators such as conjunction, disjunction or negation and checks if it is *satisfiable*, i.e. if there is an assignment of the formula's variables to either *true* or *false* so that the entirety of the formula evaluates to true [MSM18]. This problem was also the first to be proven as NP-complete [Coo71].
While SAT solving does have its areas of applications, certain problems require a more expressive modeling language. *Satisfiability modulo theories*, commonly abbreviated as *SMT*, arose from combining Boolean formulas with non-Boolean variables, function and predicate symbols from certain logical background theories such as the theory of integers, the real numbers, or different data structures. SMT solvers and their techniques, with the extended expressiveness of their corresponding formulas, have proven useful in a variety of applications, including scheduling [DMB11], program verification, test-case generation, model-based development [BdM09] as well as theory-proving systems such as PVS [ORS92].

The focus of this thesis resides in the theory of quantifier-free linear real algebra, more precisely in the satisfiability of conjunctions of linear real constraints on a set of variables. The possibly oldest approach to decide such a problem is the Fourier-Motzkin variable elimination which was first proposed in 1827 by Fourier and more than a hundred years later rediscovered in 1936 by Motzkin. The procedure eliminates each variable x by stating that each lower bound on x is less than or equal to each upper bound on x, and even strictly less, if one of the bounds is strict. However, this method was largely regarded as inefficient, as it requires each upper bound on a variable to be combined with each lower bound. This resulted in a doubly exponential worst-case runtime, becoming infeasible even for low numbers of variables. Another technique is the Simplex algorithm [Dan16]. It was originally developed for linear optimization problems, i.e. optimizing the value of a linear function that is subject to various linear constraints. It works by traversing from vertex to vertex along the edges of the polyhedron induced by the linear constraints [Dan90]. Given that this also includes determining if a feasible solution exists before optimizing it, this method can easily

be adapted to be used in SMT solving.
These two methods have recently been combined into the *FMPlex method* in [Kob21]. Instead of combining every upper with every lower bound upon eliminating a variable, one bound is assumed as greatest lower or, analogously, smallest upper bound for the variable. It is then combined with every opposite bound and every bound in the same direction. As this assumption may be incorrect, the algorithm requires backtracking when an error in the choice of greatest upper or smallest lower bound is discovered. Nevertheless, this reduces the Fourier-Motzkin elimination's worst-case runtime from doubly exponential to singly exponential.
This thesis intends to present an even further optimized version of the FMPlex algorithm. One key to that is introducing incrementality. The dominant approach in the field is *lazy SMT solving*. It is a collaboration between a SAT solver and a theory solver, with the SAT solver checking the Boolean structure of the input formula and consulting the theory solver when necessary to check feasibility in the theory. After a successful intermediate theory check, the SAT solver might add a few constraints to the ones already checked and then require a theory check for the consistency of the conjunction of old and new constraints together. In these cases, it is beneficial to still have the previous results stored instead of recomputing everything. Furthermore, we deploy effective data structures and heuristics for choosing which variables to eliminate via which and what kind of bound.

Chapter 2 introduces the basics of SMT checking, linear real arithmetic as the relevant theory for this thesis, as well as the previously mentioned two approaches for solving sets of linear constraints: the Fourier-Motzkin variable elimination and the Simplex algorithm. Chapter 3 then presents the recently developed combination of the former two, FMPlex, along with its characteristics and properties. We continue with Chapter 4 where we introduce the novel incremental version of this algorithm, including additional heuristics for further optimization. Chapter 5 covers the specifics of an experimental implementation on top of the SMT solver SMT-RAT [CKJ+15] and the evaluation of its results. We end the thesis with Chapter 6 with a brief summary, discussion and an outlook on possible future work.

Chapter 2

Preliminaries

2.1 Satisfiability Modulo Theories

The description of an SMT problem consists of a Boolean combination of atoms that originate from a certain background theory. The formula is usually given in *conjunctive normal form* (CNF) which is a conjunction of clauses. Clauses, in turn, are disjunctions of literals, i.e. atoms or negation of atoms. The SMT solver itself consists of a SAT solver interacting with a theory solver for the relevant background theory.

Most modern SAT solvers check the satisfiability of propositional logic formulas in CNF based on the *Davis–Putnam–Logemann–Loveland* (DPLL) algorithm [DP60, DLL62] which includes three key steps: `decide`, `propagate` and `backtrack`. These steps are repeated until a satisfying valuation of the propositional variables is found. In the `decide` step, the solver chooses a yet unassigned variable and assigns a truth value to it. The subsequent `propagate` step determines the consequences of that assignment decision in the context of the formula. A prominent deduction rule for that is the unit clause rule: If all but one literal in a clause have been assigned to *false*, then the last remaining literal must be assigned in such a way that it evaluates to *true* in order to not falsify the clause and, in turn, the entire formula to be satisfiable [DMB11]. If we reach a state where no extension of our current partial truth assignment can satisfy the formula, we have found a *conflict*. A conflict indicates that one of the previous decisions was not correct. We need to `backtrack` and revert decisions.

In SMT solving, the SAT solver works with a Boolean abstraction of the original formula in which every atom from the background theory is replaced by a propositional variable. The theory solver then receives a set of literals induced by the assigned truth values of the propositional variables. Its task is then to check the conjunction of these literals for satisfiability within the theory and report back to the SAT solver. Should the theory solver determine that this conjunction is not satisfiable within the theory, it will return a conflicting subset of these literals whose negation is abstracted again and included as an additional clause to the SAT solver's CNF formula. This

negation is called a *theory lemma* [DMB11] and causes the SAT solver to detect a conflict and backtrack.

The crucial factor to differentiate eager from lazy SMT solving is when the SAT solver calls on the theory solver. In eager SMT solving, the SAT solver first finds a complete truth assignment for all Boolean variables so that the abstraction evaluates to *true*, and then hands all induced literals to the theory solver. Lazy SMT solving on the other hand means that after deciding on a variable assignment, propagating and not finding a conflict in the Boolean abstraction, the SAT solver consults the theory solver to determine if the already assigned (decided or propagated) literals obtained so far are satisfiable. This allows for earlier backtracking of the SAT solver and proves especially beneficial if the theory solver can work incrementally and process additional constraints without recomputing the results of previous theory checking calls. It is this characteristic that motivates the development and implementation of an incremental version of the FMPlex method presented in this paper.

2.2 Background Theory: Linear Real Arithmetic

The background theory of an SMT solver determines what kinds of atoms appear in its formulas. Here, we consider *linear real arithmetic.*

Definition 2.2.1 (Linear terms and constraints)**.** *A linear term over variables* $x_1, \ldots, x_n$ *takes the form*

$$a_1x_1 + \ldots + a_nx_n + b$$

with coefficients $a_1, \ldots, a_n \in \mathbb{Q}$*, and a constant* $b \in \mathbb{Q}$*. A* linear constraint *is a comparison of one term against another by some relation and generally takes or can be rearranged into the form*

$$a_1x_1 + \ldots + a_nx_n \bowtie b$$

with the coefficients and the constants as in the definition above and a relation $\bowtie \in \{<, \leq, =, \geq, >, \neq\}$.

For linear terms t_1 and t_2 it holds that

$$t_1 = t_2 \quad \Leftrightarrow \quad t_1 \leq t_2 \wedge t_2 \leq t_1$$

and, for the negation of the equality, it holds that

$$t_1 \neq t_2 \quad \Leftrightarrow \quad t_1 < t_2 \vee t_2 < t_1.$$

Given this equivalence, we will limit ourselves to $\bowtie \in \{<, \leq, \geq, >\}$, although equalities and their negations may also be handled more directly.
In the following, for simplicity, we restrict ourselves to weak inequalities first and discuss strict inequalities later in Section 2.3.1.

We say that a variable x_j *occurs* in a constraint c if $a_j \neq 0$ and write $x_j \in c$ to denote this fact. We call c a *lower bound* on variable x_j if $a_j < 0$ and denote it with $lb(c, x_j)$. Analogously, if c is an upper bond on x_j as $a_j > 0$, we write $ub(c, x_j)$.
Variable-free constraints can be brought into the variable-free form of $a \leq 0$ with

$a \in \mathbb{Q}$. We then write *true* instead of the actual constraint if a is indeed less or equal to 0, making it trivially true, and write *false* otherwise.

Definition 2.2.2 (System of Linear Inequalities)**.** *A system of linear inequalities over a set of real variables $X = \{x_1, \ldots, x_n\}$ encapsulates m constraints over the variables in a matrix $A \in \mathbb{Q}^{m \times n}$ and a vector $b \in \mathbb{Q}^m$. Given these and a vector x of these variables, the system can be read as $Ax \leq b$, i. e.*

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

Thus, we simply describe the system with (A,b).

As such, each row of this system corresponds to a single constraint and each column of A correlates to a variable. A theory solver for linear real arithmetic receives just such a system with the goal to determine if there exists a valuation of the variables such that all constraints are fulfilled. A valuation is a function $\alpha : X \to \mathbb{R}$, assigning each variable a real value. If a valuation fulfills all constraints in a system, we consider it a *model* of said system.

Example 2.2.1 (System of Linear Inequalities)**.** *Assume variables x, y, and the following constraints:*

$$\begin{aligned} -x - 2y &\leq -2 \\ x - 2y &\leq 3 \\ x + 2y &\leq 5 \\ -3x + 7y &\leq 7 \\ 6x + y &\leq 18. \end{aligned}$$

We can translate these into a matrix of coefficients and a vector of constant bounds representing this system of inequalities:

$$\begin{pmatrix} -1 & -2 \\ 1 & -2 \\ 1 & 2 \\ -3 & 7 \\ 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} -2 \\ 3 \\ 5 \\ 7 \\ 18 \end{pmatrix}$$

To avoid confusion in the further course of this thesis, i will only be used to denote row / constraint indices, i.e. $1 \leq i \leq m$. Similarly, j will only denote column / variable indices, $1 \leq j \leq n$.

2.3 Fourier-Motzkin Variable Elimination

The Fourier-Motzkin variable elimination is named after Joseph Fourier, who first proposed the method in [Fou27] in 1827 and Theodore Motzkin after he rediscovered it in [Mot36] in 1936. It aims to reduce the number of variables in a given system of

linear inequalities until we obtain a variable-free constraint system whose correctness then decides the overall satisfiability of our original system. We do this by explicitly eliminating one variable per step with the possible side result of eliminating further variables implicitly. The variable order decides the order in which the variables are to be eliminated. E.g. we can assume an order simply based on the variables' indices, i.e. $x_1 < \ldots < x_n$.

To eliminate a variable x_k, we start by dividing the current constraints into three sets: U, L, and N. The set U comprises all upper bounds on x_k and L all lower bounds on x_k, while N contains exactly those constraints in which the variable does not occur. To distinguish between upper and lower bounds, we denote a lower bound $l \in L$ as $l_1x_1+\ldots+l_nx_n \leq b_l$, and analogously, an upper bound $u \in U$ as $u_1x_1+\ldots+u_nx_n \leq b_u$. To be able to eliminate the variable, we rewrite our lower bounds as

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

denoting the left-hand side as $lb(l)$ Similarly, we also rearrange upper bounds into the form

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

and call the right-hand side $ub(u)$. These forms allow us to combine the two constraints into one while simultaneously eliminating x_k:

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

This new constraint can of course be rearranged into the standard format for constraints again for the process to be repeated on other variables. The factors $\frac{1}{|l_k|}$ and $\frac{1}{|u_k|}$ may be chosen differently as long as both are positive and that choice creates the same absolute value of the coefficients of x_k in both constraints [Kob21].
Viewed as rows of a constraint system, this combination corresponds to scaling (with positive factors) and then adding the two rows representing the two constraints so that the coefficient of the resulting row for x_k is 0. This means that each constraint newly created by this operation is a positive linear combination of the upper and the lower bound that created it.
Henceforth, we will call this operation COMBINEUPPERLOWER(u, l, x_k). To eliminate a variable from the set of constraints, we extend the operation to our sets U and L. The Fourier-Motzkin algorithm combines each upper with each lower bound. We define the operation COMBINEUPPERLOWERSETS(U, L, x_k) which takes the set of upper bounds U, the set of lower bounds L and the variable to be eliminated and returns the set of constraints we obtain by executing COMBINEUPPERLOWER on every upper-lower bound combination from $U \times L$. The constraints resulting from that, united with those from N, form a *reduced* system of inequalities that is the basis for the elimination of the next variable.

Algorithm 1 Fourier-Motzkin Variable Elimination

Input: A set of constraints C, over variables with ordering $x_1 < \ldots < x_n$)

Output: SAT or UNSAT

```
1 function FOURIER-MOTZKIN(C, x_1, ..., x_n,)
2     for i = 1 to n do
3         U ← {c ∈ C | x_k ∈ c ∧ ub(c, x_k)}
4         L ← {c ∈ C | x_k ∈ c ∧ lb(c, x_k)}
5         N ← {c ∈ C | x_k ∉ c}
6         C ← N ∪ COMBINEUPPERLOWERSETS(U, L, x_k)
7         if false ∈ C then return UNSAT
8     return SAT
```

It is worth noting that a variable may disappear before being explicitly eliminated. In such cases, we speak of *implicit* elimination. Should the algorithm encounter a trivially false constraint, we can conclude UNSAT. If we at any point or at the latest, after n eliminations, arrive at a system consisting of only trivially true constraints, we can conclude SAT.

Example 2.3.1 (Fourier-Motzkin Variable Elimination)**.** *We apply the Fourier-Motzkin variable elimination to our previous Example 2.2.1. We use the variable ordering $x < y$ to determine the order of our eliminations and choose the least common multiple of the absolute values of the relevant coefficients as scaling factors.*
We begin with variable x and divide the constraints in upper bounds, lower bounds and non-bounds for it.

$$U_x = \begin{Bmatrix} u_1: & x-2y & \leq & 3 \\ u_2: & x+2y & \leq & 5 \\ u_3: & 6x+y & \leq & 18 \end{Bmatrix}, \quad L_x = \begin{Bmatrix} l_1: & -x-2y & \leq & -2 \\ l_2: & -3x+7y & \leq & 7 \end{Bmatrix}$$

In this case, the non-bound set N_x is empty. The constraint set we obtain by eliminating x then is $N_x \cup \text{COMBINEUPPERLOWERSETS}(U_x, L_x, x)$:

$$\begin{Bmatrix} \text{COMBINEUPPERLOWER}(u_1, l_1) \\ \text{COMBINEUPPERLOWER}(u_1, l_2) \\ \text{COMBINEUPPERLOWER}(u_2, l_1) \\ \text{COMBINEUPPERLOWER}(u_2, l_2) \\ \text{COMBINEUPPERLOWER}(u_3, l_1) \\ \text{COMBINEUPPERLOWER}(u_3, l_2) \end{Bmatrix} = \begin{Bmatrix} -4y & \leq & 1 \\ y & \leq & 16 \\ 0 & \leq & 3 \\ 13y & \leq & 22 \\ -11y & \leq & 6 \\ 15y & \leq & 32 \end{Bmatrix}$$

Again, we divide these constraints into upper, lower and non-bounds, this time for y:

$$U_y = \begin{Bmatrix} u_4: & y & \leq & 16 \\ u_5: & 13y & \leq & 22 \\ u_6: & 15y & \leq & 32 \end{Bmatrix}, \quad L_y = \begin{Bmatrix} l_3: & -4y & \leq & 1 \\ l_4: & -11y & \leq & 6 \end{Bmatrix}, \quad N_y = \{n_1: \quad 0 \quad \leq \quad 3\}$$

Combining the upper and lower bounds now leaves us with a set of variable-free con-

straints:

$$\begin{Bmatrix} \text{combineUpperLower}(u_4, l_3) \\ \text{combineUpperLower}(u_4, l_4) \\ \text{combineUpperLower}(u_5, l_3) \\ \text{combineUpperLower}(u_5, l_4) \\ \text{combineUpperLower}(u_6, l_3) \\ \text{combineUpperLower}(u_6, l_4) \\ n_1 \end{Bmatrix} = \begin{Bmatrix} 0 \leq 65 \\ 0 \leq 182 \\ 0 \leq 101 \\ 0 \leq 320 \\ 0 \leq 143 \\ 0 \leq 442 \\ 0 \leq 3 \end{Bmatrix}$$

All of these are trivially true, thus we can conclude that the original set of constraints is satisfiable.

2.3.1 Expansion to Strict Relations

This algorithm can easily be adjusted to handling both strict and non-strict constraints. Whenever we combine an upper and a lower bound, we can derive the relation type of the newly created constraint by looking at those of its parents. Here, the strict operators are dominant. That means as soon as one of the original constraints has a strict operator, the new constraint will have the strict operator as well. Otherwise, if both parent constraints have non-strict operators, the new constraints receives a non-strict operator too.

2.3.2 Correctness

The reason as to why we can correctly conclude SAT and UNSAT in the previously described manner is the following:

Theorem 2.3.1 (Solvability Equivalence of Reduced Systems [Dan72])**.** *Given a system of linear inequalities (A, b) over variables $x_1, \ldots, x_n$ and its reduced system of linear inequalities (A', b') over $x_2, \ldots, x_n$ derived as described by eliminating x_1, it holds that*

$$(A, b) \text{ is solvable iff } (A', b') \text{ is solvable.}$$

Proof. Given a valuation for $x_1, \ldots, x_n$ that satisfies (A, b), it is easy to see it also satisfies (A', b').
If we have a valuation for $x_2, \ldots, x_n$ satisfying (A', b'), it also satisfies that all upper bounds are greater than or equal to all lower bounds of x_1. This includes that the smallest upper bound is greater than or equal to the greatest lower bound, i.e. there exists at least one value that is on or between these bounds. Then, we can extend the valuation by assigning that value to x_1. This leads us to a valuation for $x_1, \ldots, x_n$ which now also satisfies (A,b). □

For later explanations on the FMPlex method and its different kinds of conflict, this can also be restated as the *feasibility theorem*:

Theorem 2.3.2 (Feasibility Theorem [Dan72])**.** (A,b) *is solvable iff there are no non-negative weights $\gamma_1, \ldots, \gamma_m \in \mathbb{Q}_{\geq 0}$ such that*

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

Proof. Assuming we have a solution $x = (x_1, \ldots, x_n)$ for (A, b), which implies its solvability, and there does exist a set of non-negative weights $\gamma_1, \ldots, \gamma_m \in \mathbb{Q}_{\geq 0}$ satisfying the conditions above, this would imply

$$\sum_{i=1}^{m} \sum_{j=1}^{n} (\gamma_i a_{ij}) x_j \ \leq \ \sum_{i=1}^{m} \gamma_i \cdot b_i > 0$$

which evaluates to

$$0x = 0 \ \leq \ \sum_{i=1}^{m} \gamma_i \cdot b_i > 0,$$

a trivially false constraint which is a contradiction to x being a solution to (A, b).
The reverse direction of the proof is done via contraposition: if we assume that no solution to (A, b) exists, then we can use the variable elimination to derive trivially false constraints. As the procedure only uses non-negative linear combinations of the constraints, we can thus obtain a non-negative set of weights that satisfies the aforementioned conditions that confirms that such a set does exist. □

2.3.3 Efficiency

As this method always compares and combines all upper bounds on a variable with all lower bounds on the variable, it is often considered inadequate for practical application.

Theorem 2.3.3 (Fourier-Motzkin Complexity [Kob21])**.** *The Fourier-Motzkin variable elimination method has a doubly exponential worst-case complexity.*

Proof. Let (A, b) be a system of linear inequalities with m constraints and n variables. The worst case occurs if all constraints contain the variable to be eliminated and the sets U and L have the same size, i.e. $\frac{m}{2}$. Each elimination step then creates $(\frac{m}{2})^2$ constraints. If this situation occurs for every variable, we reach $O((\frac{m}{2})^{2^n})$ many constraints. □

2.4 Simplex

In 1947, George Dantzig started developing and, later on, published what would become known as the Simplex method [Dan90]. Today it is a widely known and widely applied efficient tool in linear programming. This means that the general problem to be solved with it usually consists of a linear objective function we want to maximize or minimize whose variables are subject to a set of linear constraints. The algorithm is split into two phases. The first one determines if there exists any feasible solution while the second subsequently maximizes or minimizes the solution in regards to the objective function. Thus, the procedure of the first phase lends itself to SMT solving, where the mere existence of a solution suffices. The second phase will not be further elaborated on as it is not relevant to our work.
Before we can begin applying the algorithm, we need to convert our system of inequalities (A, b) into its *slack form* by converting the inequalities to equalities, introducing additional variables which then are subject to corresponding inequalities.

Definition 2.4.1 (Slack form [CLRS09])**.** *Each inequality in the system* (A, b) *of the form*

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

can be rewritten as the two constraints

$$\sum_{j=1}^{n} a_{ij} x_j = s_i \quad \textit{and} \quad s_i \leq b_i.$$

The resulting system of constraints is called the slack form *of* (A, b)*.*

The newly introduced variable s_i is then called a *slack variable* as it measures the difference between the two sides of the original inequality. The new constraints are satisfiable if and only if the original inequality is satisfiable. Thus, the slack form of a system of inequalities is satisfiability-equivalent to the original system. However, it proves more suitable for the algorithm.

2.4.1 Tableau Form

The simplex algorithm operates on a tableau T that represents the equalities of the slack form. We adopt the notation from [Kob21] with only slight changes.

We denote the set of variables with $\mathcal{X} = \{y_1, \ldots, y_l\}$. In our case, we have $l = n + m$ and $\mathcal{X} = \{x_1, \ldots, x_n, s_1, \ldots, s_m\}$. Furthermore, the algorithm distinguishes between *basic* and *nonbasic* variables. We denote the set of non-basic variables with $\mathcal{N} \subseteq \mathcal{X}$, and that of the basic variables with $\mathcal{B} = \mathcal{X} \setminus \mathcal{N}$. Initially, these sets contain the original variables and the slack variables respectively, i.e. $\mathcal{N} = \{x_1, \ldots, x_n\}$ and $\mathcal{B} = \{s_1, \ldots, s_m\}$.

For all variables in $y_i \in \mathcal{B}$, the tableau has a row encoding an equation expressing the basic variable as a linear combination of the non-basic variables: $y_i = \sum_{y_j \in \mathcal{N}} a_{ij} y_j$. Thus, the basic variables depend on the non-basic ones. They are recognizable by their corresponding column only containing zeros except for a single -1 entry.

Additionally, we save upper and lower bounds for variables. If a variable $y \in \mathcal{X}$ has a lower (an upper) bound, we write it as $l(y)$ ($u(y)$). The original variables have no bounds (or, equivalently, $-\infty$ and $+\infty$), as their bounds have been transferred to the slack variables, while the slack variables receive the very bounds we obtained when converting our original system into slack form.
We use an *assignment* $\alpha : \mathcal{X} \to \mathbb{R}$ indicating the current valuation of the variables. In the beginning, it is set to $\alpha(y) = 0$ for all $y \in \mathcal{X}$. This leads to all equations holding, but some of the basic variables might violate their bounds. Conversely, non-basic variables always satisfy their bounds, not just in our initial tableau, but at all points of the procedure, making this an invariant of the algorithm.

With that, we can construct our initial tableau:

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

The tableau has the dimension of $m \times (n + m)$. In it, we can read a row i as $\sum_{j=1}^{n} a_{ij}x_j - s_i = 0$. For $y_i \in \mathcal{B}$ and $y_j \in \mathcal{N}$, we denote the jth entry in the tableau row encoding y_i with T_{ij}.

2.4.2 The Algorithm

Starting with the initial tableau, we begin to apply the algorithm. It changes the tableau by *pivoting* and our current assignment by subsequent *updating*. Our explanations follow the rules given in [KBD+17] which are summarized in Figure 2.1.

For the `pivot` step, we require one basic variable $y_i \in \mathcal{B}$, the *leaving* variable, and one non-basic variable $y_j \in N$, the *entering* variable. The leaving variable y_i has to violate its bound with the current assignment, i.e. $\alpha(y_i) < l(y_i)$ or $\alpha(y_i) > u(y_i)$. The operation switches these two variables , making y_i non-basic and vice versa, by changing the tableau accordingly. However, y_j has to be a suitable candidate to be pivoted with y_i. It must not be guaranteed to violate its bound after the update operation that will follow the pivoting. Thus, we consider y_j suitable if it is in the *slack* of y_i:

$$\begin{aligned} \text{slack}^+(y_i) &= \{y_j \in \mathcal{N} \mid (T_{ij} > 0 \wedge \alpha(y_j) < u(y_j)) \vee (T_{ij} < 0 \wedge \alpha(y_j) > l(y_j))\} \\ \text{slack}^-(y_i) &= \{y_j \in \mathcal{N} \mid (T_{ij} < 0 \wedge \alpha(y_j) < u(y_j)) \vee (T_{ij} > 0 \wedge \alpha(y_j) > l(y_j))\} \end{aligned}$$

We use slack^+ if y_i violates its lower bound and slack^- if y_i violates its upper bound. If we have found a suitable pair y_i and y_j, we remove y_i from the basic variables and add y_j to them and set $T = \mathit{pivot}(T, i, j)$. Furthermore, we move y_i to the non-basic variable set. As y_j is now a basic variable, the row that until now was encoding y_i as a linear combination of non-basic variables has to be scaled with factor $-\frac{1}{T_{ij}}$ so that it now encodes y_j in the aforementioned manner. Following that, we replace all occurrences of y_j with its new linear combination in all other rows. This means we add the newly scaled row to them in such a manner that all other entries in the column of y_j become zero. Note that this makes the column recognizable as that of a basic variable since now, the only remaining non-zero entry is -1.

After the pivot step, the `update` step adjusts our current assignment to make y_i, which is now a non-basic variable, satisfy its bound and, in doing that, uphold the invariant of non-basic variables always satisfying their bounds. Let δ be a constant that, added to $\alpha(y_i)$, would put the variable between its upper and lower bound, e.g. the difference between the current valuation and the violated bound. Then, we update our assignment α to $\alpha' = \mathit{update}(\alpha, y_i, \delta)$. We set $\alpha'(y_i) = \alpha(y_i) + \delta$. As

basic variables are expressed as linear combinations of non-basic variables which now include y_i, their assignments have to be adjusted to this change as well, scaled by the coefficient of y_i in their row. So for all $y_k \in \mathcal{B}$, we set $\alpha'(y_k) = \alpha(y_k) + T_{ki} \cdot \delta$. For all remaining variables, i.e. all $y \in \mathcal{N} \setminus \{y_i\}$, the assignment remains the same: $\alpha'(y) = \alpha(y)$.
It is, of course, possible for basic variables that were previously well within their bounds, to now violate them. We can then repeat the `pivot` and `update` steps to fix their bounds as well. If, at any point, we reach a state where a basic variable violates its bound but its slack set is empty, the `Failure` rule applies and we can conclude UNSAT. If we however find all variables within their bounds after an `update` step, the result is SAT as per the `Success` rule.

$$\texttt{Pivot}_1 \quad \frac{y_i \in \mathcal{B}, \quad \alpha(y_i) < l(y_j), \quad y_j \in \text{slack}^+(y_i)}{T \leftarrow \mathit{pivot}(T, i, j), \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{y_j\} \setminus \{y_i\}}$$

$$\texttt{Pivot}_2 \quad \frac{y_i \in \mathcal{B}, \quad \alpha(y_i) > u(y_j), \quad y_j \in \text{slack}^-(y_i)}{T \leftarrow \mathit{pivot}(T, i, j), \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{y_j\} \setminus \{y_i\}}$$

$$\texttt{Update} \quad \frac{y_i \notin \mathcal{B}, \quad \big(\alpha(y_i) < l(y_i) \wedge l(y_i) = \alpha(y_i) + \delta\big) \ \vee \ \big(\alpha(y_i) > u(y_i) \wedge u(y_i) = \alpha(y_i) + \delta\big)}{\alpha \leftarrow \mathit{update}(\alpha, y_i, \delta)}$$

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

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

Figure 2.1: Rules for the Simplex algorithm [KBD+17].

Example 2.4.1 (Simplex Algorithm)**.** *Again, we use Example 2.2.1 to demonstrate the algorithm. Conversion into slack form introduces 5 new slack variables and transfers the bounds onto them:*

$$\begin{pmatrix} -1 & -2 \\ 1 & -2 \\ 1 & 2 \\ -3 & 7 \\ 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{pmatrix}, \quad \begin{pmatrix} s_1 & \leq & -2 \\ s_2 & \leq & 3 \\ s_3 & \leq & 5 \\ s_4 & \leq & 7 \\ s_5 & \leq & 18 \end{pmatrix}$$

The starting conditions for the algorithm are then given by the tableau

$$T = \left[\begin{array}{cc|ccccc} x & y & s_1 & s_2 & s_3 & s_4 & s_5 \\ -1 & -2 & -1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 0 & -1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & -1 & 0 & 0 \\ -3 & 7 & 0 & 0 & 0 & -1 & 0 \\ 6 & 1 & 0 & 0 & 0 & 0 & -1 \end{array}\right]$$

the basic variables $\mathcal{B} = \{s_1, s_2, s_3, s_4, s_5\}$ *and the non-basic variables* $\mathcal{N} = \{x, y\}$ *and the initial asssignment* α *which maps all variables to* 0*. Considering the slack variables' bounds, it becomes apparent that only* s_1 *currently violates its bound. As that bound is an upper bound, we determine the corresponding slack set:* $slack^-(s_1) = \{x, y\}$*. We choose* x *to pivot with* s_1*. As its entry in the first row, which has been, until now, encoding* s_1*, is already* -1*, no scaling is necessary. We then replace all occurrences of* x *in the other rows with its new encoding in the first row and change* T *accordingly:*

$$T = \begin{array}{c} \begin{array}{ccccccc} x & y & s_1 & s_2 & s_3 & s_4 & s_5 \end{array} \\ \left[\begin{array}{ccccccc} -1 & -2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -4 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 13 & 3 & 0 & 0 & -1 & 0 \\ 0 & -11 & -6 & 0 & 0 & 0 & -1 \end{array}\right] \end{array}$$

Now, we have $\mathcal{B} = \{x, s_2, s_3, s_4, s_5\}$ *and* $\mathcal{N} = \{s_1, y\}$*. To update* α *to* α'*, we choose* $\delta = -2$ *and set*

$$\alpha'(s_1) = \alpha(s_1) + \delta = -2$$

And adjust the basic variables' assignments accordingly:

$$\begin{aligned} \alpha'(x) &= \alpha(x) + (-1) \cdot \delta &&= 2 \\ \alpha'(s_2) &= \alpha(s_2) + (-1) \cdot \delta &&= 2 \\ \alpha'(s_3) &= \alpha(s_3) + (-1) \cdot \delta &&= 2 \\ \alpha'(s_4) &= \alpha(s_4) + 3 \cdot \delta &&= -6 \\ \alpha'(s_5) &= \alpha(s_5) + (-6) \cdot \delta &&= 12 \end{aligned}$$

While for all other variables in $\mathcal{N}$*, in this case just* y*, the assignment stays the same. Comparing this new assignment to the bounds we have, we see that none of the variables violate any bounds anymore and can conclude that the system is satisfiable.*

2.4.3 Completeness and Efficiency

It is possible for the original Simplex algorithm to cycle infinitely on certain examples. But given a variable ordering, we can execute it applying *Bland's rule* [Bla77] to ensure termination. Further pivoting rules have since been developed to increase efficiency such as the sub-exponential *Random Facet* pivoting rule in [Kal92] which has since been further improved [HZ15].
Using Bland's rule, we can make the following statement about its runtime:

Theorem 2.4.1 (Simplex Complexity [KM72])**.** *Dantzig's Simplex algorithm has an exponential worst-case complexity when using Bland's rule.*

Despite its exponential worst-case runtime, simplex has still found widespread applications as such a case rarely occurs in practice and the algorithm exhibits a polynomial average-case runtime [Sch98]. As such, it is preferred over the Fourier-Motzkin variable elimination and even algorithms like the ellipsoid method [Kha79] which have a theoretical polynomial worst-case runtime but perform poorly in practice.

Chapter 3

FMPlex

Again, we first assume only weak constraints and discuss the other constraint types in Section 3.5. The FMPlex method was presented in [Kob21] in 2021 and aims to combine the Fourier-Motzkin variable elimination with the Simplex algorithm to reduce the former's doubly exponential runtime. The application of it resembled the computation of

$$\forall u \in U, l \in L.\ lb(l) \leq x_k \leq ub(u)$$

to eliminate variable x_k on a given level with appropriate L and U. To reduce the number of constraints, instead of combining every upper bound with every lower bound upon eliminating, we choose one of the lower bounds l which we assume to be a *greatest lower bound* (GLB). We then only combine this bound, however with all upper and all other lower bounds, resulting in the computation of

$$\forall l' \in L \setminus \{l\}.\ lb(l') \leq lb(l) \quad \wedge \quad \forall u \in U.\ lb(l) \leq x_k \leq ub(u)$$

which, due to the transitivity of the relation, is equivalent to the former. Analogously, we also may choose an upper bound instead which we assume to be a *smallest upper bound* (SUB) which then is combined with all lower and all other upper bounds. We call the chosen bound the *eliminator* of the level. Such a guess of an eliminator might, however, be incorrect as it might not in fact be a GLB or SUB and thus lead to trivially false constraints. These situations are not sufficient to conclude UNSAT, but merely mean that we need to *backtrack* to the last relevant GLB / SUB choice and try another eliminator.

3.1 Notation

To simplify explanations, we adjust our representation of constraints. We choose a format where all constraints have a 0 on the right-hand side. A constraint c is now represented by a vector

$$c = (a_1, \ldots, a_n, b \mid d_1, \ldots, d_m).$$

The first section of this vector represents the constraint itself:

$$\left(\sum_{j=1}^{n} a_j \cdot x_j\right) + b \leq 0,$$

whereas the second part determines what linear combinations of the original constraints resulted in the constraint. We call these entries *derivation coefficients*. That means for the original constraints $c_1^*, \ldots, c_m^*$ with the form

$$c_i^* = (a_{i1}^*, \ldots, a_{in}^*, b_i^* \mid d_{i1}^*, \ldots, d_{im}^*)$$

it holds that $d_{ii}^* = 1$ and $\forall k \neq i.\ d_{ik}^* = 0$ for all $1 \leq i \leq m$.
Then, for any other constraint c derived via the FMPlex method from these original ones, we can also read c as:

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

The derivation coefficients always stay correct as we consider them part of the representing vector. This means any scaling of a constraint in this form also scales these coefficients and any additions of two constraints includes the addition of their coefficients.

3.2 Combining Constraints

As in the original Fourier-Motzkin variable elimination, we intend to convert one set of constraints into another by eliminating a variable until we have only trivially true constraints or a trivially false constraint. However, in this method we choose an assumed greatest lower bound or smallest upper bound to combine with all opposite bounds and all other bounds of the same direction. For simplicity, we will limit our explanations to GLBs as the handling of SUBs is done analogously.
On a given level, we choose a GLB and remove it from the set L since it now holds a special position. The combination of an upper bound with the GLB is still the standard upper-lower bound combination as in the Fourier Motzkin variable elimination as the GLB is, after all, still a lower bound. Their result is still computed the same way: Given that we want to eliminate variable x_k, we scale the bounds with appropriate positive factors so that the coefficients of the variable match and then combine them. In terms of our adjusted notation, that is equivalent to multiplying the vectors with the factors and simply adding them up.
The combination of the GLB with another lower bound, however, is a little different. Apart from the scaling with the positive factors, it requires us to multiply the eliminator with -1, transforming it into an upper bound to be able to combine the two constraints.
In the case that we have only upper or only lower bounds on a variable, i.e. $U = \emptyset$ or $L = \emptyset$, we may simply ignore all of them in our further computation as we can always choose a value for x_k that is small or large enough to satisfy these constraints. The eliminator is technically a GLB if we only have upper bounds and an SUB if we only have lower bounds, but as there is no actual choice to make, we set the eliminator to $\perp$ in either case.

3.3 Conflict Handling

As before, we might encounter trivially false constraints. But not all of these indicate that the original constraints are unsatisfiable. As we recall from the Feasibility Theorem 2.3.2, to prove the unsatisfiability of the original system, it is a necessary and sufficient condition that there exists a non-negative linear combination of the original constraints evaluating to a trivially false constraint. In the Fourier-Motzkin algorithm, this was a given for all trivially false constraints as all of our operations only scaled with positive factors and added constraints. Thus, all resulting constraints were such a non-negative linear combination.
This algorithm however includes multiplications with -1, namely in same-bound combinations. Therefore, it is possible for the trivially false constraint to include negative coefficients for the original constraints that it resulted from. In this case we do not have a non-negative linear combination of original constraints that would allow us to conclude UNSAT. Instead, this merely indicates that we have, at some point, chosen an unsuitable GLB.

This leads us to distinguish between two types of conflicts: *global* conflicts and *local* conflicts. To differentiate between them, we need to know the combination of original constraints that resulted in a given constraint. To keep track of this, we have introduced the derivation coefficients in our representation of a constraint earlier.
A global conflict occurs when we encounter a trivially false constraint that would have also been created within the original Fourier-Motzkin algorithm. It is recognizable by the derivation coefficients of the constraint being either entirely non-negative or entirely non-positive. In this case we can deduce UNSAT. Note that the derivation coefficients might not have always been non-negative or non-positive during the application of the algorithm, but that does not invalidate their non-negativity or non-positivity for this constraint.
In case of local conflict, on the other hand, the opposite is true: The constraint has both negative *and* positive derivation coefficients. It is not sufficient for an UNSAT answer. Rather, we need to re-evaluate our choices of assumed GLBs along the way that lead to the constraint, i.e. we need to *backtrack*.

The possible choices of GLBs for each variable mean that the FMPlex algorithm operates within a decision tree, starting in the root node at level 1, and descending to the next level with each decision, each option for the GLB representing a child of the current node. When we need to backtrack, we want to know the last relevant node for the involved constraint. To keep track of the last level of the decision tree where it was part of a same-bound combination, we introduce the *conflict level* of a constraint c. It is denoted $cl(c)$. Since same-bound combinations are the only kind that introduces negative factors to the coefficients, it is these we want to backtrack to if the constraint is part of a local conflict. Its initial value is 0 to indicate it has not been in any same-bound combination at all yet.
Upon combining two constraints, the conflict level of the resulting constraint is determined by the type of combination. A same-bound combination of two constraints on a given level sets the conflict level to that level. Upon a combination of an upper with a lower bound, the resulting constraint inherits the greater of its parents' conflict levels, i.e. the one denoting the more recent same-bound combination.

The different types of combinations and their effect on the resulting constraint now yield the extended combination operator FMPLEXCOMBINE.

Algorithm 2 fmPlexCombine

Input: The set of lower bounds L, the assumed GLB $c \notin L$, the set of upper bounds U, the variable to be eliminated x_k, the current level lvl

Output: The set of constraints resulting from the elimination of x_k with c as GLB

```
1  function fmPlexCombine(L, c, U, x_k, lvl)
2      if c = ⊥ then
3          return ∅
4      else
5          Constraint set R = ∅
6          for u ∈ U do                                   ▷ Upper-lower
7              Constraint c_new = 1/|u_k| · u + 1/|c_k| · c
8              cl(c_new) ← max{cl(c), cl(u)}
9              R ← R ∪ {c_new}
10         for l ∈ L do                                   ▷ Same-bound
11             Constraint c_new = 1/|l_k| · l − 1/|c_k| · c
12             cl(c_new) ← lvl
13             R ← R ∪ {c_new}
14         return R
```

Now, if we encounter a local conflict, we can, at the furthest, backtrack to the last same-bound combination that occurred in the derivation of the trivially false constraint and choose a different assumed GLB, following another branch from that node in the decision tree. In case we have no more choices for the GLB left, we backtrack one additional level at a time until we reach a level where there are other choices still available. We repeat this process whenever we come upon a local conflict. Should we have to backtrack to the first level and it has no more GLB choices, it means that we were unable to choose a GLB for each variable in a consistent way as all possibilities have been exhausted. We consider this a global conflict as well. A global conflict results in an immediate UNSAT answer, while a set of only trivially true constraints leads to a SAT conclusion.
The following example illustrates the necessity of this separate handling of local conflicts.

Example 3.3.1 (Conflict Handling)**.** *We consider two constraint sets with the same two lower bounds l_1 and l_2 on the only variable x but with a different upper bound in*

each, making the first system unsatisfiable and the second satisfiable:

$$C_{UNSAT} = \begin{Bmatrix} l_1: & -x+4 & \leq & 0 \\ l_2: & -x+8 & \leq & 0 \\ u_1: & x-4 & \leq & 0 \end{Bmatrix}, \quad C_{SAT} = \begin{Bmatrix} l_1: & -x+4 & \leq & 0 \\ l_2: & -x+8 & \leq & 0 \\ u_2: & 2x-20 & \leq & 0 \end{Bmatrix}$$

Transformed into the new constraint format, these yield

$$\begin{array}{cc|ccc} x & b & d_1 & d_2 & d_3 & & \\ \left(\begin{array}{cc|ccc} -1 & 4 & 1 & 0 & 0 \\ -1 & 8 & 0 & 1 & 0 \\ 1 & -4 & 0 & 0 & 1 \end{array}\right) & & cl=0 & (l_1) \\ & & & & & & cl=0 & (l_2) \\ & & & & & & cl=0 & (u_1) \end{array}$$

$$\begin{array}{cc|ccc} x & b & d_1 & d_2 & d_3 & & \\ \left(\begin{array}{cc|ccc} -1 & 4 & 1 & 0 & 0 \\ -1 & 8 & 0 & 1 & 0 \\ 2 & -20 & 0 & 0 & 1 \end{array}\right) & & cl=0 & (l_1) \\ & & & & & & cl=0 & (l_2) \\ & & & & & & cl=0 & (u_2) \end{array}$$

Assume that we first choose l_1 als GLB for x in both systems. The upper-lower combinations $l_1 + u_1$ and $2 \cdot l_1 + u_2$ both result in trivially true constraints. However, the same-bound combination $-l_1 + l_2$ (which is the same in both systems) results in

$$\begin{array}{cc|ccc} b & & d_1 & d_2 & d_3 & \\ \left(\begin{array}{cc|ccc} 4 & & -1 & 1 & 0 \end{array}\right) & & cl=1 \end{array}$$

This is a trivially false constraint. However, it is only a local conflict, recognizable by its derivation coefficients of mixed signedness. We observe that this occurs in both the unsatisfiable and the satisfiable constraint system. Were it to be treated the same way as a global conflict, we would now falsely conclude UNSAT for the satisfiable system. Thus, we need to backtrack in both systems. As there is only one GLB choice we made before, we move back to it and now choose l_2 as GLB instead. The result of the same-bound combination $l_1 - l_2$ is now trivially true. For the satisfiable system, the upper-lower combination yields another trivially true constraint and we can conclude SAT. For the unsatisfiable one, the upper-lower combination results in

$$\begin{array}{cc|ccc} b & & d_1 & d_2 & d_3 & \\ \left(\begin{array}{cc|ccc} 4 & & 0 & 1 & 1 \end{array}\right) & & cl=0 \end{array}$$

which is trivially false and, unlike the false constraint before, has only positive derivation coefficients. It is thus a global conflict that allows us to deduce UNSAT.

Thus we observe that local conflicts do not possess any kind of significance in regards to the satisfiability of a given constraint system. They can occur in both satisfiable and unsatisfiable systems whenever a wrong choice for an eliminator has been made. We can now apply the algorithm to larger examples as well.

Example 3.3.2 (FMPlex Algorithm)**.** *We convert our previous Example 2.2.1 into*

the new constraint format and initialize the conflict levels:

$$\begin{array}{cccc|ccccc} x & y & b & d_1 & d_2 & d_3 & d_4 & d_5 & \\ \left(\begin{array}{cccc|ccccc} -1 & -2 & 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & -3 & 0 & 1 & 0 & 0 & 0 \\ 1 & 2 & -5 & 0 & 0 & 1 & 0 & 0 \\ -3 & 7 & -7 & 0 & 0 & 0 & 1 & 0 \\ 6 & 1 & -18 & 0 & 0 & 0 & 0 & 1 \end{array}\right) & \begin{array}{l} cl = 0 \\ cl = 0 \\ cl = 0 \\ cl = 0 \\ cl = 0 \end{array} \end{array}$$

We start on level 1*. Again, our first variable to eliminate is* x *and we divide the constraints into upper, lower and non-bounds of it:*

$$U_x = \left(\begin{array}{ccc|ccccc} 1 & -2 & -3 & 0 & 1 & 0 & 0 & 0 \\ 1 & 2 & -5 & 0 & 0 & 1 & 0 & 0 \\ 6 & 1 & -18 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \begin{array}{l} (u_1) \\ (u_2) \\ (u_3) \end{array}$$

$$L_x = \left(\begin{array}{ccc|ccccc} -1 & -2 & 2 & 1 & 0 & 0 & 0 & 0 \\ -3 & 7 & -7 & 0 & 0 & 0 & 1 & 0 \end{array}\right) \begin{array}{l} (l_1) \\ (l_2) \end{array}$$

Again, N_x *is empty. We choose* l_1 *as our assumed greatest lower bound. Applying* FMPLEXCOMBINE*, albeit with the scaling method of the least common multiple of coefficients, the result of the elimination is*

$$\begin{array}{ccc|ccccc} y & b & d_1 & d_2 & d_3 & d_4 & d_5 & \\ \left(\begin{array}{ccc|ccccc} -4 & -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -3 & 1 & 0 & 1 & 0 & 0 \\ -11 & -6 & 6 & 0 & 0 & 0 & 1 \\ 13 & -13 & -3 & 0 & 0 & 1 & 0 \end{array}\right) & \begin{array}{l} cl = 0 \\ cl = 0 \\ cl = 0 \\ cl = 1 \end{array} \end{array}$$

The first three rows result from standard upper-lower combinations of the constraints of U_x *with* l_1*. All parent constraints have a conflict level of* 0*, thus theirs is* 0 *as well. The last row however is the result of the same-bound combination of* l_1 *and* l_2 *and shows how such a combination introduces a negative derivation coefficient, in this case* -3*. The conflict level of the constraint is thus set to level* 1*.*
We continue to level 2 *and divide our constraints in preparation to eliminating* y*.*

$$U_y = \left(\begin{array}{cc|ccccc} 13 & -13 & -3 & 0 & 0 & 1 & 0 \end{array}\right) \quad (u_4)$$

$$L_y = \left(\begin{array}{cc|ccccc} -4 & -1 & 1 & 1 & 0 & 0 & 0 \\ -11 & -6 & 6 & 0 & 0 & 0 & 1 \end{array}\right) \begin{array}{l} (l_3) \\ (l_4) \end{array}$$

$$N_y = \left(\begin{array}{cc|ccccc} 0 & -3 & 1 & 0 & 1 & 0 & 0 \end{array}\right) \quad (n_1)$$

We choose l_3 *as the assumed GLB for* y *and obtain*

$$\begin{array}{c|cccc|l} b & d_1 & d_2 & d_3 & d_4 & d_5 & \\ \left(\begin{array}{c|cccc} -65 & 1 & 13 & 0 & 4 & 0 \\ -13 & 13 & -11 & 0 & 0 & 4 \\ -3 & 1 & 0 & 1 & 0 & 0 \end{array}\right) & & & & & & \begin{array}{l} cl = 1 \\ cl = 2 \\ cl = 0 \end{array} \end{array}$$

The first constraint receives its conflict level from the maximum of its parents conflict levels, 1 *and* 0*. The conflict level of the second row is set to 2 as it is a result of a same-bound combination, while the constraint not containing* y *retains its conflict level as it is not part of any combination.*
As all of these constraints are trivially true, we can now conclude SAT.

3.4 Obtaining a Model

Once we have concluded SAT via the algorithm, computing a model is relatively easy. We start on the second-to-last level of the branch in the decision tree that led us to the set of trivially true constraints and iterate over each level back towards the root of the decision tree. On each level, we first substitute all variables occurring in the eliminator (except the one being eliminated on the level) with their assignment. This is always possible as each such variable falls into one of two categories. Either it has been eliminated on a level further down in the tree, which means we already have computed its assigned value, or it is one that has been implicitly eliminated along with the variable being explicitly eliminated on this level and can thus be assigned an arbitrary value. Then, we solve the eliminator for the variable that was eliminated with it and set its assignment on the resulting bound. This is correct as the previous SAT result has confirmed that the eliminator is indeed a greatest lower or a smallest upper bound and it does not conflict with the opposite bounds. For the special case of there only being one type of bound on the level and the eliminator being $\bot$, we substitute all variables except for the one being eliminated in all bounds, solve them all for the variable and assign to it the greatest bound if we had only lower bounds or the smallest bound in case there were only upper bounds.
By repeating this process until we reach the root of the decision tree which contains the original bounds, we obtain an assignment fulfilling the original set of constraints.

Example 3.4.1. *We obtain an assignment* α *for the previous constraint system based on the obtained constraint systems in Example 3.3.2.*
For the last variable which was eliminated, y*, we use the GLB chosen on its level,* l_3*, and solve for the variable:*

$$-4y - 1 \leq 0 \quad \Leftrightarrow \quad -0.25 \leq y$$

Thus we set $\alpha(y) = -0.25$*, putting it right on the bound. Moving up to the next level, we use the GLB* l_1 *and substitute* y *with* $\alpha(y)$ *to solve for* x*:*

$$-x - 2 \cdot (-0.25) + 2 \leq 0 \quad \Leftrightarrow \quad 2.5 \leq x$$

We can set $\alpha(x) = 2.5$ *and have thus created a model of the original constraint system.*

3.5 Extension to Strict Operators and Equalities

To extend the FMPlex algorithm to strict operators, we can adopt the rules given in Section 2.3.1 for the upper-lower combinations and simply treat a strict operator as dominant over a non-strict one in any case. For same-bound combinations, this is different. If we compare two lower bounds, we assume one of them to be the greatest lower bound for the variable to be eliminated, i.e. it is greater than or equal to the

other bound. Then, the only case that requires a strict operator in the resulting constraint is when the eliminator is non-strict and the other bound is strict. As the strict bound excludes equality, it cannot be equal to the eliminator as in that case it would not actually be the GLB. All other cases allow us to simply use a non-strict operator for the combination.
The obtainment of a model has to be slightly modified as well. Should the eliminator end up being a strict bound, we cannot put the assignment's value on that bound but rather have to put it between the bound and all bounds of the opposite kind. For that we not only substitute all other variables in the eliminator but also those in all the opposite bounds and evaluate them. We can then put the value anywhere between the eliminator and the smallest upper bound if the eliminator is a GLB or the greatest lower bound if the eliminator is a SUB. Note that in this case, finding the most restrictive of the opposite bounds is just a comparison of variable-free bounds as we have substituted all variables with their assignment except for the one eliminated on the level.
With both strict and non-strict inequalities at our hands, we can now additionally handle equalities and their negations, as these can be expressed as inequalities.

3.6 Properties

3.6.1 Similarities to Simplex

While the algorithm is an obvious variation of the variable elimination, the similarities with Simplex do not immediately become apparent. But upon closer inspection, there are crucial parallels. The elimination of a variable x with an eliminator c in FMPlex corresponds to pivoting x with the slack variable that was introduced to convert c into slack form. Both operations result in matching constraints which only might differ by a scalar factor. The only further difference is that Simplex retains the tableau row that, prior to the pivot operation, expressed the slack variable while in FMPlex, the eliminator is no longer included in the set of resulting constraints [Kob21].

Example 3.6.1. *We review the result from the first FMPlex elimination applied in Example 3.3.2 and the result of the first pivot operation in Example 2.4.1. The FMPlex algorithm eliminates x by using $-x - 2y \leq -2$ as eliminator. The Simplex algorithm selects s_1, the slack variable belonging to that same constraint, to pivot it with x.*
Ignoring the first row of the Simplex tableau which still holds the constraint used, we can now revert the Simplex constraints' slack forms to their original forms, extract the inequalities from both results and see that they are, in fact, the same:

$$\begin{aligned} -4y &\leq 1 \\ 0 &\leq 3 \\ -11y &\leq 6 \\ 13y &\leq 13 \end{aligned}$$

Thus, the choice of an eliminator in FMPlex to eliminate a variable corresponds to moving the eliminator's slack variable into the non-basis in the Simplex algorithm by pivoting it with the variable. A notable difference between these operations is however the selection of eliminators / slack variables: the Simplex algorithm limits itself to those non-basis variables which violate their bounds, while the FMPlex algorithm

does not. It instead backtracks in order to correct choices that did not work out and result in local conflicts.

3.6.2 Correctness and Termination

It remains to show the correctness of the algorithm as well as its termination.

Theorem 3.6.1 (FMPlex Correctness)**.** *The FMPlex algorithm is correct, i.e. if it returns SAT, the formula on which it is called is satisfiable and if it returns UNSAT then the constraint set is unsatisfiable.*

Sketch of Proof. To prove that a SAT answer of the algorithm implicates the satisfiablity of the checked formula, we assume that FMPlex returns SAT after being called on a given formula. This means that the current branch consists only of correct choices for greatest lower or smallest upper bounds and that those do not conflict with the bounds of the opposite direction. Thus, all upper and lower bounds can be satisfied at the same time and we can construct an assignment for the occurring variables just as described in Section 3.4 that proves the satisfiablitiy of the formula. The other direction of the proof can be given by showing that an UNSAT answer implicates that the given formula is not satisfiable. If we have returned UNSAT, then we have encountered a trivially false constraint that can be created by a positive linear combination of the original constraints. The coefficients of that combination can then be obtained from the derivation coefficients of the false constraint. These are, as per the Feasibility Theorem 2.3.2, sufficient to prove the formula's unsatisfiability. Should we have returned UNSAT due to backtracking to level 1 and finding that we have no untried choice of a GLB left, it means that there is no way to choose GLBs for the variables so that it does not evoke conflicts as the conflict level based backtracking does not skip any sub-trees where we could have found a SAT answer. Thus, if the system were satisfiable, such a combination of GLBs for all variables would have been found. □

Theorem 3.6.2 (FMPlex termination)**.** *The FMPlex Algorithm delivers an answer in finite time.*

Proof. Let again m be the number of constraints and n the number of variables. As mentioned before, the algorithm operates on a decision tree. On each level, at least one variable gets eliminated, limiting the maximal depth of the tree to $n + 1$ as we will have have only variable-free constraints on this level and we will either return SAT or backtrack. The number of branches of a given node is limited by the amount of choices we have for an eliminator. The initial amount of constraints m is finite. As the constraint chosen as eliminator does not appear on the next level or, in case of bounds of only one kind, we discard one or more bounds moving to the next level, the number of constraints — and, by extension, the number of possible eliminators — on a level can only decrease upon going further down along a branch in the tree. This means that the decision tree is finite. Every branch is only visited once, so the algorithm always terminates, at the latest after iterating over the entire tree. □

3.6.3 Efficiency

The FMPlex algorithm succeeds in reducing the doubly exponential complexity of the Fourier-Motzkin variable elimination.

Theorem 3.6.3 (FMPlex Complexity [Kob21])**.** *The FMPlex algorithm exhibits a singly exponential complexity.*

Proof. We again assume an initial formula consisting of m constraints over n variables. The root of the tree, containing the original constraints, can choose at most m eliminators, each leading to a different node, for which at most $m-1$ constraints are generated. Then any of these nodes on level 2 can choose at most $m-1$ eliminators with $m-2$ constraints each, and so forth. Like previously mentioned in the proof of Theorem 3.6.2, the tree's depth is limited to $n+1$. This results in at most

$$(m) \cdot (m-1) \cdot \ldots \cdot (m-n) < m^n$$

total constraints, i.e. only singly exponentially many. □

Chapter 4

Incremental FMPlex

We present the novel contribution of this thesis, which is an incremental FMPlex version for a total of m original constraints $c_1^*, \ldots, c_m^*$ and a total of n occurring variables $x_1, \ldots, x_n$. The original constraints are, due to the nature of lazy SMT solving, not all available immediately but can be added and removed from the set of currently considered constraints.
For understandability and readability, this description will be limited to eliminations via lower bounds. Using only upper bounds for eliminations can be done analogously. The actual implementation combines both, requiring an additional decision for which kind of bound to use for elimination whenever choosing a variable to eliminate. This is not included in the algorithm presented here, but the heuristic for that decision will be touched on in Section 4.3. Consequently, we also assume that if we only have one type of bound for a variable on a level, that type is upper bounds, but the opposite case can be handled analogously as well.

The implemented theory solver module interacts with its surroundings via a number of functions, namely addConstraint and removeConstraint for managing which constraints are to be checked for satisfiability as well as the main algorithm checkSat for executing the satisfiablity check and getModel, which can be called after a successful check to obtain a model that fulfills the constraints.

4.1 Data Structures and Auxiliary Functions

Based on the decision tree that is induced by the choices we are able to make for the greatest lower bound, our overarching data structure is a vector that represents the current branch of the tree our execution is in. We continue to call each entry in it a (tree) *level*. Only storing the current branch suffices as we neither need parts of the tree we already visited that we have already deemed unfit to obtain a solution from, nor do we need to pre-compute the entirety of what is still ahead. As we are eliminating at least one variable per transition from level to level, eventually reaching a variable-free set of constraints which we then evaluate, this requires the vector representing the branch to contain at most $n + 1$ levels.

For each level i with $1 \leq i \leq n+1$, we have various variables:

- C_i, the set of constraints that are not or not yet relevant on the level,
- $x_{k_i} \in \{x_1, \ldots, x_n, \bot\}$, the variable to be eliminated on the level. The special value $\bot$ appears in the case that it is not yet decided which variable this will be, that the level is not currently in use, or we are on a level with only variable-free constraints.
- c_i, the lower bound on x_{k_i} that we currently assume to be the greatest lower bound on that variable. This too can be $\bot$ if no constraint is chosen yet or we only have upper bounds on the level.
- L_i^{todo}, the set of lower bounds on x_{k_i} on the level that have not yet been tried out as greatest lower bound.
- L_i^{done}, the set of lower bounds on x_{k_i} on the level which we have already tried out as assumed greatest lower bounds but which lead to local conflicts and were thus deemed unfit for that role.
- L_i, the set of all lower bounds on x_{k_i} on the level.
- U_i, the set of all upper bounds on x_{k_i} on the level.
- N_i, the set of all constraints in which x_{k_i} does not occur.
- Additionally, for each constraint c on a level, we save its conflict level $cl(c)$.

If $L_i \neq \emptyset$, then it is compromised of c_i and the sets containing the already tried and the not yet tested lower bounds: $L_i = L_i^{todo} \cup \{c_i\} \cup L_i^{done}$. Otherwise, we set $L_i = L_i^{todo} = L_i^{done} = \emptyset$.
The division of lower bounds into L_i^{todo} and L_i^{done} is necessary for us to know which subtrees of our decision tree have already been visited and which have not. This means that if a current c_i has turned out to not work as greatest lower bound, it is added to L_i^{done}, and the next one will then be chosen from L_i^{todo}.
The theory solver initializes all levels with $C_i = U_i = L_i = L_i^{todo} = L_i^{done} = \emptyset$ as well as $c_i = \bot$ and $x_{k_i} = \bot$. A global variable $maxLvl$ is used to keep track of the deepest currently used level and is accordingly initialized with 0.

Furthermore, the theory solver possesses two sets of original constraints: C and C^{new}, containing all currently considered original constraints and new original constraints, respectively. A constraint is regarded as new if it does not appear in the branch yet. This occurs if it either has been added after the last checkSat call or the branch has been reset. It always holds that $C^{new} \subseteq C$ and both sets are initially empty.
The two sets can be accessed by the addConstraint and removeConstraint functions. For the sake of simplicity, we assume that no constraint is added twice, and that only constraints that are actually in C get removed. If the main algorithm checkSat returns SAT, it empties C^{new} to be able to keep track of the next new constraints. If the branch has been reset, all elements of C are added to C^{new} again since we will have to compute an entirely new branch at the next call of checkSat. A total reset of the branch occurs either due to an UNSAT return value by checkSat or due to removal of an original constraint that had already been incorporated into the branch. A total reset is achieved by calling resetBelow(0), while a partial reset

below a certain level i with $1 \leq i \leq n+1$, as we need it for local conflicts, is done by calling RESETBELOW(i).

Algorithm 3 Adding and removing constraints

```
1 function ADDCONSTRAINT(constraint c)
2     C ← C ∪ {c}
3     C^new ← C^new ∪ {c}
4     cl(c) = 0
5 function REMOVECONSTRAINT(constraint c)
6     if c ∈ C^new then
7         C^new ← C^new \ {c}
8     C ← C \ {c}
```

Algorithm 4 Emptying levels below a threshold

```
1 function RESETBELOW(lvl)
2     for all levels i with lvl < i ≤ n + 1 do
3         x_i ← ⊥
4         L_i^done ← ∅
5         L_i^todo ← ∅
6         L_i ← ∅
7         U_i ← ∅
8         c_i ← ⊥
9     maxLvl ← lvl
```

4.2 Incrementality

4.2.1 The Main Algorithm

Incrementality is the most influential factor on the shape of the core algorithm. We achieve incrementality by trading in the recursion of the original version for a loop iteration. It operates on the aforementioned vector that represents the current branch within the decision tree. This vector persists even between calls of the CHECKSAT() function so that when we add more constraints after a successful call, we can reuse our previous results. Thus, the algorithm is developed to be able to handle the case that there are already constraints on a level that have already been processed in a previous call. The intention is to only compute what is actually necessary, i.e. the constraints that have newly reached this level.
An aspect of this is realized by the runtime variables L^{comb} and U^{comb} which are sets of lower and upper bounds respectively which we provide as parameters to FMPLEXCOMBINE. We call them *combination sets*. They are separate from a level's U_i and

L_i for the purpose of avoiding recomputation of all combinations if it is not necessary. This enables us to incorporate new constraints into our current branch, i.e., level by level, we apply the combinations that previously led us to a set of trivially true constraints to see if these combinations still lead to only trivially true constraints. Thus, whenever new constraints 'arrive' on a level, the two sets enable us to only combine those constraints within these sets with the current eliminator. Of course, if we just chose a new eliminator, we need to apply FMPLEXCOMBINE to all constraints on the level, old and new. For that case, we keep a flag r to indicate if we need to recompute the combinations with a new eliminator.

We iterate over the vector with the lvl variable which indicates the level we are currently working on, starting on level 1. Initially, we hand over the newly added constraints in C^{new} to C_1. The C_i sets of the levels can be understood as a sort of connection point between the levels. As long as we are not working on the level they are always empty. When they receive new constraints from the level above, they pass the constraints relevant for their level on to the corresponding sets, L^{comb} and U^{comb}. Additionally, we add the remaining non-bounds to N_i, however without removing them from C_i. Then, the C_{i+1} set receives the union of the constraints remaining in the C_i set and the results of the FMPLEXCOMBINE operation on the relevant constraints of the current level. It is important for the utilization of incrementality that, at this point, C_i and *not* N_i is passed on to C_{i+1}. The reason for that is that the constraints in $N_i \setminus C_i$ have been passed to the following level in previous calls of CHECKSAT already and we do not need to process them there again.

We enter the loop which repeats as long as we cannot conclude SAT, i.e. as long as C_{lvl} contains anything other than trivially true constraints. Lines 5 to 14 describe the conflict handling. If there is at least one trivially false constraint in C_{lvl}, we call ANALYZEANDBACKTRACK to compute the backtrack level. If there is a global conflict, it returns the special value of 0. As we start on level 1, there is no such level, thus the main algorithm interprets that as the signal for a global conflict. Based on the return value, we react accordingly. In case of a global conflict, we reset the entire branch, add all constraints in C back to C^{new} and return UNSAT. If we only had a local conflict, the lvl variable now indicates the level we backtracked to. As we backtracked here because an eliminator choice led to a conflict, we choose our next eliminator from the L^{todo}_{lvl} set and move the last eliminator into L^{done}_{lvl}. With a new eliminator, we also set the r flag to $true$.
If we did not have a conflict but entered a level greater than $maxLvl$ in the current loop iteration, we execute lines 16 and 17. We have to choose a variable $x_{k_{lvl}}$ to be eliminated on this level and increment $maxLvl$.

Reaching line 18, we are now on the level we want to work on and it is guaranteed to have a variable chosen to eliminate. We then sort any constraints in C_{lvl} in which $x_{k_{lvl}}$ occurs into the combination sets, based on the type of bound they are, and remove them from C_{lvl}. The remaining non-bounds in the set are copied to N_{lvl}. We also add the newly received upper and lower bounds, now in U^{comb} and L^{comb}, to the level's sets of bounds, U_{lvl} and L^{todo}_{lvl} respectively.
If the L^{todo} set now contains lower bounds, but c_{lvl} is still set to $\bot$, we either just newly reached the level, or it was previously a level of only upper bounds but now is

not anymore. In this case, too, we need to choose a new eliminator from L_{lvl}^{todo} and set the r flag.

Should the flag be set, we now execute lines 29 to 32 and change the contents of the combination sets. L^{comb} receives all lower bounds of the level and U^{comb} all upper bounds. C_{lvl} receives all non-bounds we saved in N_{lvl} as the level below has been reset and thus all non-bounds are new bounds for it. Finally, we reset the flag. This way, if we are only on a level to incorporate new constraints into a branch already present from a previous CHECKSAT call, and that level has an eliminator that worked on that previous call, only these new constraints will be contained in the combination sets as we can simply use our previous results from the other constraints. Should we have chosen a new eliminator on this level, either because the level has just been created or it no longer has only bounds of one type or we backtracked to it, the combination sets include all upper and lower bounds on the level.

All that remains to do now is call the FMPLEXCOMBINE operation with the combination sets, the eliminator and the variable to eliminate. We then add the union of its result and the new unused constraints in C_{lvl} to C_{lvl+1} on the next level. We empty C_{lvl} and increment the lvl variable by 1 to move on to the level to which we just passed the new constraints, where we start the loop iteration anew as long as we cannot conclude SAT.

4.2.2 Checking Previous Models

Another way to utilize incrementality and previous results is when after a SAT result, we have computed a model. If the constraints added since do not contain any new variables, we can check if the computed model per chance also satisfies the new constraints as well. If it does, it implies that the old and new constraints together are still satisfiable and we can immediately return SAT instead of even entering the actual execution of the main algorithm. This offers us a quick check for the chance of not having to incorporate the new constraints into the branch for the time being. It however might be possible that we still have to do that later on at another SAT call and the model not fitting does not allow any conclusions regarding the satisfiability of the extended formula.

Example 4.2.1. *Consider again the constraint system (A, b) from Example 2.2.1 with the model $\alpha(x) = 2.5$, $\alpha(y) = -0.25$ obtained in Example 3.4.1. Assume we receive the additional inequalities*

$$\begin{aligned} 12x - 17y &\leq 35 \\ 2x + 39y &\leq -4 \end{aligned}$$

and want to know if these are satisfiable in conjunction with the constraint system. Then, instead of immediately launching into FMPlex, we can substitute x and y with their respective assignments and obtain

$$\begin{aligned} 34.25 &\leq 35 \\ -4.75 &\leq -4, \end{aligned}$$

which are both trivially true. Thus the model is also a model for the two new constraints and we can conclude SAT.

4.3 Heuristics

The algorithm offers several leverage points where different heuristic choices can be applied. When we first come onto a new level i, it only has its set C_i initialized. To continue, we have to choose a variable and, in the implementation which eliminates both via upper and lower bounds, a direction for the elimination: which variable do we eliminate with an eliminator of which bound type? Then, if the level includes both kinds of bounds for a variable, we additionally need to choose an eliminator from the candidates we have not yet tried, not just once, but whenever we backtrack to a level. Furthermore, it is also possible to deploy an alternative mode of backtracking.

4.3.1 Variable Choice

As long as we have variables in the constraints of a level, we always need to choose one of them to be eliminated. In the actual implementation, we additionally need to decide if we want to choose assumed GLBs or assumed SUBs as eliminators. We also refer to this as the *direction* in which the variable is eliminated. It is worth mentioning that on the same level in different branches of the tree, we may choose different variables and directions. This means that the variable ordering may vary from branch to branch.
The approach we chose aims to minimize same-bound combinations and consequently the number of backtracking operations we might have to execute. A secondary goal is to also minimize the total amount of constraints where possible. We make the decisions on the basis of the C_i set of a level which at the time of variable choosing contains all constraints of the level. For a variable x let $\#u(x) = |\{c \in C_i \mid ub(c, x)\}|$ denote the number of upper bounds for the variable that are in C_i, and, analogously, let $\#l(x) = |\{c \in C_i \mid lb(c, x)\}|$ be the number of lower bounds in the constraint set. We choose the variables according to the following priority list:

1. If we have variables with only one type of bound, e.g. if $\#u(x) = 0$ or $\#l(x) = 0$, these have precedence. If we have multiple of this kind, we choose the one with the largest number of total bounds. The reason for that lies in the fact that variables which only have one type of bound allow us to ignore all of these bounds further down in the decision tree, effectively decreasing the number of constraints we have to take into consideration. We do not need to choose a direction in this case as we set $c_{k_i} = \bot$.

2. If we only have variables with both kinds of bounds, we then choose in such a way that we have as few same-bound combinations as possible. The variable chosen is the one with the smallest $min\{\#u(x), \#l(x)\}$, and we decide to eliminate via upper bounds if $\#u(x) < \#l(x)$ and via lower bounds otherwise.

4.3.2 Eliminator Choice

Whenever a level eliminates a variable that has both upper and lower bounds, we need to — possibly repeatedly — choose an eliminator. This essentially means that we need to find a criterion by which we can order the set of possible eliminators. The

approach we present does this by comparing how many of the original constraints go into them, then give preference to those with less original constraints to keep the linear combinations as compact as possible.

4.3.3 Backtracking

When we encounter a local conflict, we have to decide which level we want to backtrack to. Whichever heuristic we deploy, it might happen that we land on a level on which the set of remaining eliminator candidates is empty. This happens if all possible eliminator choices on a level have been tried and resulted in a local conflict but the incorrect choice was not at the last same-bound combination, but rather another at one further up in the tree. In that case, we move back further, one level at a time, until we reach one for which that is not the case. However, apart from this case, we must not backtrack further than the conflict level of a constraint as this might mean skipping sub-trees in which we could have obtained an answer.

One option, as mentioned in Section 3.3, is making use of the conflict levels of constraints, yielding the analyzeAndBacktrack algorithm as addition to be called by the checkSat algorithm.
We iterate over all trivially false constraints on a level. If we come upon a global conflict, we immediately return the special value 0. Otherwise, we save the smallest conflict level, i.e. the furthest one from our current level, as backtrack level. From that backtrack level, we then - if necessary - go further back as long as the L^{todo} set of the backtrack level is empty.
An alternative to this is always just going back as few levels as possible, i.e. a single level plus any additional levels we need to go back because of exhausted eliminator choices. This approach enables us to reach possible global conflicts faster as they might occur in direct sibling nodes of the one we found a local conflict in. Additionally, we do not actually have to compute the conflict levels of constraints upon combination, no matter which kind.

Algorithm 5 The main algorithm for checking satisfiability

```
1  function checkSat
2      lvl ← 1
3      C_1 ← C_1 ∪ C^new
4      while C_lvl ⊈ {true} do
5          if false ∈ C_lvl then                          ▷ In case of conflict
6              lvl ← analyzeAndBacktrack(C_lvl);
7              if lvl = 0 then                            ▷ Global
8                  resetBelow(0)
9                  C^new ← C
10                 return UNSAT
11             else                                       ▷ Local
12                 chooseNextConstraint(lvl)
13                 resetBelow(lvl)
14                 r ← true
15         else if lvl > maxLvl then                      ▷ New level reached
16             x_{k_lvl} ←chooseNextVariable(C_lvl)
17             maxLvl ← maxLvl + 1
18         L^comb ← {c ∈ C_lvl | lb(c, x_i)}
19         U^comb ← {c ∈ C_lvl | ub(c, x_i)}
20         C_lvl ← C_lvl \ (L^comb ∪ U^comb)
21         N_lvl ← N_lvl ∪ C_lvl
22         L^todo_lvl ← L^todo_lvl ∪ L^comb
23         U_lvl ← U_lvl ∪ U^comb
24         if c_lvl = ⊥ ∧ L^todo_lvl ≠ ∅ then
25             chooseNextConstraint(lvl)
26             L^comb ← L^comb \ {c_lvl}
27             r ← true
28         if r = true then                               ▷ Recomputation with new eliminator
29             L^comb ← L^done_lvl ∪ L^todo_lvl
30             U^comb ← U_lvl
31             C_lvl ← N_lvl
32             r ← false
33         C_lvl+1 ← C_lvl+1 ∪ C_lvl ∪ fmPlexCombine(L^comb, c_lvl, U^comb, x_{k_lvl})
34         C_lvl ← ∅
35         lvl + +
36     C^new ← ∅
37     return SAT
```

Algorithm 6 Determining the backtracking level

```
1  function analyzeAndBacktrack(C_lvl)
2      btLevel ← lvl
3      for each trivially false constraint c = (a_1, ..., a_n, b | d_1, ..., d_m) in C_lvl do
4          if d_1, ..., d_m > 0 or d_1, ..., d_m < 0 then
5              return 0
6          else
7              if cl(c) < btLevel then
8                  btLevel ← cl(c)
9      while L^todo_btLevel = ∅ do
10         btLevel ← btLevel − 1
11     return btLevel
```

Chapter 5

Implementation and Testing

5.1 Implementation

An experimental implementation of the incremental method has been developed on top of the SMT Solver SMT-RAT [CKJ+15] in the programming language C++.

5.1.1 Differences to Theory

There are a number of differences between theory and implementation, most of them being simplified for the explanation of the basic principle in this thesis while being adapted for higher time and memory efficiency in the actual implementation.
For example, the branch in the implementation is not a vector, but instead a list of levels so as to only use the storage space we need. What were sets of constraints in the algorithm presented here have been implemented as lists of constraints as well. For the constraints themselves, the pre-existing SimpleConstraint data type has been used. The conflict level of a constraint is given as a list iterator indicating the corresponding element in the branch list. Derivation coefficients are implemented as a map as we, unlike assumed here in the paper for simplicity, do not know the possible total number of constraints the theory solver would receive for checking. The map was therefore an obvious choice as we can add new derivation coefficients easily and only need to keep track of those necessary.
Another crucial difference, as already touched upon in Section 4.3, is that we eliminated not solely via assumed GLBs but also via assumed SUBs to provide more adaptability, which is why the choice of a variable to eliminate also brought the choice of a direction with it.
The implementation has furthermore been created in such a way that possible additions and extensions, especially further heuristics for variable, direction and eliminator choices may be integrated easily, should the need arise. To allow a comparison between the utilization of incrementality and the lack thereof, it can be turned on and off via the settings of the module.

5.2 Testing Results and Discussion

The implementation was tested on the quantifier-free linear real arithmetic benchmark set from SMT-LIB [BFT16], containing 1753 individual benchmarks. They were modified only by replacing equalities with two weak inequalities as described in Section 3.5. The tests were conducted on the High Performance Cluster of RWTH Aachen University with a time limit of 5 minutes and a memory limit of 5 Gigabyte. The *Standard* configuration deploys incrementality, the previously presented heuristics and the previously discussed ANALYZEANDBACKTRACK backtracking mode. For a general overview of the overall effect of the presented improvements of the algorithm, we first test it against a *None* configuration that uses neither incrementality, nor heuristics, nor the conflict level based backtracking.
Following that, we compare it to three different configurations, each missing only one of the presented aspects: The *No-Incr* configuration without incrementality, the *No-Heu* configuration where we do not apply heuristics and always choose the first variable and constraint we can find and the *One-Step* configuration that uses the alternative backtracking mode of only backtracking as far as absolutely necessary.

The possible results of an instance are

- *MO*: The given memory has been exceeded and the instance could thus not be solved,
- *TO*: The given time has been exceeded and the instance could thus not be solved,
- *SAT*: The instance was correctly determined to be satisfiable,
- *UNSAT*: The instance was correctly determined to be unsatisfiable,
- *WRONG*: The instance was incorrectly determined to be satisfiable or unsatisfiable and
- *SEGFAULT*: An error occurred during execution of the algorithm.

As the latter two results WRONG and SEGFAULT did not occur for any of the configurations, they will be omitted in the following evaluation.
Table 5.1 gives a first overview of the amount of instances solved per configuration while Figure 5.1 shows the total runtime for the number of solved instances.

Configuration	**MO**	**TO**	**SAT**	**UNSAT**	**Total Unsolved**	**Total Solved**
Standard	358	682	426	287	1040	713
None	167	1461	49	76	1628	125
No-Heu	195	1417	56	85	1612	141
One-Step	354	731	403	265	1085	668
No-Incr	216	794	431	312	1010	743

Table 5.1: An overview of the different configurations

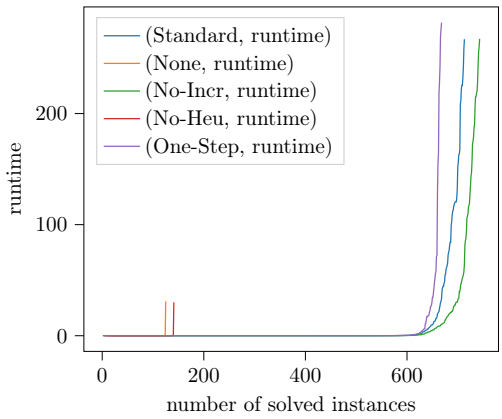
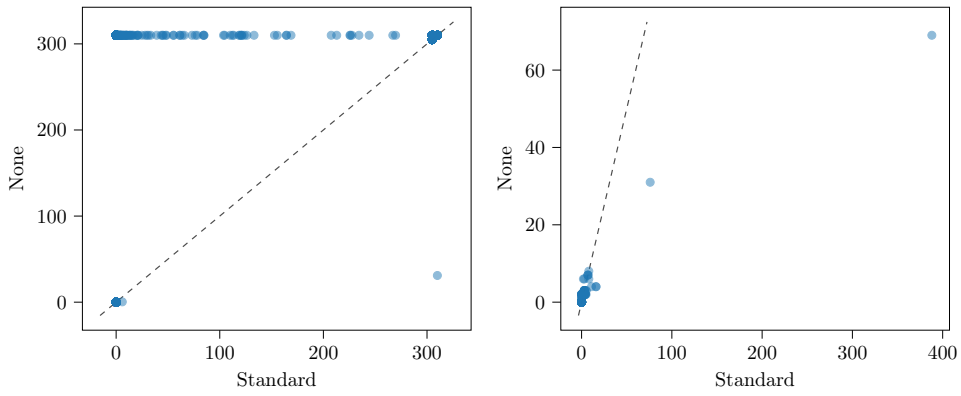


Figure 5.1: Performance profile in regards to runtime

5.2.1 Overall Effects

Both Table 5.1 and Figure 5.1 illustrate the severe difference in solved instances and required runtime between the *Standard* configuration and the *None* configuration which does not apply any of the presented improvements. It becomes obvious that the former is performing significantly better than the latter. The greater part of unsolved instances of the *None* configuration can be attributed to timeouts: In comparison to the *Standard* configuration, 589 instances that were originally solved now created timeouts, almost all timeouts were carried over and there were an additional 191 instances that, instead of exceeding memory limits, now also created timeouts.



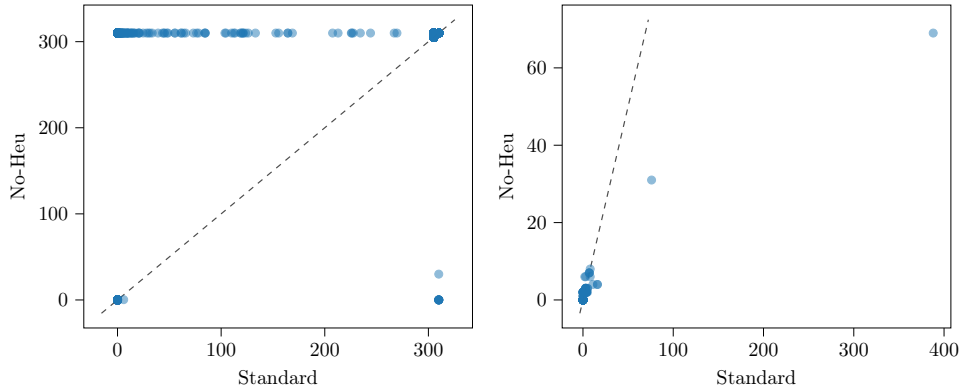
(a) Runtime (b) Number of generated constraints

Figure 5.2: *Standard* and *None* configuration in comparison

The instances that were solved by both configurations seem to behave similarly, as shown in Figure 5.2. However, each of these instances consisted of at most a single checkSat call and, with a few exceptions, barely generated constraints, thus not offering a reliable basis to make a conclusive statement on.

5.2.2 Heuristics: Variable, Direction and Eliminator Choice

Surprisingly, testing revealed the heuristics for variable, direction and eliminator choices to be the most influential factor on the performance of the algorithm. Out of the configurations which each have a single improvement area disabled, the *No-Heu* configuration exhibited the strongest deviation from the *Standard* configuration. This becomes evident in Table 5.1 and both Figures 5.1 and 5.3.



(a) Runtime (b) Number of generated constraints

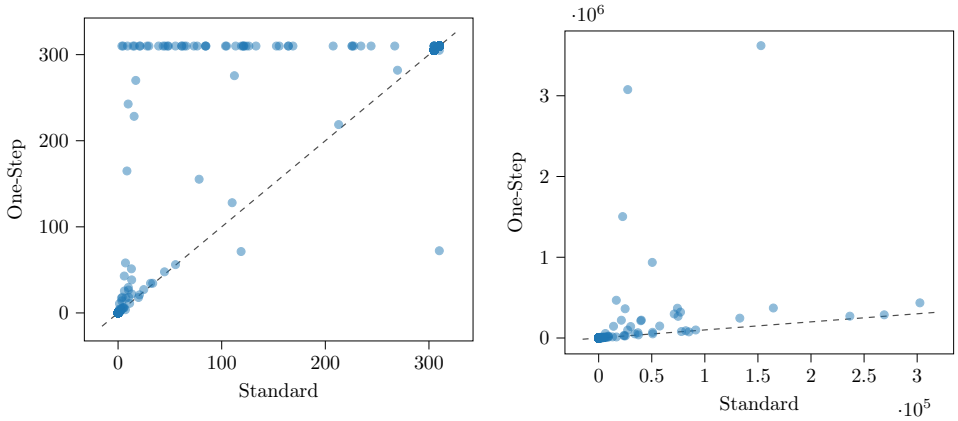
Figure 5.3: *Standard* and *No-Heu* configuration in comparison

The results moreover bear a striking resemblance to these of the *None* configuration with only slight variations, further corroborating the crucial impact these heuristics have on the overall performance. The majority of the differences in the number of timeouts and exceeded memory between the *Standard* and the *None* configurations can thus be attributed to the heuristics.
On one hand, this implies that efficiently chosen variables, directions and eliminators are decisive for the runtime of a problem instance. On the other hand, the 163 instances that exceeded the memory limit in the *Standard* configuration but resulted in timeouts without the heuristics are of note as well: They indicate that, while the heuristics allow for a significant reduction of runtime, they pose a considerable contribution to memory usage. On this set of benchmarks however, this might be considered negligible as there were no cases of MO results from the *Standard* configuration that were actually solved in the *No-Heu* configuration as well as numerous additional timeouts. Whether this holds up for other problems remains an open question.
Again, the instances solved by both configurations consisted of very few or even no checkSat calls at all and do not lend themselves to any further substantial conclusion except for the importance of heuristics for the algorithm.

5.2.3 Heuristics: Backtracking Modes

As outlined in Figure 5.1, the *Standard* configuration with conflict level based backtracking mostly outperformed the *One-Step* configuration in regards to runtime. The reasoning for the latter was that, given a local conflict, just backtracking as little as possible might allow us to find a potential nearby global conflict faster than in the conflict level based backtracking. However, apart from the slightly greater number of

unsolved instances, largely due to timeouts, there is only a single (UNSAT) instance which was unsolved in the *Standard* configuration but solved in the *One-Step* configuration, suggesting that this is not the case.



(a) Runtime

(b) Number of generated constraints

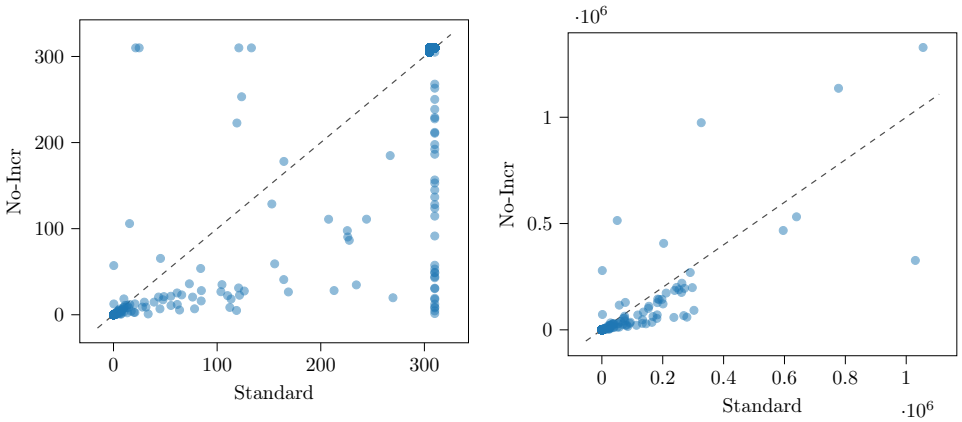
Figure 5.4: *Standard* and *One-Step* configuration in comparison

Figure 5.4 further confirms that this backtracking mode does not provide us with any advantage, or at least not with one that is sufficient to outweigh the additional runtime and generated constraints that are arise when deploying this method. We thus conclude that the backtracking according to the conflict levels of the trivially false constraints has proven itself to be more beneficial.

5.2.4 Incrementality

The comparison between the incremental *Standard* and the non-incremental *No-Incr* configurations produced the unexpected result of the non-incremental version overall faring better than the incremental version. One remarkable aspect is that the *No-Incr* configuration produced less MO results but more TO results. Of the 182 instances with differing results, 143 ended in an MO result for the *Standard* version but a TO result in the *No-Incr* configuration. Yet, the total number of instances solved by the latter is higher with most of the additionally solved ones being unsatisfiable.

The scatter plots of Figure 5.5 further illustrate that the non-incremental version of the algorithm generally tends to outperform the incremental approach in regards to both runtime and number of generated constraints.
In the implementation, the only additional step the *No-Incr* configuration takes is resetting both the branch and the current model at the beginning of a checkSat call, otherwise both versions proceed in the same manner. One possible reason for the differing performances would thus be that testing a previous model costs more resources than it eventually saves. This, however, was disproven by further testing. Whether or not the model of a previous call is tested or not showed to have very little to no effect on runtime and generated constraints, only saving approximately as much time as it requires to be applied.



(a) Runtime

(b) Number of generated constraints

Figure 5.5: *Standard* and *No-Incr* configuration in comparison

The main cause of the *No-Incr* configuration being more efficient are thus likely the heuristics for variable and direction choices. If a previous checkSat call returned SAT, we retain the created decision tree branch in the incremental version and later on merely incorporate additional constraints into that tree, i.e. all decisions for variables and directions that have been made in the tree so far must be adhered to as long as we do not backtrack above them and reset the nodes below in the branch. This entails that we have to adhere to these decisions even if the heuristics would no longer deem them ideal given the additional constraints of the next call of checkSat. This can result in the decision tree becoming unnecessarily large and, in turn, increased runtime for iterating through it. In contrast, the non-incremental version always completely rebuilds the decision tree in each call. This means the algorithm can, at each node, make more informed choices with its heuristics as they can consider the entire current set of constraints and does not have to follow previous choices that were based on only a subset of these constraints that were available in a previous checkSat call. Figure 5.5 further supports this. The smaller instances that required less time and generated constraints mostly result in similar behavior for both versions, but the larger the instances, the more efficient the *No-Incr* configuration proves itself. The resulting advantage effects the results in a number of ways: Firstly, there are less MO results in the non-incremental version as the decision trees in each checkSat call are kept as small as possible while this does not happen. Secondly, if we disregard the MO results turned into TO results, there are also less timeouts, resulting in a greater total number of solved instances which generally also were completed in less time and with fewer generated constraints.
Of course this raises the question if this result would still look similar if no heuristics were used. However, as already mentioned in Section 5.2.2, the instances that are solved without heuristics are so few and so small that this would not be viable to obtain any kind of reliable statement about that.

Chapter 6

Conclusion

6.1 Summary

We presented the FMPlex algorithm for satisfiability checking of sets of linear real constraints as a combination of the Fourier-Motzkin variable elimination and the Simplex algorithm to reduce the former's doubly exponential runtime. It eliminates variables by choosing an assumed greatest lower or smallest upper bound for them and combining these with the other bounds of the given variable. As such choices can be incorrect and lead to conflicts that however do not allow us to conclude UNSAT immediately, we introduced the notion of backtracking.
As a theory solver for a lazy SMT solver, this algorithm is expected to profit from incrementality. Making use of appropriate data structures, we thus presented an incremental version of it that reuses previous results and, if available, models. Furthermore, we introduced heuristics for choosing variables, directions and constraints to eliminate them with on a given level of the algorithm as well as two kinds of backtracking modes.
After a short description of the implementation of the incremental version and its differences to the theory given in this thesis, we presented the results of testing the experimental implementation on a set of benchmarks. While the heuristics for variable, direction and eliminator choice had an unexpectedly large impact on performance improvement and the conflict level based backtracking offered an additional moderate speed-up, the incremental version actually fell behind the non-incremental version. We arrived at the conclusion that this can most likely be attributed to the shape and size of the tree in later checkSat calls where the incremental version has to adhere to previous choices of variables and directions while the non-incremental one can choose these anew tailored to the actual set of constraints in every call.

6.2 Future Work

There are various aspects of this method still offering potential for further research. One of these is whether incrementality can possibly still be of use under different circumstances or with different or additional heuristics. As mentioned in Section 5.1.1, such heuristics may be easily added and tested. To utilize it even further, it also might

prove beneficial to adapt the algorithm in such a way that removing a constraint does not necessitate resetting all previous results.
Furthermore it might be worth investigating the utilization of models of previous checkSat calls in regards to what it means when a constraint is satisfied by a model even if not all new constraints are satisfied by it and if it could be used to make the model testing actually improve efficiency.

While this thesis treated equalities and their negations by converting them into equivalent Boolean combinations of inequalities, the algorithm might benefit from a more specific and direct handling as e.g. equalities always represent a GLB *and* an SUB.

Bibliography

[BdM09] Nikolaj Bjørner and Leonardo de Moura. Z3[10]: Applications, enablers, challenges and directions. In *Sixth International Workshop on Constraints in Formal Verification Grenoble, France*, 2009.

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

[Bla77] Robert G Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368. Springer, 2015.

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

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[Dan72] George B Dantzig. Fourier-Motzkin elimination and its dual. Technical report, STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972.

[Dan90] George B Dantzig. Origins of the simplex method. In *A History of Scientific Computing*, pages 141–151. Association for Computing Machinery, 1990.

[Dan16] George Dantzig. *Linear programming and extensions*. Princeton University Press, 2016.

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

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, sep 2011.

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

[Fou27] Jean-Baptiste-Joseph Fourier. Analyse des travaux de l'académie royale des sciences pendant l'année 1824, partie mathématique. *Histoire de l'Acad emie Royale des Sciences de l'Institut de France*, 7, 1827.

[HZ15] Thomas Dueholm Hansen and Uri Zwick. An improved version of the random-facet pivoting rule for the simplex algorithm. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 209–218. Association for Computing Machinery, 2015.

[Kal92] Gil Kalai. A subexponential randomized simplex algorithm. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, pages 475–482, 1992.

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

[Kha79] Leonid Genrikhovich Khachiyan. A polynomial algorithm in linear programming. In *Doklady Akademii Nauk*, volume 244, pages 1093–1096. Russian Academy of Sciences, 1979.

[KM72] Victor Klee and George J Minty. How good is the simplex algorithm. *Inequalities*, 3(3):159–175, 1972.

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

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

[MSM18] Joao Marques-Silva and Sharad Malik. *Propositional SAT Solving*, pages 247–275. Springer International Publishing, Cham, 2018.

[ORS92] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.