MASTER OF SCIENCE THESIS

A NOVEL ADAPTION OF THE SIMPLEX ALGORITHM FOR LINEAR REAL ARITHMETIC

Jasper Nalbach

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

Additional Advisor:
Gereon Kremer

Aachen, 4th March 2020

Abstract

Satisfiability modulo theories (SMT) solving is a technology for checking the satisfiability of *quantifier-free first-order logic formulas*, i.e. Boolean combinations of constraints from some theories. For solving the theory of *linear real arithmetic*, the *Simplex algorithm* has successfully been adapted for the *DPLL(T)* framework. However, this approach reaches its limits at instances with complex Boolean structure. This thesis presents a novel adaption of the Simplex method attempting to overcome these limits by interleaving Simplex pivot steps with the SAT solving process.

Acknowledgements

I am grateful for the opportunity to work on this topic and the insights achieved throughout the work on this thesis. My thanks go to Prof. Erika Ábrahám for the discussions and valuable suggestions, to Gereon Kremer for joining the discussions and giving technical advise and to the additional examiner Prof. Christina Büsing.

Thanks to my family for encouraging and supporting me.

Contents

Chapter 1

Introduction

Undoubtedly, there are only few technologies that influenced human life as much as computerized systems. Algorithms, step-by-step recipes for information transformation, form the aorta of modern life. Most of these algorithms are tailored for solving a specific problem allowing highly efficient computations. However, ensuring that their solutions are correct is a hard challenge and sometimes no less important than their efficiency. Hence, developing these algorithms is hard and not always feasible. This is why general frameworks admitting the formulation of a wide range of problem statements have been developed, where the user does not need to know how to solve the problem - this task is handed over to a solver. These solvers help to verify that problem-specific algorithms work correctly and are able to solve logical problems deductively. Developing an efficient solver for a general problem can only succeed by focussing heuristically on practical instances. This is a consequence of the so called *no free lunch theorems* [WM95, Wol96, WM97], stating, shortly speaking, that in expectation, each algorithm performs equally bad (or well) on the set of all possible instances. Hence, it is theoretically proven that this discipline will never render itself obsolete and the strive for mighty solvers is never ending.

A long-known and well-studied problem is the *Boolean satisfiability problem (SAT)* being the first problem proven to be *NP-complete* [Coo71, Lev73]. That means, simply speaking, a wide range of interesting problems in computer science can be reduced to this problem. As SAT admits a simple but natural language, there has been intensive research over the past decades resulting in reasonably fast SAT solvers. The most common approach are the *conflict-driven clause-learning (CDCL)* [MSS96, MMZ$^+$01] solvers building on the *DPLL* [DP60, DLL61] algorithm.

The problem statement of SAT has been extended for first-order logic, namely *satisfiability modulo theories (SMT)*. SMT formulas are often restricted to the quantifier-free fragment, that is, they are Boolean combinations of constraints from some theories. The theory of interest in this thesis is *linear real arithmetic (LRA)*. A successful technique for solving such formulas is the *DPLL(T)* framework, a modular framework consisting of a CDCL-style SAT solver and a theory solver. The former solves a Boolean abstraction of the formula while the latter ensures the consistency of the Boolean solution with the underlying theory.

The constraints of linear real arithmetic are linear equalities and strict or weak inequalities. Well-known decision procedures for this theory are the *Fourier-Motzkin variable elimination* [Fou27, Mot36], the ellipsoid method [Kha79] and an adaption of the *Simplex* [Dan98] method, where the latter is the most popular one being implemented in many solvers. In this thesis, we will present a novel adaption of the Simplex method into the DPLL(T) framework aiming for better performance on problems with complex Boolean structure.

This thesis starts in Chapter 2 with an introduction of the DPLL(T) framework, systems of linear constraints, the fundamental theorem of linear programming and the general Simplex algorithm. The fundamental theorem of linear programming is extended for strict inequalities in Chapter 3 as a requirement for solving LRA formulas. In Chapter 4, an adaption of the Simplex algorithm and a novel embedding into the DPLL(T) framework is presented. In Chapter 5, the procedure is compared with the general Simplex method, experimental results are analysed and possible extensions as well as alternative approaches are discussed. Finally, in Chapter 6, we will conclude the thesis.

Chapter 2

Preliminaries

2.1 SAT and SMT

2.1.1 Boolean satisfiability problem (SAT)

Given a set of *Boolean variables* $B = \{b_1, \ldots, b_n\}$, the set of *Boolean formulas* (*in* B) is inductively defined: All variables in B are Boolean formulas and if ψ and ϕ are Boolean formulas then their *negation* $\neg\psi$ and *conjunction* $(\psi \wedge \phi)$ are Boolean formulas, too. As syntactic sugar, we introduce *disjunction* $(\psi \vee \phi)$, *implication* $(\psi \rightarrow \phi)$ etc. as usual.

Given a Boolean formula ϕ in variables B, an *assignment* $\alpha : B \rightarrow \{0{,}1\}$ assigns each Boolean variable to a truth value allowing to evaluate ϕ to 0 or 1 (*false* respectively *true*) using the usual interpretation of the *connectives* $\neg, \vee, \wedge, \rightarrow$, etc. We say that a Boolean formula ϕ is *satisfiable* if and only if there exists an assignment α such that ϕ evaluates to *true*, written as $\alpha \models \phi$; otherwise, we call ϕ *unsatisfiable* or *conflicting*. The *Boolean satisfiability problem (SAT)* is the problem to decide the satisfiability of a given formula.

SAT is the first problem known to be NP-complete [Coo71, Lev73]. This means on the one hand, that many interesting and practical problems in computer science can be reduced to SAT; but on the other hand, that most probably there exists no polynomial-time algorithm for deciding the SAT problem. Nevertheless, there exist algorithms and tools - so called *SAT solvers* - working considerably well on practical problem instances.

2.1.1.1 The DPLL-CDCL algorithm

The most popular SAT-solving algorithms extend the *Davis-Putnam-Logemann-Loveland (DPLL) algorithm* [DP60, DLL61] with *Conflict-Driven-Clause-Learning (CDCL)* [MSS96]. The original DPLL algorithm used enumeration and propagation for the search and in case of a conflict backtracked to the last decision chronologically. This approach has been extended with conflict-driven clause-learning to explain the roots of conflicts using Boolean resolution. Explanation clauses are an implication of the original clause set and thus can be added to the original clause set to guide the solving process (*clause learning*).

Such SAT solving algorithms assume the input to be in CNF:

Definition 2.1.1 (Conjunctive normal form (CNF))**.** *A formula is in* conjunctive normal form (CNF) *if and only if it is of the form*

$$\varphi = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{m_i} l_{i,j} \right)$$

where $l_{i,j} \in \{b, \neg b \mid b \in B\}$*. The* $l_{i,j}$ *terms are called* literals*, and the* $\vee_{j=1}^{m_i} l_{i,j}$ *terms are called* clauses*.*

Using Tseitin's transformation [Tse83], any formula can be transformed into a satisfiability-equivalent CNF formula in polynomial time at the cost of additional variables.

An overview of the DPLL-CDCL procedure is given in Algorithm 1. The procedure maintains a partial assignment that is consistent with every clause (i.e. no clause evaluates to *false*). Whenever possible, the procedure deduces values for yet unassigned variables: If all literals of a clause except one evaluate to *false* under the current assignment and the remaining literal does not evaluate to a value yet, a unique value for the variable in the remaining literal is implied that satisfies the clause; this is called *unit clause propagation*. If no propagations are possible, a heuristic picks a yet unassigned variable and assigns it to a value (called a *decision*). Iterating decision and propagation terminates when either a complete assignment satisfying φ is created this way, or a conflict occurs, where a clause evaluates to *false* under the current assignment. In the latter case, *Boolean resolution* is applied on this *conflict clause* and some other clauses that contributed to the conflict by unit propagation; this results in an *explanation clause* excluding the current assignment and some others that would lead to the same conflict. If the conflict happened before any decisions have been made, then the whole formula is unsatisfiable. Otherwise, some conflicting decisions are backtracked and the procedure continues.

The performance of this procedure depends highly on the decision heuristic and a bunch of optimizations; for more details, we refer to [BHvM09].

Algorithm 1: DPLL

```
function DPLL($C$)
    Input: a set of clauses $C$
    Output: SAT if $C$ is satisfiable or UNSAT otherwise
    while $true$ do
        while exists unit clause $c \in C$ do
            Propagate($c$)
            if exists conflicting clause $c \in C$ then
                if $\neg$ Resolve($c$) then
                    return $UNSAT$
        if all variables assigned then
            return $SAT$
        Decide()
```

2.1.1.2 Minisat

The experimental part of this thesis will rely on `Minisat` [ES03], a DPLL-CDCL-style SAT solver featuring some popular SAT heuristics (amongst others):

- The order in which the Boolean variables are decided is determined by the *variable state independent decaying sum (VSIDS)* [MMZ$^+$01] scheme. Shortly speaking, it tracks for each variable an activity score which is increased whenever a clause that was used for resolution to derive an explanation for a conflict contained this variable or its negation. Whenever a decision is made, an unassigned variable with the highest activity score is picked. In order to focus on more recent conflicts, the increment is increased after each conflict. To avoid overflow, all activity scores are regularly divided by a constant factor.
- The first time a variable is decided, `Minisat` assigns the value *false* to it. If a decision variable has already been assigned before (but this assignment has been reverted), then it will be set to its last value.
- As the SAT solver progresses, the clause database grows as clauses are learned. While learned clauses drive the solving process, too many clauses cause a remarkable additional effort for propagation and memory management. This is why the clause database is reduced regularly: For each clause, an activity score similarly to the variables is maintained. Whenever the number of learned clauses reaches a certain threshold *max_ learnts* (relative to the initial number of clauses), the learned clauses with an activity below a certain threshold are deleted. To ensure completeness, *max_ learnts* is regularly increased.

2.1.2 Satisfiability modulo theories (SMT)

SMT is an extension to the SAT problem, asking whether a sentence in the *existential fragment of first order logic* is satisfiable with respect to one or multiple background theories. We omit a formal definition here, but give a restricted view, motivating an algorithmic solution to the problem: An SMT formula is a SAT formula where some of the Boolean variables are replaced by predicates (*theory constraints*), which are Boolean-valued functions of *theory variables.* These constraints are evaluated with respect to its associated theory. For *linear real arithmetic*, those are linear inequalities such as $4x_1 + 5x_2 \leq 0$ where x_1 and x_2 are real-valued *variables.* There exists a variety of further theories, such as *non-linear real arithmetic*, *uninterpreted functions*, etc., where some of them can be combined. For more details, we refer to [BHvM09].

An SMT solver solves such problems, and in case of satisfiability generates also a satisfying assignment for the theory variables. The SMT-LIB standard [BST$^+$10] defines theories, a protocol and a file format for SMT problems.

2.1.2.1 DPLL(T)

There are several approaches for SMT solving. The *eager* approach reduces an SMT formula to a SAT instance and then passes the problem to a SAT solver. While this works well enough for simple theories, another approach is employed for most theories: In *lazy* SMT solving, a SAT solver communicates with a theory solver, meaning that the SAT solver solves a Boolean abstraction of the formula while the theory solver checks the consistency with the underlying theories. If the current Boolean solution does conflict with the theory, the theory solver generates *explanations* expressing the theory conflict by a Boolean formula. We go into more detail:

Definition 2.1.2 (Boolean abstraction)**.** *Let φ be a quantifier-free first-order logic formula. By abstraction(φ) we denote the Boolean formula obtained by replacing each constraint c in φ by a fresh Boolean variable b_c (and the constraint equivalent to $\neg c$ by $\neg b_c$).*

The *DPLL(T)* algorithm is an extension to the DPLL algorithm. Let in the following α denote the current Boolean variable assignment in the solver. The theory solver regularly checks the set

$$I := \{c \mid \alpha(b_c) = 1\} \cup \{\neg c \mid \alpha(b_c) = 0\}$$

of constraints for consistency with the underlying theory, thus needs only to be able to handle conjunctions of theory constraints. For inconsistent constraints, the theory solver returns an explanation in form of a theory lemma that is violated by the current assignment. In the simplest case, the theory solver might return $\vee_{c \in I} \neg c$ as explanation. However, most solvers are able to compute a smaller *infeasible subset* of I, excluding bigger search areas. The Boolean abstraction of this lemma lifts the conflict from the theory to the Boolean level, thus advises the SAT solver to backtrack and continue the search.

There are two variants of DPLL(T): A *full-lazy* variant calling the theory solver only if a complete Boolean assignment has been found, satisfying the whole abstraction; and a *less-lazy* variant calling the theory solver before every decision. In the latter case, a theory solver should be able to keep information across theory calls: Building up the theory model *incrementally* and be able to *backtrack* conflicts.

An overview of the less-lazy procedure is given in Algorithm 2.

Algorithm 2: Less-lazy DPLL(T)

```
function DPLL($C$)
    Input: a set of clauses $C$
    Output: SAT if $C$ is satisfiable or UNSAT otherwise
    while true do
        while no new clauses are added to $C$ do
            while exists unit clause $c \in C$ do
                Propagate($c$)
                if exists conflicting clause $c \in C$ then
                    if $\neg$ Resolve($c$) then
                        return UNSAT
            $C := C \cup$ TheoryCheck()
        if all variables assigned then
            return SAT
        Decide()
```

2.2 Linear constraints

We point out that we introduce a slightly peculiar notation and usage of the mathematical objects deviating from usual presentations. For a proper introduction to the basics of linear algebra, we refer to [Axl97, Hef18, SR12].

In the following, let $\mathcal{F}$ be a field, $(\mathcal{U}, <)$ an ordered vector space over $\mathcal{F}$ (or $\mathcal{F}$-vector space) and $X = \{x_1, \ldots, x_n\}$ a set of $\mathcal{U}$-valued variables.

For $a_1, \ldots, a_n \in \mathcal{F}$, $b \in \mathcal{U}$ and a *relation symbol* $\sim \in \{= , \leq , \geq , < , > , \neq\}$, we call $p = a_1 \cdot x_1 + \ldots + a_n \cdot x_n$ a *linear $\mathcal{F}$-combination of X* and $p \sim b$ a *linear constraint (over $(\mathcal{U}, <)$ in variables X)*. An *equation* is a linear constraint with $\sim \in \{=\}$. A constraint $p \sim b$ is called *weak* if $\sim \in \{=, \leq, \geq\}$ and *strict* otherwise. By $\bar{c}$ we denote the negation of a linear constraint c, i.e. $=$ is replaced by $\neq$, $<$ is replaced by $\geq$ and so on.

A constraint c can be evaluated under an *assignment* $\alpha : X \to \mathcal{U}$ the standard way. If c is satisfied by α (denoted as $\alpha \models c$), it is called a *solution*. The *solution set of c* is defined as $\mathit{sol}(c) = \{\alpha : X \to \mathcal{U} \mid \alpha \models c\}$.

Furthermore, let p be a linear $\mathcal{F}$-combination of X, then $\alpha(p) \in \mathcal{U}$ denotes the value of p under α.

Furthermore we introduce some notation for an assignment $\alpha : X \to \mathcal{U}$: Let $X' \subset X$, then $\alpha \downharpoonright_{X'}: X' \to \mathcal{U}$ denotes the *restriction* of α to X'. Let $x \notin X$ be a variable and $v \in \mathcal{U}$, then $\alpha[x \mapsto v] : (X \cup \{x\}) \to \mathcal{U}$ denotes the *extension* of α where x is assigned to v.

Let $\mathcal{U}$ be a structure and x be a variable, then $\mathcal{U}[x]$ denotes the structure $\mathcal{U}$ extended by x such that $\mathcal{U}[x]$ is closed under all its operations. Let $\alpha : X \to \mathcal{U}[x]$ be an assignment, $x \notin X$ and $v \in \mathcal{U}$, then $\alpha[v/x] : X \to \mathcal{U}$ is the assignment where x is replaced by v in the values of the target domain $\mathcal{U}[x]$.

Notation In the following, instead of $i = 1, \ldots, n$, we write $i \in [n]$.

Linear real arithmetic (LRA) A formula φ in *linear real arithmetic (LRA)* is a Boolean combination of linear constraints $p \sim b$ over $(\mathbb{R}, <)$ where $\mathbb{R}$ is viewed as a $\mathbb{Q}$-vector space and $b \in \mathbb{Q}$. The set of linear constraints in a formula φ is denoted by *Constraints*(φ); the set of $\mathbb{R}$-valued variables in φ is denoted by *Vars*(φ).

Note that if such a formula φ has a solution over $(\mathbb{R}, <)$, then it has also a solution over $(\mathbb{Q}, <)$.

Systems of linear constraints A *systems of linear constraints over* $(\mathcal{U}, <)$ is a finite set of linear constraints. The notions for solutions can be extended for systems by treating the system as conjunction of its elements. Two systems C_1, C_2 of linear constraints are *equivalent* (denoted as $C_1 \equiv C_2$) if and only if $\mathit{sol}(C_1) = \mathit{sol}(C_2)$.

A system of linear constraints with only equations is called an *equation system.*

Furthermore, given a set $C = \{p_i \sim_i b_i \mid i \in [m]\}$ of linear constraints, we define the set of left hand sides of the constraints in C as $P_C = \{p_i \mid i \in [m]\}$.

Definition 2.2.1. *A system C of linear constraints is called* consistent *or* satisfiable *if and only if $\mathit{sol}(C) \neq \emptyset$, otherwise it is called* inconsistent *or* conflicting.

Definition 2.2.2. *A set $\{p_i \mid i \in [m]\}$ of linear $\mathcal{F}$-combinations of a set X of variables is called* linearly independent *if and only if for all $f_1, \ldots, f_m \in \mathcal{F}$*

$$f_1 \cdot p_1 + \ldots + f_m \cdot p_m = 0 \iff f_1 = \ldots = f_m = 0$$

and linearly dependent *otherwise.*

A set C of linear constraints is called linearly independent (dependent) if and only if P_C is linearly independent (dependent).

Note that in the above definition, if a constraint is trivial (i.e. some p_i is 0), then the whole set is linearly dependent.

Definition 2.2.3. *The* rank *of a system C of linear constraints is defined as*

$$\mathit{rank}(C) = \max \{ |C'| \mid C' \subseteq C, C' \text{ linearly independent}\}.$$

C is called underdetermined *if and only if $\mathit{rank}(C) < n$ where n is the number of variables in C.*

2.3 Satisfiability of systems of weak linear constraints

In the following, we first focus on the satisfiability of systems of weak linear constraints. Furthermore, we will assume $\leq$ as the only relation, as a constraint with relation $=$ can be replaced by two constraints with $\leq$ and $\geq$; constraints with $\geq$ in turn can be turned into ones with $\leq$ by multiplying them with -1.

2.3.1 Linear programming

In this thesis, we make use of some results from the theory of *linear programming*. A *linear program* is an optimization problem $\max_{\boldsymbol{x} \in \mathcal{U}^n} \boldsymbol{c}^T \cdot \boldsymbol{x}$ given the cost vector $\boldsymbol{c} \in \mathcal{F}^n$ respecting side conditions $A\boldsymbol{x} \leq \boldsymbol{b}$ where $A \in \mathcal{F}^{m \times n}, \boldsymbol{b} \in \mathcal{U}^m$. Note that the side conditions $A\boldsymbol{x} \leq \boldsymbol{b}$ form a system of weak linear constraints - its satisfiability is our interest for SMT solving.

2.3.2 Fundamental theorem of linear programming

In the following, we prove a slightly stronger variant of the first part of the *fundamental theorem of linear programming*. A proof for the original theorem is presented for example in [LY84].

Definition 2.3.1 (Tight constraint)**.** *Let c be a weak linear constraint $p \sim b$, i.e. $\sim \in \{=, \leq, \geq\}$. We call $p = b$ the* tightness equality *of c, denoted by $\tilde{c}$.*

Let $\alpha : \{x_1, \dots, x_n\} \to \mathcal{U}$, then c is called tight *at α if $\alpha \models \tilde{c}$.*

Given a set V of weak linear constraints, we define $\tilde{V} := \{\tilde{c} \mid c \in V\}$.

Theorem 2.3.1 (Adaption of the fundamental theorem of linear programming)**.** *Let C be a system of weak linear constraints over $(\mathcal{U}, <)$ in variables X.*

Then C is satisfiable if and only if there exists a maximal linearly independent set $V \subseteq C$ (i.e. $|V| = \mathit{rank}(C)$) such that

$$\exists \alpha : \{x_1, \dots, x_n\} \to \mathcal{U}.\ \alpha \models \tilde{V} \cup C.$$

Before proving this theorem, we present a well known fact about matrices: Let $A \in \mathcal{F}^{m \times n}$ be a matrix. We denote the j-th column vector of A by $\boldsymbol{a_{-,j}} \in \mathcal{F}^{m \times 1}$ and the i-th row vector of A by $\boldsymbol{a_{i,-}} \in \mathcal{F}^{1 \times n}$. Linear independence for column and row vectors is defined analogously to Definition 2.2.2. Inspired by Definition 2.2.3, the *row rank* of A is defined as the size of a maximal linearly independent subset of $\{\boldsymbol{a_{i,-}} \mid i \in [m]\}$. The *column rank* of A is defined analogously.

Theorem 2.3.2 (Row rank equals column rank)**.** *The column rank of a matrix A is equal to its row rank.*

We omit a proof here and refer to [Axl97] instead.

Proof of Theorem 2.3.1. The backward direction is trivial. For the other direction, let $m = |C|$ and assume $C = \{p_i \leq b_i \mid i \in [m]\}$ with $p_i = a_{i,1} \cdot x_1 + \ldots + a_{i,n} \cdot x_n$ for $i \in [m]$ and let $\alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}$ such that $\alpha \models C$. Let $I = \{(p_i \leq b_i) \in C \mid \alpha(p_i) = b_i\}$ be the constraints that are tight at α.

If $\mathit{rank}(I) = \mathit{rank}(C)$, then there exists a maximal linear subset $V \subseteq I$ such that $|V| = \mathit{rank}(C)$ and we are done.

Otherwise, $\mathit{rank}(I) < \mathit{rank}(C)$ and by assumption, $\alpha(p_i) < b_i$ for every constraint $(p_i \leq b_i) \in C \setminus I$ and at least one constraint in $C \setminus I$ is linearly independent from I. In the following, we construct an assignment α' so that such a constraint becomes tight, i.e. $\mathit{rank}(I) < \mathit{rank}(I')$ where I' is defined analogously to I for α'. By iterating this, we obtain an assignment α'' such that $\mathit{rank}(I'') = \mathit{rank}(C)$.

W.l.o.g. I consists of the first k elements of C. The left hand sides of the constraints in C can be written as matrix $A = (a_{i,j})_{i \in [m], j \in [n]}$, where the coefficients of each constraint form a row. Then C can be restated as $A \cdot \boldsymbol{x} \leq \boldsymbol{b}$, thus

$$\boldsymbol{a_{i,-}} \cdot [\alpha(x_1), \ldots, \alpha(x_n)]^T \leq b_i \text{ for all } i \in [m]$$

which is equivalent to

$$\alpha(x_1) \cdot \boldsymbol{a_{-,1}} + \ldots + \alpha(x_n) \cdot \boldsymbol{a_{-,n}} =: \boldsymbol{s} \leq \boldsymbol{b}$$

and by assumption, it holds additionally

$$s_i = b_i \text{ for } i \in [k].$$

We apply Theorem 2.3.2 to the submatrix A_I corresponding to the first k rows of A: From $\mathit{rank}(I) < \mathit{rank}(C) \leq n$ and, by construction, $\mathit{rank}(I)$ equals to the row rank of A_I, we follow that the columns of A_I are linearly dependent. That is, there are $f_1, \ldots, f_n \in \mathcal{F}$ such that $f_i \neq 0$ for at least one i and for

$$f_1 \cdot \boldsymbol{a_{-,1}} + \ldots + f_n \cdot \boldsymbol{a_{-,n}} =: \boldsymbol{t}$$

it holds $t_i = 0$ for $i \in [k]$.

By using Theorem 2.3.2 again on A, from $\mathit{rank}(I) < \mathit{rank}(C)$ follows that $t_i \neq 0$ for at least one $i = k+1, \ldots, m$: Towards a contradiction, assume that all such $t_i = 0$. Then the columns $\{\boldsymbol{a_{-,i}} \mid f_i \neq 0\}$ are linearly dependent and thus $\mathit{rank}(I) \geq \mathit{rank}(C)$, which is a contradiction.

Additionally, we can show that $t_i = 0$ for all $(p_i \leq b_i) \in C \setminus I$ that are linearly dependent on I: Let $p_i = f'_1 \cdot p_1 + \ldots + f'_k \cdot p_k$, then

$$\begin{aligned} t_i = f_1 \cdot a_{i,1} + \ldots + f_n \cdot a_{i,n} &= f_1 \cdot \sum_{j \in [k]} (f'_j \cdot a_{j,1}) + \ldots + f_n \cdot \sum_{j \in [k]} (f'_j \cdot a_{j,n}) \\ &= \sum_{j \in [k]} f'_j \cdot (f_1 \cdot a_{j,1} + \ldots + f_k \cdot a_{j,k}) = 0 \end{aligned}$$

Now, we define

$$\alpha^e(x_i) = \alpha(x_i) + e \cdot f_i$$

and observe that for all $e \in \mathcal{U}$ it holds

$$\alpha^e(x_1) \cdot \boldsymbol{a_{-,1}} + \ldots + \alpha^e(x_n) \cdot \boldsymbol{a_{-,n}} = \boldsymbol{s} + e \cdot \boldsymbol{t} =: \boldsymbol{s^e} \text{ with } s^e_i = b_i \text{ for } i \in [k].$$

Note that for $i = k+1, \ldots, m$ where $t_i \neq 0$, s_i^e depends linearly on e.

Despite $\boldsymbol{s^e} \leq \boldsymbol{b}$ does not necessarily hold, we can increase or decrease e such that $\boldsymbol{s^e} \leq \boldsymbol{b}$ still holds - and we can find such an e such that additionally $s_i^e = b_i$ for one or more $i = k+1, \ldots, m$ where $t_i \neq 0$. Thus, for such an e, $\alpha_e : \alpha'$ is an assignment where I and one or more additional linearly independent constraints are tight. □

2.3.3 Relation to polyhedra

An assignment $\alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}$ can be interpreted as a vector in $\mathcal{U}^n$; analogously, given a linear constraint c or a system C of linear constraints, its solution set can be viewed as $sol(c) \subseteq \mathcal{U}^n$ respectively $sol(C) \subseteq \mathcal{U}^n$.

Given a constraint $p \leq b$, its solution set $sol(p \leq b)$ defines a *half space*; an intersection of half spaces is called a *polyhedron*. Thus, the solution set $sol(C)$ of a system C of weak linear constraints is a such a polyhedron. Conversely, each polyhedron has a defining system of weak linear constraints.

Polyhedra are convex sets. A set $P \subseteq \mathcal{U}^n$ is called *convex* if and only if the line between each two points in P lies in P, that is

$$\boldsymbol{x}, \boldsymbol{y} \in P \implies \lambda \cdot \boldsymbol{x} + (1 - \lambda) \cdot \boldsymbol{y} \in P \text{ for all } \lambda \in [0,1].$$

A point $\boldsymbol{x} \in P$ of a convex set is an *extreme point* if it does not lie on a line between two different points in P, that is, $\boldsymbol{x}$ cannot be written as

$$\boldsymbol{x} = \boldsymbol{y} + (1 - \lambda) \cdot \boldsymbol{z} \text{ for } \boldsymbol{y}, \boldsymbol{z} \in P \setminus \{x\} \text{ and } \lambda \in [0,1].$$

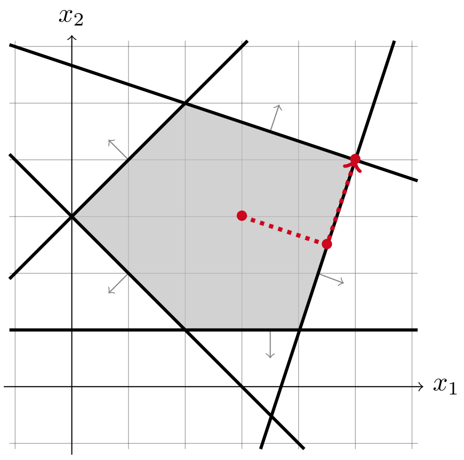


Figure 2.1: Intuitively, in Theorem 2.3.1, we start with a satisfiable assignment, that is, any point in the polyhedron induced by the solution set. We make constraint by constraint tight; in each step, we move the assignment into a direction until a constraint that defines a face of the polyhedron becomes tight while tight constraints remain tight. As the polyhedron is convex, we do not leave the satisfiable region.

The extreme points of a polyhedron are called *vertices* and only exist for polyhedra whose defining system C of weak linear constraints is of full rank $rank(C) = n$. It turns out that in this case, the extreme points are exactly the points induced by the satisfying sets $\tilde{V}$ from Theorem 2.3.1. If $rank(C) < n$, these definitions do not match perfectly, as the satisfying sets $\tilde{V}$ do not induce single points, but lines, planes or higher dimensional subspaces; however, the notion of vertices could be extended analogously. For this thesis, we are content with this intuition. Although not immediately related to this thesis, for more intuition we refer to [Zie12].

2.4 The general Simplex algorithm

The Simplex algorithm is a well known algorithm for linear optimization over the reals and is able to solve *linear programs*. The algorithm was first introduced by Dantzig in 1947 [Dan98]; an evolution of this algorithm called the *dual Simplex method* was introduced by Lemke [Lem54] in 1954. Here, we present a variant of the latter based on the *general Simplex* from [KS16], first introduced in [DDM06], which is an adaption for checking the satisfiability of systems of weak linear constraints.

Normal form We can transform any system of weak linear constraints

$$a_{i,1} \cdot x_1 + \ldots + a_{i,n} \cdot x_n \leq b_i \text{ for } i \in [m]$$

to the form

$$a_{i,1} \cdot x_1 + \ldots + a_{i,n} \cdot x_n = s_i \wedge s_i \leq b_i \text{ for } i \in [m]$$

where the s_i are called *slack variables* in contrast to the *original variables* x_i. Let $E \in \mathcal{F}^{m \times m}$ be the identity matrix, then the system can be written as

$$[A \mid -E] \cdot [x_1, \ldots, x_n, s_1, \ldots, s_m]^T = 0 \wedge \bigwedge_{i \in [m]} s_i \leq b_i.$$

Tableau $[A \mid -E]$ has a special form, which we will introduce now.

A *tableau* is a matrix of the form

$$M = \begin{array}{c} \begin{array}{cccccc} x_1 & \ldots & x_n & s_1 & \ldots & s_m \end{array} \\ \left[\begin{array}{ccc|ccc} a_{1,1} & \cdots & a_{1,n} & f_{1,1} & \cdots & f_{1,m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} & f_{m,1} & \cdots & f_{m,m} \end{array}\right] \end{array} \tag{2.1}$$

together with *bounds* $s_i \leq b_i$ for $i \in [m]$.

Note that $[A \mid -E]$ has full rank as E has full rank. As the algorithm progresses, starting from $[A \mid -E]$, the tableau is transformed retaining a similar form: The columns could always be reordered such that some of them form a negative identity matrix, or equivalently, for every standard vector $\boldsymbol{e}_{-,i} \in \mathcal{F}^{m \times 1}$ with $e_{i,i} = 1$ and $e_{j,i} = 0$ for $j \neq i$, there is a column in M equal to $-\boldsymbol{e}_{-,i}$. For every $i \in [m]$, by $\mathfrak{b}_i$, we refer to the labelling variable of the first column equal to $-\boldsymbol{e}_{-,i}$. Then, the variables $\mathcal{B} = \{\mathfrak{b}_1, \ldots, \mathfrak{b}_m\}$ are called *basic* (or *dependent*) and the remaining variables $\mathcal{N} = \{x_1, \ldots, x_n, s_1, \ldots, s_m\} \setminus \mathcal{B}$ are called *non-basic.*

Intuitively, by this notation, each basic variable $\mathfrak{b}_i \in \mathcal{B}$ is rewritten in terms of non-basic variables: As $\mathfrak{b}_i$ occurs by definition only in the i-th row (with a non-zero coefficient) and all other variables occurring in the i-th row are non-basic, $\mathfrak{b}_i$ depends on the non-basic variables:

$$\mathfrak{b}_i = \sum_{j\in[n],\ x_j\in\mathcal{N}} a_{i,j} \cdot x_j + \sum_{j\in[m],\ s_j\in\mathcal{N}} f_{i,j} \cdot s_j$$

Note that initially, for $[A \mid -E]$, it holds $\mathcal{N} = \{x_1, \dots ,x_n\}$ and $\mathcal{B} = \{s_1, \dots ,s_m\}$.

Assignment The algorithm maintains an assignment $\alpha : \{x_1, \dots , x_n, s_1, \dots , s_m\} \to \mathcal{U}$ such that the following two invariants hold:

$$M \cdot [\alpha(x_1), \dots , \alpha(x_n), \alpha(s_1), \dots , \alpha(s_m)]^T = 0 \tag{2.2}$$

$$\begin{gathered}\alpha(s_i) = b_i \text{ for all slack variables } s_i \in \mathcal{N} \\ \text{and } \alpha(x_i) = 0 \text{ for all original variables } x_i \in \mathcal{N}\ .\end{gathered} \tag{2.3}$$

Note that by this definition, if a slack variable s_i is non-basic ($s_i \in \mathcal{N}$), then this is equivalent to replacing $s_i \leq b_i$ by $s_i = b_i$. Initially, the assignment $\alpha(x_i) = 0$ for $i \in [n]$ and $\alpha(s_i) = 0$ for $i \in [m]$ is chosen; it is easy to see that this does fulfil the invariants.

A tableau is satisfied by α if and only if all bounds on the slack variables are satisfied, i.e. $\alpha(s_i) \leq b_i$ for all i. Note that if α satisfies the above invariants, this property holds trivially for all non-basic slack variables $s_i \in \mathcal{N}$, thus only the basic slack variables might cause a conflict.

Pivot step The idea is to improve the initial tableau to reach such a satisfying assignment; this is done by a sequence of *pivot steps.*

If the given tableau produces a satisfying assignment, then we are already done. Otherwise, there exists a $s_i \in \mathcal{B}$ such that $\alpha(s_i) \leq b_i$ is violated. Thus $\alpha(s_i) > b_i$ and $\alpha(s_i)$ needs to be decreased. Recall that the value of s_i depends on the non-basic variables. Assume that s_i is the k-th basic variable, i.e. $\mathfrak{b}_k = s_i$, then:

$$s_i = \sum_{j\in[n],\ x_j\in\mathcal{N}} a_{k,j} \cdot x_j + \sum_{j\in[m],\ s_j\in\mathcal{N}} f_{k,j} \cdot s_j .$$

Decreasing $\alpha(s_i)$ can be achieved by decreasing or increasing the value of a variable occurring in the summand with a non-zero factor such that its bounds are not violated:

- A slack variable $s_j \in \mathcal{N}$ such that $f_{k,j} > 0$ and thus $\alpha(s_j)$ can be decreased or
- an original variable $x_j \in \mathcal{N}$ such that $a_{k,j} \neq 0$ (depending on the sign of $a_{k,j}$, the value $\alpha(x_j)$ can be increased or decreased as it has no bound).

Such a variable is called *suitable* for pivoting with s_i. If no such variable is found, the problem is unsatisfiable and the algorithm terminates.

Otherwise, we use a suitable non-basic variable $\mathfrak{v}$ to push $\alpha(s_i)$ to its boundary, that is, making s_i a non-basic variable by swapping it with $\mathfrak{v}$ (making $\mathfrak{v}$ a basic variable).

W.l.o.g. let $\mathfrak{v} = x_j$. We then solve the k-th row for x_j and eliminate x_j for all rows $k' \neq k$ using the equality obtained from row k. Formally, we update:

- $a'_{k,l} = -\frac{a_{k,l}}{a_{k,j}}$ for all l respectively $f'_{k,l} = -\frac{f_{k,l}}{a_{k,j}}$ for all l

- $a'_{k',l} = a_{k',l} + a_{k',j} \cdot a'_{k,l}$ for all l and all $k' \neq k$ resp. $f'_{k',l} = f_{k',l} + a_{k',j} \cdot f'_{k,l}$ for all l and all $k' \neq k$

We call the element $a_{k,j}$ *pivot element.*

The thoughtful reader might have recognized the similarity to the *Gaussian elimination* algorithm: A pivot step in the Simplex algorithm corresponds to a forward and backward elimination step.

Termination The procedure is repeated until either unsatisfiability is detected or a satisfying assignment is found. The completeness is guaranteed by Theorem 2.3.1, as it can be reformulated to: Given the normal form of a system C of weak linear constraints, C is satisfiable if and only if there exists a set S of slack variables such that the assignment corresponding to the tableau where $S \subseteq \mathcal{N}$ is satisfying.

In general, it is possible that the same tableau is reached again and thus, the algorithm cycles due to a phenomenon called *degeneracy.* However, when selecting the pivot element according to a fixed ordering for the basic and non-basic variables, this case never occurs. This rule is called *Bland's rule.* It should be mentioned that there are several heuristics for choosing a pivot element among the suitable ones, deviating from *Bland's rule*, but performing better in practice.

Geometric interpretation We can transform the system $C = \{p_i \leq b_i \mid i \in [m]\}$ into an optimization problem $C' = \{p_i \leq b_i + e_i \mid i \in [m]\} \cup \{e_i \geq 0 \mid i \in [m]\}$ where the e_i are fresh variables with the objective function $e := \sum_{i \in [m]} e_i$ to be minimized. Note for a maximal linearly independent subset $V \subseteq C$, its counterpart $V' \subseteq C'$ implies minimal values for the e_i and thus for e as well, denoted by e_V. Assuming that C has full rank and $sol(C)$ does not intersect with an axis, the general Simplex on C minimizes e as follows:

First, a tableau is constructed by a sequence of pivot steps such that $\mathcal{N} = V$ for a maximal linearly independent subset $V \subseteq C$; let α be the current assignment. Observe that the assignment $\alpha[e \mapsto e_V]$ corresponds to a vertex of $sol(C')$ interpreted as polyhedron.

From now on, given any tableau, pivoting with a non-zero $f_{i,j}$ as pivot element, an assignment corresponding to a neighbouring vertex in $sol(C')$ can be obtained. Thus, the Simplex algorithm essentially jumps from vertex to vertex along the edges of the polyhedron induced by $sol(C')$ as illustrated in Figure 2.2.

Note that there is no guarantee that e_V will be strictly decreased after a pivot step - in fact, it can even increase in some cases; however, following Bland's rule and applying the notion of suitable pivot elements, the objective value decreases in "bigger" steps. Moreover, the *local improvements* induced by the suitable pivot elements can be so small such that in theory, the algorithm has exponential running time. However, following them is a good heuristic on practical instances.

2.4.1 Embedding into DPLL(T)

In the DPLL(T) embedding, all constraints occurring in the formula are added to the Simplex tableau, but there is a distinction made between active and non-active bounds: While active bounds on a slack variable are considered for pivoting as described above, non-active bounds are ignored and might be violated in a satisfying tableau.

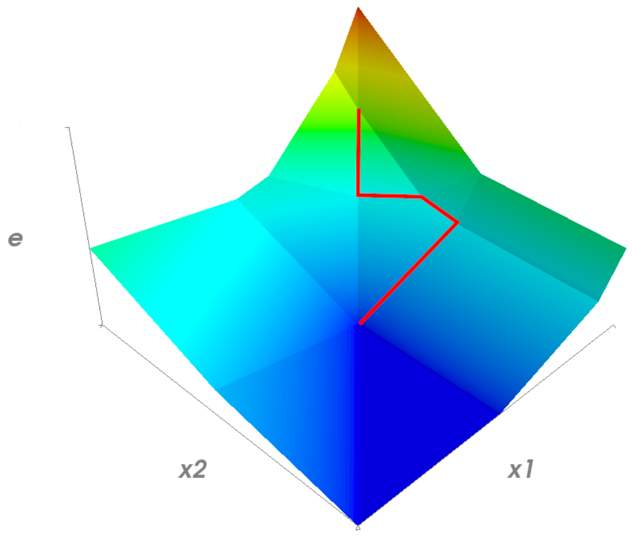


Figure 2.2: The objective value e is plotted on the upward direction dependent on the x_1 and x_2 values. The dark blue area is the satisfying region of the given problem. The red line indicates the on which the Simplex jumps from vertex to vertex. The notion of a suitable pivot element ensures that the algorithm goes into a direction that tends to lead down the hill.

A theory call with input I looks as follows: First, all bounds on slack variables are set active if and only if they correspond to a constraint $c \in I$. Then the pivot steps as described above are performed until either a satisfying assignment is found, or a row i without a suitable pivot variable is found and the infeasible subset

$$\{c \mid c \text{ corresponds to a } s_j \text{ such that } f_{i,j} \neq 0\}$$

is returned.

Furthermore, to be able to handle SMT problems properly, support for the relations $<$, $>$ and $\neq$ needs to be added. This is discussed in Chapter 3.

Chapter 3

Satisfiability of general systems of linear constraints

Theorem 2.3.1 gives a very useful condition to find a solution for weak systems of linear constraints, forming the basis of the general Simplex algorithm. For SMT solving however, we need to be able to solve general systems of linear constraints containing also strict constraints.

First, we will reduce systems containing the relations $<$ and $>$ to the case of weak linear constraints in Section 3.1 by an equisatisfiable transformation. In Section 3.2, we will introduce the notion of *infinitesimals*, giving a nice formalism for dealing with strict inequalities. Finally, in Section 3.3, we will extend this to the relation $\neq$ and apply the obtained results to the original system.

In the following, $(\mathcal{U}[\varepsilon], <)$ is an ordered $\mathcal{F}$-vector space and $X = \{x_1, \ldots, x_n\}$ be a set of $\mathcal{U}[\varepsilon]$-valued variables.

3.1 Strictly-greater and strictly-less constraints

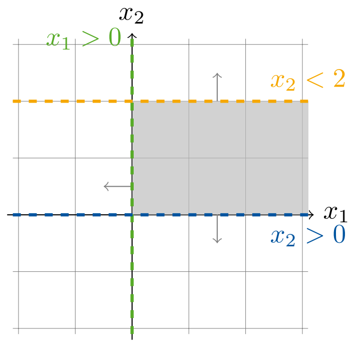
In this section, we assume that constraints with relation $\neq$ do not occur. Furthermore, constraints with relations $>$ and $\geq$ can be transformed to constraints with $<$ respectively $\leq$ by multiplying both sides by -1. Thus, without loss of generality, all relations are one of $<, \leq$ or $=$.

First, we introduce a weakened version of a system of linear constraints:

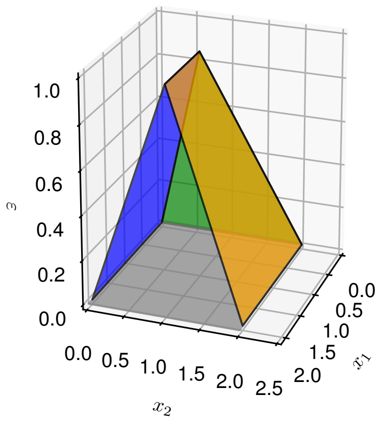
Definition 3.1.1. *Let C be a system of linear constraints over $(\mathcal{U}, <)$ in variables X. We define the system C_w of linear constraints over $(\mathcal{U}, <)$ in variables $X \cup \{\varepsilon\}$ where ε is a fresh variable such that*

$$\begin{aligned} (p \leq b - \varepsilon) \in C_w &\iff p < b \in C \\ (p \sim b) \in C_w &\iff (p \sim b) \in C \text{ where } \sim \in \{\leq, =\} \end{aligned}$$

Example 3.1.1. *Consider the system $C = \{x_1 > 0, x_2 > 0, x_2 < 2\}$ depicted in Figure 3.1a, and its weak version $C_w = \{x_1 \geq \varepsilon, x_2 \geq \varepsilon, x_2 \leq 2 - \varepsilon\}$ depicted in Figure 3.1b.*



(a) The original system C. The satisfiable regions of C are grey.



(b) The transformed system C_w whose satisfiable region has a "tent" shape. The satisfiable regions of C are grey.

Figure 3.1: An illustration of Definition 3.1.1.

It is easy to see that a system C and its weakened version C_w are satisfiability equivalent. We start with the easy direction:

Lemma 3.1.1. *Let C be a system of linear constraints over $(\mathcal{U}, <)$ in variables X. Then for any $\alpha_w : \{x_1, \ldots x_n, \varepsilon\} \to \mathcal{U}$ such that $\alpha_w(\varepsilon) > 0$ and $\alpha_w \models C_w$,*

$$\alpha_w \downharpoonright_{\{x_1,\ldots,x_n\}} \models C.$$

Example 3.1.2. *Continuing the previous example, Figure 3.2 illustrates the application of Lemma 3.1.1.*

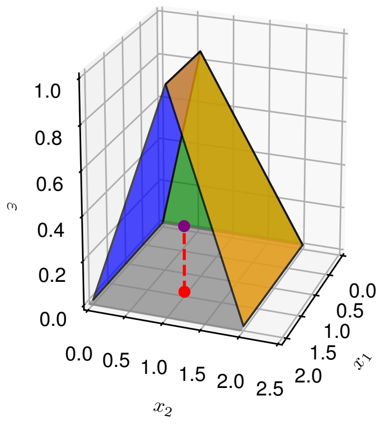


Figure 3.2: Any point that is inside the "tent" of C_w can be turned into a solution of C by projecting out ε.

For the converse direction, we prove stronger statement:

Lemma 3.1.2. *Let C be a system of linear constraints over $(\mathcal{U}, <)$ in variables X. Then for any $\alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}$ such that $\alpha \models C$,*

$$\exists g \in \mathcal{U}.\ (g > 0 \wedge \forall e \in \mathcal{U}.\ (0 < e \leq g \to \alpha[\varepsilon \mapsto e] \models C_w)).$$

Proof. Note that for $(p_i \sim_i b_i) \in C \cap C_w$, $\alpha[\varepsilon \mapsto e] \models p_i \sim_i b_i$ for all $e \in \mathcal{U}$. Thus, only the remaining case $(p_i \leq b_i - \varepsilon) \in C_w$ needs to be considered.

Let $(p_i \leq b_i - \varepsilon) \in C_w$, thus $(p_i < b_i) \in C$. Then by assumption,

$$\alpha(p_i) < b_i \iff 0 < b_i - \alpha(p_i) =: g_i.$$

Now, let $e \in \mathcal{U}$ such that $0 < e \leq g_i$, then

$$\begin{aligned} & b_i - g_i \leq b_i - e \\ \iff & \alpha(p_i) \leq b_i - e \\ \iff & \alpha[\varepsilon \mapsto e] \models p_i \leq b_i - \varepsilon \end{aligned}$$

Note that if $C_w = C$, there are no bounds on ε and hence can be chosen as 1. Thus, for all $e \in \mathcal{U}, 0 < e \leq g := \min(\{(g_i \mid p_i \leq b_i - \varepsilon) \in C_w\} \cup \{1\})$, it holds $\alpha[\varepsilon \mapsto e] \models C_w$. $\square$

Example 3.1.3. *Considering again the familiar example, Figure 3.3 illustrates the application of Lemma 3.1.2.*

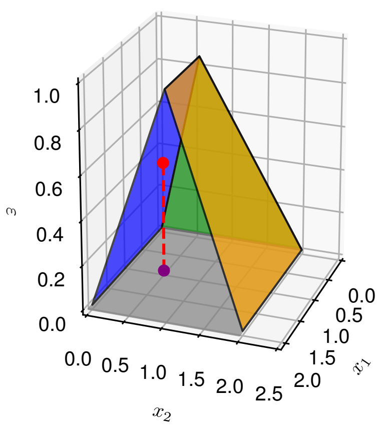


Figure 3.3: Any point satisfying C can be transformed into a solution of $C_w \wedge \varepsilon > 0$ by moving the satisfying point up until it hits the "tent roof".

Now, combining these lemmas with the fundamental theorem of linear programming from Theorem 2.3.1, we obtain an elegant condition for the satisfiability of a system of linear constraints C. Remind that given a system of linear constraints C, P_C is the set of the left hand sides of the constraints in C.

Theorem 3.1.3. *Let C be a system of linear constraints over $(\mathcal{U},<)$ in variables X.*
Then C is satisfiable if and only if there exists a subset $V \subseteq C_w$ with $|V| = rank(C)$ such that $P_V \cup \{\varepsilon\}$ is linearly independent and

$$\exists g \in \mathcal{U}.\ (g > 0 \wedge \forall e \in \mathcal{U}.\ (0 < e \leq g \to \exists \alpha : X \to \mathcal{U}.\ \alpha[\varepsilon \mapsto e] \models \tilde{V} \cup C_w)).$$

Proof.

$$\begin{aligned}
&\exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}.\ \alpha \models C\\
\overset{(1)}{\iff}\ &\exists \alpha : \{x_1, \ldots, x_n, \varepsilon\} \to \mathcal{U}.\ (\alpha \models C_w \wedge \alpha(\varepsilon) > 0)\\
\overset{(2)}{\iff}\ &\exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}.\ \exists g \in \mathcal{U}.\ (g > 0 \wedge \forall e \in \mathcal{U}.\ (0 < e \leq g \to \alpha[\varepsilon \mapsto e] \models C_w))\\
\iff\ &\exists \alpha : X \to \mathcal{U}.\ \exists g \in \mathcal{U}.\ (g > 0 \wedge \forall e \in \mathcal{U}.\ (0 < e \leq g \to \alpha[\varepsilon \mapsto e] \models C_w \cup \{\varepsilon = e\}))\\
\overset{(4)}{\iff}\ &\exists g \in \mathcal{U}.\ (g > 0 \wedge \forall e \in \mathcal{U}.\ (0 < e \leq g \to \exists \text{ max. lin. indep. } V \subseteq C_w \cup \{\varepsilon = e\}.\\
&\exists \alpha : X \to \mathcal{U}.\ \alpha[\varepsilon \mapsto e] \models \tilde{V} \cup C_w \cup \{\varepsilon = e\})).
\end{aligned}$$

where (1) $\Longrightarrow$: Lemma 3.1.2; $\Longleftarrow$: Lemma 3.1.1
(2) $\Longrightarrow$: Lemma 3.1.1, Lemma 3.1.2; $\Longleftarrow$: trivial
(4) Theorem 2.3.1

Now, we prove the equivalence of the last statement above and the statement from the theorem. While the backwards direction is clear, for the forward direction, we need additional arguments how the quantified unknowns can be chosen:

First observe that we can choose V such that $(\varepsilon = e) \in V$ and the remaining constraints in $V \setminus \{\varepsilon = e\}$ can be chosen from C_w.

Additionally, if g is chosen small enough, we can choose V such that $V \setminus \{\varepsilon = e\}$ is the same for all $0 < e \leq g$; this follows from Claim 3.1.3.1 and the fact that there are only finitely many subsets $V \subseteq C_w \cup \{\varepsilon = e\}$. Thus, $V \setminus \{\varepsilon = e\}$ can be chosen before g, which proves the theorem.

It remains to prove the following claim:

Claim 3.1.3.1. *Let $V \subseteq C_w$ with $|V| = rank(C)$ such that $P_V \cup \{\varepsilon\}$ is linearly independent. Let $e,e' \in \mathcal{U}$ such that $0 < e' < e$. Let $\alpha : \{x_1, \ldots, x_n, \varepsilon\} \to \mathcal{U}$ such that $\alpha \models \tilde{V} \cup C_w \cup \{\varepsilon = e\}$. Furthermore, assume there exists no α' such that $\alpha' \models \tilde{V} \cup C_w \cup \{\varepsilon = e'\}$.*
Then for every $e'' \in \mathcal{U}, e'' < e'$ and every assignment α'', $\alpha'' \not\models \tilde{V} \cup C_w \cup \{\varepsilon = e''\}$.

Proof of Claim. Assume $V = \{p_i \leq b_i \mid i = 1, \ldots, k\}$. Note that for any $\beta \models \tilde{V}$, $\beta(p_i) = b_i$ for $i = 1, \ldots, k$; analogously for any $e \in \mathcal{U}$ and $\beta \models (\varepsilon = e)$, $\beta(\varepsilon) = e$ holds.

As $V \cup \{\varepsilon = e\}$ is maximal linearly independent, for all $(p \leq b) \in C_w \setminus (C \cup V)$, there exists $f_1, \ldots, f_k$ and $f_\varepsilon \neq 0$ such that

$$p = f_1 \cdot p_1 + \ldots + f_k \cdot p_k + f_\varepsilon \cdot \varepsilon.$$

Let $(p \leq b) \in C_w \setminus (C \cup V)$. Then for any assignment $\beta \models \tilde{V} \cup \{\varepsilon = e\}$,

$$\beta(p) = f_1 \cdot \underbrace{\beta(p_1)}_{b_1} + \ldots + f_k \cdot \underbrace{\beta(p_k)}_{b_k} + f_\varepsilon \cdot \underbrace{\beta(\varepsilon)}_{e}$$

and $\beta \models (p \leq b)$ if and only if $\beta(p) \leq b$.

Now, let $\beta \models \tilde{V} \cup \{\varepsilon = e\}$ and $\beta' \models \tilde{V} \cup \{\varepsilon = e'\}$. Then

$$\beta'(p) - \beta(p) = f_\varepsilon \cdot (e' - e).$$

As $\alpha \models \tilde{V} \cup \{p \leq b, \varepsilon = e\}$, $\alpha(p) \leq b$. As $\forall \alpha'.\ \alpha' \not\models \tilde{V} \cup C_w \cup \{\varepsilon = e'\}$, we can choose a $p \leq b$ such that $\alpha'(p) > b$ for any $\alpha' \models \tilde{V} \cup \{\varepsilon = e'\}$. Observe that $\alpha'(p) - \alpha(p) > 0$. Furthermore, let $\alpha'' \models \tilde{V} \cup \{\varepsilon = e''\}$. As $e'' < e' < e$, we observe that $\alpha'(p) - \alpha(p) = f_\varepsilon \cdot (e' - e) > 0$ implies that $\alpha''(p) - \alpha'(p) = f_\varepsilon \cdot (e'' - e') > 0$. It follows that

$$\alpha''(p) = \underbrace{\alpha'(p)}_{>b} + \underbrace{\alpha''(p) - \alpha'(p)}_{>0} > b$$

and thus, $\alpha'' \not\models p \leq b$. ■

□

Example 3.1.4. *We consider the system* $C = \{2 \cdot x_1 > 0, \frac{1}{2} \cdot x_1 > -2\}$ *respectively*

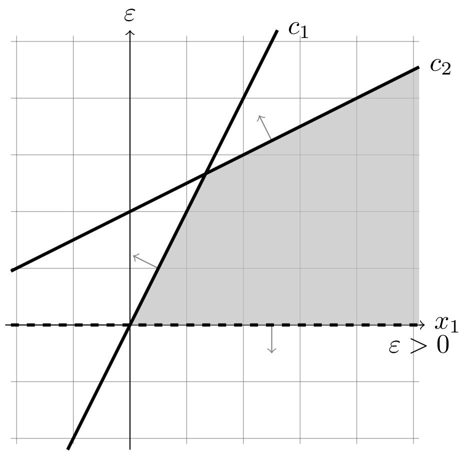
$$C_w = \{\underbrace{2 \cdot x_1 \geq \varepsilon}_{c_1}, \underbrace{\frac{1}{2} \cdot x_1 \geq -2 + \varepsilon}_{c_2}\}.$$

Let us go through the proof of Theorem 3.1.3 step by step. Equivalence (1) states that C *is satisfiable if and only if* $C_w \wedge \varepsilon > 0$ *is satisfiable. The latter is depicted in Figure 3.4a.*

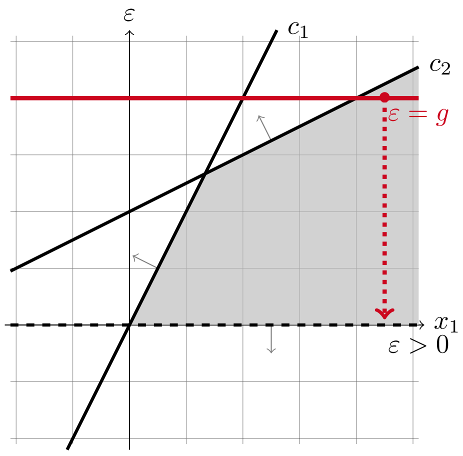
Equivalence (2) and the following one state that there is a value g *for* ε *such that for this and any lower value (arbitrarily close to* 0*), a satisfying point in the solution set can be found, see Figure 3.4b.*

By the application of the fundamental theorem of linear programming in equivalence (4), the problem is reduced to moving along the ridge of the induced polyhedron, Figure 3.4c.

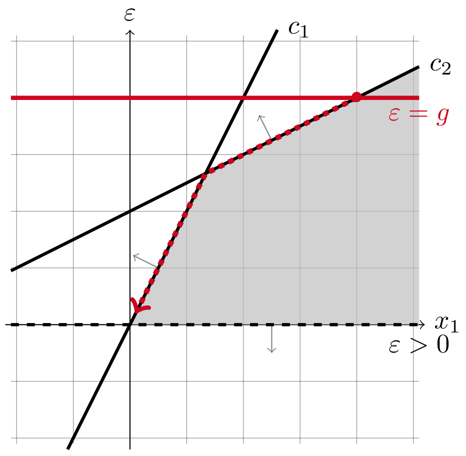
The aim is to find an inducing set of constraints defining the ridge that we are moving along - this set is what is denoted by $V \setminus \{\varepsilon = e\}$ *in the proof. However, in the current situation, these constraints change when lowering the value for* ε*. To address this, we prove that when lowering the value for* ε*, from some point onwards, the set of constraints defining the ridge does not change any more. Intuitively this is clear: Given a ridge that we moved along but is cut off by another ridge at some point, we will never move along this ridge again; this is exactly, what is proven in Claim 3.1.3.1. As there are only finitely many constraints, the set of ridges that we can move along is also finite. Thus, it is always possible to follow a single ridge arbitrarily small to* 0 *while staying in the satisfiable region, see Figure 3.4d.*



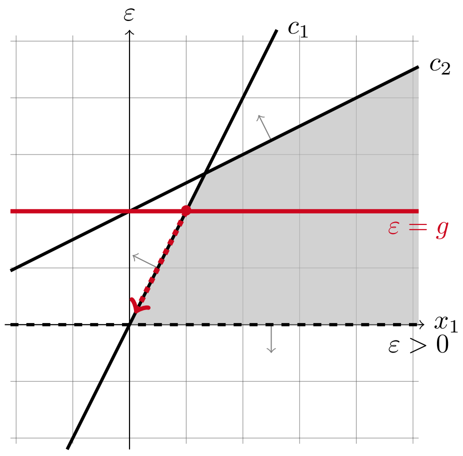
(a) $C_w \wedge \varepsilon > 0$; the satisfying region is grey.



(b) For any value for ε less or equal to g and greater than 0, a satisfying solution can be found.



(c) For any value for ε less or equal to g and greater than 0, a satisfying solution on a ridge of the solution set can be found.



(d) If g is chosen small enough, for any value for ε less or equal to g and greater than 0, a satisfying solution on the same ridge can be found.

Figure 3.4: Illustration of Theorem 3.1.3.

3.2 Infinitesimal arithmetic

Now, we turn Theorem 3.1.3 into a handy theorem by using *infinitesimal arithmetic*, as inspired from the *virtual substitution quantifier elimination method* [LW93, Wei97]:

Definition 3.2.1 (Infinitesimal)**.** *Let* $(\mathcal{U}, <)$ *be an ordered vector space over* $\mathcal{F}$*. We define* ε *as positive infinitesimal, that is*

$$\forall c \in \mathcal{U}.\ (c > 0 \rightarrow 0 < \varepsilon < c).$$

We define the extension of $(\mathcal{U}, <)$ for ε *as the ordered vector space* $(\mathcal{U}[\varepsilon], <)$ *over* $\mathcal{F}$ *such that:*

- $+ : \mathcal{U}[\varepsilon] \times \mathcal{U}[\varepsilon] \to \mathcal{U}[\varepsilon]$ *with* $(d_1 + e_1 \cdot \varepsilon) + (d_2 + e_2 \cdot \varepsilon) \mapsto (d_1 + d_2) + (e_1 + e_2) \cdot \varepsilon$
- $\cdot : \mathcal{F} \times \mathcal{U}[\varepsilon] \to \mathcal{U}[\varepsilon]$ *with* $a \cdot (d + e \cdot \varepsilon) \mapsto a \cdot d + a \cdot e \cdot \varepsilon$
- $<\subseteq \mathcal{U}[\varepsilon] \times \mathcal{U}[\varepsilon]$ *with* $(d_1 + e_1 \cdot \varepsilon) < (d_2 + e_2 \cdot \varepsilon) :\iff d_1 < d_2 \vee (d_1 = d_2 \wedge e_1 < e_2)$

Corollary 3.2.1. *Let* $d + e \cdot \varepsilon \in \mathcal{U}[\varepsilon]$ *and* $<$ *an ordering on* $\mathcal{U}[\varepsilon]$*. Then*

$$\begin{aligned}
0 = d + e \cdot \varepsilon &\iff d = 0 \wedge e = 0 \\
0 < d + e \cdot \varepsilon &\iff d > 0 \vee (d = 0 \wedge e > 0) \\
0 \leq d + e \cdot \varepsilon &\iff d > 0 \vee (d = 0 \wedge e \geq 0) \\
0 \neq d + e \cdot \varepsilon &\iff d \neq 0 \vee e \neq 0
\end{aligned}$$

From now on, $(\mathcal{U}[\varepsilon], <)$ denotes the extension of $(\mathcal{U}, <)$ for ε. Given a system C of linear constraints over $(\mathcal{U}, <)$ in variables $\{x_1, \dots, x_n, \varepsilon\}$, we denote by C^* the system C but view it as a system over $(\mathcal{U}[\varepsilon], <)$ in variables $\{x_1, \dots, x_n\}$.

Theorem 3.2.2. *Let C be a system of linear constraints over $(\mathcal{U}, <)$ in variables X. Then C is satisfiable if and only if there exists a maximal linearly independent subset $V \subseteq C_w^*$ such that*

$$\exists \alpha : \{x_1, \dots, x_n\} \to \mathcal{U}[\varepsilon].\ \alpha \models \tilde{V} \cup C_w^*.$$

For proving this, the following result is needed:

Theorem 3.2.3. *Any set C of linearly independent equations is satisfiable.*

We omit a proof here, as it follows immediately from the *Rouché-Capelli theorem*, which is proven in [SR12].

Proof of Theorem 3.2.2. From Theorem 3.1.3 it follows that

$$\exists \alpha : \{x_1, \dots, x_n\} \to \mathcal{U}.\ \alpha \models C$$

is equivalent to

$\exists V \subseteq C_w.\ (|V| = \mathit{rank}(C) \wedge P_V \cup \{\varepsilon\}$ linearly independent $\wedge$

$\exists g \in \mathcal{U}.\ (g > 0 \wedge \forall e \in \mathcal{U}.\ (0 < e \leq g \to \exists \alpha : \{x_1, \dots, x_n\} \to \mathcal{U}.\ \alpha[\varepsilon \mapsto e] \models \tilde{V} \cup C_w)))$.

As $P_V \cup \{\varepsilon\}$ is linearly independent, from Theorem 3.2.3 follows that $\tilde{V}$ admits a solution for any value of ε. It is easy to see that such a set of solutions in $\{x_1, \dots, x_n, \varepsilon\}$ can be described by an $\alpha : \{x_1, \dots, x_n\} \to \mathcal{U}[\varepsilon]$. Thus, the equation above is equivalent to:

$$\begin{aligned}
&\exists \text{ maximal linearly independent } V \subseteq C_w^*.\ \exists \alpha : \{x_1, \dots, x_n\} \to \mathcal{U}[\varepsilon]. \\
&(\alpha \models \tilde{V} \wedge \exists g \in \mathcal{U}.\ (g > 0 \wedge \forall e \in \mathcal{U}.\ (0 < e \leq g \to \alpha[e/\varepsilon] \models C_w)))
\end{aligned}$$

By analysis of the definition of $<$ on $\mathcal{U}[\varepsilon]$, this is equivalent to

$$\begin{aligned}
&\exists \text{ maximal linearly independent } V \subseteq C_w^*. \\
&\exists \alpha : \{x_1, \dots, x_n\} \to \mathcal{U}[\varepsilon].\ (\alpha \models \tilde{V} \wedge \alpha \models C_w^*).
\end{aligned}$$

The forward direction follows immediately from $\varepsilon < g$ for $g \in \mathcal{U}$. For the backward direction, plugging in α into C_w^* produces bounds on ε, which are, by the semantics of $<$ on $\mathcal{U}[\varepsilon]$, only positive upper bounds or non-positive lower bounds. Thus, g can be chosen as any value smaller than the smallest upper bound (or as any positive value if no such bound exists). □

Before we continue, we will present some examples.

Example 3.2.1. *We consider first the weak system*

$$C = \{\underbrace{x_2 \le 3}_{c_1}, \underbrace{x_1 - x_2 \ge 0}_{c_2}, \underbrace{x_1 + x_2 \le 4}_{c_3}\}$$

illustrated in Figure 3.5. Then we have a single choice for constructing a satisfying vertex, which is induced by $\tilde{c}_2 \wedge \tilde{c}_3 \equiv x_1 = 2 \wedge x_2 = 2$. It is easy to see that c_1 is satisfied under the corresponding assignment.

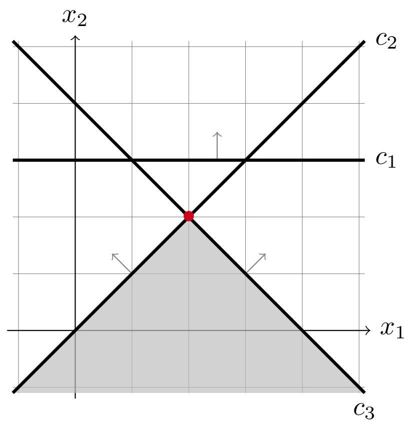


Figure 3.5: A vertex induced by a unique set of two constraints.

Example 3.2.2. *We consider the modified system*

$$C = \{\underbrace{x_2 \le 2}_{c_1}, \underbrace{x_1 - x_2 \ge 0}_{c_2}, \underbrace{x_1 + x_2 \le 4}_{c_3}\}$$

as illustrated in Figure 3.6. Now, we have multiple choices to construct a satisfying vertex, as all constraints are tight under $[x_1 \mapsto 2, x_2 \mapsto 2]$. Thus, for any i,j,k $\in$ {1,2,3} such that i,j,k are pairwise not equal, $\tilde{c}_i \wedge \tilde{c}_j \equiv x_1 = 2 \wedge x_2 = 2$.

So far, nothing exciting happened. Now, we examine how strict bounds are handled:

Example 3.2.3. *We consider the system $C = \{x_2 < 2, x_1 - x_2 \ge 0, x_1 + x_2 \le 4\}$ and*

$$C_w = \{\underbrace{x_2 \le 2 - \varepsilon}_{c_1}, \underbrace{x_1 - x_2 \ge 0}_{c_2}, \underbrace{x_1 + x_2 \le 4}_{c_3}\}$$

depicted in Figure 3.7. In the figure, the dashed line represents the ε part of c_1. In the plot, the red dots correspond to the vertices $\tilde{c}_1 \wedge \tilde{c}_2 \equiv x_1 = 2 - \varepsilon \wedge x_2 = 2 - \varepsilon$ respectively $\tilde{c}_1 \wedge \tilde{c}_3 \equiv x_1 = 2 + \varepsilon \wedge x_2 = 2 - \varepsilon$. It is a crucial observation that there is a difference between vertices over $(\mathcal{U}, <)$ and $(\mathcal{U}[\varepsilon], <)$; vertices over the latter field are not just vertices over the first plus some infinitesimal part, but increase the number of vertices and thus the complexity of the problem.

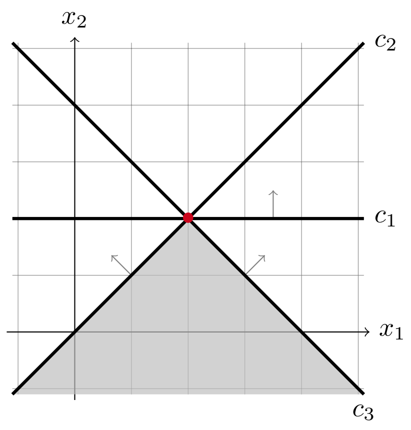


Figure 3.6: An “overdetermined” vertex which is induced by multiple constraint sets.

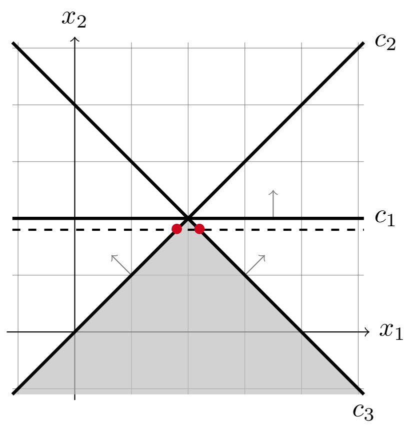


Figure 3.7: Vertices with infinitesimal part.

Example 3.2.4. *We consider the system* $C = \{x_2 < 2, x_1 - x_2 > 0, x_1 + x_2 < 4\}$ *and*

$$C_w = \{\underbrace{x_2 \leq 2 - \varepsilon}_{c_1}, \underbrace{x_1 - x_2 \geq \varepsilon}_{c_2}, \underbrace{x_1 + x_2 \leq 4 - \varepsilon}_{c_3}\}$$

depicted in Figure 3.8. Note that in this case, all constraints are tight under the choice $x_1 = 2 \wedge x_2 = 2 - \varepsilon$*; thus, any combination might be used as the set inducing the vertex.*

However, this is in some sense fragile: When replacing c_1 *by*

$$(2 \cdot x_2 < 4)_w = 2 \cdot x_2 \leq 4 - \varepsilon,$$

then c_1 *is not tight under the satisfying assignment of* $\tilde{c}_2 \wedge \tilde{c}_3 \equiv x_1 = 2 \wedge x_2 = 2 - \varepsilon$*. Moreover, when replacing* c_1 *by*

$$(\frac{1}{2} \cdot x_2 < 1)_w = \frac{1}{2} \cdot x_2 \leq 1 - \varepsilon,$$

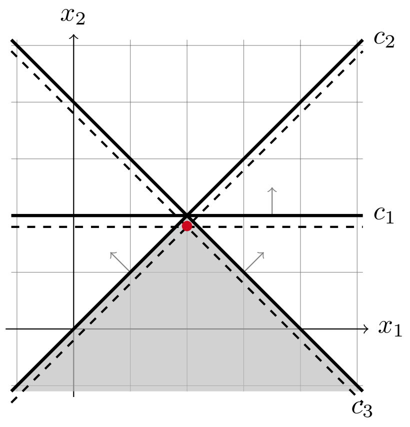


Figure 3.8: "Fragile" vertices with infinitesimal part.

c_1 is even conflicting under $\tilde{c}_2 \wedge \tilde{c}_3 \equiv x_1 = 2 \wedge x_2 = 2 - \varepsilon$; however, $\tilde{c}_1 \wedge \tilde{c}_2$ or $\tilde{c}_1 \wedge \tilde{c}_3$ can be chosen to induce a satisfying vertex instead. Despite the fact that the underlying original systems are all equivalent (as we only multiplied a constraint by a constant factor), the set of constraints inducing a satisfying vertex may vary.

As seen, our extension of the notion of a vertex could be counter-intuitive in some cases; nevertheless, Theorem 3.2.2 guarantees the existence of a satisfying vertex w.r.t. infinitesimal arithmetic if and only if the original system is satisfiable.

3.3 Not-equal constraints

From now on, we consider systems of linear constraints with arbitrary relation symbols.

Lemma 3.3.1. *Let C be a system of linear constraints over any ordered vector space and α be an assignment. Then*

$$\alpha \models C \cup \{p \neq b\} \iff \alpha \models C \cup \{p < b\} \vee \alpha \models C \cup \{p > b\}.$$

Lemma 3.3.1 allows various variants of handling not-equal-constraints: The simplest method is to check the satisfiability of two systems instead of one, as suggested immediately by the theorem. However, increasing the number of not-equal-constraints, the number of systems to be checked grows exponentially. Our aim is to avoid those splits as far as possible by deferring them to the selection of the set V of tight constraints.

Similarly, we defer replacing a strict constraint (with relation $<$ or $>$) in C by its weak correspondence in C_w to the selection of V. Thus, we extend Definition 2.3.1 for strict inequalities by introducing the notion of a *vertex candidate* where the name is inspired by the geometrical interpretation:

Definition 3.3.1 (Vertex candidate)**.** *Let C be a system of linear constraints over*

$(\mathcal{U}, <)$ and $V \subseteq C$ be linearly independent. Then $\tilde{V}$ is defined such that

$$p \sim b \in V \iff \begin{cases} p = b \in \tilde{V} & \textit{if} \sim \in \{=, \leq, \geq\} \\ p = b - \varepsilon \in \tilde{V} & \textit{if} \sim \in \{<\} \\ p = b + \varepsilon \in \tilde{V} & \textit{if} \sim \in \{>\} \\ p = b + \varepsilon \in \tilde{V} \textit{ xor } p = b - \varepsilon \in \tilde{V} & \textit{if} \sim \in \{\neq\} \end{cases}$$

$\tilde{V}$ is called a partial vertex candidate *of C. If V is maximal with this property, then $\tilde{V}$ is called a* vertex candidate*.*

Let $c \in V$, then its corresponding constraint $\tilde{c} \in \tilde{V}$ is called the tightness equality *of c in $\tilde{V}$.*

Lemma 3.3.2. *Let C be a system of linear constraints over $(\mathcal{U}, <)$ in variables X.*

Then if C is satisfiable, then there exists a vertex candidate $\tilde{V}$ of C^* such that

$$\exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ \alpha \models \tilde{V} \cup (C^* \setminus V).$$

Proof.

(a) First assume C does not contain constraints with relation $\neq$. Then the statement follows immediately from Theorem 3.2.2 and the observation

$$\forall \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ (\alpha \models p \leq b - \varepsilon \to \alpha \models p < b). \tag{3.1}$$

(b) Now, we allow constraints with relation $\neq$: Let *Conv*(C) be the set of all systems containing the same constraints as C, but each $\neq$ is replaced by either $<$ or $>$ respectively. Thus, for every $\neq$ occurring in C, the size of *Conv*(C) is doubled. In the following, we abbreviate *vertex candidate* with *v.c.*

$$\begin{aligned} & \exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ \alpha \models C \\ \overset{(1)}{\iff} & \bigvee_{C' \in Conv(C)} \exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ \alpha \models C' \\ \overset{(2)}{\implies} & \bigvee_{C' \in Conv(C)} \exists \text{ v.c. } \tilde{V} \text{ of } C'.\ \exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ \alpha \models \tilde{V} \cup (C')^* \\ \implies & \exists \text{ v.c. } \tilde{V} \text{ of } C.\ \exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ \bigvee_{C' \in Conv(C)} \alpha \models \tilde{V} \cup (C')^* \\ \overset{(4)}{\iff} & \exists \text{ vertex candidate } \tilde{V} \text{ of } C.\ \exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ \alpha \models \tilde{V} \cup C^* \end{aligned}$$

where	(1), (4)	Lemma 3.3.1
	(2)	Claim (a)

□

Lemma 3.3.3. *Let C_1 and C_2 be systems of linear constraints over $(\mathcal{U}, <)$ in variables X. Then $((C_1)_w \cup C_2)^*$ is a corresponding system of linear constraints over $(\mathcal{U}[\varepsilon],)$ where the constraints of C_1 are "weakened" and the constraints in C_2 remain untouched but are interpreted differently. Assume there exists an $\alpha^* : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon]$ such that $\alpha^* \models ((C_1)_w \cup C_2)^*$.*

Then there is an $\alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}$ such that $\alpha \models C_1 \cup C_2$.

Proof. The idea is that, by the semantics of $<$ on $\mathcal{U}[\varepsilon]$, each $p \sim b \in ((C_1)_w \cup C_2)^*$ simplifies under α^* in $(\mathcal{U}, <)$ (where ε is treated as regular variable) to one of the following constraints (with coefficients in $\mathcal{U}$):

- $0 = 0$ (trivially true),
- $g < \varepsilon$ or $g \leq \varepsilon$ with $g \in \mathcal{U}$ such that $g < 0$;
- $\varepsilon < g$, $\varepsilon \leq g$ with $g \in \mathcal{U}$ such that $g > 0$,
- $\varepsilon \neq g$ with $g \in \mathcal{U}$.

Thus, there are only not-equal and strict and weak upper bounds on the value of ε. By choosing an $e \in \mathcal{U}$ being strictly smaller than all these bounds, we obtain $\alpha^*[e/\varepsilon] \models C_1 \cup C_2$. □

Summarizing Lemma 3.3.2 and Lemma 3.3.3 leads to:

Corollary 3.3.4. *Let C be a system of linear constraints over $(\mathcal{U}, <)$ in variables X. Then C is satisfiable if and only if there exists a vertex candidate $\tilde{V}$ of C^* such that*

$$\exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon].\ \alpha \models \tilde{V} \cup C^*.$$

Example 3.3.1. *Consider Example 3.2.4 again, which shows that by scaling a single constraint of a system C, the set of satisfying vertex candidates of C_w varies. However, note that by application of vertex candidates to the original constraints, this can be circumvented:*

We consider the system $C = \{\underbrace{x_2 < 2}_{c_1}, \underbrace{x_1 - x_2 > 0}_{c_2}, \underbrace{x_1 + x_2 < 4}_{c_3}\}$. Then still, $\tilde{c}_2 \wedge \tilde{c}_3 \equiv x_1 = 2 \wedge x_2 = 2 - \varepsilon$. However,

$$[x_1 \mapsto 2, x_2 \mapsto 2 - \varepsilon] \models h \cdot c_1 \text{ for any } h \in \mathcal{F}$$

and hence, by the notion of a vertex candidate and Corollary 3.3.4, vertex candidates are independent from the scaling of the constraints.

Note that Theorem 3.2.2 and Corollary 3.3.4 can be applied to the general Simplex algorithm to add support for strict inequalities.

3.4 Alternative approaches

3.4.1 Transformation to a maximization problem

Lemma 3.1.1 and Lemma 3.1.2 already yield an obvious transformation of a system C of linear constraints over $(\mathcal{U}, <)$ in variables $\{x_1, \ldots, x_n\}$ to the linear program

$$\max_{\alpha:\{x_1,\ldots,x_n,\varepsilon\} \to \mathcal{U}} \varepsilon \text{ subject to } C_w$$

C is satisfiable if and only if there exists a solution for C_w with a positive value for ε, that is, the outcome of the linear program is positive.

3.4.2 Transformation to system in a single variable ε

We defined a specific semantic for ε to establish a theorem speaking about vertex candidates for general systems of linear constraints analogously to the case with weak constraints only. To do so, we restricted ourselves to vertex candidates satisfying the system for an arbitrarily small positive value for ε; however, there are further vertex candidates satisfying the system for a positive (but not necessarily arbitrarily small) value for ε. Thus, we could reformulate Theorem 3.1.3 to

Theorem 3.4.1. *Let C be a system of linear constraints over $(\mathcal{U}, <)$ in variables X. Then C is satisfiable if and only if there exists a linearly independent $V \subseteq C_w$ with $|V| = \mathit{rank}(C_w) - 1$ such that*

$$\exists e \in \mathcal{U}.\ (e > 0 \wedge \exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}.\ \alpha[\varepsilon \mapsto e] \models \tilde{V} \cup C_w).$$

Following this path, we end up with something analogous to Corollary 3.3.4: A system C of linear constraints over $(\mathcal{U}, <)$ in variables X is satisfiable if and only if there exists a vertex candidate $\tilde{V}$ of C such that

$$\exists e \in \mathcal{U}.\ (e > 0 \wedge \exists \alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}.\ \alpha[\varepsilon \mapsto e] \models \tilde{V} \cup C).$$

Let $C_{\tilde{V}}$ be the system after simplifying $\tilde{V} \cup C$ via Gaussian variable elimination. Then, the only variable in $C_{\tilde{V}}$ is ε, thus the above is equivalent to

$$\exists e \in \mathcal{U}.\ (e > 0 \wedge [\varepsilon \mapsto e] \models C_{\tilde{V}})$$

Solving $C' = C_{\tilde{V}} \cup \{\varepsilon > 0\}$ for ε is straight forward: The system C' of linear constraints in one variable ε is unsatisfiable if and only if one of the following is true:

- There exists $\varepsilon \sim_u e_u, \varepsilon \sim_l e_l \in C'$ for $e_u, e_l \in \mathcal{U}$ such that $\sim_u \in \{<, \leq, =\}$ and $\sim_l \in \{>, \geq, =\}$ and $e_u < e_l$,
- there exists $\varepsilon \sim_u e, \varepsilon \sim_l e \in C'$ for an $e \in \mathcal{U}$ such that $\sim_u \in \{<, \leq, =\}$ and $\sim_l \in \{>, \geq, =\}$ and at least one of $\{\sim_u, \sim_l\}$ is strict,
- there exists $\varepsilon \neq e, \varepsilon \leq e, \varepsilon \geq e \in C'$ for an $e \in \mathcal{U}$ or
- there exists $\varepsilon \neq e, \varepsilon = e \in C'$ for an $e \in \mathcal{U}$.

This is essentially the result of the application of the *Fourier-Motzkin variable elimination method* [Fou27, Mot36]) on C'.

Originally, this method allows the elimination of a single variable $x_i \in X$ in a system C of linear constraints over $(\mathcal{U}, <)$ in variables X containing only relations $\leq$ and $\geq$. Then, all constraints containing x_i can be reformulated to be a weak lower or upper bound on x_i; let m^l and m^u be the number of lower respectively upper bounds on x_i and let $C^l = \{\sum_{j \in [n] \setminus \{i\}} a^l_{k,i} \cdot x_j - b^l_k \leq x_i \mid k \in [m^l]\}$ be the set of lower bounds and $C^u = \{x_i \leq \sum_{j \in [n] \setminus \{i\}} a^u_{k,i} \cdot x_j - b^u_k \mid k \in [m^u]\}$ be the set of upper bounds on x_i. Then the sentence $\exists x_i.\ C$ is equivalent to

$$\bigwedge_{k^l \in [m^l], k^u \in [m^u]} \sum_{j \in [n] \setminus \{i\}} a^l_{k^l,i} \cdot x_j - b^l_{k^l} \leq \sum_{j \in [n] \setminus \{i\}} a^u_{k^l,i} \cdot x_j - b^u_{k^l}.$$

Extending this for equalities and strict relations is straight forward.

Chapter 4

Solving linear real arithmetic

In this section, we first elaborate the weaknesses of the classical embedding of the Simplex algorithm in DPLL(T). Then we present an adaption of the Simplex algorithm for checking the satisfiability of a vertex candidate of a system of linear constraints. Finally, we present a novel embedding into DPLL(T).

4.1 Motivation

The obvious approach based on Corollary 3.3.4 is to enumerate all vertex candidates $\tilde{V}$ of a system C of linear constraints and check whether $C \cup \tilde{V}$ is satisfiable. It is easy to see that this approach as in Algorithm 3 is sound and complete.

Algorithm 3: Sketch of the algorithm

```
1 function Check($C$)
      Input: A system of linear constraints
      Output: SAT or UNSAT
2     foreach vertex candidate $\tilde{V}$ of $C$ do
3         compute $sol(\tilde{V} \cup (C \setminus V))$
4         if $sol(\tilde{V} \cup (C \setminus V)) \neq \emptyset$ then
5             return $SAT$
6     return $UNSAT$
```

4.1.1 Weaknesses of the general Simplex algorithm

As already mentioned, the Simplex method checks vertex candidates as in Algorithm 3 with a smart heuristic.

The general Simplex method (with support for strict inequalities) is implemented in SMT solvers such as `Z3` [DDM06] and `SMT-RAT` [CKJ+15] as theory solver for solving sets of linear real arithmetic constraints and performs well in practise. However, on benchmarks with complex combinatorial respectively Boolean structure, the running time increases steeply. The intuition is that constraints are added and removed from the theory solver frequently, causing two issues:

- In the general Simplex algorithm, all constraints are maintained in the tableau - removing or adding constraints only disables or enables the corresponding bounds on the slack variables. That means that all calculations during pivot steps are applied to all constraints despite their relevance to the Boolean model. But especially this is often the case for problems with complex Boolean structure.

  Thus, it might be desirable to remove constraints from the tableau completely and add them later when they become relevant again.

- The Simplex algorithm is a local search algorithm as it chooses a pivot step based on the current tableau. Thus, it does not need to represent "dead end" search paths explicitly - in fact, the progress of the search is purely heuristic, as no improvement is guaranteed. However, as progress is stored implicitly, improvement steps are relatively cheap and a wisely chosen heuristic makes the procedure highly efficient on conjunctions.

  But if constraints are added and removed often, as in case of complex Boolean structure, some of the implicitly stored progress is lost and those "dead end" search paths are explored more often.

  Thus, it might be desirable to store the progress made by the Simplex algorithm explicitly to avoid re-exploring known dead ends.

4.1.2 Interleaving the Simplex method with the SAT solver

In the following, we develop a procedure that addresses these problems. The idea is to keep information across theory calls by moving the selection of the vertex candidate $\tilde{V}$ of the system C induced by the currently asserted theory constraints from the theory to the SAT solver.

The central change is that now, the theory solver does not only receive a set of theory constraints C whose satisfiability is to be checked, but along with C also a (possibly partial) vertex candidate $\tilde{V}$ of C. The task is now to check whether C is satisfied at $\tilde{V}$, that is, whether $(C \setminus V) \cup \tilde{V}$ is satisfiable. This corresponds to constructing a Simplex tableau of the systems C where the slack variable corresponding to the constraints in V are made non-basic. In Section 4.2, we will develop an algorithm to achieve this, based on the Gaussian elimination procedure.

Subsequently, also the role of the theory lemmas changes: While in the usual Simplex embedding, unsatisfiable subsets of the currently asserted constraints are returned, in the new approach a conflict is caused by a constraint that is implied to be *false* under the selected (partial) vertex candidate. In Section 4.3, we will examine how the reasons for such a conflict are propagated to the SAT solver by theory lemmas and which additional knowledge can be learned from a conflict. Moreover, as these conflicts depend usually only on subsets of C and $\tilde{V}$, the procedure is modified to admit incremental solving by recomputing only the changes of the input compared to the previous theory call.

A further prerequisite is that the SAT solver is able to reason about vertex candidates. It is responsible to enumerate all possible vertex candidates to be conform with Corollary 3.3.4 while cutting search paths that are already known to be unsatisfiable. The latter involves the ability to generalize from the lemmas learned from the theory solver in terms of conflict resolution and Boolean propagations. The obvious approach is to encode the semantics of a vertex candidate (or an over-approximation of them) eagerly in a Boolean formula and append it to the input formula; this and an alternative approach is introduced in Section 4.4.

Albeit not immediately relevant for the understanding of this thesis, it is worth noting that it is not a new idea to shift (parts of) mathematical reasoning to a SAT solver. Examples for solving mathematical problems by encoding them as SAT formulas are given in [HKM16, Heu09]; there exist also approaches combining SAT encodings with a specialized solver, as [ZBH$^+$17, BKHG18].

Although this method clearly introduces some overhead by explicitly encoding the excluded vertex candidates, the method might have two advantages: First, the theory calls get smaller and the theory and Boolean reasoning are more interleaved. Furthermore, by minimizing the generated lemmas, a larger portion of the search space can be excluded than in one pivot step of the Simplex method.

4.2 Checking the satisfiability at a vertex candidate

In the following, $(\mathcal{U}[\varepsilon], <)$ is an ordered $\mathcal{F}$-vector space and $X = \{x_1, \ldots, x_n\}$ be a set of $\mathcal{U}[\varepsilon]$-valued variables. We assume all systems of linear constraints to be over $(\mathcal{U}[\varepsilon], <)$ in variables X. Moreover, m refers to the number of constraints in the given system of linear constraints from the context.

The goal is to check the consistency of $C \cup \tilde{V}$ given a system C of linear constraint and a vertex candidate $\tilde{V}$ of C. To do so, we first introduce a convenient notation for a system of linear constraints, reminiscent of the Simplex tableau and the matrix as in the Gaussian elimination:

Definition 4.2.1 (Extended tableau)**.** *Let $a_{i,j}, f_{i,j} \in \mathcal{F}$ coefficients, $b_i \in \mathcal{U}[\varepsilon]$, $\sim_i \in \{=, <, >, \leq, \geq, \neq\}$ relation symbols (w.r.t. the ordering $<$ on $\mathcal{U}[\varepsilon]$) and p_j linear $\mathcal{F}$-combinations of $x_1, \ldots, x_n$ for $i \in [n], j \in [m]$, then the notation*

$$\begin{array}{c}
\begin{array}{ccccc|ccc}
x_1 & \dots & x_n & & & p_1 & \dots & p_m \\
\end{array}\\
\left[\begin{array}{ccccc|ccc}
a_{1,1} & \dots & a_{1,n} & \sim_1 & b_1 & f_{1,1} & \dots & f_{1,m} \\
\vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
a_{m,1} & \dots & a_{m,n} & \sim_m & b_m & f_{m,1} & \dots & f_{m,m}
\end{array}\right]
\end{array}$$

is called an (extended) tableau over $(\mathcal{U}[\varepsilon], <)$ in X and $\{p_1, \ldots, p_m\}$, *denoted as $M = (A, \boldsymbol{\sim}, \boldsymbol{b}, F)$. The* linear constraint system associated with M *is defined as $C_M = \{a_{i,1} \cdot x_1 + \ldots + a_{i,n} \cdot x_n \sim b_i \mid i \in [m]\}$. Thus, we can apply the same semantics to M and its rows as to C_M and its constraints; in particular $sol(M) := sol(C_M)$.*

Conversely, let $C = \{c_1, \ldots, c_m\}$ be a system of linear constraints over $(\mathcal{U}[\varepsilon], <)$ in variables X with $c_i = (p_i \sim_i b_i)$, $p_i = a_{i,1} \cdot x_1 + \ldots + a_{i,n} \cdot x_n$, $a_{i,j} \in \mathcal{F}$, $b_i \in \mathcal{U}[\varepsilon]$ and $\sim_i \in \{=, <, >, \leq, \geq, \neq\}$. Additionally, let $F = -E$ be the negative independent matrix. Then $M_C = (A, \boldsymbol{\sim}, \boldsymbol{b}, F)$ denotes the tableau over $(\mathcal{U}[\varepsilon], <)$ in X and $\{p_1, \ldots, p_m\}$:

$$\begin{array}{c}
\begin{array}{cccccc|ccccc}
x_1 & \dots & x_n & & & & p_1 & \dots & p_i & \dots & p_m
\end{array}\\
\left[\begin{array}{ccccc|ccccc}
a_{1,1} & \dots & a_{1,n} & \sim_1 & b_1 & -1 & & 0 & \dots & 0\\
\vdots & \ddots & \vdots & \vdots & \vdots & & \ddots & & \ddots & \vdots\\
a_{i,1} & \dots & a_{i,n} & \sim_i & b_i & 0 & & -1 & & 0\\
\vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & & \ddots & \\
a_{m,1} & \dots & a_{m,n} & \sim_m & b_m & 0 & \dots & 0 & & -1
\end{array}\right]
\end{array}$$

In the following, we simplify notation when referring to a tableau M; depending on the context, $A, \boldsymbol{\sim}, \boldsymbol{b}, F$ will refer to the entries of M.

Note that the p_i correspond to the slack variables in the Simplex algorithm; here, we refer to the p_i as linear $\mathcal{F}$-combinations of variables directly instead of introducing slack variables for them, because we forego maintaining an explicit assignment but treat them symbolically as in the Gaussian elimination procedure. Still, we maintain a similar invariant as in the Simplex algorithm:

Lemma 4.2.1. *Given a system C of linear constraints, for M_C it holds that*

$$\sum_{j\in[n]} a_{i,j}\cdot x_j + \sum_{j\in[m]} f_{i,j}\cdot p_j = 0 \quad \forall i = 1,\dots,m. \tag{4.1}$$

Proof. Let $i \in [m]$. By definition, $\boldsymbol{f_{i,-}}$ is the negative of the i-th standard vector, and thus Equation (4.1) is equivalent to $\sum_{j\in[n]} a_{i,j}\cdot x_j - p_j = 0$, which is the assumption $p_i = \sum_{j\in[n]} a_{i,j}\cdot x_j$. □

Theorem 4.2.2 (Row operation)**.** *Let M be a matrix such that Equation* (4.1) *holds and let $i, j \in [m], i \neq j$ be rows of M such that $\sim_i \in \{=\}$ and $g \in \mathcal{F}$. Let M' be the matrix after replacing $\boldsymbol{a_{j,-}}$ by $\boldsymbol{a'_{j,-}} := \boldsymbol{a_{j,-}} + g\cdot \boldsymbol{a_{i,-}}$, b_j by $b'_j := b_j + g\cdot b_i$ and $\boldsymbol{f_{j,-}}$ by $\boldsymbol{f'_{j,-}} := \boldsymbol{f_{j,-}} + g\cdot \boldsymbol{f_{i,-}}$. Then*

(i) Equation (4.1) *still holds for M',*

(ii) sol(M) = sol(M') and

(iii) rank(M) = rank(M').

We call such an operation row operation. *For matrices M' that emerged from a matrix M by a sequence of row operations, we write $M \rightsquigarrow M'$.*

Proof.

(i) Let $i \in [m]$ and $\alpha : X \to \mathcal{U}[\varepsilon]$, then

$$\begin{aligned}
&\alpha \models \boldsymbol{a_{i,-}x} + \boldsymbol{f_{i,-}p} = 0\\
\iff &\alpha \models g\cdot \boldsymbol{a_{i,-}x} + g\cdot \boldsymbol{f_{i,-}p} = 0 \iff \alpha \models \boldsymbol{a'_{i,-}x} + \boldsymbol{f'_{i,-}p} = 0.
\end{aligned}$$

(ii) As $\alpha \models M \iff \forall i \in [m].\ \alpha \models \boldsymbol{a_{i,-}}\cdot \boldsymbol{x}$, it is sufficient to prove

$$sol(\{\boldsymbol{a_{i,-}x} \sim_i b_i, \boldsymbol{a_{j,-}x} \sim_j b_j\}) = sol(\{\boldsymbol{a_{i,-}x} \sim_i b_i, \boldsymbol{a'_{j,-}x} \sim_j b_j\}).$$

Let $\alpha \models \boldsymbol{a_{i,-}x} \sim_i b_i$, then

$$\begin{aligned}
&\alpha \models \boldsymbol{a_{j,-}x} \sim_j b_j\\
\iff &\alpha \models \boldsymbol{a_{j,-}x} + g \cdot b_i \sim_j b_j + g \cdot b_i\\
\iff &\alpha \models \boldsymbol{a_{j,-}x} + g \cdot (\boldsymbol{a_{i,-}x}) \sim_j b_j + g \cdot b_i\\
\iff &\alpha \models \boldsymbol{a'_{j,-}x} \sim_j b_j.
\end{aligned}$$

(iii) Observe that the new row $\boldsymbol{a'_{j,-}}$ is a linear combination of $\boldsymbol{a_{i,-}}$ and $\boldsymbol{a_{j,-}}$, hence $\mathit{rank}(M') \leq \mathit{rank}(M)$. Furthermore, note that a row operation is reversible, that is $M' \rightsquigarrow M$, and thus, $\mathit{rank}(M) \leq \mathit{rank}(M')$.

□

Definition 4.2.2. *Let M be a matrix and $i \in [m]$ a row index. The i-th row is called* redundant in M *if and only if* $\boldsymbol{a_{i,-}} = \boldsymbol{0}$.

Note that a redundant row i can be evaluated using Corollary 3.2.1. If the result is contradictory, we call row i *conflicting* and *satisfied* otherwise.

Remark 4.2.3. *If any row in M is conflicting, then $\mathit{sol}(M) = \emptyset$.*

Obviously, the converse direction does not hold for all systems, but it holds for a class of matrices:

Given a system of linear constraints C and a vertex candidate $\tilde{V}$ of C, the idea is to solve $M_{(C \setminus V) \cup \tilde{V}}$ by applying Gaussian elimination on V and applying all pivot steps on the remaining constraints $C \setminus V$. I.e. for the case $\mathit{rank}(C) = n$, assume w.l.o.g. $V = \{c_1, \dots, c_n\}$, the system $M_{(C \setminus V) \cup \tilde{V}}$ is transformed by row operations (and swapping rows) to the following form, which is known as *row echelon form*:

$$\begin{array}{c}
\begin{array}{ccccc|cccccc}
x_1 & \cdots & x_n & & & p_1 & \cdots & p_n & p_{n+1} & \cdots & p_m
\end{array}\\
\left[\begin{array}{ccccc|cccccc}
a'_{1,1} & \cdots & a'_{1,n} & = & b'_1 & -1 & \cdots & 0 & 0 & \cdots & 0\\
\vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots\\
0 & \cdots & a'_{n,n} & = & b'_n & f'_{n,1} & \cdots & -1 & 0 & \cdots & 0\\
0 & \cdots & 0 & \sim_{n+1} & b'_{n+1} & f'_{n+1,1} & \cdots & f'_{n+1,n} & -1 & \cdots & 0\\
\vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & 0\\
0 & \cdots & 0 & \sim_m & b'_m & f'_{m,1} & \cdots & f'_{m,n} & 0 & \cdots & -1
\end{array}\right]
\end{array}$$

This works for underdetermined systems as well, for the same reasons the Gaussian elimination method is correct and complete, but it is tedious to show that. Thus, here we give an outline of the proof to build our terminology; for some details and deeper understanding, we refer to [Hef18, SR12].

Definition 4.2.3 (Pivot)**.** *Let M be a tableau. Let $\mathcal{P} = ((i_1, j_1), \dots, (i_k, j_k))$ be a k-tuple of pairs of row indices $i_1, \dots, i_k \in [m]$ and column indices $j_1, \dots, j_k \in [n]$.*

Then $\mathcal{P}$ is called a pivot ordering *on M if for every $k' \in [k]$*

- $\sim_{i_{k'}} \in \{=\}$,
- $a_{i_{k'}, j_{k'}} \neq 0$ *and*

- $a_{i_{k''},j_{k'}} = 0$ *for every* $k'' \in [k'-1]$.

The rows $i_{k'}$ *for* $k' \in [k]$ *are called* pivots *of* M *with respect to* $\mathcal{P}$*. Additionally, rows that are not a pivot of* M *w.r.t.* $\mathcal{P}$ *are called* dependent *w.r.t.* $\mathcal{P}$*. The set of dependent rows w.r.t.* $\mathcal{P}'$ *is denoted by* $\mathcal{B}_{M',\mathcal{P}'}$*.*

From now on, let $C = \{p_1 \sim_1 b_1, \dots, p_m \sim_m b_m\}$ be a system of linear constraints and $\tilde{V}$ be a partial vertex candidate of C. During our procedure, we maintain a tableau $M' \leftrightsquigarrow M_{(C\setminus V)\cup\tilde{V}}$ together with a pivot ordering $\mathcal{P}'$ on M'. The resulting algorithm state is denoted as $(M',\mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}}, \emptyset)$. Moreover, we maintain some invariants for such a state $(M',\mathcal{P}')$:

$$p_{i,i} = -1 \text{ for all } i \in [m] \text{ and } p_{i,j} = 0 \text{ for all } i \in [m],\ j \in \mathcal{B}_{M',\mathcal{P}'} \setminus \{i\} \tag{4.2}$$

Thus, for a dependent row $i \in \mathcal{B}_{M',\mathcal{P}'}$, the column $\boldsymbol{f'_{-,i}}$ labelled with p_i is the negative of the i-th standard vector and by Equation (4.1), p_i can be written as

$$p_i = \sum_{j\in[n]} a'_{i,j} \cdot x_j + \sum_{j\in\mathcal{P}'} f'_{i,j} \cdot p_j.$$

and for any row $i \in \mathcal{B}_{M',\mathcal{P}'}$, it makes sense to associate the original constraint $c_i \in C$ - what we will do implicitly from now on. Note that by the properties of row operations, c_i is equivalent to

$$\sum_{j\in[n]} a'_{i,j} \cdot x_j + \sum_{j\in\mathcal{P}'} f'_{i,j} \cdot p_j \sim_i \sum_{j\in\mathcal{P}'} f'_{i,j} \cdot b'_j.$$

Intuitively, this is the partial evaluation of the constraint c_i under the original constraints in $(C \setminus V) \cup \tilde{V}$ corresponding to the pivot rows w.r.t. $\mathcal{P}'$. Thus, we associate every pivot row j w.r.t. $\mathcal{P}'$ with its corresponding constraint $\tilde{c}_j \in \tilde{V}$. Our aim is to choose the pivots to match $\tilde{V}$:

Definition 4.2.4 (Normalized tableau)**.** $C = \{c_1, \dots, c_m\}$ *be a system of linear constraints,* $\tilde{V}$ *be a partial vertex candidate of* C*,* $M' \leftrightsquigarrow M_{(C\setminus V)\cup\tilde{V}}$ *and* $\mathcal{P}'$ *be a pivot ordering on* M'*.* $(M',\mathcal{P}')$ *is called* normalized *if and only if Equation* (4.2) *holds and* $\{i \mid (i,j) \in \mathcal{P}'\} = \{i \mid \tilde{c}_i \in \tilde{V}\}$*, that is, the pivots of* M' *w.r.t.* $\mathcal{P}'$ *correspond to* $\tilde{V}$*.*

Note that in contrast to the row echelon form in the Gaussian elimination, we do not require the rows to be sorted nor require the rows have a 1 as leading coefficient.

Furthermore, we can normalize any such state $(M',\mathcal{P}')$ by forward elimination:

Lemma 4.2.4. *Let* C *be a system of linear constraints,* $\tilde{V}$ *be a partial vertex candidate and* $M' \leftrightsquigarrow M_{(C\setminus V)\cup\tilde{V}}$ *such that Equation* (4.2) *holds.*

Then there exists a matrix $M'' \leftrightsquigarrow M'$ *and a pivot ordering* $\mathcal{P}''$ *on* M'' *such that* $(M'',\mathcal{P}'')$ *is normalized.*

Proof. W.l.o.g. let the first k rows of M' correspond to $\tilde{V}$ (i.e. the left hand sides of V are given as $P_V = \{p_1, \dots, p_k\}$). Remind that as $\tilde{V}$ is a partial vertex candidate, the set V is linearly independent.

Let $\mathcal{P}'$ be a pivot ordering on M' such that $\{i \mid (i,j) \in \mathcal{P}'\} \subseteq \{i \mid \tilde{c}_i \in \tilde{V}\}$, that is, only the first k rows might be pivots w.r.t. $\mathcal{P}'$. If the first k rows of M' are pivots w.r.t. $\mathcal{P}'$, then M' is already normalized. Otherwise, let $i \in [k] \cap \mathcal{B}_{M',\mathcal{P}'}$. Let

$J = \{j \mid (i',j) \in \mathcal{P}'\}$ be the columns of M' for which a pivot exists. By assumption, $a_{i,j} = 0$ for all $j \in J$. However, there must exist a $j \in \{1,\dots,n\} \setminus J$ such that $a_{i,j} \neq 0$, as otherwise, V would be linearly dependent. Thus, we can transform M' to an equivalent tableau M'' such that $\mathcal{P}'' := (\mathcal{P}', (i,j))$ is a pivot ordering on M'' by a sequence of row operations as in the Gaussian elimination. By iterating this, we obtain a normalized tableau. □

We can transfer some well known properties of the Gaussian elimination:

Theorem 4.2.5. *Let C be a system of linear constraints, $\tilde{V}$ be a partial vertex candidate and $(M', \mathcal{P}') \leadsto (M_{(C\setminus V)\cup\tilde{V}}, \emptyset)$ be normalized. Then*

(i) a row $\boldsymbol{a_{i,-}}$ in M' is redundant if and only if p_i is linearly dependent on P_V.

If V is maximal, then

(ii) a row is redundant if and only if it is dependent w.r.t. $\mathcal{P}'$ and

(iii) if no row is conflicting, then $sol(M') = sol(\tilde{V}) \neq \emptyset$.

Proof. W.l.o.g. $\mathcal{P}' = ((1,1), \dots, (k,k))$ with $|V| = k$, i.e. the first k rows are the pivots of M' w.r.t. $\mathcal{P}'$, are ordered and their columns are the first k columns.

(i) Let $i \in [m] \setminus [k]$. If $a_{i,-}$ is redundant, Equation (4.1) implies that p_i is linear dependent on the left hand sides P_V of V.

For the other direction, let p_i be linearly dependent on V and $A'' := A'_{[1,\dots,k,i],-}$ be the submatrix of A' consisting of the first k and the i-th row. Then by Theorem 4.2.2, $A'' = rank(P_V \cup \{p_i\}) = rank(V)$.

By assumption, $a''_{i,j} = 0$ for $j = 1,\dots,k$. Now assume for contradiction, that $a''_{i,-}$ is not redundant, i.e. there exists $l > k$ such that $a''_{i,l} \neq 0$. Thus, A'' is of the form

$$\begin{bmatrix} a''_{1,1} \neq 0 & \dots & a''_{1,k} & \dots & a''_{1,l} & \dots \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots \\ 0 & \dots & a''_{k,k} \neq 0 & \dots & a''_{k,l} & \dots \\ 0 & \dots & 0 & \dots & a''_{k+1,l} \neq 0 & \dots \end{bmatrix}$$

and clearly, the columns $\boldsymbol{a''_{-,1}}, \dots, \boldsymbol{a''_{-,k}}, \boldsymbol{a''_{-,l}}$ are linearly independent, thus the column rank is greater than $|V| = rank(A'')$. This is a contradiction to Theorem 2.3.2 - the row rank must be equal to its column rank.

(ii) This statement is an immediate consequence from (i) and the maximality of V.

(ii) Note that $sol(M') = sol(C_{M'})$ with $C_{M'} = \{\boldsymbol{a'_{i,-}x} \sim_i b'_i \mid i = 1,\dots,m\}$. Let $C' = \{\boldsymbol{a'_{i,-}x} \sim_i b'_i \mid i = 1,\dots,k\} \subseteq C_{M'}$ the subset of $C_{M'}$ corresponding to the vertex candidate.

We observe that $sol(C_{M'} \setminus C')$ is the set of all assignments, as all rows in $C_{M'} \setminus C'$ are redundant. Thus $sol(M') = sol(C_{M'} \setminus C') \cap sol(C') = sol(C')$. Furthermore, as C' is linearly independent, from Theorem 3.2.3 follows $sol(C') \neq \emptyset$.

□

Corollary 4.2.6. *Let C be a system of linear constraints and $\tilde{V}$ be a vertex candidate of C.*

Then there exists a normalized $(M', \mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}}, \emptyset)$ such that

$$sol(C \cup \tilde{V}) = \emptyset \text{ if and only if } M' \text{ has a conflicting row.}$$

4.3 Incrementality

In our application of the method presented above, neither the set of constraint nor the vertex candidate is given a priori. Instead, these sets are received incrementally allowing for detection of conflicts in partial systems and vertex candidates with less effort. Also, changing the vertex candidate or changing the considered system of linear constraints is possible without recomputing the whole matrix - only the deltas need to be computed.

To do so, we define procedures maintaining a normalized $(M', \mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}}, \emptyset)$ for a system C and a partial vertex candidate $\tilde{V}$ of C. Note that $M' = \emptyset$ fulfils the mentioned properties for $C = \tilde{V} = \emptyset$. Then, starting from there, we can construct such a matrix for any system of linear constraints and any vertex candidate of this systems by methods allowing for adding/removing constraints to/from the system C and adding/removing constraints from a partial vertex candidate $\tilde{V}$.

Algorithms 4 and 6 define procedures for extending and shrinking the current vertex candidate. If an element $\tilde{c}_i$ is added to $\tilde{V}$, then its corresponding row i will be altered such that is originates from $\tilde{c}_i$ instead of c_i by adjusting its right hand side and will then be made a pivot in the spirit of Lemma 4.2.4. Removing an element $\tilde{c}_i$ from $\tilde{V}$ is simply reverting the operations that were performed for adding it to $\tilde{V}$ if $\tilde{c}_i$ was the last element added to $\tilde{V}$, that is (i,j) is the greatest element w.r.t. $\mathcal{P}'$ for some $j \in [m]$. Otherwise, pivots greater than i w.r.t. $\mathcal{P}'$ need to be reverted first and re-pivoted again after removing $\tilde{c}_i$. This case is quite expensive, but will not occur frequently in practise as the embeddings presented in Section 4.4 shrink the vertex candidate chronologically.

Furthermore, the procedures in Algorithms 5 and 7 allow adding and removing constraints to an existing normalized tableau. Adding a constraint is done by adding it as a row to the current tableau in the sense of Definition 4.2.1 and replaying pivot steps according to the current pivot ordering $\mathcal{P}'$ on it. Removing a constraint (that does not correspond to a pivot) is as simple as removing a row from the tableau.

Algorithm 4: Adding constraints to the vertex candidate

```
1 function SelectVertexDefining(($M'$,$\mathcal{P}'$), $\tilde{c}_i$)
      Input: normalized $(M',\mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}},\emptyset)$ such that $\tilde{V}$ is a partial
             vertex candidate of $C$, $\tilde{c}_i$ such that $c_i \in C \setminus V$ and $\boldsymbol{a_{i,-}} \neq 0$
      Output: normalized $(M'',\mathcal{P}'') \leftrightsquigarrow (M_{(C\setminus V')\cup\tilde{V}'},\emptyset)$ such that $\tilde{V}' = \tilde{V} \cup \{\tilde{c}_i\}$
2     $M'' \longleftarrow M'$
      // set constraint tight
3     $\sim''_i \longleftarrow =$
4     $b''_i \longleftarrow b'_i$ respectively $b'_i + \varepsilon$ or $b'_i - \varepsilon$
      // Now: $M'' \leftrightsquigarrow M_{(C\setminus V')\cup\tilde{V}'}$
      // make row $i$ a pivot
5     choose $j$ such that $a''_{i,j} \neq 0$
6     foreach $i' \in [m]$ such that $i' \neq i$ and $i' \notin \{i' \mid (i',j') \in \mathcal{P}_{M''}\}$ do
7         $a''_{i',-} \longleftarrow a''_{i',-} - \frac{a''_{i',j}}{a''_{i,j}} \cdot a''_{i,-}$
8         update $b''_{i'}$ and $f''_{i',-}$ analogously
      // Now: $(\mathcal{P}',(i,j))$ is a pivot ordering on $M''$
9     return $(M'',(\mathcal{P}',(i,j)))$
```

Algorithm 5: Adding constraints

```
1 function AddRow(($M'$,$\mathcal{P}'$), $c$)
      Input: normalized $(M',\mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}},\emptyset)$ such that $\tilde{V}$ is a partial
             vertex candidate, constraint $c = p_{m+1} \sim_{m+1} b_{m+1} \notin C$
      Output: normalized $(M'',\mathcal{P}') \leftrightsquigarrow (M_{(C'\setminus V)\cup\tilde{V}},\emptyset)$ such that
              $C' = C \cup \{p_{m+1} \sim_{m+1} b_{m+1}\}$
2     Assume $\mathcal{P}' = ((i_1,j_1),\ldots,(i_l,j_l))$
3     $M'' \longleftarrow M'$ except that $M''$ has one more row and $F''$ has one more
       column
4     $a''_{m+1,-} \longleftarrow a_{m+1,-}$ where $a_{m+1,-}$ are the coefficients of $p_{m+1}$
5     $f''_{m+1,m+1} \longleftarrow -1$, $f''_{i',m+1} \longleftarrow 0$, $f''_{m+1,i'} \longleftarrow 0$ for $i' \in [m]$
6     foreach $k' = 1,\ldots,l$        // eliminate existing pivots w.r.t.
       $\mathcal{P}'$
7      do
8         $a''_{m+1,-} \longleftarrow a''_{m+1,-} - \frac{a''_{m+1,j_{k'}}}{a''_{i_{k'},j_{k'}}} \cdot a''_{i_{k'},-}$
9         update $b''_{m+1}$ and $f''_{m+1,-}$ analogously
10    return $(M'',\mathcal{P}')$
```

Algorithm 6: Removing constraints from the vertex candidate

```
1  function ResetVertexDefining(($M'$,$\mathcal{P}'$), $c_i$)
       Input: normalized $(M',\mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}},\emptyset)$ such that $\tilde{V}$ is a partial
              vertex candidate of $C$, $c_i \in V$
       Output: normalized $(M'',\mathcal{P}'') \leftrightsquigarrow (M_{(C\setminus V')\cup\tilde{V}'},\emptyset)$ such that $\tilde{V}' = \tilde{V} \setminus \{\tilde{c}_i\}$
2      Assume $\mathcal{P}' = ((i_1,j_1),\dots,(i_l,j_l))$ and $i = i_k$
3      $M'' \longleftarrow M'$
4      foreach $k' = l,\dots,k$        // revert SelectVertexDefining in
        reverse order w.r.t. $\mathcal{P}'$
5       do
           // remove pivot
6          foreach $i' \in [m]$ do
7              $a''_{i',-} \longleftarrow a''_{i',-} - f_{i',j_{k'}} \cdot a''_{i_{k'},-}$
8              update $b''_{i'}$ and $f''_{i',-}$ analogously
           // revert setting constraint tight
9          $\sim''_{i_{k'}} \longleftarrow$ relation of $c_{i_{k'}}$
10         $b''_{i_{k'}} \longleftarrow b'_{i_{k'}}$ respectively $b'_{i_{k'}} - \varepsilon$ or $b'_{i_{k'}} + \varepsilon$
11     $\mathcal{P}'' \longleftarrow (i_1,j_1),\dots,(i_{k-1},j_{k-1})$
12     foreach $k' = k+1,\dots,l$     // redo pivots after $k$ w.r.t. $\mathcal{P}'$
13      do
14         $(M'',\mathcal{P}'') \longleftarrow$ SelectVertexDefining (($M''$,$\mathcal{P}''$), $\tilde{c}_{i_{k'}}$)
15     return $(M'',\mathcal{P}'')$
```

Algorithm 7: Removing constraints

```
1  function RemoveRow(($M'$,$\mathcal{P}'$), $c_i$)
       Input: normalized $(M',\mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}},\emptyset)$ such that $\tilde{V}$ is a partial
              vertex candidate, constraint $c_i \in C \setminus V$
       Output: normalized $(M'',\mathcal{P}') \leftrightsquigarrow (M_{(C'\setminus V)\cup\tilde{V}},\emptyset)$ such that $C' = C \setminus \{c_i\}$
2      $M'' \longleftarrow M'$ without the $i$-th row and the $i$-th column of $F$
3      return $(M'',\mathcal{P}')$
```

4.3.1 Computing an assignment

For any consistent tableau M, a model corresponding to the selected vertex can be generated by backward substitution on the equation system induced the pivot rows as described in Algorithm 8.

Algorithm 8: Computing an assignment

```
1 function ComputeModel(($M'$,$\mathcal{P}'$))
      Input: normalized $(M', \mathcal{P}') \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}}, \emptyset)$ such that $\tilde{V}$ is a (full)
             vertex candidate for a system $C$ and $M'$ does not contain a
             conflicting row
      Output: $\alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon]$ such that $\alpha \models M'$
2     Assume $\mathcal{P}' = ((i_1,j_1), \ldots, (i_l,j_l))$
3     $\alpha \longleftarrow$ empty assignment
4     foreach $k' = l, \ldots, 1$ do
5         foreach $j'$ such that $a'_{i_{k'},j'} \neq 0, j' \neq j_{k'}$ and $\alpha(x_{j'})$ undefined  // set
           all undetermined variables to 0
6          do
7          |  $\alpha(x_{j'}) \longleftarrow 0$
8         $\alpha(x_{i_{k'}}) \longleftarrow \left(b'_{i_{k'}} - \sum_{j' \mid a_{i_{k'},j'} \neq 0, j' \neq j_{k'}} a'_{i_{k'},j'} \cdot \alpha(x_{j'})\right) / a'_{i_{k'},j_{k'}}$
9     return $\alpha$
```

Theorem 4.3.1. *Let M' be a valid input to Algorithm 8, then the algorithm returns an assignment $\alpha \models M'$.*

Proof. By Corollary 4.2.6, $sol(M') = sol(\tilde{V}) \neq \emptyset$. Assume first that A' has full rank, i.e. no undetermined variables occur. Then it is easy to see that α encodes the assignment obtained by backward substitution as in the Gaussian elimination method.

It remains to show that we can assign all undetermined variables to 0: Let x be an undetermined variable. Adding the constraint $x = 0$ to C would not make the solution set empty: First, it is linearly independent from $\tilde{V}$, thus $\tilde{V}' := \tilde{V} \cup \{x = 0\}$ is still linearly independent. As all constraints $C \setminus V$ are linearly dependent on $\tilde{V}$, they do as well on $\tilde{V}'$, thus $sol(C \cup \{x = 0\}) = sol(\tilde{V}')$. By Theorem 3.2.3, it follows $sol(\tilde{V}') \neq \emptyset$, which proves the claim. □

Afterwards, an assignment in $\mathcal{U}$ for the original problem can be computed using Lemma 3.3.3.

4.3.2 Computing lemmas

If there is a conflicting row in a tableau, then we do not only want to continue the search by trying the next vertex candidate, but want to generalize the current conflict to exclude further (partial) vertex candidates and - for applications in SMT - unsatisfiable sets of constraints.

In the following, let C be a system of linear constraints, $\tilde{V}$ be a vertex candidate of C and $(M', \mathcal{P}') \leftrightsquigarrow M_{(C\setminus V)\cup\tilde{V}}$ be normalized.

4.3.2.1 Excluding the current vertex candidate

Let $i \in [m]$ be a conflicting row of M'. Then, we define

$$K_i := \{c_j \text{ respectively } \tilde{c}_j \in (C \setminus V) \cup \tilde{V} \mid f'_{i,j} \neq 0\}$$

as the *origins* of row i. Note that as $(M', \mathcal{P}')$ is normalized, $c_i \in K_i$ and $K_i \setminus \{c_i\} \subseteq \tilde{V}$. As p_i is a linear combination of the right hand sides $P_{K_i \setminus \{c_i\}}$ of $K_i \setminus \{c_i\}$, K_i is already unsatisfiable:

Lemma 4.3.2. *Let $(\mathcal{U}, <)$ be any ordered vector space over $\mathcal{F}$, $p \sim b$ and $p_i = b_i, i \in [k]$ be linear constraints over $(\mathcal{U}, <)$ in variables X such that there exists $f_1, \ldots, f_k \in \mathcal{F}$ such that*

$$p = f_1 \cdot p_1 + \ldots + f_k \cdot p_k.$$

Then $p_1 \sim_1 b_1 \wedge \ldots \wedge p_k \sim_k b_k \wedge p \sim b$ is satisfiable if and only if $\sum_{i \in [k]} f_i \cdot b_i \sim b$ is valid (i.e. trivially true).

Proof. Let $\alpha : X \to \mathcal{U}$. Then

$$\alpha \models p \sim b \iff \alpha \models f_1 \cdot p_1 + \ldots + f_k \cdot p_k \sim b \iff \alpha \models f_1 \cdot b_1 + \ldots + f_k \cdot b_k \sim b$$

□

In other words, by selecting $K_i \setminus \{c_i\} \subseteq \tilde{V}$ into the partial vertex candidate, c_i is conflicting. The following theorem proves that $K_i \setminus \{c_i\}$ is the smallest subset of $\tilde{V}$ causing c_i to be conflicting:

Theorem 4.3.3. *Let $(\mathcal{U}, <)$ be any ordered vector space over $\mathcal{F}$, p and $p_i, i \in [k]$ be linear $\mathcal{F}$-combinations of X such that $p_1, \ldots, p_k$ are linearly independent and there exists $f_1, \ldots, f_k \in \mathcal{F} \setminus \{0\}$ such that*

$$p = f_1 \cdot p_1 + \ldots + f_k \cdot p_k.$$

Then $\{p_1, \ldots, p_k\}$ is minimal with this property.

Proof. W.l.o.g, assume we can leave out p_k. Then there exist $f'_1, \ldots, f'_{k-1}$ such that $p = f'_1 \cdot p_1 + \ldots + f'_{k-1} \cdot p_{k-1}$ and

$$\begin{aligned} & f_1 \cdot p_1 + \ldots + f_k \cdot p_k = f'_1 \cdot p_1 + \ldots + f'_{k-1} \cdot p_{k-1} \\ \iff & f_k \cdot p_k = (f'_1 - f_1) \cdot p_1 + \ldots + (f'_{k-1} - f_{k-1}) \cdot p_{k-1} \end{aligned}$$

But then, $\{p_1, \ldots, p_k\}$ would not be linearly independent, which is a contradiction. □

This fact can be expressed as the lemma

$$\left(\bigwedge_{\tilde{c}_j \in K_i \setminus \{c_i\}} \tilde{c}_j \right) \to \neg c_i.$$

Note that this does not only exclude the current (partial) vertex candidate, as the sets K_i might be smaller and thus excludes all partial vertex candidates including K_i. Furthermore, K_i only depends on $\{i' \mid (i', j') \in \mathcal{P}'\}$, i.e. it is independent from the order of $\mathcal{P}'$ and the selected columns.

4.3.2.2 Equalities

Note that if the relation of c_j (for $\tilde{c}_j \in K_i$) is $=$, then $\tilde{c}_j = c_j$; thus, in this case, the lemma does not only state something about the current vertex candidate, but about the original constraint as well.

4.3.2.3 Simplex-based lemmas

In the presentation of the Simplex algorithm, we introduced the notion of suitable non-basic variables for selecting a pivot element. This is based on the observation that pivoting with non-suitable variables obtains a conflicting tableau. We already observed that M' can be seen as Simplex tableau and thus, we can use the non-suitable variables to exclude further vertex candidates. Unsatisfiability of sets of constraints can be concluded similarly to Simplex as well.

Assuming weakened systems Let us first assume we execute our procedure on the weakened version C_w^* (over $(\mathcal{U}[\varepsilon], <)$) of a system C of linear constraints over $(\mathcal{U}, <)$ without constraints with relation $\neq$:

Lemma 4.3.4. *Let $(\mathcal{U}, <)$ be any vector space over $\mathcal{F}$, $p \sim b$, $p_i \sim_i b_i, i = 1, \ldots, k$ be linear constraints over $(\mathcal{U}, <)$ and $f_1, \ldots, f_k \in \mathcal{F} \setminus \{0\}$ such that*

$$p = f_1 \cdot p_1 + \ldots + f_k \cdot p_k.$$

Let $V = \{p_i \sim_i b_i \mid i = 1, \ldots, k\}$ and $\tilde{V} \cup \{p \sim b\}$ be conflicting. Define $V' = (V \setminus \{p_k \sim_k b_k\}) \cup \{p \sim b\}$. Then $\tilde{V}' \cup \{p_k \sim_k b_k\}$ is conflicting if and only if one of the following conditions is satisfied:

- *If $\sim \in \{\leq\}$, then $f_k < 0$ and $\sim_k \in \{\leq\}$ or $f_k > 0$ and $\sim_k \in \{\geq\}$;*
- *if $\sim \in \{\geq\}$, then $f_k < 0$ and $\sim_k \in \{\geq\}$ or $f_k > 0$ and $\sim_k \in \{\leq\}$;*
- *if $\sim \in \{=\}$ and $f_1 \cdot b_1 + \ldots + f_k \cdot b_k > b$, then as in the case $\sim \in \{\leq\}$;*
- *if $\sim \in \{=\}$ and $f_1 \cdot b_1 + \ldots + f_k \cdot b_k < b$, then as in the case $\sim \in \{\geq\}$.*

Proof. Assume that $\sim \in \{\leq\}$. Then

$$\begin{aligned}
& p_1 = b_1 \wedge \ldots \wedge p_k = b_k \wedge p \leq b \text{ is conflicting} \\
\iff & f_1 \cdot b_1 + \ldots + f_k \cdot b_k > b \\
\iff & f_1 \cdot b_1 + \ldots + f_{k-1} \cdot b_{k-1} - b > -f_k \cdot b_k \\
\iff & \frac{f_1}{-f_k} \cdot b_1 + \ldots + \frac{f_{k-1}}{-f_k} \cdot b_{k-1} + \frac{1}{f_k} \cdot b > b_k \wedge f_k < 0 \\
& \vee \frac{f_1}{-f_k} \cdot b_1 + \ldots + \frac{f_{k-1}}{-f_k} \cdot b_{k-1} + \frac{1}{f_k} \cdot b < b_k \wedge f_k > 0 \\
\iff & \text{if } f_k < 0 \text{, then } p_1 = b_1 \wedge \ldots \wedge p_{k-1} = b_{k-1} \wedge p = b \wedge p_k \leq b_k \text{ is conflicting;} \\
& \text{if } f_k > 0 \text{, then } p_1 = b_1 \wedge \ldots \wedge p_{k-1} = b_{k-1} \wedge p = b \wedge p_k \geq b_k \text{ is conflicting}
\end{aligned}$$

$\sim \in \{\geq\}$ is analogous. For $\sim \in \{=\}$, we use that $p = b \iff p \leq b \wedge p \geq b$. □

Thus, by application of this lemma to a conflicting row $i \in [m]$ of M' and K_i as defined above, we obtain further partial vertex candidates that can be excluded in the same way as for the current one: W.l.o.g. $K_i = \{\tilde{c}_1, \ldots, \tilde{c}_{i-1}, c_i\}$ and define $\tilde{V}_k = (K_i \setminus \{\tilde{c}_k, c_i\}) \cup \{\tilde{c}_i\}$ for $k = 1, \ldots, i-1$ and $\tilde{V}_i = K_i \setminus \{c_i\}$. Then, $\tilde{V}_i \cup \{c_i\}$ is conflicting and for all k such that the condition in Lemma 4.3.4 applies, $\tilde{V}_k \cup \{c_k\}$ is unsatisfiable. Hence, the lemmas

$$\left(\bigwedge_{\tilde{c}_j \in \tilde{V}_j} \tilde{c}_j \right) \to \neg c_k$$

are valid for those k.

Furthermore, observe that by Theorem 3.2.2, there must be a satisfying vertex candidate in $C_i := \{c_1, \ldots, c_i\}$ if this set is satisfiable. Thus, if Lemma 4.3.4 excludes all "neighbouring" vertex candidates, that is $\tilde{V}_k \cup \{c_k\}$ is unsatisfiable for all $k = 1, \ldots, i$, we can conclude the unsatisfiability of C_i and the lemma

$$\bigvee_{j=1,\ldots,i} \neg c_j$$

is valid.

Extension to strict inequalities Extending these observations to the procedure based on Corollary 3.3.4 is a bit hacky: Given a partial vertex candidate $\tilde{V}$ of a system of linear constraints C (without the relation $\neq$), we observe that V is a maximal linearly independent subset of C_w, that is $\tilde{V_w} = \tilde{V}$. The idea is given a conflicting row $i \in [m]$ of M, we consider instead of $M' \leftrightsquigarrow M_{(C \setminus V) \cup \tilde{V}}$ the corresponding matrix $(M')_w \leftrightsquigarrow M_{(C_w \setminus V_w) \cup \tilde{V}}$.

Let us first see, why a conflicting row i in M' corresponds to a conflicting row in $(M')_w$: While this is clear for constraints with relation $=$ and $\leq$ in C (as they are also in C_w), constraints of the form $(p_i < b_i) \in C$ might differ from their counterparts in $(p_i \leq b_i - \varepsilon) \in C_w$. We observe that for any $\alpha : \{x_1, \ldots, x_n\} \to \mathcal{U}[\varepsilon]$,

$$\alpha \models p_i \leq b_i - \varepsilon \implies \alpha \models p_i < b_i$$

and can follow, that if $\tilde{V} \cup \{p_i < b_i\}$ is conflicting, then $\tilde{V} \cup \{p_i \leq b_i - \varepsilon\}$ is as well. Thus, the row i is conflicting in $(M')_w$, the preconditions for Lemma 4.3.4 are fulfilled and we can generate the lemmas φ'_k for the conflicting row i in $(M')_w$ as defined above.

Now, we obtain lemmas φ_k corresponding to the conflicting row i in M' by replacing all $c_w \in C_w$ by their original constraints in $c \in C$ in φ'_k. While this might not raise doubts in the case that $c_k \in C$ is of the form $p_k \leq b_k$ or $p_k = b_k$, it does for the case $p_k < b_k$: φ'_k encodes that $\tilde{V}_k \cup \{p_k \leq b_k - \varepsilon\}$ is unsatisfiable. But this does not imply that $\tilde{V}_k \cup \{p_k < b_k\}$ is unsatisfiable as well - which is encoded by φ_k. However, it is okay to exclude this vertex candidate here as if C_i is satisfiable, then there exists a vertex candidate $\tilde{V}_{k'}$ for $(C_i)_w$ which will also satisfy C_i. Thus, there are still be enough vertex candidates left to prove satisfiability.

Extension to not-equal constraints If the conflicting row i in M' corresponds to a constraint c_i of the form $(p_i \neq b_i)$, then we handle it as the cases $(p_i < b_i)$ and

$(p_i > b_i)$ separately. That is checking for each $c_k \in K_i \setminus \{c_i\}$ the sets $\tilde{V}_k^< \cup \{c_k\}$ and $\tilde{V}_k^> \cup \{c_k\}$ for satisfiability, where $\tilde{V}_k^<$ and $\tilde{V}_k^>$ represent each a choice for $\tilde{c}_i$.

If a constraint $(p_k \neq b_k) \in V_i$ for a conflicting row $i \neq k$ in M', then we extend Lemma 4.3.4 to handle the case "$p_k < b_k \in V_i$ or $p_k > b_k \in V_i$" simultaneously. That is, $\tilde{V}_k \cup \{p_k \neq b_k\}$ is conflicting if and only if c_i is of the form $p_i = b_i$.

4.4 Embedding into DPLL(T)

We present several variants of embedding our incremental procedure into the DPLL(T) framework.

4.4.1 Encoding vertex candidates in a SAT formula - First variant

Here, we encode the decisions which constraints should be added to the vertex candidate as Boolean formula with the help of additional helper variables. This formula is eagerly generated in a preprocessing step and passed to the SAT solver together with the original formula. The theory solver then collects the helper variables and adapts its current state using the procedures defined in the previous section.

4.4.1.1 Preprocessing

The preprocessing is applied before the formula is passed to the SAT solver. It extends the formula by a Boolean formula containing variables select_c where c is a theory constraint: select_c is assigned to be *true* if and only if $\tilde{c}$ should be added to the vertex candidate. Additionally, it makes sure that only those constraints assigned to true can define the vertex.

However, a priori, it is not known which constraints are linearly dependent nor the rank of the system induced by the currently asserted constraints. Thus, these circumstances cannot be encoded into the formula entirely. We run into two encoding-related issues:

- First, the SAT solver might select linearly dependent constraints into the vertex candidate. In this case, we defer handling these cases to later by adding lemmas excluding those selections in the theory solver.
- Second, the possibilities for encoding the number of constraints that need to be selected to define a vertex are limited. However, we know certain upper bounds on the rank, among them the number of theory variables. Thus, our formula uses such an upper bound to select an appropriate number of constraints. Unluckily, this is not always possible in the case of an underdetermined system (caused by linear dependencies or simply not enough asserted constraints): For this reason, the formula may select artificial constraints of the form $x = 0$ for any theory variable x. Note that this does not cause problems, as in case of an underdetermined system, the remaining variables can be chosen arbitrarily (as shown in the proof of Theorem 4.3.1).

Now, let φ be the original input formula, and $x_1, \ldots, x_n$ all (real-valued) theory variables in φ.

Each constraint that might be asserted should be eligible for selecting the vertex candidate. That are all constraints c occurring in the formula as well as their negations $\bar{c}$, thus we define $C := \{c, \bar{c} \mid c \in \mathit{Constraints}(\varphi)\}$.

We can employ an observation to reduce the size of C: If φ is in CNF, then it is particularly in *negation normal form (NNF)*: Thus, if a constraint c occurs only positively in the formula, and it is not asserted to be true, then its truth value does not matter; therefore those constraints are not passed to the theory solver and need not to be considered for selecting the vertex candidate. Thus, if φ is in CNF, then we can set $C = \mathit{Constraints}(\varphi)$.

We introduce for each constraint $c \in C$ a selection variable select_c. According to Definition 3.3.1, to select a vertex, for $\neq$-constraints we need to split: Thus, for those constraints, we instead introduce selection variables $\mathit{select}_c^<$ and $\mathit{select}_c^>$ meaning that the constraint should be treated as $<$-constraint respectively $>$-constraint. Additionally, for the artificial constraints, we create selection variables $\mathit{select}_{x_i=0}$ for $i \in [n]$.

Our formula asserts that for every variable x_i, a constraint containing x_i is selected (either original ones or of the form $x = 0$); for encoding this, we additionally define $\mathit{select}_{x,c}$ for $c \in C, x \in \mathit{Vars}(c)$. If an $\neq$-constraint is selected, then also the decision whether it should be treated as $<$-constraint or $>$-constraint. Furthermore, let $C_x = \{c \in C \mid x \in \mathit{Vars}(C)\}$ for $x \in \mathit{Vars}(C)$. We add the following formulas to φ by connecting them with $\wedge$:

- Only asserted formulas can be selected; for constraints c of the form $p \neq 0$, at most one of $\mathit{select}_c^<$ and $\mathit{select}_c^>$ can be selected.

$$\begin{aligned} &\bigwedge_{\substack{c \in C \\ c \neq (p \neq b)}} (\mathit{select}_c \to c) \\ \wedge &\bigwedge_{\substack{c \in C \\ c = (p \neq b)}} \big((\mathit{select}_c^< \to c) \wedge (\mathit{select}_c^> \to c) \wedge (\neg\mathit{select}_c^< \vee \neg\mathit{select}_c^>)\big) \end{aligned}$$

- For every variable, exactly one constraint is selected:

$$\begin{aligned} &\bigwedge_{x \in \mathit{Vars}(C)} \left(\left(\bigvee_{c \in C_x} \mathit{select}_{x,c}\right) \vee \mathit{select}_{x=0}\right) \\ \wedge &\bigwedge_{x \in \mathit{Vars}(C)} \text{at-most-one-of}(\{\mathit{select}_{x,c} \mid c \in C_x\} \cup \{\mathit{select}_{x=0}\}) \end{aligned}$$

- Each constraint is selected at most once for a variable:

$$\bigwedge_{c \in C} \text{at-most-one-of}(\{\mathit{select}_{x,c} \mid x \in \mathit{Vars}(c)\})$$

- A constraint is selected if it is selected for a variable:

$$\begin{aligned} &\bigwedge_{\substack{c \in C, x \in \mathit{Vars}(c) \\ c \neq (p \neq b)}} (\mathit{select}_{x,c} \to \mathit{select}_c) \\ \wedge &\bigwedge_{\substack{c \in C, x \in \mathit{Vars}(c) \\ c = (p \neq b)}} \big(\mathit{select}_{x,c} \to (\mathit{select}_c^< \vee \mathit{select}_c^>)\big) \end{aligned}$$

At-most-one encoding at-most-one-of($\{x_1, \dots, x_k\}$) encodes that at most one of $x_1, \dots, x_n$ is selected.

A possible implementation is the *pairwise encoding* defined as

$$\text{at-most-one-of}(\{x_1, \dots, x_k\}) = \bigwedge_{i=1,\dots,k-1} \bigwedge_{j=i+1,\dots,k} (\neg x_i \vee \neg x_j).$$

However, there are better encodings, as described for example in [HN13]. In this thesis, the following *binary encoding* was implemented.

Fresh Boolean variables $b_1, \dots, b_{\log_2 \lceil k \rceil}$ are introduced. The idea is that if x_i is the variable set to *true*, then the model of $b_1, \dots, b_{\log_2 \lceil k \rceil}$ is a binary encoding of $i - 1$:

$$\text{at-most-one-of}(\{x_1, \dots, x_k\}) = \bigwedge_{i=1,\dots,k} \bigwedge_{j=1,\dots,\log_2 \lceil k \rceil} (\neg x_i \vee \phi(i,j))$$

where $\phi(i,j) = b_j$ ($\phi(i,j) = \neg b_j$) if the j-th bit of the binary encoding of $i - 1$ is 1 (0).

4.4.1.2 Theory solver

The theory solver is an SMT solver which also receives a model for the $\textit{select}_c$ variables. Thus, it can update the set of asserted constraints and the partial vertex candidate accordingly using the procedures defined in the previous section.

As mentioned before, a selected constraint might be linearly dependent to the current partial vertex candidate. However, the corresponding row is then redundant - that means that a minimal subset of the current partial vertex candidate can be computed such that the given constraint is linearly dependent on this set according to Theorem 4.3.3. Thus, the corresponding sets of selection variables can be excluded by an infeasible subset.

If a conflict occurs, the lemmas are generated as described above. Note that the conflict needs to be explained in terms of the selection variables, which is achieved by replacing the elements of the partial vertex candidate by their corresponding selection variables.

The embedding is presented in Algorithm 9.

4.4.2 Encoding vertex candidates in a SAT formula - Second variant

As the formula size grows quickly, especially because of the *at-most-one-of* subformulas, we present an evolution of the first variant.

We also observe, that if the left hand sides of two constraints are equal, then they cannot be part of the same vertex candidate and thus should not be selected twice. Furthermore, the linear dependencies of selected constraints are actually properties of their left hand sides.

Thus given the input formula φ, let C be defined as in the first variant with the additional constraints $\{x = 0 \mid x \in \textit{Vars}(\varphi)\} \subseteq C$ and C_x be defined as before. Furthermore, let $P = \{p \mid (p \sim b) \in C\}$ and P_x analogously to C_x. At this point, we note that we assume the constraints in C to be normalized in some way such that the leading coefficients of P are positive.

Algorithm 9: Theory solver

```
1  function Check($(M,\mathcal{P})$, $C$, $S$)
       Input: $(M,\mathcal{P})$ current state, $C_+$,$C_-$ set of added/removed asserted
              constraints, $S_+, S_-$ set of added/removed asserted selection
              variables
       Output: SAT if the select-variables induce a satisfying vertex candidate,
               UNSAT if there is a conflict or UNKNOWN otherwise
2      foreach $select_c \in S_-$, $select_c^< \in S_-$, $select_c^> \in S_-$ do
3          $(M,\mathcal{P}) \longleftarrow$ ResetVertexDefining($(M,\mathcal{P})$,$c$)
4      foreach $c \in C_-$ do
5          $(M,\mathcal{P}) \longleftarrow$ RemoveRow($(M,\mathcal{P})$,$c$)
6      foreach $c \in C_+$ do
7          $(M,\mathcal{P}) \longleftarrow$ AddRow($(M,\mathcal{P})$,$c$)
8      foreach $select_c \in S_+$, $select_c^< \in S_+$, $select_c^> \in S_+$ do
9          if the row of $c$ in $M$ is redundant then
10             $K \longleftarrow$ ComputeOrigins($(M,\mathcal{P})$,$c$)
11             $R \longleftarrow \{select_c \mid c \in K, c$ is not $\neq\} \cup$
                 $\{select_c^<$ respectively $select_c^> \mid c \in K, c$ is $\neq\}$
12             return (UNSAT, R)

13
14         $(M,\mathcal{P}) \longleftarrow$ SelectVertexDefining($(M,\mathcal{P})$,$\tilde{c}$) with $\tilde{c}$ according to
             the active selection variable

15
16         $K \longleftarrow \emptyset$
17         foreach conflicting row $d$ in $M$ do
18             $K \longleftarrow K\cup$ ComputeConflict($(M,\mathcal{P})$,$d$)
                 where every $\tilde{c}'$ is replaced by $select_{c'}$ resp. $select_{c'}^<$ or $select_{c'}^>$
19         if $K \neq \emptyset$ then
20             return (UNSAT, K)

21
22     if all rows in $M$ are either redundant or a pivot then // full vertex
         selected
23         return (SAT, ComputeModel($(M,\mathcal{P})$))
24     else // only partial vertex selected
25         return UNKNOWN
```

We define Boolean variables select_p for all $p \in P$ and $\mathit{select}_{x,p}$ for all $p \in P$ and $x \in \mathit{Vars}(p)$ and append the following formulas to the input formula φ:

- Only asserted formulas can be selected.

$$\bigwedge_{\substack{c \in C \\ c \neq (p \neq b)}} (\mathit{select}_c \to c) \land \bigwedge_{\substack{c \in C \\ c = (p \neq b)}} \big((\mathit{select}_c^< \to c) \land (\mathit{select}_c^> \to c)\big)$$

- For every variable, exactly one polynomial is selected:

$$\bigwedge_{x \in \mathit{Vars}(P)} \left(\bigvee_{p \in P_x} \mathit{select}_{x,p} \right)$$
$$\land \bigwedge_{x \in \mathit{Vars}(P)} \text{at-most-one-of}(\{\mathit{select}_{x,p} \mid p \in P_x\})$$

- Each polynomial is selected if and only if it is selected at most once for a variable:

$$\bigwedge_{p \in P} \left(\left(\bigvee_{x \in \mathit{Vars}(p)} \mathit{select}_{x,p} \right) \leftrightarrow \mathit{select}_p \right)$$
$$\land \bigwedge_{p \in P} \text{at-most-one-of}(\{\mathit{select}_{x,p} \mid x \in \mathit{Vars}(p)\})$$

- Each polynomial is selected if and only if a constraint is selected for this polynomial:

$$\bigwedge_{p \in P} \left(\mathit{select}_p \leftrightarrow \left(\bigvee_{\substack{c \in C \\ c = (p \sim b), \sim \text{ is not } \neq}} \mathit{select}_c \lor \bigvee_{\substack{c \in C \\ c = (p \neq b)}} \big(\mathit{select}_c^< \lor \mathit{select}_c^>\big) \right) \right)$$
$$\land \bigwedge_{p \in P} \text{at-most-one-of}(\{\mathit{select}_c \mid c \in C, c = (p \sim b), \sim \text{ is not } \neq\} \cup$$
$$\{\mathit{select}_c^<, \mathit{select}_c^> \mid c \in C, c = (p \neq b)\})$$

The theory solver is mostly equal the first variant. However, if a linear dependency is detected, its reason is returned as a lemma in the select_p variables.

4.4.3 Modifying the decision heuristic of the SAT solver

Encoding the selection of a vertex in a SAT formula has some disadvantages:

- The encoding grows rapidly and might slow down the SAT solver,
- the selection of helper constraints $x = 0$ adds unnecessary overhead,
- and the linear dependencies between constraints are added explicitly as clauses to the SAT solver, although the theory solver tracks these linear dependencies already.

A possible alternative is to enforce the selection of a vertex candidate in the decision heuristic of the SAT solver. We again define C and the selection variables select_c, $\mathit{select}_c^<$, $\mathit{select}_c^>$ as described in Section 4.4.1 and append

$$\bigwedge_{\substack{c\in C\\ c\neq(p\neq b)}} (\mathit{select}_c \to c) \wedge \bigwedge_{\substack{c\in C\\ c=(p\neq b)}} \big((\mathit{select}_c^< \to c) \wedge (\mathit{select}_c^> \to c)\big)$$

to φ to make sure that all constraints corresponding to the vertex candidate are also asserted.

Now, when the SAT solver makes the next decision and picks a selection variable, w.l.o.g. select_c for $c \in C$, $c \neq (p \neq b)$. Also, let $\tilde{V}$ be the currently selected partial vertex candidate. Then,

- select_c is decided to be *true* if c is asserted and linearly independent from V,
- select_c is decided to be *false* if c is asserted and linearly dependent on V,
- and the decision is deferred to later otherwise.

The information whether an asserted constraint is linearly dependent or independent on V is tracked in the theory solver: The maintained tableau M is always normalized, thus a row i of M is redundant if and only if its corresponding constraint c_i is linearly dependent on V.

It is also easy to see that a complete Boolean assignment always selects exactly a vertex candidate. However, the SAT solver does not have the information to conclude that given a set of constraints $C' \subseteq C$, and the lemmas added by the theory solver refute the existence of a satisfying vertex candidate, that C' is unsatisfiable. Thus, the theory solver needs to add this information lazily: If the Boolean model is complete and the given partial vertex candidate V is not a full vertex candidate of C', then it returns $\vee_{c\in C'} \neg c$.

Another advantage of this embedding is that decisions can be more interleaved with theory-specific heuristics. A simple observation is that if an equality is asserted to be true, then it can safely be added to the vertex candidate: Any equality satisfied under a vertex candidate is also tight under a vertex candidate - it is easy to see that then, a constraint in the vertex candidate can be switched with the equality without changing the solution set.

The disadvantage is that the SAT solver loses some information about the problem and is limited in its decisions that can be made.

4.4.4 Theory propagations

In order to save unnecessary computations in the SAT and theory solving process, we can combine the approaches before with generating additional theory lemmas guiding the SAT solver and hinting the theory solver.

A simple approach is to not only generate lemmas excluding certain combinations of constraints and selection variables that cause a conflict, but also generate positive lemmas explaining implications: Given a vertex candidate $\tilde{V}$ for a system C and a normalized $(M', \mathcal{P})' \leftrightsquigarrow (M_{(C\setminus V)\cup\tilde{V}}, \emptyset)$, then for any satisfied row $i \in [m]$ of M',

$$\left(\bigwedge_{\tilde{c}_j \in \tilde{V}_i} \tilde{c}_j\right) \to c_i$$

where $\tilde{V}_i := \{\tilde{c}_j \in \tilde{V} \mid j \neq i, f'_{i,j} \neq 0\}$ is valid and might prevent the SAT solver to make wrong decisions on constraints that have already been seen or help to detect conflicts.

We can go one step further: By introducing a variable $\mathit{satisfied}_c$ and adding $\mathit{satisfied}_c \to c$ for each constraint $c \in \mathit{Constraints}(\varphi)$ where φ is the input formula, we can write the lemma above as

$$\left(\bigwedge_{\tilde{c}_j \in \tilde{V}_i} \tilde{c}_j \right) \to \mathit{satisfied}_c.$$

By preventing the SAT solver from deciding upon $\mathit{satisfied}_c$, whenever $\mathit{satisfied}_c$ is set to *true*, then c is known to be true under the currently selected partial vertex candidate and thus, the theory solver can safely ignore c to save unnecessary computations. Furthermore, by adding

$$\mathit{satisfied}_c \to \neg \mathit{select}_c \text{ respectively } \mathit{satisfied}_c \to (\neg \mathit{select}_c^{<} \wedge \neg \mathit{select}_c^{>})$$

for every $c \in \mathit{Constraints}(\varphi)$, constraints known to be redundant are not considered for the selection of the vertex candidate.

Chapter 5

Discussion

5.1 Comparison with the general Simplex method

5.1.1 Similarities of the tableau representation

The tableau from Definition 4.2.1 is similar to the tableau in Equation (2.1) from the Simplex algorithm. However, there are some differences:

- While the Simplex algorithms maintains bounds on slack variables referring to the original system, our notation represents a transformed (equivalent) system by maintaining the relations $\sim_i$ and bounds b_i. The p_i columns track how the transformed system emerged from the original system.

  This is mainly a different interpretation: In the Simplex algorithm, the bounds could be tracked in a similar way; or our procedure could be modified to maintain a set of bounds on the p_i that are tight or evaluated as soon as the row corresponding to p_i is pivoted respectively becomes redundant.

- Furthermore, Simplex maintains a full assignment for all variables, while our procedure defers computing the assignment to later. This has the advantage that in case of an underdetermined system, our procedure can compute all satisfying assignments; the Simplex method sets all undetermined original variables to 0, instead of handling them symbolically.

- In the Simplex algorithm, slack variables and original variables are treated equally, thus every row can be pivoted with any column. In our method, every row is associated with a constraint of the original system and can only be pivoted with a column corresponding to an original variable once.

- In the Simplex algorithm, the notion of a pivot step is slightly different from our method: In the Simplex method, a pivot element is chosen and made the only non-zero entry in its column; we define an ordering on the pivot elements and do not change the rows of smaller pivot elements during a pivot step. This idea originates from the forward direction of the Gaussian elimination method.

  With some effort, we could lift this restriction together with the previous one. However, our embedding into DPLL(T) would not be able to exploit this possibility.

Thus, a single non-incremental theory call in our procedure with a system C and vertex candidate $\tilde{V}$ as input could be viewed as an instance of the Simplex algorithm: From the initial tableau (corresponding to M_C), we choose pivot elements such that all slack variables corresponding to $\tilde{V}$ become non-basic (the result is corresponding to a normalized (M',$\mathcal{P}'$) such that $M_{(C\setminus V)\cup\tilde{V}} \rightsquigarrow M'$).

5.1.2 Heuristic similarities

In Section 4.1.1, we already discussed how the general Simplex method behaves on conjunctions as well as its embedding into DPLL(T).

Let us deliberate how our implementation acts on conjunctions: First, observe that by learning the Simplex-based lemmas as described in Section 4.3.2, we exclude the same "neighbouring" tableaus as the Simplex algorithm does. Observe further that Simplex also guarantees an improvement in every step by choosing a suitable "neighbouring" tableau by a pivot step; at first sight, this seems to be a clear advantage over our proposed adaption. However, the activity-based scheme implemented by `Minisat` to schedule decisions (as described in Section 2.1.1.2) resembles this kind of locality by some degree. Still, there is only a rough correspondence, and our implementation does not exploit improvements achieved by pivot heuristics from the Simplex algorithm.

While it is unlikely to be competitive on purely conjunctive problems, the hope is that our procedure has advantages in case of highly combinatorial problems due to the interleaving with the Boolean reasoning.

5.1.3 Differences in the notion of a decision

In the Simplex method, a non-basic variable can be pivoted with any basic variable without the need to touch the other non-basic variables, even if they were made non-basic afterwards. In fact, the order in which variables are made non-basic does not have an influence on the result. Thus, one can say that the decisions on which variables are made non-basic are non-chronological.

In contrast, in our current approach, we do not make use of this observation despite the heuristic similarities: Decisions in DPLL are chronological and all Boolean propagations may depend on all previous decisions. If the decision for a variable is reverted and its value is switched, all propagations and decisions made after this decision need to be backtracked - this applies to the selection variables as well. Thus, when we just exchange a constraint in the vertex candidate with another, we need to backtrack and recompute the selection of the partial vertex candidate entirely.

5.2 Implementation

5.2.1 `SMT-RAT`

`SMT-RAT` [CKJ+15] is a modular SMT framework implementing modules for preprocessing and SAT solving as well as theory solvers for various logics. It is written in *C++* and maintained as open source project at the Theory of Hybrid Systems group at RWTH Aachen University.

The SAT solver module is an adaption of the `Minisat` SAT solver for less-lazy SMT solving. Moreover, the SAT module is modified such that custom variable ordering heuristics can be implemented, in particular theory specific orderings.

While its main focus is on *non-linear real arithmetic* problems, `SMT-RAT` also implements a Simplex-based LRA solver.

5.2.2 The `NewLRA` module

Our procedure is implemented by two `SMT-RAT` modules: a preprocessing module and a theory solver module. The preprocessing module passes the extended formula to the SAT solver, which in turn passes SMT-compliant subformulas to the theory solver module during SAT solving.

The implementation of the theory solver in its simplest form is straight forward from the presentation of the incremental methods presented in Chapter 4. For representing rational numbers with arbitrary precision, `GNU-MP` library [Fre00] is used.

The data structure for a tableau M is implemented as a two-dimensional array of rationals using $2n$ columns: This is possible as at any time, at most n columns of F differ from a standard vector; thus, only the columns in A and the columns in F differing from a standard vector need to be stored, the other columns can be stored implicitly. Moreover, many optimizations for efficiency are thinkable, e.g. A and F are mostly sparse and thus, maintaining only non-zero cells might safe some overhead; however, this optimizations have not been implemented for this thesis.

As already mentioned, `Minisat` assigns decision variables to *false* if they have never been assigned before. This might not be desired for selection variables as it counteracts the selection of a vertex candidate. Thus, the decision heuristic is altered to set the default initial value to *true* for the selection variables (while for the remaining variables, we stick to *false* as the default initial value).

5.3 Experimental results

The following variants of our procedure were tested:

NewLRA A less lazy solver based on the first version of the eager encoding described in Section 4.4.1 implementing all theory lemmas described in Section 4.3.2.

NewLRA-NewEnc As `NewLRA`, but based on the second version of the eager encoding described in Section 4.4.2.

This variant aimed to reduce combinatorial overhead in the vertex candidate selection by encoding more specific information, but introduces more variables.

NewLRA-PairwiseAtM As `NewLRA`, but using the pairwise encoding for *at-most-one-of* (see Section 4.4.1).

As the *at-most-one-of* is usually large and the pairwise encoding introduced more clauses than the binary encoding, this variant is expected to be inferior to `NewLRA`.

NewLRA-NoEnc As `NewLRA`, but instead of the eager encoding, the variant described in Section 4.4.3 is implemented, moving the responsibility for the vertex candidate selection to the decision heuristic of the SAT solver.

The aim of this variant is to reduce the overhead introduced by the large eager encodings.

NewLRA-NegSelVars As `NewLRA`, but the default initial value for selection variables is *false* (as for any other variable).

The aim is to illustrate the influence of small tweaks of the solver.

NewLRA-NoLemmas As `NewLRA`, but without the Simplex-based lemmas, thus only excluding the current (partial) vertex candidate on conflict.

This variant should illustrate the importance of extracting information from the current theory solver state.

NewLRA-TheoryProp As `NewLRA` with the theory propagations as described in Section 4.4.4.

The theory propagations are additional lemmas generated by the theory solver directing the SAT solver to avoid unnecessary conflicts in the theory.

NewLRA-FullLazy A full lazy version of `NewLRA`.

Full lazy means that the theory solver is called only when the SAT solver constructed a full Boolean model, in contrast to the less lazy variant where theory checks are performed after each completed Boolean constraint propagation.

NewLRA-EpsLowerBounds A basic variant based on the approach described in Section 3.4.2 where we also admit lower bounds on ε.

This variant is tested non-competitively for showing that this alternative approach also works correctly.

The solvers were executed on the *QF_ LRA* benchmark set from *SMT-LIB* on a *2.1 GHz AMD Opteron* CPU with a time limit of 5 minutes and a memory limit of 4 gigabytes. For comparison, we included the Simplex implementations in `Z3` and `SMT-RAT` (the latter is denoted by `Simplex` in the following); all `NewLRA*` and `Simplex` were executed without preprocessing.

A run can have the outcomes

- *sat* (correctly solved satisfiable instance),
- *unsat* (correctly solved unsatisfiable instance),
- *wrong* (wrong solver result),
- *timeout* (computation took longer than 5 minutes) and
- *memout* (computation exceeded 4 gigabytes of memory).

The outcomes are shown in Figure 5.1 and Table 5.1. None of the variants produced any wrong outputs and thus, there is strong evidence that the implementations are correct.

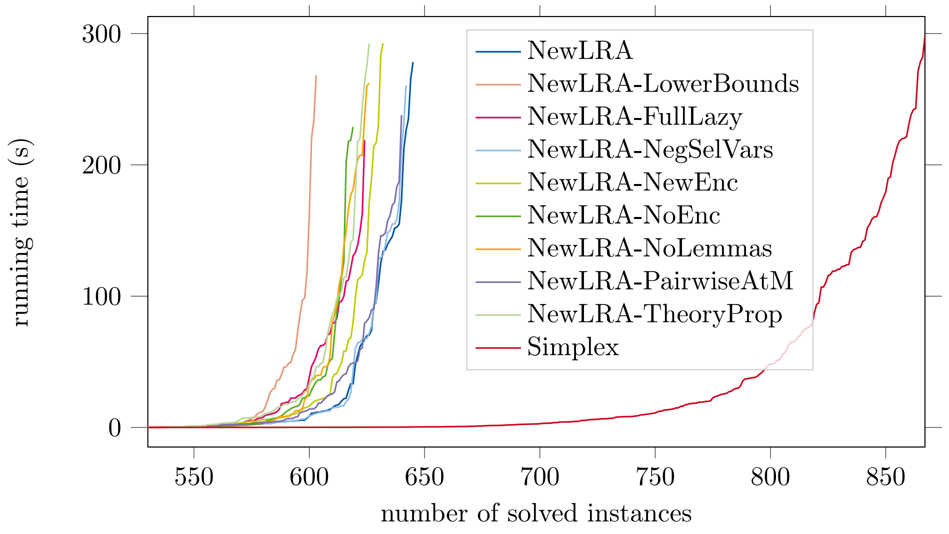


Figure 5.1: Performance profile.

	NewLRA	NewLRA-LowerBounds	NewLRA-FullLazy	NewLRA-NegSelVars
sat	372	361	365	372
unsat	273	242	259	270
wrong	0	0	0	0
timeout	924	965	944	932
memout	79	80	80	74

	NewLRA-NewEnc	NewLRA-NoEnc	NewLRA-NoLemmas	NewLRA-PairwiseAtM
sat	366	353	370	370
unsat	266	266	256	270
wrong	0	0	0	0
timeout	1012	1009	946	762
memout	4	20	76	246

	NewLRA-TheoryProp	Simplex	Z3
sat	367	499	885
unsat	259	368	609
wrong	0	0	0
timeout	951	776	151
memout	71	5	3

Table 5.1: Number of instances by outcome.

5.3.1 Performance of **NewLRA** compared to **Simplex**

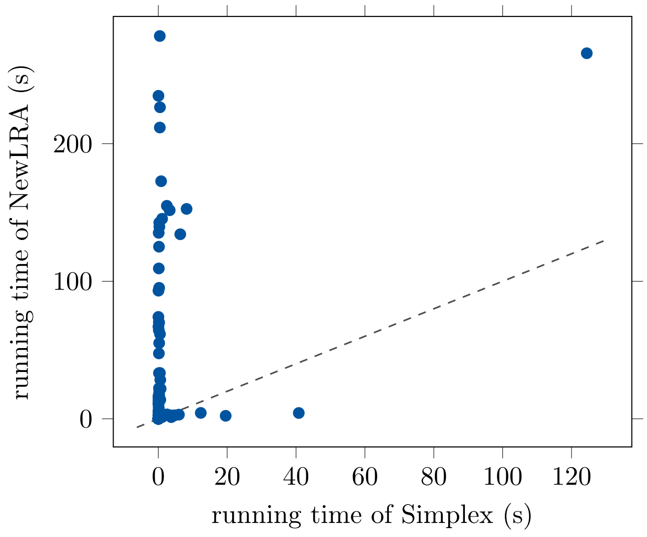


Figure 5.2: Comparison of running times on instances solved by both solvers.

For about the first 520 examples, `NewLRA` can compete with the `Simplex` solver - these instances are solved in less than 100 milliseconds and can be assumed to be almost trivial. From there, the running times are rapidly growing such that only 125 additional instances are solved, which are mostly trivial for the `Simplex` solver, as seen in Figure 5.2.

	Type		Simplex		NewLRA	
	sat	unsat	sat	unsat	sat	unsat
set						
2017-Heizmann-UltimateInvariantSynthesis	3	50	0	0	0	0
DTP-Scheduling	91	0	82	0	28	0
LassoRanker	267	143	0	1	0	0
TM	24	1	11	1	1	1
check	1	1	1	1	1	1
clock_synchro	0	36	0	23	0	9
keymaera	0	21	0	21	0	21
latendresse	16	2	0	0	0	0
meti-tarski	338	150	338	150	338	150
miplib	22	20	2	3	0	1
sal	11	96	10	84	4	58
sc	108	36	46	9	0	0
spider_benchmarks	0	42	0	42	0	24
tropical-matrix	5	0	0	0	0	0
tta_startup	24	48	3	28	0	8
uart	36	37	6	5	0	0

Table 5.2: Number of solved instances from the *SMT-LIB's QF_LRA* benchmark collection by set and type.

Furthermore, with the given data, it is hard to measure a certain structural property on which NewLRA performs better or disproportionately worse than Simplex given the number of solved instances by test set in Table 5.2. Similarly, drawing conclusions from single measures like density (the ratio of number of variable occurrences to $n \cdot m$), the ratio of constraints to variables, the average or maximal size of clauses as measures for Boolean structure and the number of vertex candidates (m choose n where m is the number of constraints and n the number of theory variables) is like reading future off coffee ground - the performance relies heavily on the test set and its structure.

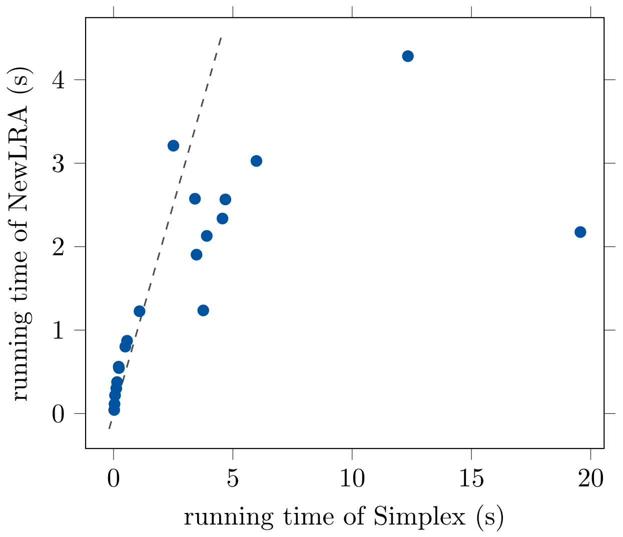


Figure 5.3: Comparison of running times on the *SAL/windowreal/windowreal-no_t_deadlock-** benchmarks.

However, almost all instances where NewLRA is superior to Simplex stem from a single class of instances from the *SAL benchmark suite*, and the performance seems not to be accidental, as Figure 5.3 shows. Similar to the other sets *meti-tarski*, *DTP-Scheduling* and *spider_benchmarks* for which NewLRA solves a good amount of instances, the number of theory variables n is relatively small. However, a relatively small n is not a sufficient condition but seemingly a necessary one for benchmarks to be solved; the reason is most likely the combinatorial blow up - remember that the number of possible vertex candidates is the binomial coefficient $\binom{m}{n}$. Apart from that, it is not clear why NewLRA performs better than Simplex on these instances.

More insights could be gained by systematic testing on sets with an isolated property. Especially the influence of a complex Boolean structure needs to be investigated, as this was the main motivation for the new procedure.

5.3.2 Analysis of **NewLRA**

Theory vs SAT running times Compared to the Simplex solver, NewLRA shifts complexity to the SAT solver, as the relative running time of the theory solver tends to be smaller compared to the Simplex solver, see Figure 5.4.

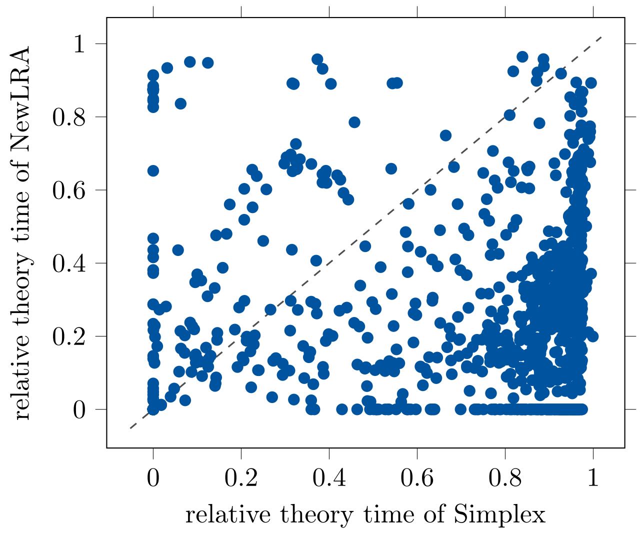


Figure 5.4: Ratio of theory solver to running time without parsing and encoding on all instances.

Encoding size It should be mentioned that the eager encoding of the selection variable semantic grows rapidly and for large instances, its generation can take several minutes (though the latter could be mitigated technically to some extend) and dominate the set of clauses by far (compared to original and learned clauses). All instances solved by `NewLRA` have a relatively small encoding, with few exceptions as the family of instances from the SAL benchmark suite, see Figure 5.5.

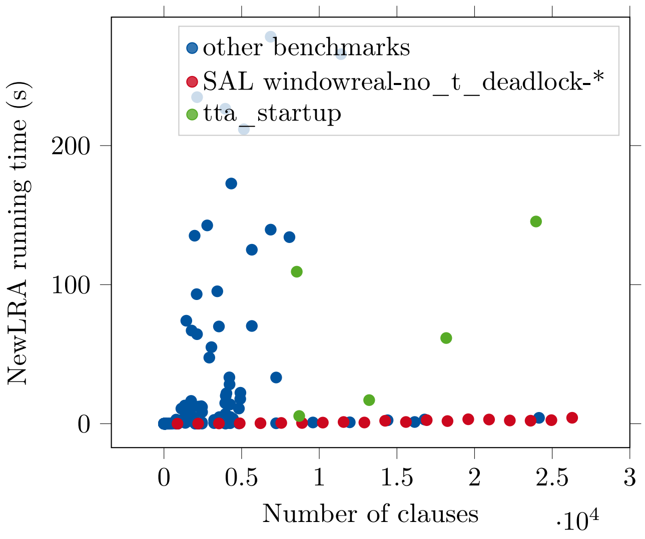


Figure 5.5: Running time on solved instances by number of initial (original and encoding) clauses.

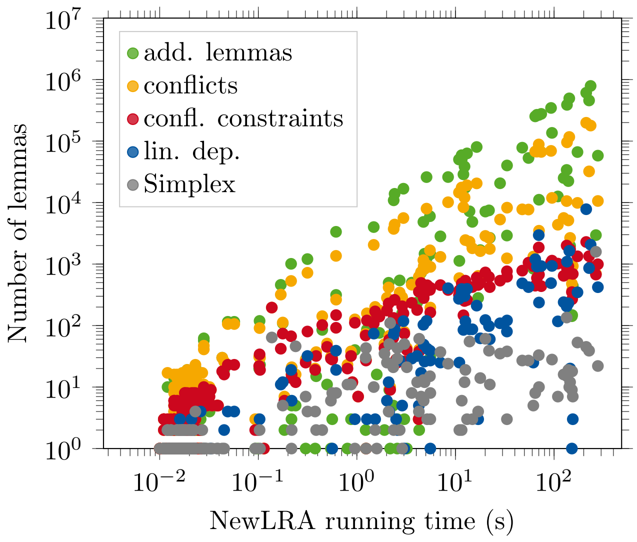


Figure 5.6: Learned lemmas by type on solved instances.

Learned lemmas Figure 5.6 shows the number of lemmas learned: *add. lemmas* are the additional Simplex-based lemmas excluding "neighbouring" tableaus, *conflicts* are constraints implied to be conflicting by a partial vertex candidate, *inf. constraints* are the Simplex-based lemmas excluding a set of constraints and *lin. dep.* are the learned linear dependencies between selection variables. Note that lemmas of the latter type are only generated if no other conflict occurred - otherwise, this number would overtop the other types.

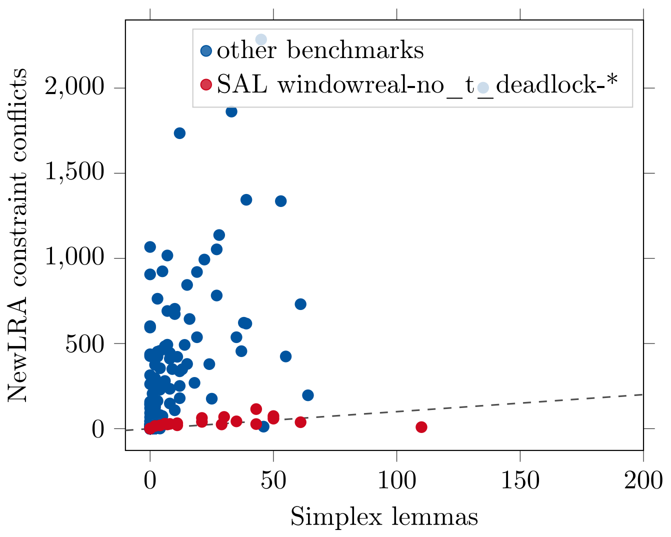


Figure 5.7: Number of `Simplex` lemmas vs constraint conflicts in `NewLRA` on instances solved by both solvers.

Note that the *inf. constraints* lemmas correspond to the `Simplex` lemmas. Thus, as `Simplex` needs less lemmas to solve the instance, the Simplex algorithm seems also to be heuristically more successful. Interestingly, this picture is completely different for the family of instances where `NewLRA` outplays `Simplex`, as shown in Figure 5.7.

Furthermore, the size of the lemmas generated by `NewLRA` are small: For the half of all instances, the number of literals in a lemma divided by the number of asserted constraints (excluding selection variables) is below 0.014; and for 75% of the instances, this value is below 0.15. Thus, these lemmas generalize well.

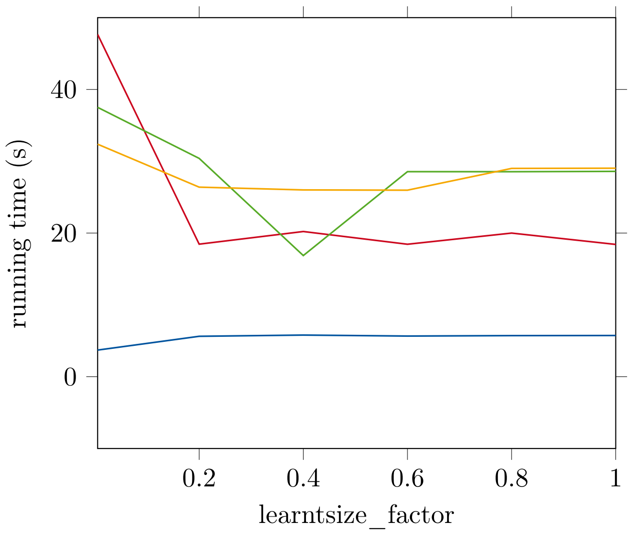


Figure 5.8: Performance of some instances dependent on *learntsize_factor*.

SAT solver internals Given the high number of clauses which lead to higher load on the SAT solver, the clause deletion feature could help to make the number of clauses tractable. The impact of this feature relies heavily on the limit of learned clauses. As this limit is the initial number of clauses (including the eager encoding) passed to the SAT solver times some factor *learntsize_factor*, this feature is essentially disabled for the default setting *learntsize_factor* = 1. Due to the disproportionate differences in the size of the encoding, it is impossible to find an optimal *learntsize_factor*, as indicated in Figure 5.8.

5.3.3 Performance of `NewLRA-NoEnc` compared to `NewLRA`

Given the large encoding of the semantics of the selection variables and the observation that a large portion of the running time of `NewLRA` is spent in the SAT solver, one would expect that `NewLRA-NoEnc` performs better. While this is true regarding memory consumption, as indicated by the lower number of *memouts* (see Table 5.1), the contrary is the case for the running times and the number of solved instances.

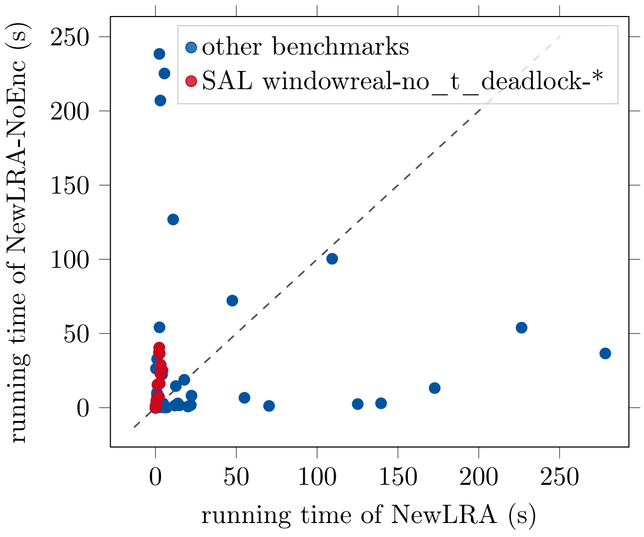


Figure 5.9: Comparison of running time on instances solved by both solvers.

First, the solvers seem to behave completely different on the given instances, regarding relative theory time or overall running time (see Figure 5.9). Conspicuously, the promising instance class from the SAL benchmarks got disproportionately slower. This might be a hint that the structural information that was eagerly encoded in `NewLRA` is missing in `NewLRA-NoEnc` or the restriction of the decision heuristic makes it inferior. However, drawing a conclusion is hard as the search for a justification was unsuccessful.

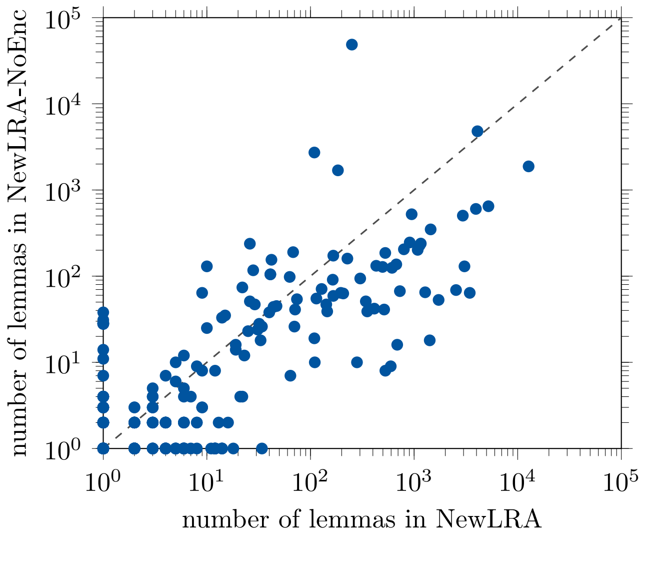


Figure 5.10: Lemmas learned on instances solved by both solvers, excluding linear dependencies.

Interestingly, `NewLRA-NoEnc` tends to need less lemmas (see Figure 5.10) than `NewLRA` on the instances that it solved. The obvious conclusion would be that `NewLRA-NoEnc` is more efficient than `NewLRA` - but reality is different. Also, implementation specific slowdown cannot justify the worse performance.

5.3.4 Further variants of **`NewLRA`**

The differences in the performance between the solver variants are not huge; sometimes they differ in a handful of solved instances - which could well be caused one solver luckily guessing right. Thus, these results should be interpreted carefully. However, we gain a few insights from Figure 5.1 and Table 5.1:

- The encoding could indeed make a difference, as shown by the differences in the performance of `NewLRA`, `NewLRA-NewEnc`, `NewLRA-PairwiseAtM` and `NewLRA-NoEnc`. Still, drawing a conclusion beyond the aspects already discussed is hardly possible.
- Incremental solving performs better than full lazy solving, as `NewLRA` outplays `NewLRA-FullLazy`. The incremental approach interleaves Boolean and theory solving by propagating knowledge from the theory during the Boolean model construction, which seems to be worth it.
- Also, the additional Simplex-inspired lemmas pay off, as leaving them out in `NewLRA-NoLemmas` is slower.
- The theory propagations in `NewLRA-TheoryProp` are not worth the effort. A possible reason could be that they cannot be applied that often by the SAT solver as they are only generated for constraints that are already asserted by the SAT solver.
- The result of `NewLRA-NegSelVars` does not allow drawing a conclusion of the influence of the altered decision heuristic regarding the selection variables.
- The variant `NewLRA-LowerBounds` stems from a by-product of our extension of vertex candidates to systems with strict inequalities. Although it works correctly, it needs more effort to detect conflicts and does not allow learning the Simplex-like lemmas; this is why this path has not been followed further.

5.3.5 Summary

As outlined multiple times, drawing a conclusion is hard. However, the fact that there is a specific class of instances where the new algorithm is systematically superior to the Simplex algorithm could be an indication that our algorithm is able to learn and generalize more knowledge than the Simplex procedure. Nonetheless, given the large number of learned lemmas, especially regarding the constraint conflicts with a correspondence to the Simplex procedure, the heuristic defining the ordering in which the tableaus are checked might be inferior to the Simplex.

Also, it turned out that learned lemmas and the eager encoding do add valuable knowledge to speed up the solving process, as replacing the eager encoding or disabling the Simplex-inspired lemmas performed worse. Also, it is worth noting that an incremental consistency check helps directing the solving process.

It is also likely that the large number of learned clauses is intractable for the SAT solver, as indicated by the portion of running time spent on the SAT solver and the role of the clause deletion heuristic. The Simplex procedure has the major advantage that it tracks progress implicitly - while the new procedure progresses by learning lemmas explicitly. Managing them is technically hard and is a major hurdle for a competitive implementation.

5.4 Future work

5.4.1 Altering the decision heuristic

Figure 5.7 indicates that the decision heuristic in the new procedure is inferior to the Simplex heuristic regarding reducing the number of necessary lemmas to conclude satisfiability respectively unsatisfiability.

As discussed, the employed activity-based decision heuristic resembles vaguely the Simplex heuristic by making a compromise between local search and a more dynamic exploration. It is be arguable that for the selection variables, local search might be more efficient as it is closer to the Simplex heuristic. Thus, a separate ordering for original and selection variables might be desirable.

Moreover, having separate heuristics, pivot rules from the Simplex algorithm could be adapted for the decision heuristic.

5.4.2 Learning from redundant rows

So far, only lemmas excluding “neighbouring” tableaus that were generated based on conflicting rows. However, symmetric lemmas from redundant rows can be learned: Restricting the relation to $\leq$ for simplicity, Lemma 4.3.4 can be extended in the sense that if $\tilde{V} \cup \{p \sim b\}$ is satisfiable, then $\tilde{V}' \cup \{p_k \sim b_k\}$ is conflicting if and only if $f_k > 0$.

Moreover, if any dependent row i is tight (that is, $b'_i = 0$), then swapping the corresponding constraint c_i with any constraint $c_j \in V$ being an origin of row i (that is, $f'_{i,j} \neq 0$) leads to essentially the same solution - known as *degenerate* solution. Formally, if $\tilde{V} \cup \{p_i = b_i\}$ is satisfiable, then for $V' = (V \setminus \{p_j \sim_j b_j\}) \cup \{p_i \sim_i b_i\}$ it holds $C \cup \tilde{V} \equiv C \cup \tilde{V}'$. Thus, one of these vertex candidates can safely be excluded.

5.4.3 Specializations of the SAT solver

As we move a huge amount of information to the SAT solver, it needs to manage its clause set efficiently.

First, our experiments have shown that, by varying *learntsize_factor*, on long running examples, the clause set is never reduced, while on some others, it happens frequently; on the latter examples, the clause deletion heuristic does influence the performance of an instance positively or negatively. Currently, the initial upper limit on the number of learned clauses depends on the number of original and encoding clauses. An obvious idea is to make it depend only on the original clauses, or on another property.

Second, observe that our procedure learns different kinds of clauses: Lemmas excluding linear dependent sets of constraints (for selecting a linear independent vertex candidate), lemmas implying the (un-)satisfiability of a single constraint together with a partial vertex candidate or lemmas excluding sets of original constraints. Depending on its type, a lemma has a different importance for the solving process: While the lemmas excluding sets of constraints are the most valuable, linear dependence could be more of local importance. Hence, introducing separate conditions for removing lemmas from the clause set might help to keep the SAT solving process fast while retaining relevant information.

5.4.4 Theory propagations

Theory propagations are additional lemmas generated by the theory solver to pass information to the SAT solver. This could prevent wrong decisions, as theory lemmas allow for more propagations during SAT solving. Especially for the `NewLRA-NoEnc` variant, additional lemmas could compensate the missing information that the other variants receive by the eager encoding of the selection variable semantics.

5.4.4.1 Propagating truth values of constraints

A possible way is to extend the approach from Section 4.4.4. There, we add lemmas implying constraints to be satisfied by a partial vertex candidate. However, we only consider the currently asserted constraints for generating such lemmas; an extension for considering all constraints occurring in the input formula allows propagations of truth values for a constraint without waiting until the SAT solver decided or propagated the constraint. Clearly, it causes additional effort to evaluate all constraints in the theory solver, which is why we forwent this approach in our implementation. However, experiments have shown that the theory solver is not the bottleneck and the additional effort might pay off.

5.4.4.2 Look-ahead for extending a partial vertex candidate

So far, we detect the unsatisfiability of a system C at a partial vertex candidate $\tilde{V}$ only after the normalized tableau corresponding $(C \setminus V) \cup \tilde{V}$ has already been computed. The idea is now, to allow extending a non-conflicting partial vertex candidate by a constraint only if the corresponding extension is still a non-conflicting partial vertex candidate. To determine the set of those extensions, we make use of the following observation:

Remark 5.4.1. *Let C be a weak system of linear constraints over $(\mathcal{U}, <)$ and $\tilde{V}$ a partial vertex candidate of C_w. Consider a normalized $M' \leftsquigarrow M_{(C_w \setminus V) \cup \tilde{V}}$. Given two non-redundant dependent rows i,$j \in [m]$ in M', we define $\tilde{V}' = \tilde{V} \cup \{\tilde{c}_i\}$ and let $M'' \leftsquigarrow M_{(C_w \setminus V') \cup \tilde{V}'}$ be normalized. Then $\boldsymbol{a''_{j,-}} = 0 \iff \exists h \in \mathcal{F}.\ \boldsymbol{a'_{j,-}} = h \cdot \boldsymbol{a'_{i,-}}$.*

Simply speaking, whenever we extend a partial vertex candidate $\tilde{V}$ of a system C by a constraint $\tilde{c}_i$, the rows j that will be redundant after making row i a pivot are exactly the ones for which $\boldsymbol{a'_{j,-}} = h \cdot \boldsymbol{a'_{i,-}}$ for some h. Furthermore, such a row j will be satisfied after pivoting row i if and only if $0 \sim_j b''_j$ respectively $0 \sim_j b'_j - h \cdot b'_i$ is fulfilled. Thus, we got a condition for constraints c_i that cause an immediate conflict of a row j after adding it to the partial vertex candidate. We could exclude those selections by generating the lemma

$$\left(\bigwedge_{\tilde{c}_k \in \tilde{V}_i \cup \tilde{V}_j} \tilde{c}_k \wedge \tilde{c}_i \right) \to \neg c_j$$

where $\tilde{V}_i := \{\tilde{c}_k \in \tilde{V} \mid k \neq i, f'_{i,k} \neq 0\}$ and $\tilde{V}_j$ defined analogously.

We got even further: We partition the dependent non-redundant rows of M' into sets G'_p where p are a linear $\mathcal{F}$-combinations of X such that for each row $i \in G'_p$, $\boldsymbol{a'_{i,\_}} \cdot \boldsymbol{x}$ is a multiple of p. Then, each row $i \in G'_p$ induces a bound on p and by pairwise comparison of all bounds, we can compute all pairs causing a conflict of the kind described above.

Furthermore, observe that whenever $\tilde{V}$ is extended, for the corresponding tableau M'', $i,j \in G'_p \implies i,j \in G''_{p'}$ for some p,p'; in other words by extending a vertex candidate, groups might be merged, but never split. Thus, once two rows are a multiple of each other, they will remain multiple of each other for any extension of the current vertex candidate. It can be shown that for two $i,j \in G'_p$ with conflicting lower and upper bounds, we can generate the lemma

$$\left(\bigwedge_{\tilde{c}_k \in \tilde{V}_i \cup \tilde{V}_j} \tilde{c}_k \right) \to \neg(c_i \land c_j).$$

5.4.5 Exploiting non-chronological decisions

In Section 5.1.3, we already discussed a downside of our approach - that DPLL needs chronological decision while the Simplex allows changing decisions non-chronologically. A possible improvement would be to change our tableau data structure to match more with the one of the Simplex algorithm as discussed in Section 5.1.1; this would allow the direct exchange of a constraint in the vertex candidate with a dependent constraint. While mapping the notion of exchanging a non-basic with a basic variable in Simplex to the notion of a decision in DPLL would require the SAT solver to maintain complete implication graph - which is possibly practically infeasible - we propose the following compromise, which we call *lazy backtracking*:

The theory solver (as in Algorithm 9) is adapted such that the constraints $c \in C_-$ removed by the SAT solver are not removed from the tableau. Instead, they remain in the tableau until its negation $\neg c$ is in the set C_+ of added constraints. Similarly, we proceed with the selection variables in S_- and S_+ respectively.

To be conform with the DPLL(T) framework, for each occurring conflict in the tableau, it needs to be checked if its origins are currently asserted by the SAT solver: If the conflict stems from the set of currently asserted constraints and the selected vertex candidate, it is returned as infeasible subset. Otherwise, if the conflict depends on assertions that were already removed by the SAT solver, the conflict is added as a theory propagation, which is inserted into the clause set of the SAT solver.

5.4.6 An alternative embedding into MCSAT

Throughout working on this approach, it turned out that some ideas fit better into another solving framework, the *model-constructing satisfiability calculus (MCSAT)* [DMJ13].

5.4.6.1 MCSAT

In contrast to the DPLL(T) framework where the SAT solving and theory solving are done in a separated and alternating fashion, the MCSAT framework aims to solve both the Boolean and the theory level simultaneously.

In MCSAT, Boolean reasoning is done as in DPLL. In addition to Boolean decisions, also values of theory variables can be decided that are *consistent* with the Boolean abstraction, that is, the *value* of a constraint under the theory assignment does not contradict with the truth value of its abstraction. As a theory assignment might imply truth values on the constraints, the Boolean reasoning is *directed* by the theory assignment in the sense that the implied truth values are propagated to the Boolean level.

Whenever the current theory assignment α should be extended by a variable x, a method *assign* is invoked which either returns a value for x such that the extended assignment is consistent with the Boolean abstraction or returns a set of constraints C for which α cannot be extended for x. This conflict is passed to the *explain* function which returns a lemma that is conflicting under α. This lemma is then backtracked analogously to a conflict clause in the Boolean reasoning.

Note that the *explain* function is allowed to introduce new constraints; however, to ensure termination, it needs to fulfil the so called *finite basis property* that all constraints must stem from a finite set dependent on the set of initial constraints.

MCSAT is mainly used for solving non-linear real arithmetic, the logic with constraints over polynomials, and is the basis for the most successful LRA solvers, most prominently Z3 [JDM12].

5.4.6.2 Embedding Simplex into MCSAT

Our method already feels similar to MCSAT, especially the encoding-less variant as described in Section 4.4.3 and gets even more similar by combining it with the theory propagations described in Section 5.4.4. However, MCSAT seems to define a more suitable framework as it defines explicit interfaces that allow avoiding the creation of lemmas expressing simple facts like theory propagations.

For our purposes, we generalize the theory assignments in MCSAT such that instead of theory variables, linear $\mathcal{F}$-combinations of X are assigned to values in $\mathcal{U}[\varepsilon]$. The *assign* function produces an assignment corresponding to a partial vertex candidate of the set of currently asserted constraints while maintaining consistency using the look-ahead described in Section 5.4.4.2. The direction by the theory, that is propagating truth values from the theory to the Boolean level, is analogous to the theory propagations described in Section 5.4.4.1. The *explain* function can be chosen trivially by returning the set of conflicting constraints together with the relevant subset of the current assignment; for the latter, special literals need to be introduced representing a theory assignment. Note that the finite basis property still holds by the choice of the *assign* function, which selects a vertex candidate of which only finitely many exist.

Advantages of this approach over the current one would be that the solver does not chose unsatisfying vertex candidates while being able to extract possibly more information about unsatisfying partial vertex candidates thanks to the look-ahead described in Section 5.4.4.2. Furthermore, a lot of knowledge can be propagated implicitly to the SAT solver without the need of generating lemmas, which relieves the SAT solver. It should be noted that the *value*, *assign* and *explain* functions can share information by maintaining a tableau in parallel to the SAT solver. However, again, the performance will be highly dependent on the decision heuristic (see Section 5.4.1). Moreover, there might be ways to exploit the non-chronological nature of the Simplex algorithm (see Section 5.4.5).

Chapter 6

Conclusion

This thesis gave another view on the Simplex method unfastened from usual presentations and applications. The reformulation of the fundamental theorem of linear programming in Theorem 2.3.1 allowed us to extend the theory behind the Simplex algorithm for strict inequalities, leading to an appealing analogous formulation in Corollary 3.3.4.

From there, a similar procedure - a tribute to the Gaussian elimination procedure - has been developed that allows "jumping" to a specific vertex candidate in Section 4.2 respectively Section 4.3 and extracting information about the original problem from the results. Several embeddings into the DPLL(T) framework were presented in Section 4.4, which base on the idea of encoding some information from the Simplex algorithm in SAT formulas to retain information throughout the solving process that gets otherwise lost in the usual embedding of the Simplex algorithm into DPLL(T).

The procedure was compared to the usual Simplex adaption for DPLL(T) in Section 5.1 and all its variations developed along this thesis were evaluated in Section 5.3. Although it turned out that the developed procedure does not scale well, there were some indications of some structural advantages for some cases; however, more investigations are needed before drawing a conclusion.

Finally, unexplored ideas were presented in Section 5.4 - among it another novel approach based on MCSAT developed from the ideas of this thesis.

Although the experimental results were below expectations, the ideas were worth trying out, not least because of the insights gained during the work on this thesis. It is thinkable that some of these ideas lead to another impartial attempt or can be reused for improving existing solvers.

To come back to the sentences that started this thesis, let me cite a well known adage that I find appropriate in this context:

"There ain't no such thing as a free lunch." [Key07]

Bibliography

[Axl97] Sheldon Jay Axler. *Linear Algebra Done Right*, volume 2 of *Undergraduate Texts in Mathematics*. Springer, 1997.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*, volume 185. IOS Press, 2009.

[BKHG18] Curtis Bright, Ilias Kotsireas, Albert Heinle, and Vijay Ganesh. Enumeration of complex Golay pairs via programmatic SAT. *arXiv preprint arXiv:1805.05488*, 2018.

[BST$^+$10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, volume 13, page 14, 2010.

[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 *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing*, volume 9340 of *LNCS*, pages 360–368. Springer, 2015.

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

[Dan98] George Bernard Dantzig. *Linear Programming and Extensions*, volume 48. Princeton University Press, 1998.

[DDM06] Bruno Dutertre and Leonardo De Moura. Integrating simplex with DPLL(T). *Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-06-01*, 2006.

[DLL61] Martin Davis, George Logemann, and Donald W Loveland. *A Machine Program for Theorem-proving.* New York University, Institute of Mathematical Sciences, 1961.

[DMJ13] Leonardo De Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *LNCS*, pages 1–12. Springer, 2013.

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

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[Fou27] JBJ Fourier. Analyse des travaux de l'académie royale des sciences pendant l'année 1824. *Partie mathématique*, 1827.

[Fre00] Free Software Foundation. The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`, 2000.

[Hef18] Jim Hefferon. *Linear Algebra*. Virginia Commonwealth University Mathematics, third edition, 2018.

[Heu09] Marijn JH Heule. Solving edge-matching problems with satisfiability solvers. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.148.8746`, 2009.

[HKM16] Marijn JH Heule, Oliver Kullmann, and Victor W Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016.

[HN13] Steffen Hölldobler and Van-Hau Nguyen. An efficient encoding of the at-most-one constraint. *Technical Report 1304. Technische Universitäat Dresden*, 2013.

[JDM12] Dejan Jovanović and Leonardo De Moura. Solving non-linear arithmetic. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *LNCS*, pages 339–354. Springer, 2012.

[Key07] Ralph Keyes. *The Quote Verifier: Who Said What, Where, and When*. St. Martin's Griffin, 2007.

[Kha79] Leonid G Khachiyan. A polynomial algorithm in linear programming. *Dokl. Akad. Nauk SSSR*, 5:1093–1096, 1979.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2016.

[Lem54] Carlton E Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1(1):36–47, 1954.

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

[LW93] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The computer journal*, 36(5):450–462, 1993.

[LY84] David G Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*, volume 228 of *International Series in Operations Research & Management Science*. Springer, 1984.

[MMZ$^+$01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, pages 530–535. ACM, 2001.

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

[MSS96] João P Marques-Silva and Karem A Sakallah. GRASP - A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227. IEEE/ACM, 1996.

[SR12] Igor R Shafarevich and Alexey O Remizov. *Linear Algebra and Geometry.* Springer, 2012.

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pages 466–483. Springer, 1983.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, 1997.

[WM95] David H Wolpert and William G Macready. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, The Santa Fe Institute, 1995.

[WM97] David H. Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[Wol96] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

[ZBH$^+$17] Edward Zulkoski, Curtis Bright, Albert Heinle, Ilias Kotsireas, Krzysztof Czarnecki, and Vijay Ganesh. Combining SAT solvers with computer algebra systems to verify combinatorial conjectures. *Journal of Automated Reasoning*, 58(3):313–339, 2017.

[Zie12] Günter M Ziegler. *Lectures on Polytopes*, volume 152 of *Graduate Texts in Mathematics.* Springer, 2012.