

The present work has been carried out at the LuFG Theory of Hybrid
Systems

BACHELOR OF SCIENCE THESIS

ON THE MECHANISMS OF SIMPLEX HEURISTICS

Jonas Spang

Examiners:

Prof. Dr. Erika Ábrahám

Prof. Dr. Thomas Noll

Additional Advisor:

Jasper Nalbach, M. Sc.

Aachen, 4.8.2022

Abstract

General Simplex [DdM06] is a variation of the well known Simplex algorithm [Dan63] used as a theory solver within the DPLL(T) framework for solving *satisfiability modulo theories (SMT)* formulas. The algorithm decides the satisfiability of a conjunction of constraints from the theory of *linear real arithmetic* by first constructing a *tableau* and then executing *pivot steps* on it until it detects either a *conflict* or a *satisfying assignment*. If multiple pivot steps are possible, a *heuristic* is used to decide which pivot operation should be performed. This thesis examines the behavior of these Simplex heuristics when implemented in the SMT solver SMT-RAT [CKJ⁺15], developed at the *Theory of Hybrid Systems Group* at RWTH Aachen University. We focus on four of these heuristics: *Bland's rule*, the *difference-based* heuristic, the *conflict-based* heuristic and *sum of infeasibilities*. We give a proof of termination for *Bland's rule*, implement the four heuristics as well as a modular framework to simplify exchanging one heuristic for another and perform extensive testing to find the cause for the surprising similarity of the four heuristics performance-wise. The results suggest that the test setup leaves little room for optimization in terms of Simplex heuristics.

Erklärung

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

Jonas Spang
Aachen, den 4. August 2022

Acknowledgements

Writing this thesis was a long and difficult process and I would like to thank those who have helped me along the way. My thanks go to Prof. Dr. Erika Ábrahám for the discussions and for taking the time to look at preliminary versions of my thesis and provide feedback, to Jasper Nalbach for helping me through various technical problems and to Prof. Dr. Thomas Noll for agreeing to be my second adviser. Finally, I want to give a huge thanks to my family and friends for always supporting and encouraging me when I was struggling. I wouldn't have succeeded without you.

Contents

1	Introduction	9
2	Preliminaries	11
2.1	SAT and SMT	11
2.2	DPLL and DPLL(T)	13
2.3	Linear real arithmetic	16
2.4	General Simplex	17
2.5	SMT-RAT	27
3	Simplex heuristics	29
3.1	Bland's rule	29
3.2	Difference-based	33
3.3	Conflict-based	33
3.4	Sum of infeasibilities	34
4	Implementation	39
4.1	Overview	39
4.2	Making pivot selection more modular	39
4.3	The class PivotStrategy	40
4.4	Difficulties with sum of infeasibilities	41
5	Evaluation	43
5.1	The setup	43
5.2	Initial Results	43
5.3	Two examples to show divergence	45
5.4	Going into detail	49
6	Conclusion	59
6.1	Summary	59
6.2	Discussion	59
6.3	Future work	60
	Bibliography	61

Chapter 1

Introduction

Satisfiability modulo theories (SMT) is a generalization of the *Boolean satisfiability problem (SAT)* from propositional to first-order logic. Instead of deciding the satisfiability of Boolean formulas, SMT solvers decide the satisfiability of Boolean combinations of constraints from some *first-order theory*. SMT solvers play an important role in the industry, with practical applications including software and hardware verification, test case generation, planning and optimization [MB11, KS16, NOT06].

In order to decide the satisfiability of formulas from the first-order theory of *linear real arithmetic*, many SMT solvers utilize *general Simplex* [DdM06], an adaptation of the famous *Simplex* algorithm [Dan63]. General Simplex takes a set of *linear constraints*, i.e. linear equalities, weak inequalities or strict inequalities, as input and decides whether the conjunction over these linear constraints is satisfiable. To do this, it constructs a *tableau* and an *assignment* and then performs a series of *pivot operations* until the assignment satisfies all input constraints or a *conflict* is detected in the tableau. At the start of each pivot operation, general Simplex has to decide which changes are going to be applied to the tableau and the assignment. Since there is usually no clear indication which decision is the best in terms of performance, general Simplex uses a *heuristic* to try and maximize performance as much as possible. In this thesis, we take a closer look at four of these heuristics (*Bland's rule*, the *conflict-based* heuristic, the *difference-based* heuristic and *sum of infeasibilities*) and examine their influence on the overall performance when implemented in the SMT solver SMT-RAT [CKJ⁺15]. The following contributions are made in this thesis:

- a proof of termination for *Bland's rule* is presented;
- the four heuristics are implemented in the `LRAModule` of SMT-RAT together with a modular framework that enables easy switching of the used heuristics during runtime and simplifies changes to existing heuristics and the addition of new ones;
- extensive testing is performed to find the cause of the surprising similarity of the implemented heuristics in terms of performance.

In Section 2 we lay the groundwork for the rest of the thesis. We introduce SAT and its generalization SMT as well as DPLL and DPLL(T), two popular algorithms for solving SAT and SMT, respectively. We then introduce the first-order theory of linear real arithmetic, followed by an introduction of general Simplex as a theory

solver for this theory in the DPLL(T) framework. Finally, we give a brief introduction of the SMT solver `SMT-RAT`. In Section 3, we explain the four heuristics which are the focus of this thesis as well as the ideas behind them. Section 4 focuses on the implementation of the heuristics and the modular framework in the `LRAModule` of `SMT-RAT` and Section 5 presents the surprising results of the initial benchmarks and then focuses on the different tests performed in search for an explanation and the insights gained from them. Section 6 concludes the thesis by giving a quick summary, discussing the results of Section 5 and highlighting possibilities for future work.

Chapter 2

Preliminaries

2.1 SAT and SMT

2.1.1 SAT

This section is intended as a quick recap of Boolean logic and the definition of the SAT problem. For the rest of this section, τ denotes a given set of Boolean variables. We start with the definition of *Boolean formulas*:

Definition 2.1.1. *The set of Boolean formulas is inductively defined as follows:*

- *Each Boolean variable is a formula.*
- *The constants 0 and 1 are Boolean formulas (0 means false and 1 means true).*
- *If φ and ψ are Boolean formulas, then $\neg\varphi$ and $\varphi \wedge \psi$ are Boolean formulas.*
- *Only constructs formed by the above rules are Boolean formulas.*

Additionally, we define as usual syntactic sugar $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi := \neg\varphi \vee \psi$ etc.

For a set of Boolean formulas Φ we denote the conjunction of all its elements by $\bigwedge \Phi$ (e.g. for $\Phi = \{A \vee \neg B, A \rightarrow C\}$ we have $\bigwedge \Phi = (A \vee \neg B) \wedge (A \rightarrow C)$). For $\Phi = \emptyset$, we set $\bigwedge \Phi = 1$. A *literal* is either a Boolean variable or its negation. For a literal l , $\bar{l}$ denotes the *complement* of l . If $l = B$ for a Boolean variable $B \in \tau$ then $\bar{l} = \neg B$ and if $l = \neg B$ then $\bar{l} = B$. A *clause* is a disjunction of literals. A formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses.

For a Boolean formula φ , we denote the set of Boolean variables contained in φ as $\tau(\varphi)$. An *assignment* of a Boolean formula φ is a mapping $\alpha : D \mapsto \{0,1\}$ where $D \subseteq \tau(\varphi)$. An assignment α is called *full* if $D = \tau(\varphi)$ and *partial* otherwise. A variable $B \in \tau(\varphi)$ is *assigned* by an assignment α of φ if $B \in D$ and *unassigned* otherwise. An assignment $\alpha' : D' \mapsto \{0,1\}$ of φ is called an *expansion* of another assignment $\alpha : D \mapsto \{0,1\}$ of φ if $D \subseteq D'$. We define a notation for expansions where $D' = D \dot{\cup} \{B\}$ for a variable $B \in \tau(\varphi) \setminus D$:

Definition 2.1.2. Let φ be a Boolean formula, $\alpha : D \rightarrow \{0,1\}$ be a partial assignment of φ , $B \in \tau(\varphi) \setminus D$ and $b \in \{0,1\}$. We define $\alpha[B \mapsto b]$ as follows:

$$\alpha[B \mapsto b] : D \cup \{B\} \rightarrow \{0,1\}$$

with $\alpha[B \mapsto b](B') = \alpha(B')$ for $B' \in D$ and $\alpha[B \mapsto b](B) = b$.

Semantically, each full assignment evaluates a formula by substituting each Boolean variable for its mapped value and deciding whether the resulting Boolean term is equivalent to 0 or 1 using the usual interpretation of $\neg$ and $\wedge$. We say that a full assignment α of a formula φ *satisfies* φ (denoted as $\alpha \models \varphi$) if and only if the resulting term is equivalent to 1. Otherwise, we say that α *doesn't satisfy* φ (denoted as $\alpha \not\models \varphi$). A Boolean formula φ is called *satisfiable* if there exists a full assignment α that satisfies φ and *unsatisfiable* otherwise. The *Boolean satisfiability problem (SAT)* is the problem of deciding whether a given Boolean formula φ is satisfiable or unsatisfiable.

2.1.2 First-order theories and SMT

Formally, SMT is the problem of deciding whether a given formula in the *existential fragment of first-order logic* is satisfiable in the context of a given *theory*. Since detailed knowledge about first-order logic is not necessary for the topic of this thesis, a simpler, less formal definition is given in this section. For a formal definition, we refer the reader to [KS16].

Given a *first-order theory* T , a *T-formula* is a Boolean formula where the Boolean variables are replaced by *T-atoms*. The syntax of *T-atoms* depends on the theory T . For example, $x + y < 5$ is a *T-atom* for the theory of *linear real arithmetic* while $a[i] = 5$ is a *T-atom* for the theory of *arrays*. Similar to Boolean formulas, *T-formulas* are evaluated by *assignments*. An assignment of a *T-formula* evaluates each *T-atom* of that formula to either *true* (1) or *false* (0). The truth value of the *T-formula* is then determined the same way as with Boolean formulas, by substituting each *T-atom* by its truth value and then using the usual interpretation of $\neg$ and $\wedge$. If a *T-formula* φ is evaluated to 1 by an assignment α , we say that α *satisfies* φ (written $\alpha \models \varphi$). Otherwise, we say that α *doesn't satisfy* φ ($\alpha \not\models \varphi$). A *T-formula* φ is called *satisfiable* if there exists an assignment α that satisfies φ and *unsatisfiable* otherwise.

Analog to the *complement* of a literal for Boolean logic, we also define the *complement* $\bar{l}$ of a (possibly negated) *T-atom* l for a first-order theory T . If $l = a$ for a *T-atom* a then $\bar{l} = \neg a$ and if $l = \neg a$ for a *T-atom* a then $\bar{l} = a$.

Example 2.1.1. Let T be the theory of linear real arithmetic (see Section 2.3). An example of a *T-formula* is $\varphi = x + y < 5 \wedge x > 3 \wedge (x < y \vee x > 10)$. An assignment of φ corresponds to mapping x and y to real values and then evaluating the *T-atoms* using the usual interpretation of $+$, $<$ and $>$. Consider for example an assignment α that maps x to 12 and y to -10. The function φ is evaluated as follows:

1. *T-atoms*:

- $x + y < 5$ evaluates to 1 since $12 - 10 = 2 < 5$
- $x > 3$ evaluates to 1 since $12 > 3$
- $x < y$ evaluates to 0 since $12 \not< -10$
- $x > 10$ evaluates to 1 since $12 > 10$

2. Substituting T -atoms by truth values and evaluating the formula:

$$1 \wedge 1 \wedge (0 \vee 1) \equiv 1 \wedge 1 \wedge 1 \equiv 1$$

Therefore, α satisfies φ . Since we have found an assignment that satisfies φ , we conclude that φ is satisfiable.

Example 2.1.2. Let T be the theory of arrays. An example of a T -formula is $\varphi = a[i] < b[j] \wedge i = j \wedge a = b$ where a and b are array variables and i and j are index variables. An assignment of φ corresponds to assigning arrays to a and b and index values to i and j (and then using the usual interpretation of $[[$, $=$ and $<$ for evaluation). For an assignment α which maps i to 2, j to 2, a to an array of length at least 3 with $a[2] = 3.5$ and b to an array of length at least 3 with $b[2] = 4$, φ would be evaluated as follows:

1. T -atoms:

- $a[i] < b[j]$ evaluates to 1 ($a[i] = a[2] = 3.5 < 4 = b[2] = b[j]$)
- $i = j$ evaluates to 1
- $a = b$ evaluates to 0 since $a[2] \neq b[2]$ and therefore a and b cannot be equal (two arrays are equal if they have the same length and contain the same value for each index)

2. Substituting T -atoms by truth values and evaluating the formula:

$$1 \wedge 1 \wedge 0 \equiv 0$$

Therefore, α doesn't satisfy φ . Furthermore, we can argue that φ is unsatisfiable, since $a[i] < b[j] \wedge i = j$ implies that a and b differ at least at one index. Therefore, an assignment that satisfies $a[i] < b[j] \wedge i = j$ cannot satisfy $a = b$.

The *Satisfiability Modulo Theories (SMT)* problem is the problem of deciding for a given first-order theory T and a given T -formula φ whether φ is satisfiable or unsatisfiable.

2.2 DPLL and DPLL(T)

The contents of this section are heavily inspired by course materials from the research group *Theory of Hybrid Systems* at RWTH Aachen University which in turn draw inspiration from [KS16]. Inspiration was also drawn from [Na120].

2.2.1 DPLL

The *Davis-Putnam-Loveland-Logemann procedure (DPLL)* [DLL62] is an algorithm for solving the SAT problem that forms the backbone of many modern SAT solvers. Although DPLL assumes the input formula to be in CNF, it can be easily extended to work on any Boolean formula, since there exists an algorithm for transforming an arbitrary Boolean formula into a satisfiability-equivalent formula in CNF in linear time [Tse83].

DPLL solves a formula φ in CNF by maintaining an assignment α of φ . Figure 2.1 shows a recursive version of DPLL. In each step (corresponding to a call to

```

1: function DPLL( $\varphi$ )
2:    $\alpha \leftarrow$  empty assignment (no variables assigned)
3:   return DPLL_SEARCH( $\varphi, \alpha$ )
4: end function
1: function DPLL_SEARCH( $\varphi, \alpha$ )
2:   repeat
3:     if  $\alpha$  evaluates a clause in  $\varphi$  to 0 then
4:       return UNSAT
5:     end if
6:     if  $\alpha$  evaluates all clauses in  $\varphi$  to 1 then
7:       return SAT
8:     end if
9:   until DERIVE( $\varphi, \alpha$ ) = false
10:  select an unassigned variable  $V$ 
11:   $\alpha_0 \leftarrow \alpha[V \mapsto 0]$ 
12:  if DPLL_SEARCH( $\varphi, \alpha_0$ ) = SAT then
13:    return SAT
14:  end if
15:   $\alpha_1 \leftarrow \alpha[V \mapsto 1]$ 
16:  return DPLL_SEARCH( $\varphi_1, \alpha_1$ )
17: end function

```

Figure 2.1: Pseudocode of a recursive version of DPLL. DPLL(φ) returns SAT if φ is satisfiable and UNSAT otherwise. DPLL_SEARCH(φ, α) returns SAT if there exists an expansion of α that satisfies φ and UNSAT otherwise. DERIVE(φ, α) returns *true* if additional assignments can be derived from α and *false* otherwise. If additional assignments can be derived, DERIVE derives one such assignment and expands α accordingly.

DPLL_SEARCH in the algorithm) DPLL first tries to *derive* mappings of unassigned variables from the already assigned variables based on the current assignment α (e.g. if $\varphi = (A \vee \neg B) \wedge (A \vee C)$ and $\alpha : A \mapsto 0$, DPLL deduces that B must be mapped to 0 for φ to be satisfied and expands α accordingly). If no more new mappings can be derived, the algorithm selects an unassigned variable B and then checks recursively whether there exists an expansion of $\alpha[B \mapsto 0]$ or $\alpha[B \mapsto 1]$ that satisfies φ . This process is called a *decision* (since the assignment of B cannot be derived and the algorithm therefore must decide which value is assigned to B). There exists an expansion of α that satisfies φ if and only if there exists an expansion of $\alpha[B \mapsto 0]$ or $\alpha[B \mapsto 1]$ that satisfies φ .

2.2.2 CDCL

Conflict-driven clause learning (CDCL) [MSS96] is a popular extension of DPLL which is often used in modern SAT solving [KS16]. CDCL solvers additionally maintain an *implication graph* during the solving process. Each time a *conflict* is encountered, i.e. the current assignment evaluates a clause to 0, the implication graph is used to determine a set of decided or derived assignments (see Section 2.2.1) that together imply the conflict. The solver then adds a clause to the input formula which prevents that particular combination of assignments from occurring again. A

detailed description is omitted here, since it is not necessary for understanding the contributions in this thesis.

2.2.3 DPLL(T)

The DPLL(T) framework was first introduced in [GHN⁺04] and is implemented in most modern SMT solvers due to its performance advantages. DPLL(T) has two main components: A SAT solver using the DPLL algorithm with the CDCL extension and a theory solver. The SAT solver decides the satisfiability of a T -formula φ by solving a *Boolean abstraction* of φ and calling the theory solver during the solving process to check whether the current assignment is compliant with the underlying theory T .

A Boolean abstraction of a T -formula φ is constructed by replacing each T -atom a in φ with a new Boolean variable B_a . Note that while multiple occurrences of the same T -atom in φ are replaced by the same variable, two T -atoms that are logically equivalent but don't have the same syntax are typically replaced by two different variables.

Example 2.2.1. *Let T be the first-order theory of linear real arithmetic and $\varphi = y > x \wedge (x \geq y + 1 \vee \neg(y > x) \vee x > 0) \wedge x \geq 1 + y$. The Boolean abstraction of φ is constructed by replacing the T -atoms of φ as follows:*

- $y > x$ is replaced by $B_{y>x}$
- $x \geq y + 1$ is replaced by $B_{x \geq y+1}$
- $x > 0$ is replaced by $B_{x>0}$
- $x \geq 1 + y$ is replaced by $B_{x \geq 1+y}$ (note that $B_{x \geq y+1}$ and $B_{x \geq 1+y}$ are two different Boolean variables since $x \geq y + 1$ and $x \geq 1 + y$ are logically equivalent but differ syntactically)

The Boolean abstraction of φ is therefore $B_{y>x} \wedge (B_{x \geq y+1} \vee \neg B_{y>x} \vee B_{x>0}) \wedge B_{x \geq 1+y}$.

When given an input formula φ to solve, DPLL(T) first constructs the Boolean abstraction φ_B of φ . After the Boolean abstraction is constructed, the SAT solver starts solving it. If the SAT solver returns UNSAT then DPLL(T) terminates with UNSAT as well. At certain stages during the solving process, which are determined by the chosen *approach*, the theory solver is called for the current assignment α_B to check the conjunction of all (potentially negated) T -atoms in the set

$$I_{\alpha_B} := \{a \mid a \text{ is a } T\text{-atom in } \varphi \text{ and } \alpha(B_a) = 1\} \\ \cup \{\neg a \mid a \text{ is a } T\text{-atom in } \varphi \text{ and } \alpha(B_a) = 0\}$$

for satisfiability. If $\bigwedge I_{\alpha_B}$ is satisfiable then the theory solver returns a satisfying assignment α for $\bigwedge I_{\alpha_B}$. If α_B was a full assignment of φ_B this causes DPLL(T) to terminate with α as a satisfying assignment for φ , otherwise the SAT solver continues with the solving process. If $\bigwedge I_{\alpha_B}$ is unsatisfiable then the theory solver returns an *explanation clause* E , a disjunction

$$E := \bigvee_{i \in I'} \bar{i}$$

with $I' \subseteq I_{\alpha_B}$ such that $\bigwedge I'$ is unsatisfiable. The SAT solver then adds the Boolean abstraction of E to φ as a new clause and undoes its assignments (starting with the last variable assigned) as long as the new clause remains violated. Subsequently, the SAT solver applies the CDCL method to resolve the conflict and continues solving the formula.

There are two approaches used in DPPL(T). In the *full-lazy* approach, the theory solver is only called if the SAT solver finds a satisfying assignment α_B of φ_B . In the *less-lazy* approach, the theory solver is additionally called before each decision.

2.3 Linear real arithmetic

The first-order theory of interest in this thesis is the theory of *linear real arithmetic*. Let τ be a set of real-valued variables. *T-atoms* of the theory of linear real arithmetic are constructed using *T-terms*.

Definition 2.3.1. *For the theory of linear real arithmetic, a T-term is of the form*

$$a_1x_1 + \dots + a_nx_n + b$$

where $a_1, \dots, a_n \in \mathbb{Q}, b \in \mathbb{Q}$ and $x_1, \dots, x_n$ are real-valued variables. A *T-atom* is of the form

$$t_0 \sim t_1$$

where t_0 and t_1 are *T-terms* and $\sim \in \{<, \leq, =, \neq, \geq, >\}$.

As already shown in Section 2.1.2, *T-terms* and *T-atoms* are evaluated using the usual interpretation of $+, \cdot, <, \leq, =, \neq, \geq$ and $>$. For the theory of linear real arithmetic, we also call a *T-atom* a *linear constraint*. A linear constraint is called *strict* if $\sim \in \{<, >, \neq\}$ and *weak* otherwise. If $\sim \in \{=\}$, the constraint is called a *linear equation*.

In the following, we assume without loss of generality that all linear constraints are of the form

$$a_1x_1 + \dots + a_nx_n \sim b$$

where $a_1, \dots, a_n \in \mathbb{Q}, b \in \mathbb{Q}, x_1, \dots, x_n$ are real-valued variables and $\sim \in \{<, \leq, =, \neq, \geq, >\}$. This is possible because for an arbitrary linear constraint c of the form

$$a_1x_1 + \dots + a_nx_n + b \sim a'_1x'_1 + \dots + a'_nx'_n + b'$$

a logically equivalent linear constraint $\hat{c}$ of the desired form can be constructed by subtracting $a'_1x'_1 + \dots + a'_nx'_n + b$ from both sides and simplifying according to the rules of real arithmetic.

In order to explain an example presented in Section 5, we need one more minor definition:

Definition 2.3.2. *Given a weak linear constraint of the form $c := a_1x_1 + a_2x_2 \sim b$, the equality-line of c is defined by the equation $a_1x_1 + a_2x_2 = b$.*

This definition is only used to visualize constraints in a two-dimensional coordinate system as shown in Figure 2.2.

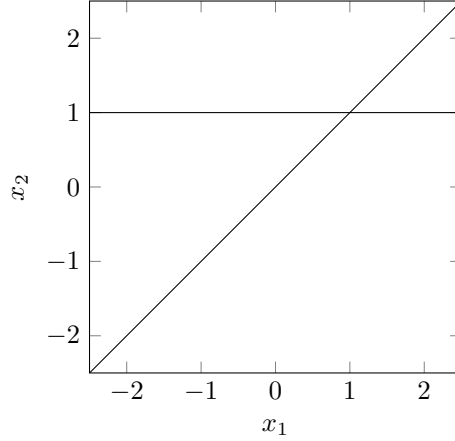


Figure 2.2: Visual representation of the equality-lines of the constraints $x_2 \leq 1$ and $-x_1 + x_2 \geq 0$ in a two-dimensional coordinate system.

2.4 General Simplex

General Simplex is a variation of the well known *Simplex* algorithm [Dan63] which is used to optimize a *linear objective function* under certain restrictions represented by a set of weak linear constraints. The version of general Simplex presented here is based on the algorithm from [KS16, Nal20], which was first introduced in [DdM06]. Given a set Φ of weak linear constraints, general Simplex determines whether $\varphi := \bigwedge \Phi$ is satisfiable. If φ is satisfiable then the algorithm returns a satisfying assignment α , otherwise it returns a set $\Psi \subseteq \Phi$ such that $\bigwedge \Psi$ is unsatisfiable.

2.4.1 General Form

In order to start the solving process, general Simplex requires its input to be in general form. An input in general form consists of two components. The first component is a conjunction φ_{eq} of linear equations of the form

$$\varphi_{eq} = \bigwedge_{i=1}^m \left(\sum_{j=1}^n a_{i,j} x_j = s_i \right)$$

where $a_{i,j} \in \mathbb{Q}$ for each $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n\}$ and $x_1, \dots, x_n, s_1, \dots, s_m$ are real-valued variables ($x_1, \dots, x_n$ are called *original variables* and $s_1, \dots, s_m$ are called *slack variables*). The second component φ_{bounds} is a conjunction of *bounds*. A bound is a linear constraint of the form

$$s_i \sim b$$

where $\sim \in \{\leq, \geq\}$, $b \in \mathbb{Q}$ and s_i is a slack variable. For such a constraint $s_i \sim b$, we will sometimes also refer to b as a *bound*.

An arbitrary conjunction of weak linear constraints $\varphi = \bigwedge_{i=1}^m c_i$ can be translated into normal form by first replacing each linear constraint $c_i = (\sum_{j=1}^n a_j x_j \sim b)$ by a

linear equation $\sum_{j=1}^n a_j x_j = s_i$ (where s_i is a newly introduced variable) and either one bound $s_i \sim b$ (if $\sim \in \{\leq, \geq\}$) or two bounds $s_i \leq b$ and $s_i \geq b$ (if $\sim \in \{=\}$). If two constraints c_i and c_j ($i < j$) have the same T-term on the left side, only one linear equation with s_i on the right side is added.

Example 2.4.1. Let $\varphi = (x - y = 4 \wedge 2x + z \leq -1 \wedge -2y \geq 1 \wedge 2x + z \geq -6)$. The general form is constructed as follows:

- replace $x - y = 4$ by $x - y = s_1$, $s_1 \leq 4$ and $s_1 \geq 4$
- replace $2x + z \leq -1$ by $2x + z = s_2$ and $s_2 \leq -1$
- replace $-2y \geq 1$ by $-2y = s_3$ and $s_3 \geq 1$
- replace $2x + z \geq -6$ by $s_2 \geq -6$ (there already is a constraint with $2x + z$ on the left side)

Therefore, $\varphi_{eq} = (x - y = s_1 \wedge 2x + z = s_2 \wedge -2y = s_3)$ and $\varphi_{bounds} = (s_1 \leq 4 \wedge s_1 \geq 4 \wedge s_2 \leq -1 \wedge s_3 \geq 1 \wedge s_2 \geq -6)$.

2.4.2 Tableau, assignment and bounds

During the solving process, general Simplex maintains both a representation of a tableau (which we will call a tableau for convenience) in the form

$$x_i = \sum_{x_j \in N} a_{i,j} x_j \quad x_i \in B$$

and an assignment $\alpha : \tau \rightarrow \mathbb{R}$, where $\tau = \{x_1, \dots, x_k\}$ are the variables of the tableau, $B \subset \tau$ is a set of *basic variables* (also called the *basis*), $N = \tau \setminus B$ is the set of *non-basic variables* (also called the *non-basis*) and $(a_{i,j})_{x_i \in B, x_j \in N}$ are the *coefficients* of the tableau. Note that we silently assume a mapping that maps each *basic variable* to a *row index* and each *non-basic variable* to a *column index* of the tableau. Additionally, general Simplex stores a *lower bound* $l(x_i) \in \mathbb{Q} \cup \{-\infty\}$ and an *upper bound* $u(x_i) \in \mathbb{Q} \cup \{\infty\}$ for each variable $x_i \in \tau$. For simplicity, we will sometimes denote a tableau as a tuple

$$T = (B, N, A := (a_{i,j})_{x_i \in B, x_j \in N})$$

For an input $(\varphi_{eq} = \bigwedge_{i=1}^m (\sum_{j=1}^n a_{i,j} x_j = s_i), \varphi_{bounds})$ in general form, tableau, assignment and lower/upper bounds are initialized as follows ($\max \emptyset = -\infty$, $\min \emptyset = \infty$):

- the tableau consists of the equations from φ_{eq} (this implies $B = \{s_1, \dots, s_m\}$ and $N = \{x_1, \dots, x_n\}$)
- $\alpha(x_j) = 0$ for $j \in \{1, \dots, n\}$, $\alpha(s_i) = 0$ for $i \in \{1, \dots, m\}$
- $l(x_j) = -\infty$, $u(x_j) = \infty$ for $j \in \{1, \dots, n\}$ (no bounds)
- $l(s_i) = \max \{b \mid s_i \geq b \text{ is a constraint in } \varphi_{bounds}\}$ for $i \in \{1, \dots, m\}$ (greatest lower bound)

- $u(s_i) = \min \{b \mid s_i \leq b \text{ is a constraint in } \varphi_{\text{bounds}}\}$ for $i \in \{1, \dots, m\}$ (least upper bound)

Example 2.4.2. Consider the following input in general form:

$$\begin{aligned}\varphi_{eq} &= (x + y = s_1 \wedge -2x + y = s_2 \wedge 3y = s_3) \\ \varphi_{\text{bounds}} &= (s_1 \geq 1 \wedge s_2 \geq -2 \wedge s_1 \leq 3 \wedge s_3 \leq -3 \wedge s_2 \geq -5)\end{aligned}$$

The initial tableau constructed from this input is

$$\begin{array}{rcl} x & +y & = s_1 \\ -2x & +y & = s_2 \\ & 3y & = s_3 \end{array}$$

The stored bounds are $l(x) = l(y) = -\infty$, $u(x) = u(y) = \infty$ for the non-basic variables and $l(s_1) = \max\{1\} = 1$, $u(s_1) = \min\{3\} = 3$, $l(s_2) = \max\{-2, -5\} = -2$, $u(s_2) = \min \emptyset = \infty$, $l(s_3) = \max \emptyset = -\infty$, $u(s_3) = -3$. We denote the bounds in the form

$$\begin{aligned} 1 &\leq s_1 \leq 3 \\ s_2 &\geq -2 \\ s_3 &\leq -3 \end{aligned}$$

The initial assignment is $\alpha : x \mapsto 0, y \mapsto 0, s_1 \mapsto 0, s_2 \mapsto 0, s_3 \mapsto 0$.

After initializing tableau, assignment and bounds, general Simplex starts altering tableau and assignment by performing *pivot operations* until either a satisfying assignment is found or the tableau produces a *conflict*. The following theorem (whose proof we omit here) states three invariants that hold initially and after each pivot operation.

Theorem 2.4.1. *The invariants*

1. $\alpha(x_i) = \sum_{x_j \in N} a_{i,j} \alpha(x_j)$ for all $x_i \in B$ (the assignment fulfills the equations)
2. $l(x_j) \leq \alpha(x_j) \leq u(x_j)$ for all $x_j \in N$ (all non-basic variables satisfy their bounds)
3. an arbitrary assignment α' satisfies the equations of the current tableau if and only if α' is a satisfying assignment of φ_{eq}

hold initially and after each pivot operation.

We note that the third invariant of this theorem proves that if at any point the current assignment α satisfies the equations of the tableau and all bounds, α is also a satisfying assignment of the original input $\varphi_{eq} \wedge \varphi_{\text{bounds}}$ in general form.

2.4.3 Pivot operation

A satisfying assignment $\alpha : \tau \rightarrow \mathbb{R}$ needs to satisfy the equations

$$x_i = \sum_{x_j \in N} a_{i,j} x_j \quad x_i \in B$$

and the bounds

$$l(x_i) \leq x_i \leq u(x_i) \quad x_i \in \tau$$

Since the invariants state that the assignment always satisfies both the equations and the bounds of the non-basic variables, the only constraints that can be violated before a pivot operation are bounds of basic variables. If a variable $x_i \in \tau$ violates one of its bounds, we denote that bound by v_i ($v_i = l(x_i)$ if $l(x_i) > \alpha(x_i)$ and $v_i = u(x_i)$ if $u(x_i) < \alpha(x_i)$). A pivot operation consists of the following steps:

1. Pivot selection:

- (a) Select a basic variable that violates one of its bounds as the *leaving variable* (the variable that leaves the basis B).

Formally, select a $x_l \in B$ with $l(x_l) > \alpha(x_l)$ or $u(x_l) < \alpha(x_l)$.

- (b) Select a non-basic variable as the *entering variable* (the variable that enters the basis) according to its bounds and the equation $\sum_{x_j \in N} a_{l,j} x_j = x_l$. If the leaving variable x_l violates its *lower bound* (*upper bound*), select an entering variable whose assignment can be changed such that the assignment of x_l *increases* (*decreases*).

Formally, select a $x_e \in N$ with

$$(a_{l,e} < 0 \wedge l(x_e) < \alpha(x_e)) \vee (a_{l,e} > 0 \wedge u(x_e) > \alpha(x_e))$$

if $l(x_l) > \alpha(x_l)$ and

$$(a_{l,e} < 0 \wedge u(x_e) > \alpha(x_e)) \vee (a_{l,e} > 0 \wedge l(x_e) < \alpha(x_e))$$

if $u(x_l) < \alpha(x_l)$.

2. Pivot step:

- (a) Modify the assignments:

- i. $\alpha(x_e) := \alpha(x_e) + \delta$ with $\delta = \frac{v_l - \alpha(x_l)}{a_{l,e}}$ (change the assignment of the entering variable)
- ii. $\alpha(x_l) := \alpha(x_l) + a_{l,e} \cdot \delta = v_l$ (change the assignment of x_l according to its equation; this doesn't change δ)
- iii. $\alpha(x_i) := \alpha(x_i) + a_{i,e} \cdot \delta$ for $x_i \in B \setminus \{x_l\}$ (the assignments of the other basic variables are changed according to their equations)

- (b) Solve $\sum_{x_j \in N} a_{l,j} x_j = x_l$ for x_e :

$$\begin{aligned} & \sum_{x_j \in N} a_{l,j} x_j = x_l \\ \Leftrightarrow & \frac{1}{a_{l,e}} \cdot x_l + \sum_{x_j \in N \setminus \{x_e\}} -\frac{a_{l,j}}{a_{l,e}} \cdot x_j = x_e \end{aligned} \quad (2.1)$$

- (c) Eliminate x_e from all other equations by substitution using Equation (2.1).
The new system of equations is now

$$\begin{aligned} a'_{e,l}x_l + \sum_{x_j \in N \setminus \{x_e\}} a'_{e,j}x_j &= x_e \\ a'_{i,l}x_l + \sum_{x_j \in N \setminus \{x_e\}} a'_{i,j}x_j &= x_i \quad x_i \in B \setminus \{x_l\} \end{aligned}$$

with

$$\begin{aligned} a'_{e,l} &= \frac{1}{a_{l,e}} \\ a'_{e,j} &= -\frac{a_{l,j}}{a_{l,e}} & x_j \in N \setminus \{x_e\} \\ a'_{i,l} &= a_{i,e} \cdot a'_{e,l} & x_i \in B \setminus \{x_l\} \\ a'_{i,j} &= a_{i,j} + a_{i,e} \cdot a'_{e,j} & x_i \in B \setminus \{x_l\}, \quad x_j \in N \setminus \{x_e\} \end{aligned}$$

- (d) $B := (B \setminus \{x_l\}) \cup \{x_e\}$, $N := \tau \setminus B$ (x_e becomes basic and x_l becomes non-basic)

We call a basic variable $x_i \in B$ *suitable* if it can be selected as the leaving variable in Step 1a, that is if $l(x_i) > \alpha(x_i)$ or $u(x_i) < \alpha(x_i)$. We call a non-basic variable *suitable* if it can be selected as entering variable in Step 1b.

Example 2.4.3. We continue Example 2.4.2. For the initial state

$$\begin{array}{rcl} x & +y & = s_1 \\ -2x & +y & = s_2 \\ & 3y & = s_3 \end{array} \qquad \begin{array}{rcl} 1 & \leq & s_1 \leq 3 \\ & & s_2 \geq -2 \\ & & s_3 \leq -3 \end{array}$$

with assignment $\alpha : x \mapsto 0, y \mapsto 0, s_1 \mapsto 0, s_2 \mapsto 0, s_3 \mapsto 0$, only s_1 and s_3 violate their bounds. Therefore, the leaving variable must be either s_1 or s_3 . We select $x_l := s_1$. Since $\alpha(s_1) = 0$, the violated bound is $v_l = l(s_1) = 1$. Both x and y are unbounded with a non-zero coefficient in the first equation and therefore suitable for the entering variable. We select $x_e := x$. We now modify α (Step 2a):

$$\delta := \frac{v_l - \alpha(x_l)}{a_{l,e}} = \frac{1 - 0}{1} = 1$$

$$\alpha(x) := \alpha(x) + \delta = 0 + 1 = 1$$

$$\alpha(s_1) := \alpha(s_1) + a_{l,e} \cdot \delta = 0 + 1 \cdot 1 = 1$$

$$\alpha(s_2) := 0 + (-2) \cdot 1 = -2$$

$$\alpha(s_3) := 0 + 0 \cdot 1 = 0$$

Solving $x + y = s_1$ for x produces

$$s_1 - y = x$$

and substituting x for $s_1 - y$ in the other equations leads to the tableau

$$\begin{array}{rcl} s_1 & -y & = x \\ -2s_1 & +3y & = s_2 \\ & 3y & = s_3 \end{array}$$

Special cases

There are two special cases that can occur during a pivot operation. In Step 1a, it is possible that there is no suitable basic variable because all basic variables satisfy their bounds. Since α always satisfies the equations and the bounds of the non-basic variables, this means that α is a satisfying assignment. Likewise, it is possible that there is no suitable non-basic variable in Step 1b. In this case, $\varphi_{eq} \wedge \varphi_{bounds}$ is unsatisfiable and a *conflict set* $\Psi_{conflict}$ is generated.

A conflict set $\Psi_{conflict}$ is a set consisting of bounds from φ_{bounds} such that $\varphi_{eq} \wedge \Psi_{conflict}$ is unsatisfiable. Recall that we have already selected a leaving variable $x_l \in B$ in Step 1a and either $l(x_l) > \alpha(x_l)$ or $u(x_l) < \alpha(x_l)$. Assume $l(x_l) > \alpha(x_l)$. Due to Step 1b finding no suitable variables, we know that

$$(a_{l,j} < 0 \wedge l(x_j) \geq \alpha(x_j)) \vee (a_{l,j} > 0 \wedge u(x_j) \leq \alpha(x_j)) \vee a_{l,j} = 0 \quad (2.2)$$

for each $x_j \in N_j$. This leads to

$$\begin{aligned} l(x_l) &> \alpha(x_l) \\ &= \sum_{x_j \in N} a_{l,j} \alpha(x_j) \\ &= \sum_{\substack{x_j \in N \\ a_{l,j} < 0}} a_{l,j} \alpha(x_j) + \sum_{\substack{x_j \in N \\ a_{l,j} > 0}} a_{l,j} \alpha(x_j) \\ &\geq \sum_{\substack{x_j \in N \\ a_{l,j} < 0}} a_{l,j} l(x_j) + \sum_{\substack{x_j \in N \\ a_{l,j} > 0}} a_{l,j} u(x_j) \end{aligned}$$

Therefore

$$\begin{aligned} \Psi_{conflict} &= \{x_l \geq l(x_l)\} \cup \{x_j \geq l(x_j) \mid a_{l,j} < 0\} \\ &\quad \cup \{x_j \leq u(x_j) \mid a_{l,j} > 0\} \end{aligned}$$

is a valid conflict set. The case that $u(x_l) < \alpha(x_l)$ is handled in a similar manner.

Example 2.4.4. *If we perform a second pivot operation on Example 2.4.3 with s_3 as the leaving and y as the entering variable, we obtain*

$$\begin{array}{rcl} s_1 & -\frac{1}{3}s_3 & = x \\ -2s_1 & s_3 & = s_2 \\ & \frac{1}{3}s_3 & = y \end{array} \qquad \begin{array}{rcl} 1 & \leq & s_1 \leq 3 \\ & & s_2 \geq -2 \\ & & s_3 \leq -3 \end{array}$$

with assignment $\alpha : x \mapsto 2, y \mapsto -1, s_1 \mapsto 1, s_2 \mapsto -5, s_3 \mapsto -3$. Only s_2 is suitable for the leaving variable, because $\alpha(s_2) = -5 < -2$. However, s_1 is not suitable as the entering variable because

$$-2 < 0 \wedge l(s_1) \geq \alpha(s_1)$$

and s_3 is not suitable because

$$1 > 0 \wedge u(s_3) \leq \alpha(s_3)$$

Therefore the input formula in general form is unsatisfiable and

$$\Psi_{\text{conflict}} = \{s_2 \geq -2, s_1 \geq 1, s_3 \leq -3\}$$

If we have the same state except that $s_2 \geq -2$ is replaced with $s_2 \geq -8$, the pivot operation would find no suitable candidates for the leaving variable and the algorithm would terminate with α as a satisfying assignment.

Useful lemmas

We will now prove a few lemmas that we will use in Section 3.1 to prove the termination of Bland's rule.

Lemma 2.4.2. *Let T be the tableau before the pivot operation, T' be the tableau after the pivot operation and $\alpha : \tau \rightarrow \{0,1\}$ be an arbitrary assignment. Then α satisfies the equations of T if and only if α satisfies the equations of T' .*

Proof. The only changes made to the equations are the rewriting of one equation in Step 2b and the subsequent use of the rewritten equation to substitute a variable in the other equations in Step 2c. Neither rewriting an equation in a system (of equations) nor substituting one side of an equation in a system for the other in all other equations changes whether an assignment α satisfies all equations. Therefore, α satisfies the equations of T if and only if α satisfies the equations of T' . $\square$

Lemma 2.4.3. *Let $T := (B, N, A := (a_{i,j})_{x_i \in B, x_j \in N})$ and $T' := (B', N', A' := (a'_{i,j})_{x_i \in B, x_j \in N})$ be two tableaus such that T' is obtained from T by a series of pivot operations. Then*

1. *an arbitrary assignment α satisfies the equations of T if and only if α satisfies the equations of T'*
2. *$B = B'$ implies $T = T'$.*

Proof. We prove the first statement and then use it to prove the second one.

1. Since T' is obtained from T by a series of pivot operations, there exist tableaus $T_0, \dots, T_n$ such that $T_0 = T$, $T_n = T'$ and T_{i+1} is obtained from T_i by one pivot operation. According to Lemma 2.4.2, α satisfies the equations of T_{i+1} if and only if α satisfies the equations of T_i . Therefore α satisfies the equations of $T_0 = T$ if and only if α satisfies the equations of $T_n = T'$.
2. Assume $B = B'$ and $T \neq T'$.

Since $B = B'$ and $N = \tau \setminus B = \tau \setminus B' = N'$ there exists at least one $x_k \in B$, $x_h \in N$ such that $a_{k,h} \neq a'_{k,h}$. Let $\alpha : \tau \rightarrow \{0,1\}$ with $\alpha(x_h) = 1$, $\alpha(x_j) = 0$ for all $x_j \in N \setminus \{x_h\}$ and $\alpha(x_i) = a_{i,h}$ for all $x_i \in B$. Then

$$\begin{aligned} \alpha(x_i) &= a_{i,h} \\ &= a_{i,h} \cdot 1 + \sum_{x_j \in N \setminus \{x_h\}} a_{i,j} \cdot 0 \\ &= \sum_{x_j \in N} a_{i,j} \alpha(x_j) \end{aligned}$$

for all $x_i \in B$. Therefore α satisfies all equations in T .

According to the first statement, α must also satisfy all equations in T' . This leads to

$$\begin{aligned} a_{k,h} &= \alpha(x_k) \\ &= \sum_{x_j \in N'} a'_{k,j} \alpha(x_j) \\ &= a'_{k,h} \end{aligned}$$

Contradiction!

Therefore, $B = B'$ implies $T = T'$.

□

Lemma 2.4.4. *The invariants*

1. $\alpha(x_j) = 0$ for all $x_j \in N$ that are unbounded ($l(x_j) = -\infty$ and $u(x_j) = \infty$)
2. $\alpha(x_j) \in \{l(x_j), u(x_j)\}$ for all other $x_j \in N$

hold initially and after each pivot operation.

Proof. We prove the lemma by induction over the number $n \in \mathbb{N}$ of pivot operations. For $n > 0$, $S_n := (T_n := (B_n, N_n, A_n := (a_{i,j})_{x_i \in B_n, x_j \in N_n}^n), \alpha_n)$ denotes the state of the algorithm after the n -th pivot operation, where T_n is the tableau and α_n the assignment. $S_0 = (T_0 := (B_0, N_0, A_0 := (a_{i,j})_{x_i \in B_0, x_j \in N_0}^0), \alpha_0)$ denotes the initial state.

Base case:

We have

$$\begin{aligned} \alpha_0(x_j) &= 0 \\ l(x_j) &= -\infty \\ u(x_j) &= \infty \end{aligned}$$

for each $x_j \in N_0$ per definition (see Section 2.4.2).

Inductive step: Assume $\alpha_n(x_j) = 0$ for all $x_j \in N_n$ that are unbounded and $\alpha_n(x_j) \in \{l(x_j), u(x_j)\}$ for all other $x_j \in N_n$.

Let $x_e \in N_n$ be the entering variable and $x_l \in B_n$ be the leaving variable of the n -th pivot operation. We know that $N_{n+1} = (N_n \setminus \{x_e\}) \cup \{x_l\}$ and $\alpha_{n+1}(x_j) = \alpha_n(x_j)$ for $x_j \in N_n \setminus \{x_e\} = N_{n+1} \setminus \{x_l\}$ due to Step 2a. Therefore we only need to prove that $\alpha_{n+1}(x_l) = 0$ if x_l is unbounded and $\alpha_{n+1}(x_l) \in \{l(x_l), u(x_l)\}$ otherwise. But x_l cannot be unbounded due to Step 1a and

$$\alpha_{n+1}(x_l) = \alpha_n(x_l) + a_{l,e}^n \cdot \frac{v_l - \alpha_n(x_l)}{a_{l,e}^n} = v_l$$

with $v_l \in \{l(x_l), u(x_l)\}$ due to Step 2a. Therefore both invariants hold for S_{n+1} .

□

```

1: function GENERAL_SIMPLEX( $\varphi_{eq}, \varphi_{bounds}$ )
2:    $(T, \alpha) \leftarrow$  initial tableau and assignment ▷ Section 2.4.2
3:   loop ▷ pivot operation, Section 2.4.3
4:     select a leaving variable  $x_l \in B$  ▷ Step 1a
5:     if there is no suitable basic variable then
6:       return (SAT,  $\alpha$ )
7:     end if
8:     select a entering Variable  $x_e \in N$  ▷ Step 1b
9:     if there is no suitable non-basic variable then
10:      return (UNSAT, CONFLICT( $T, \alpha, x_l$ ))
11:    end if
12:     $(T, \alpha) \leftarrow$  PIVOT_STEP( $T, \alpha, x_l, x_e$ ) ▷ Step 2
13:  end loop
14: end function

```

Figure 2.3: Pseudocode for general Simplex without transformation into general form (Section 2.4.1) and reverse transformation of output. CONFLICT constructs a conflict set the way described in Section 2.4.3 and returns it.

2.4.4 The complete algorithm

The pseudocode of the complete algorithm (without preprocessing and processing of the output) is given by the function GENERAL_SIMPLEX in Figure 2.3.

GENERAL_SIMPLEX is given an input $(\varphi_{eq}, \varphi_{bounds})$ in general form and determines if $\varphi_{eq} \wedge \varphi_{bounds}$ is satisfiable. After constructing the initial tableau, GENERAL_SIMPLEX starts to perform pivot operations until there is either no suitable basic variable or no suitable non-basic variable. If there is no suitable basic variable, GENERAL_SIMPLEX returns SAT together with the current assignment α as a satisfying assignment. If there is no suitable non-basic variable, GENERAL_SIMPLEX generates a conflict set and returns UNSAT together with the conflict set.

Note that a selection rule is needed in line 4 and 8 if there are multiple suitable variables. The choice of this selection rule can have a big impact on the number and cost of pivoting operations performed before termination and therefore on the general performance. Depending on the chosen selection rule, it is even possible that the algorithm gets stuck in a *cycle* and doesn't terminate at all. Such selection rules are called *heuristics* and will be discussed in detail later.

2.4.5 General Simplex in DPLL(T)

Extending General Simplex for strict linear constraints

A theory solver in DPLL(T) accepts a set Φ of literals (in this case linear constraints or their negation) as input and returns a satisfying assignment if $\varphi := \bigwedge \Phi$ is satisfiable and an explanation clause constructed from a set $\Psi \subseteq \Phi$ for which $\bigwedge \Psi$ is unsatisfiable otherwise. The base form of general Simplex accepts a set of weak linear constraints and [Nal20] describes an extension of general Simplex that also allows strict linear constraints. Such an extension is sufficient for handling any set of literals, since the negation of a linear constraint $\neg c$ can always be replaced by a logically equivalent linear constraint c' .



Figure 2.4: The strategy used in this thesis. The root module is blue.

Incrementality

When embedding general Simplex in DPLL(T), the tableau is initialized with the equations and bounds derived from all constraints in the input formula. Each bound is either *active* or *inactive* and inactive bounds are simply ignored during the execution of general Simplex. Each time the theory solver is called with Φ as input, only the bounds corresponding to a constraint $c \in \Phi$ are active. Furthermore, after a call to the theory solver is complete, the tableau remains in its final state and instead of re-initializing the tableau on the next call, the final state of the last call is used as initial state. In case some of the newly activated bounds are violated by non-basic variables, the assignment of these variables is increased/decreased and the assignment of the basic variables is modified accordingly.

This approach saves overhead from repeated re-initializing of the tableau (especially when using a less-lazy approach).

2.5 SMT-RAT

The implementation was done in SMT-RAT, *an open source C++ toolbox for strategic and parallel SMT solving* [CKJ⁺15] developed at the *Theory of Hybrid Systems* group at RWTH Aachen University. SMT-RAT uses a *modular* architecture: Each solving technique (e.g. general Simplex or a SAT solver) is implemented in a *module*. A *strategy* is a directed tree where each node is a module and the child nodes of a module are its *backends*. When SMT-RAT is run on an input formula, the selected strategy determines which modules are used to solve it: Initially, only the root module is called to solve the input formula, but during the solving process, a module can pass formulas to its backends to check them for satisfiability. In the DPLL(T) framework for example, the SAT solver would pass conjunctions of literals to its backend, the theory solver.

The strategy used in this thesis is shown in Figure 2.4. It is composed of three different modules: FPPModule, SATModule and LRAModule. FPPModule is a preprocessing module used to simplify the formula and SATModule and LRAModule serve as the SAT solver and the theory solver of the DPLL(T) framework, respectively. SATModule uses Minisat [ES03] to solve the boolean abstraction of the input formula and calls the theory solver according to the less-lazy approach of DPLL(T) and LRAModule implements general Simplex.

2.5.1 Tableaus in LRAModule

LRAModule models the state of the tableau as a table M where each *row* of M corresponds to a basic variable and each *column* of M corresponds to a non-basic

$$\begin{array}{rrrr}
x_1 & +x_2 & +x_6 & = x_4 \\
-x_1 & & +x_6 & = x_5 \\
& -5x_2 & -2x_6 & = x_3
\end{array}
\longrightarrow
\begin{array}{c|c|c|c}
& x_1 & x_2 & x_6 \\
\hline
x_4 & 1 & 1 & 1 \\
x_5 & -1 & \textcolor{red}{0} & 1 \\
x_3 & \textcolor{red}{0} & -5 & -2
\end{array}$$

Figure 2.5: Exemplary tableau on the left and the corresponding table on the right. Red coefficients in the table are not stored internally.

variable. The coefficients of the tableau are the entries of the table: If the basic variable of the row r is x_i and the non-basic variable of the column c is x_j , the entry at row r and column c is $a_{i,j}$. Internally, only coefficients $a_{i,j} \neq 0$ are stored for performance reasons. Figure 2.5 shows an exemplary tableau and the corresponding table in `LRAModule`.

Chapter 3

Simplex heuristics

In this thesis, we take a closer look at the mechanisms of several *heuristics* of general Simplex. A heuristic is a selection rule for the leaving and entering variable of a pivot operation if there are multiple suitable variables. We call the rule for selecting the leaving variable the *leaving rule* and the rule for selecting the entering variable the *entering rule*. This chapter describes the heuristics evaluated in this thesis: *Bland's rule*, *difference-based*, *conflict-based* and *sum of infeasibilities*.

3.1 Bland's rule

Bland's rule was first introduced in [Bla77] for the normal Simplex algorithm and adapted to general Simplex in [DdM06]. It is very simple and guarantees termination, but its performance in practice is rather poor. *Bland's rule* uses an arbitrary *fixed order* of the variables and both the entering and the leaving rule always pick the smallest suitable variable according to the chosen order.

We will now present a proof for the termination of *Bland's rule* for general Simplex which is a more in-depth version of the proof from [DdM06] (inspiration was also drawn from [AC78]).

Theorem 3.1.1. *If general Simplex uses Bland's rule for the selection of the leaving and entering variable, it always terminates.*

Proof. Assume general Simplex doesn't always terminate when using Bland's rule. Let $x_1 < x_2 < \dots < x_n$ be the ordered variables of τ . Then Bland's rule picks the suitable variable with the smallest index. We choose an input for which general Simplex doesn't terminate.

General Simplex cycles:

The state of general Simplex before each pivot iteration is defined by the tableau T and the assignment α (the bounds never change). According to Lemma 2.4.3 the basis B uniquely defines the tableau T of the state. Therefore, we can write the state before the k -th pivot operation as a pair $S_k = (B_k, \alpha_k)$ where B_k is the basis and α_k the assignment. We denote the unique tableau defined by B_k as

$$T_k = (B_k, N_k, A_k := (a_{i,j})_{x_i \in B_k, x_j \in N_k}^k)$$

There is only a finite number of possible bases since τ is finite and there is only a finite number of valid assignments α_k for a given base B_k since

- $\alpha_k(x_j) \in \{0, l(x_j), u(x_j)\}$ for each $x_j \in N_k$ according to Lemma 2.4.4.
- $\alpha_k(x_i) = \sum_{x_j \in N_k} a_{i,j}^k \alpha_k(x_j)$ for each $x_i \in B_k$ according to the first invariant of Theorem 2.4.1.

Therefore, the set of states that are reachable during execution is finite.

Since general Simplex doesn't terminate, this means that the sequence of states $S_0, S_1, \dots$ must contain a cycle $S_k, \dots, S_t, S_{t+1} = S_k$.

General observations:

We call a variable $x_i \in \tau$ *fickle*, if it leaves the basis during at least one pivot operation of the cycle. Since each pivot operation changes the basis we have at least one fickle variable.

Let x_r be the fickle variable with the largest index. Then there exists a state S_l in the cycle where x_r leaves the basis ($x_r \in B_l$ and $x_r \in N_{l+1}$). Since $S_k, \dots, S_t, S_{t+1} = S_k$ is a cycle, x_r must eventually re-enter the basis. Let S_p be the first state in the cycle after S_l where x_r enters the basis and let x_s be the variable that is pivoted with x_r in the pivot operation between S_p and S_{p+1} ($x_s \in B_p$ and $x_s \in N_{p+1}$).

We make a few observations about S_l (denoted in blue) and S_p (denoted in orange) as well as some additional observations (denoted in purple) that form the basis of the proof.

The following observations hold for S_l :

- (L1) $\alpha_l(x_r) < l(x_r) \vee \alpha_l(x_r) > u(x_r)$
- (L2) $\alpha_l(x_s) = \sum_{x_j \in N_p} a_{s,j}^p \alpha_l(x_j)$
- (L3) $l(x_j) \leq \alpha_l(x_j) \leq u(x_j)$ for all $x_j \in \tau$ with $j < r$

(L1) holds because x_r is selected as leaving variable in S_l . (L2) holds because α_l satisfies the equations of S_l (first invariant of Theorem 2.4.1) and therefore must also satisfy the equations of S_p (Lemma 2.4.3). (L3) holds because $l(x_j) \leq \alpha_l(x_j) \leq u(x_j)$ for each $x_j \in N_l$ (second invariant of Theorem 2.4.1) and if there was an $x_j \in B_l$ with $j < r$ and $\alpha_l(x_j) < l(x_j) \vee \alpha_l(x_j) > u(x_j)$, Bland's rule would have picked x_j as leaving variable instead of x_r .

The following observations hold for S_p :

- (P1) $\alpha_p(x_s) < l(x_s) \vee \alpha_p(x_s) > u(x_s)$
- (P2) $\alpha_p(x_s) = \sum_{x_j \in N_p} a_{s,j}^p \alpha_p(x_j)$

(P1) holds because x_s is selected as leaving variable in S_p . (P2) holds according to the first invariant of Theorem 2.4.1.

The following additional observations hold:

$$(A1) \quad \alpha_p(x_r) = \alpha_{l+1}(x_r)$$

$$(A2) \quad \alpha_l(x_s) - \alpha_p(x_s) = \sum_{x_j \in N_p} a_{s,j}^p (\alpha_l(x_j) - \alpha_p(x_j))$$

(A1) holds because x_r is non-basic in the states $S_{l+1}, \dots, S_p$ and therefore its assignment is not changed. (A2) holds due to (L2) and (P2).

According to (L1) and (P1), we can distinguish between four cases.

Case 1: $\alpha_l(x_r) < l(x_r) \wedge \alpha_p(x_s) < l(x_s)$

Since $\alpha_l(x_r) < l(x_r)$, we get $\alpha_{l+1}(x_r) = l(x_r)$. With (A1), we get

$$\alpha_{l+1}(x_r) = \alpha_p(x_r) = l(x_r) \quad (3.1)$$

With $\alpha_p(x_s) < l(x_s)$ and because x_r is chosen as entering variable in S_p , we get

$$a_{s,r}^p > 0 \quad (3.2)$$

Furthermore, we know that $s < r$ due to the definition of x_r . With (L3), we get $l(x_s) \leq \alpha_l(x_s) \leq u(x_s)$ and with $\alpha_p(x_s) < l(x_s)$ we get $\alpha_l(x_s) > \alpha_p(x_s)$. According to (A2), this means that there exists at least one $x_j \in N_p$ such that

$$a_{s,j}^p (\alpha_l(x_j) - \alpha_p(x_j)) > 0 \quad (3.3)$$

Case 1.1: $j < r$

With (L3) we get

$$l(x_j) \leq \alpha_l(x_j) \leq u(x_j) \quad (3.4)$$

Since $\alpha_p(x_s) < l(x_s)$ and $x_j \in N_p$ was not picked as entering variable in S_p , we know that

$$\begin{aligned} &(\alpha_p(x_j) \leq l(x_j) \wedge a_{s,j}^p < 0) \\ &\vee (\alpha_p(x_j) \geq u(x_j) \wedge a_{s,j}^p > 0) \\ &\vee a_{s,j}^p = 0 \end{aligned}$$

In each case, we have

$$a_{s,j}^p (\alpha_l(x_j) - \alpha_p(x_j)) \leq 0$$

with (3.4). Contradiction!

Case 1.2: $j = r$

We get

$$a_{s,r}^p (\alpha_l(x_r) - \alpha_p(x_r)) < 0$$

with (3.2), (3.1) and $\alpha_l(x_r) < l(x_r)$. Contradiction!

Case 1.3: $j > r$

Then x_j is always non-basic in the cycle and $\alpha_l(x_j) = \alpha_p(x_j)$. Therefore

$$a_{s,j}^p (\alpha_l(x_j) - \alpha_p(x_j)) = 0$$

Contradiction!

Case 2: $\alpha_l(x_r) < l(x_r) \wedge \alpha_p(x_s) > u(x_s)$

Case 2 is proven analog to Case 1, with the following changes:

- in Equation (3.2), we get $a_{s,r}^p < 0$
- we infer that there is a $x_j \in N_p$ with $a_{s,j}^p(\alpha_l(x_j) - \alpha_p(x_j)) < 0$ in Equation (3.3)
- in Case 1.1, we infer

$$\begin{aligned} &(\alpha_p(x_j) \leq l(x_j) \wedge a_{s,j}^p > 0) \\ &\vee (\alpha_p(x_j) \geq u(x_j) \wedge a_{s,j}^p < 0) \\ &\vee a_{s,j}^p = 0 \end{aligned}$$

and therefore $a_{s,j}^p(\alpha_l(x_j) - \alpha_p(x_j)) \geq 0$

- in Case 1.2 we infer $a_{s,r}^p(\alpha_l(x_r) - \alpha_p(x_r)) > 0$

Case 3: $\alpha_l(x_r) > u(x_r) \wedge \alpha_p(x_s) < l(x_s)$

Case 3 is proven analog to Case 1, with the following changes:

- in Equation (3.1), we get $\alpha_{l+1}(x_r) = \alpha_p(x_r) = u(x_r)$
- in Equation (3.2), we get $a_{s,r}^p < 0$
- in Case 1.2, we derive the same equation $a_{s,r}^p(\alpha_l(x_r) - \alpha_p(x_r)) < 0$ from the modified equations above and $\alpha_l(x_r) > u(x_r)$

Case 4: $\alpha_l(x_r) > u(x_r) \wedge \alpha_p(x_s) > u(x_s)$

Case 4 is proven analog to Case 1, with the following changes:

- in Equation (3.1), we get $\alpha_{l+1}(x_r) = \alpha_p(x_r) = u(x_r)$
- in Equation (3.2), we derive $a_{s,r}^p > 0$ from the modified equation above and $\alpha_p(x_s) > u(x_s)$
- we infer that there is a $x_j \in N_p$ with $a_{s,j}^p(\alpha_l(x_j) - \alpha_p(x_j)) < 0$ in Equation (3.3)
- in Case 1.1, we infer

$$\begin{aligned} &(\alpha_p(x_j) \leq l(x_j) \wedge a_{s,j}^p > 0) \\ &\vee (\alpha_p(x_j) \geq u(x_j) \wedge a_{s,j}^p < 0) \\ &\vee a_{s,j}^p = 0 \end{aligned}$$

and therefore $a_{s,j}^p(\alpha_l(x_j) - \alpha_p(x_j)) \geq 0$

- in Case 1.2 we infer $a_{s,r}^p(\alpha_l(x_r) - \alpha_p(x_r)) > 0$

Conclusion:

Since every case has led to a contradiction, the initial assumption must be false. Therefore, general Simplex always terminates when using Bland's rule.

□

3.2 Difference-based

The *difference-based* heuristic was taken from [Gri09] where experiments showed a significant performance advantage compared to Bland’s rule. The heuristic works as follows:

1. The *leaving rule* computes the absolute difference $d_i := |v_i - \alpha(x_i)|$ between the violated bound v_i and the assignment $\alpha(x_i)$ for each suitable basic variable $x_i \in B$. The variable with the smallest d_i is selected as leaving variable. Ties are broken by using Bland’s rule.
2. The *entering rule* picks the suitable non-basic variable $x_e \in N$ which occurs in the smallest number of equations (that is $\{x_i \in B \mid a_{i,e} \neq 0\}$ is minimal). Ties are broken using Bland’s rule.

The idea behind the *difference-based* heuristic is to minimize the changes of the assignment α during pivoting in order to reduce the risk of a pivot operation generating new violations that need to be solved later. The *leaving rule* tries to minimize the amount by which the assignment of the entering variable and consequently the assignments of the basic variables are changed while the *entering rule* tries to maximize the number of basic variables that are not changed at all. This has the added benefit of reducing the overhead of the pivot step.

However, while the difference-based heuristic performed better in experiments than Bland’s rule, it doesn’t guarantee terminating and can lead to *cycling*.

3.3 Conflict-based

The *conflict-based* heuristic was taken from [Ali20] (it is called *priority-based strategy* there). This heuristic uses knowledge about the *less-lazy* DPLL(T) framework, specifically the knowledge that the theory solver is often called many times with only minor changes to the formula. Knowledge about *conflicts* encountered in previous calls is preserved and the heuristic tries to force the tableau into a state which produced a conflict in a previous call. This is done as follows:

- for each variable $x_i \in \tau$, two values are stored: its *basic priority* and its *non-basic priority* (both values are initialized with zero)
- both values are preserved between calls to the theory solver and each time the theory solver returns *unsatisfiable* to the SAT solver, the final state of the tableau is checked: all basic variables gain *basic priority* (the value is incremented by one) and all non-basic variables gain *non-basic priority*
- the *leaving rule* picks the suitable variable with maximal *non-basic priority*
- the *entering rule* picks the suitable variable with maximal *basic priority*

The idea is that returning the tableau to a previously conflicting state increases the likelihood of quickly encountering a conflict again and therefore saves runtime. The *conflict-based* heuristic doesn’t guarantee termination and may also be particularly prone to cycling due to it trying to restore a fixed tableau state.

A *secondary heuristic* to break ties is particularly important for the *conflict-based* heuristic since in the beginning there won’t be a lot of previously encountered conflicts

and ties will therefore occur often. A secondary heuristic can be any other heuristic, e.g. Bland's rule or the difference-based heuristic.

3.4 Sum of infeasibilities

3.4.1 General idea

Sum of infeasibilities was introduced in [KBD13] and tries to minimize the *total violation* of all variables (also called the *sum of infeasibilities*). The total violation $Vio(\alpha)$ for an assignment α is defined as follows:

$$Vio(\alpha) = \sum_{x_i \in \tau} Vio_{x_i}(\alpha)$$

$Vio_{x_i}(\alpha)$ denotes the amount by which the assignment $\alpha(x_i)$ of the variable x_i violates its bound, that is

$$Vio_{x_i}(\alpha) = \begin{cases} \alpha(x_i) - u(x_i) & \alpha(x_i) > u(x_i) \\ l(x_i) - \alpha(x_i) & \alpha(x_i) < l(x_i) \\ 0 & \text{otherwise} \end{cases}$$

Sum of infeasibilities selects the leaving/entering variable in a way that ensures the total violation either shrinks or remains unchanged during each pivot operation. If that is not possible, *sum of infeasibilities* derives a conflict.

3.4.2 Preliminary definitions

In order to select the leaving and entering variable, *sum of infeasibilities* defines *directions*. The direction $d_i(\alpha)$ of a variable $x_i \in \tau$ for an assignment α is defined as follows:

$$d_i(\alpha) = \begin{cases} 1 & \alpha(x_i) > u(x_i) \\ -1 & \alpha(x_i) < l(x_i) \\ 0 & \text{otherwise} \end{cases}$$

For a given assignment α , the *violated bound* $v_i(\alpha)$ of a variable $x_i \in \tau$ is defined as follows:

$$v_i(\alpha) = \begin{cases} u(x_i) & \alpha(x_i) > u(x_i) \\ l(x_i) & \alpha(x_i) < l(x_i) \\ 0 & \text{otherwise} \end{cases}$$

With these definitions, we get

$$\begin{aligned} Vio_{x_i}(\alpha) &= d_i(\alpha) \cdot (\alpha(x_i) - v_i(\alpha)) \\ Vio(\alpha) &= \sum_{x_i \in \tau} d_i(\alpha) \cdot (\alpha(x_i) - v_i(\alpha)) \end{aligned}$$

Another important definition is that of a *break point*. Given a current tableau T_c and assignment α_c as well as a non-basic variable x_j , a break point for T_c, α_c and x_j is a value δ that fulfills the following condition: If δ is added to $\alpha_c(x_j)$ and the assignments of the basic variables are modified according to their equations (see Step 2a of the pivot operation) then at least one basic variable is set to one of its bounds.

3.4.3 Determining the total violation

Let $T_c = (B_c, N_c, A_c := (a_{i,j}^c)_{x_i \in B_c, x_j \in N_c})$ be the current tableau and α_c be the current assignment before a pivot operation. We know that in the pivot step (Step 2) the assignment of the entering variable is changed and then the assignments of the basic variables are modified according to the equations. Therefore, given a value δ and an entering variable $x_e \in N_c$, our goal is to determine how the *total violation* changes if we add δ to $\alpha_c(x_e)$ and modify the assignments of the basic variables according to their equations.

To do this, we take a look at the current total violation

$$\begin{aligned} Vio(\alpha_c) &= \sum_{x_i \in \tau} d_i(\alpha_c) \cdot (\alpha_c(x_i) - v_i(\alpha_c)) \\ &= \sum_{x_i \in B_c} d_i(\alpha_c) \cdot (\alpha_c(x_i) - v_i(\alpha_c)) \\ &= \left(\sum_{x_i \in B_c} d_i(\alpha_c) \cdot \alpha_c(x_i) \right) - C \end{aligned}$$

with $C = \sum_{x_i \in B_c} d_i(\alpha_c) \cdot v_i(\alpha_c)$. By replacing $\alpha_c(x_i)$ by x_i , we get

$$Vio_F(\alpha_c) := \left(\sum_{x_i \in B_c} d_i(\alpha_c) \cdot x_i \right) - C$$

Since the basic variables are modified according to their equations, we substitute using said equations and get

$$\begin{aligned} Vio_F(\alpha_c) &= \left(\sum_{x_i \in B_c} d_i(\alpha_c) \cdot x_i \right) - C \\ &= \left(\sum_{x_i \in B_c} d_i(\alpha_c) \left(\sum_{x_j \in N_c} a_{i,j}^c x_j \right) \right) - C \\ &= \left(\sum_{x_j \in N_c} \left(\sum_{x_i \in B_c} d_i(\alpha_c) a_{i,j}^c \right) x_j \right) - C \end{aligned} \tag{3.5}$$

We call $t_j := \sum_{x_i \in B_c} d_i(\alpha_c) a_{i,j}^c$ the *infeasibility coefficient* of x_j . Note that for every assignment α that fulfills the equations of the tableau and has the same *violated bounds* and *directions* as α_c , the total violation $Vio(\alpha)$ can be computed by replacing x_j with $\alpha(x_j)$ in Equation (3.5).

We now consider the smallest break point $\delta_{j,+} \geq 0$ and the biggest break point $\delta_{j,-} \leq 0$ for each non-basic variable $x_j \in N_c$. For every $\delta \in (\delta_{j,-}, \delta_{j,+})$, adding δ to the assignment of an entering variable $x_e \in N_c$ and modifying the assignments of the basic variables accordingly doesn't change the *violated bounds* or *directions*. Therefore, the new total violation is

$$Vio_{old} + t_j \cdot \delta$$

where Vio_{old} is the total violation before the modifications. This motivates a new definition of *suitable* variables in the context of this heuristic. We call a non-basic variable $x_j \in N$ suitable, if

$$(t_j > 0 \wedge \alpha_c(x_j) > l(x_j)) \vee (t_j < 0 \wedge \alpha_c(x_j) < u(x_j))$$

The set of suitable non-basic variables is

$$\begin{aligned} Suit = & \{x_j \in N_c \mid t_j > 0 \wedge \alpha_c(x_j) > l(x_j)\} \\ & \cup \{x_j \in N_c \mid t_j < 0 \wedge \alpha_c(x_j) < u(x_j)\} \end{aligned}$$

3.4.4 Algorithm

An algorithm for general Simplex with *sum of infeasibilities* is given in Figure 3.1. Note that this algorithm has a different structure than the standard algorithm depicted in Figure 2.3. Before each pivot operation, the algorithm performs three checks: First it checks whether there are any basic variables that violate their bounds. If that is not the case, the algorithm terminates with α as a satisfying assignment. Otherwise, the algorithm checks whether there is a conflict caused by a single equation (see Section 2.4.3). Unlike the standard version of general Simplex, which only checks the equation of the leaving variable for a conflict, SOI_SIMPLEX checks *all* equations for conflicts. If no single-line conflict is found, the algorithm then checks if there are suitable non-basic variables. If not, a conflict set $\Psi_{conflict}$ can be constructed as described in Section 3.4.5. Otherwise, there are basic variables that violate their bounds and no conflicts have been detected. The algorithm then selects a leaving and entering variable as well as a value δ to be added to the entering variable. This again is a deviation from standard general Simplex where δ is determined by the leaving and entering variable (see Step 2a of the pivot operation).

Pivot selection

The pivot selection is depicted in Line 12 to 22 of Figure 3.1. Pivot selection generates a set S of triples containing the *entering candidate* x_j , the *leaving candidate* x_i and the *update* δ and then picks one triple for pivoting.

For this, the algorithm generates a set L of tuples for each suitable non-basic variable x_j . L contains all breakpoints of x_j together with the basic variable that is set to its bound (if a break point δ sets multiple basic variables to one of their bounds, multiple tuples are generated). SOI_SIMPLEX then selects one tuple $\langle x_i, \delta \rangle$ from L by minimizing $\Delta Vio(x_j, \delta)$ (the amount that the total violation would change if δ was added to $\alpha(x_j)$ and the assignments of the basic variables were modified accordingly). The absolute value $|\delta|$ of the break point and Bland's rule (the variables are ordered

```

1: function SOI_SIMPLEX( $\varphi_{eq}, \varphi_{bounds}$ )
2:    $(T, \alpha) \leftarrow$  initial tableau and assignment
3:   loop
4:     if  $l(x_i) \leq \alpha(x_i) \leq u(x_i)$  for all  $x_i \in B$  then
5:       return (SAT,  $\alpha$ )
6:     else if there is a conflicting equation  $x_i = \sum_{x_j \in N} a_{i,j}x_j$  then
7:       return (UNSAT, CONFLICT( $T, \alpha, x_i$ ))  $\triangleright$  single-line conflict
8:     else if  $Suit = \emptyset$  then
9:       return (UNSAT, MULTI_CONFLICT( $T, \alpha$ ))  $\triangleright$  Section 3.4.5
10:    end if
11:     $S \leftarrow \emptyset$ 
12:    for all  $x_j \in Suit$  do
13:       $L \leftarrow \emptyset$ 
14:      for all  $x_i \in B$  with  $a_{i,j} \neq 0$  do
15:         $L \leftarrow L \cup \{ \langle x_i, \delta_{b,l}(x_j, x_i) \rangle \}$   $\triangleright$  skipped if  $x_i$  has no lower bound
16:         $L \leftarrow L \cup \{ \langle x_i, \delta_{b,u}(x_j, x_i) \rangle \}$   $\triangleright$  skipped if  $x_i$  has no upper bound
17:      end for
18:      select  $\langle x_i, \delta \rangle \in L$  by minimizing  $\langle \Delta Vio(x_j, \delta), |\delta|, i \rangle$ 
19:       $S \leftarrow S \cup \{ \langle x_j, x_i, \delta \rangle \}$ 
20:    end for
21:    select  $\langle x_e, x_l, \delta \rangle$  by minimizing  $\langle sgn(\Delta Vio(x_j, \delta)) \cdot |t_j|, j \rangle$ 
22:     $(T, \alpha) \leftarrow$  PIVOT_STEP( $T, \alpha, x_l, x_e, \delta$ )
23:  end loop
24: end function

```

Figure 3.1: Pseudocode for general Simplex with *sum of infeasibilities*. $\delta_{b,l}(x_j, x_i) := (l(x_i) - \alpha(x_i))/a_{i,j}$ is the break point of x_j that sets x_i to its lower bound and $\delta_{b,u}(x_j, x_i) := (u(x_i) - \alpha(x_i))/a_{i,j}$ is the break point of x_j that sets x_i to its upper bound. $sgn(x)$ denotes the *signum* of x . Single-line conflicts have been explained in Section 2.4.3.

by index) are used as *secondary* and *tertiary* heuristic to break ties. x_j is then added to S together with the selected break point δ and the basic variable x_i .

The triple $\langle x_e, x_l, \delta \rangle$ used for pivoting is selected from the triples with negative $\Delta Vio(x_j, \delta)$ by maximizing the absolute value $|t_j|$ of the infeasibility coefficient. If there are no triples with negative $\Delta Vio(x_j, \delta)$ in S , the triple is selected according to Bland's rule from the triples with $\Delta Vio(x_j, \delta) = 0$ (it can be shown that there always exists a triple $\langle x_e, x_l, \delta \rangle$ with $\Delta Vio(x_j, \delta) \leq 0$). Ties are also broken using Bland's rule.

3.4.5 Multi-line conflicts

A *multi-line conflict* occurs if the checks in Line 4 and Line 6 of Figure 3.1 fail (there are basic variables violating their bounds and no single-line conflicts are detected), but the check in Line 8 discovers that $Suit = \emptyset$ (there are no suitable non-basic variables).

Let E_l be the set of basic variables that violate their lower bound and E_u be the set of basic variables that violate their upper bound, that is

$$E_l = \{x_i \in B \mid \alpha(x_i) < l(x_i)\}$$

$$E_u = \{x_i \in B \mid \alpha(x_i) > u(x_i)\}$$

Furthermore, let

$$\Psi_t = \{x_j \geq l(x_j) \mid t_j > 0\}$$

$$\cup \{x_j \leq u(x_j) \mid t_j < 0\}$$

It can be shown that $E \neq \emptyset$ (check Line 4) and $Suit = \emptyset$ (check Line 8) imply that every assignment α which satisfies the bounds of Ψ_t and the equations from the final tableau cannot satisfy all bounds from

$$\Psi_E = \{x_i \geq l(x_i) \mid x_i \in E_l\} \cup \{x_i \leq u(x_i) \mid x_i \in E_u\}$$

Therefore, we can return

$$\Psi_{conflict} = \Psi_E \cup \Psi_t$$

as a conflict set.

Chapter 4

Implementation

4.1 Overview

The heuristics presented in Section 3 were implemented in the `LRAModule` of `SMT-RAT` (see Section 2.5). Since ties can occur when using the *difference-based* heuristic or the *conflict-based* heuristic, both heuristics use *Bland’s rule* as a secondary heuristic to break said ties. Additionally, the code for the pivot selection was rewritten in order to enable modular and easy switching of the implemented heuristics and simplify changes to the heuristics as well as the addition of new ones. Sections 4.2 and 4.3 describe this in more detail.

4.2 Making pivot selection more modular

Within the `LRAModule`, pivot selection is realized by calling the function

```
std::pair<EntryID, bool> nextPivotingElement()
```

from the `Tableau` class. It returns a pair of an `EntryID` and a `bool`. The `bool` is *true* if there is no conflict in the tableau and *false* if there is such a conflict (either caused by a single row or multiple rows, see Section 3.4.5). If a conflicting row exists then the `EntryID` is used to identify the conflicting row and if the leaving and entering variable were selected successfully then the `EntryID` is used to identify the leaving and entering variable. In all other cases, the `EntryID` is set to the reserved value `LAST_ENTRY_ID` to signal that either the current assignment satisfies all bounds or there is a multi-line conflict.

The rewritten code for this function implements the algorithm for pivot selection depicted in Section 2.4.3, which is used by most heuristics. Figure 4.1 shows the interaction between `nextPivotingElement` and the four most important subroutines used in the algorithm: `selectLeavingVar`, `selectEnteringVar`, `betterBVar` and `betterNbVarEntry`. During pivot selection, `nextPivotingElement` calls `selectLeavingVar` to select the leaving variable. In order to do this, `selectLeavingVar` iterates over all *suitable basic variables* and uses `betterBVar` to determine if the current variable is “better” (according to the used heuristic) as leaving variable then the “best” suitable variable encountered so far. After the leaving variable was selected, `selectEnteringVar` utilizes `betterNbVarEntry` in a similar manner to select the entering variable out of the *suitable non-basic variables*.

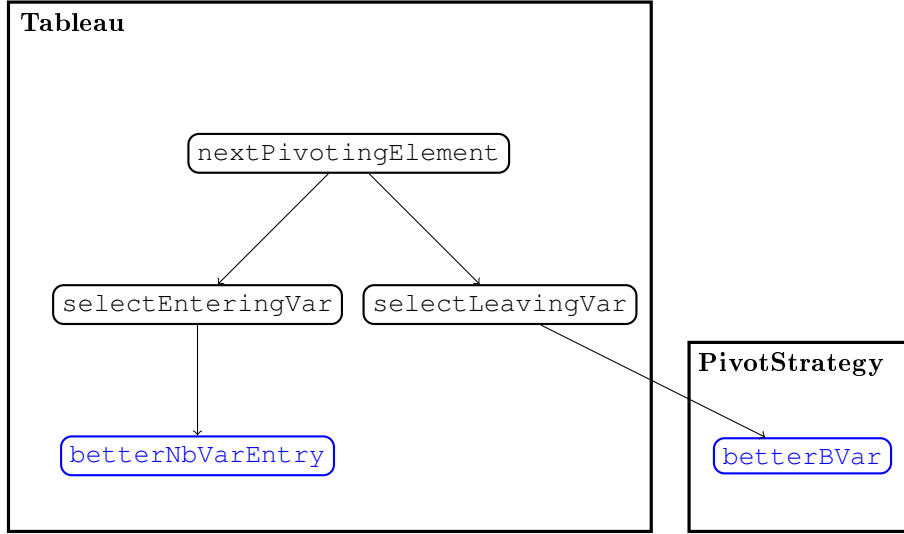


Figure 4.1: Graph visualizing the interaction between `nextPivotingElement` and four of its subroutines. Rectangles with rounded corners represent functions, while rectangles without represent classes. An arrow from function *a* to function *b* means that *b* is called by *a*. If a new heuristic is implemented, only the code within the **blue functions** (namely `betterBVar` and `betterNbVarEntry`) needs to be changed.

This structure has the benefit that only `betterBVar` and `betterNbVarEntry` have to know about the current heuristic and therefore changes to the implemented heuristics (or the addition of new ones) only require changes in these two functions.

However, it should be noted that this only works for heuristics that use the algorithm from Section 2.4.3. *Sum of infeasibilities* for example uses a completely different procedure and introduces multi-line conflicts. Consequently, the structure from Figure 4.1 could not be used and a new structure had to be implemented. Additional changes to the `LRAModule` were necessary in order to implement the correct handling of multi-line conflicts.

4.3 The class `PivotStrategy`

As the original intent of this thesis was to develop a *dynamical heuristic* which switches between *static heuristics* during runtime, a new class was created to store information and subroutines for this purpose, most importantly the currently selected static heuristic. Unfortunately, due to the disappointing results of the static heuristics implemented (see Section 5) this idea was later abandoned, leaving this class with little content. It was also tried to capsule subroutines related to heuristics into this class, but this lead to problems with *mutual recursion*. Therefore, a lot of subroutines that were originally planned to be implemented within the *PivotStrategy* class, were moved to the *Tableau* class instead. This can also be seen in Figure 4.1, as of the four subroutines of `nextPivotingElement`, only `betterBVar` could be implemented in the *PivotStrategy* class.

4.4 Difficulties with *sum of infeasibilities*

While implementing the first three heuristics was fairly simple and straightforward, implementing *sum of infeasibilities* proved to be very difficult due to two reasons:

1. The computation of ΔVio required complex calculations with various edge cases that had to be detected (since they were not mentioned in [KBD13]) and dealt with in the right manner.
2. The pivot operation for *sum of infeasibilities* was completely different from the standard procedure and therefore required various changes to the underlying solver structure. Furthermore, the `LRAModule` itself turned out to be rather complex, containing various extensions on top of the standard general Simplex algorithm. Changes to the underlying structure therefore required a great deal of care to avoid unintended side effects.

As a result, the time required to construct a working implementation of *sum of infeasibilities* was roughly 10 to 20 times higher than the time required for implementing all other heuristics together.

Chapter 5

Evaluation

The original intent of this thesis was to develop a *dynamic heuristic* that switches between a set of *static heuristics* during runtime. The idea was to have the dynamic heuristic monitor the solving process and always select the “best” static heuristic according to the collected data. However, the results obtained from benchmarking SMT-RAT with the static heuristics described in Section 3 showed little difference between the heuristics performance-wise (except for *sum of infeasibilities*, which was significantly worse than the other heuristics) and therefore little promise for such a dynamic heuristic. As a result, the focus of the thesis was shifted to examining these rather surprising results more closely with the goal of finding an explanation as to why the selected heuristic seems to have little influence on the overall performance of SMT-RAT.

5.1 The setup

Benchmarking was performed on the cluster of the *Informatics 2* chair at the RWTH Aachen University, featuring 4x 2.1 Ghz AMD Opteron with 12 Cores each and 192 GB RAM. For benchmarking, we used the tool BENCHMAX, which is provided alongside SMT-RAT. BENCHMAX was executed with a memory limit of 4 gigabyte, using an SSH backend on localhost with 9 connections and 5 parallel jobs per connection. As a benchmark set, we used the repository QF_LRA with non-incremental benchmarks from the SMT-LIB library [BFT16]. Our benchmark set contained a total of 1753 instances.

Usually, heuristics that can *cycle* switch to *Bland’s rule* after a certain number of pivot operations to guarantee termination, a feature that is implemented in SMT-RAT as well. However, since we wanted to specifically examine the differences between the heuristics, we disabled this feature for our tests.

5.2 Initial Results

The initial tests were run with a time limit of 1 minute. As shown in Figure 5.1, *Bland’s rule*, the *difference-based* heuristic and the *conflict-based* heuristic showed only minimal differences with regards to the number of instances solved (the best of them solving only 15 instances more than the worst). This is especially surprising for

	Bland	difference	conflict	soi	virtual best
sat	513	510	510	411	522
unsat	375	384	369	305	386
timeout	822	817	831	992	
memout	43	42	43	45	
total solved	888	894	879	716	908
total unsolved	865	859	874	1037	845

Figure 5.1: Results of the initial tests run with a time limit of 1 minute. The table depicts the number of instances were the heuristics returned *sat/unsat*, the number of instances where they timed out and the number of instances where the solver ran out of memory. Additionally, the total number of *solved instances* (*sat + unsat*) and the number of *unsolved instances* (*timeout + memout*) is shown. *Sum of infeasibilities* was shortened to *soi* for convenience.

	Bland	difference	conflict	virtual best
sat	512	513	512	522
unsat	372	385	369	387
timeout	826	812	831	
memout	43	43	41	
total solved	884	898	881	909
total unsolved	869	855	872	844

Figure 5.2: Results of the second set of tests.

the *difference-based* heuristic, since this heuristic is directly taken from [Gri09], where it is stated that using the *difference-based* heuristic led to a significant performance boost compared to *Bland's rule*. *Sum of infeasibilities* performs significantly worse than the other heuristics even though [KBD13] states that *sum of infeasibilities* was able to compete with a heuristic superior to *Bland's rule* when implemented in their solver CVC4. Most importantly, the *virtual best*, determined by taking the performance of the fastest solver for each instance, only solves slightly (2-3%) more instances than the first three heuristics. This means that not only do *Bland's rule*, the *difference-based* heuristic and the *conflict-based* heuristic solve a similar number of instances, but the instances they solve are mostly the same. This indicates that there would be little benefit to the *dynamic heuristic* explained at the start of this section.

For the second set of tests, we modified *Bland's rule*, the *difference-based* heuristic and the *conflict-based* heuristic. They now perform the same conflict check as *sum of infeasibilities*, checking *all equations* for conflicts instead of just the equation of the leaving variable. Furthermore, we implemented a data structure to dynamically maintain the set of *violating rows* (rows in the tableau where the basic variable is violating its bounds) to increase efficiency. The idea for this was again taken from [KBD13], where it led to a 46% speedup in performance. Figure 5.2 shows the results.

The results of this second set of tests were rather disappointing. Rather than leading to a significant performance increase for the three heuristics, the effects of the new conflict check were negligible. The *difference-based* heuristic solved 4 more instances, the *conflict-based* heuristic solved 2 more and *Bland's rule* even solved 4

instances less than in the initial tests. The fact that the *difference-based* heuristic and the *conflict-based* heuristic seemed to get at least somewhat of a performance increase is probably due to them being better in capitalizing on the new conflict check and the underlying data structure. The *entering rule* of the *difference-based* heuristic tries to minimize the changes to the tableau, therefore reducing the cost of maintaining the data structure for the *violating rows*. The *conflict-based* heuristic aims to force the tableau in a conflicting state and can therefore make good use of a more thorough check for conflicts. On a final note, the behavior of the *virtual best* also didn't change much, with it solving only 1-3% more instances than the other heuristics.

5.3 Two examples to show divergence

Due to the surprising similarities of our heuristics performance-wise, we constructed two examples to verify that the chosen heuristic can have a significant influence on the solving process. Both examples were additionally solved by hand for each heuristic and the obtained results (including most intermediate results) were compared to the ones computed by SMT-RAT. When testing the heuristics on the examples we disabled the `FPPModule` to prevent it from prematurely solving or modifying the input formula.

The first example causes SMT-RAT to execute a single theory call, checking the conjunction of the following linear constraints for satisfiability:

$$\begin{aligned} -x_1 + x_2 &\geq 1 \\ -x_1 + 2x_2 &\leq 4 \\ x_1 + x_2 &\geq 2 \\ 5x_1 + 11x_2 &\geq 55/2 \\ 2x_1 + x_2 &\leq 10 \\ x_1 + 3x_2 &\geq 3/2 \end{aligned}$$

We tested all four heuristics on this example and traced the assignment of x_1 and x_2 during the solving process. Figure 5.3, Figure 5.4 and Figure 5.5 show a geometric representation of the results. Each point corresponds to an assignment. Apart from the two dashed lines, each line is the *equality-line* of a constraint (see Section 2.3). The blue shading signals that one or multiple constraints are violated: The darker the shades at one point, the more constraints are violated by the corresponding assignment. Therefore, the white region corresponds to the set of satisfying assignments. The changes to the assignment during the solving process are shown by a series of red arrows, each arrow representing a single pivot operation.

The figures show that the traversed assignments of the four heuristics differ significantly from each other, with the only exception being *Bland's rule* and the *conflict-based* heuristic, where the traversed assignments are identical. This is because the example causes only a single theory call. At the start of this singular theory call, there are no prior theory calls and therefore no prior conflicts for the *conflict-based* heuristic to utilize. The heuristic then selects the leaving and entering variable solely according to its *secondary heuristic*, which is *Bland's rule*.

In order to show the difference between *Bland's rule* and the *conflict-based* heuristic, we constructed a second example. The second example causes three theory calls to be executed. The third theory call is executed with the same set of constraints as the

theory call in the first example. The other two theory calls aim to showcase the difference between the *conflict-based* heuristic and *Bland's rule* by increasing the *non-basic priority* of the *slack variables* generated by the constraints $5x_1 + 11x_2 \geq 55/2 =: c_1$ and $2x_1 + x_2 \leq 10 =: c_2$ before the third theory call. Figures 5.6, 5.7, 5.8 and 5.9 show the geometric representation of the traversed assignments for the third theory call. Note that due to the incrementality of the Simplex algorithm implemented in the `LRAModule`, the initial assignments of the third theory call differ from each other. Additionally, the variable order assigned by `SMT-RAT` in the second example is different from the one assigned in the first example and therefore the traversed assignments for *Bland's rule* differ as well.

As expected, the higher *non-basic priority* of the *slack variables* generated by the constraints c_1 and c_2 causes the `LRAModule` to try and keep these *slack variables* non-basic. This leads to a notable difference between the traversed assignments of *Bland's rule* and the *conflict-based* heuristic.

Overall, the two examples show that even for a relatively small example, the solving process can indeed differ significantly depending on the chosen heuristic. Therefore, the results of the initial benchmarks are unlikely to be caused by a lack of divergence between the different heuristics.

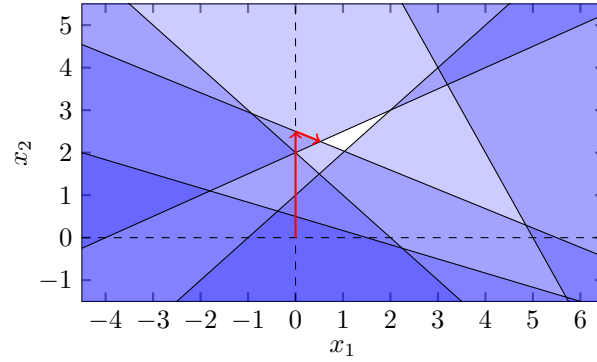


Figure 5.3: Solving process for the first example when using *sum of infeasibilities*.

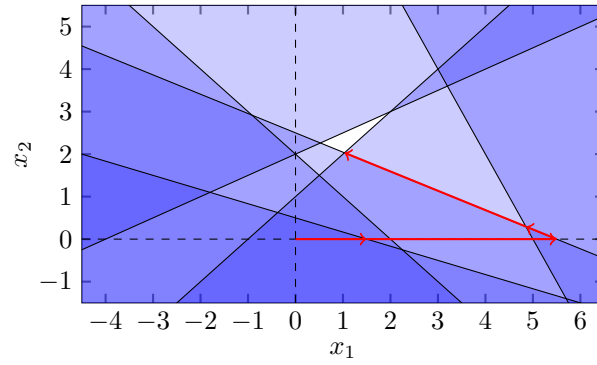


Figure 5.4: Solving process for the first example when using *Bland's rule* or the *conflict-based* heuristic.

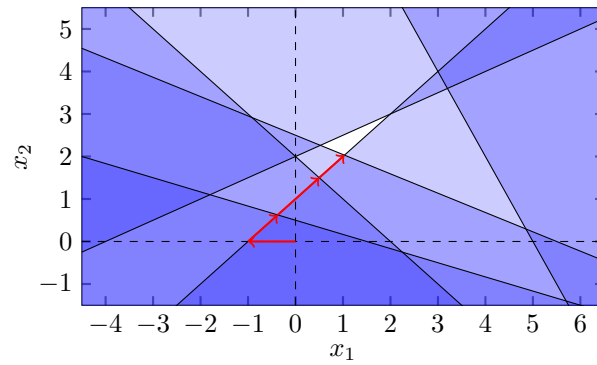


Figure 5.5: Solving process for the first example when using the *difference-based* heuristic.

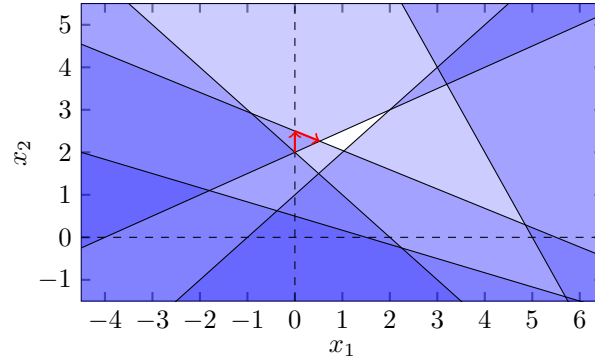


Figure 5.6: Solving process for the third theory call of the second example when using *sum of infeasibilities*.

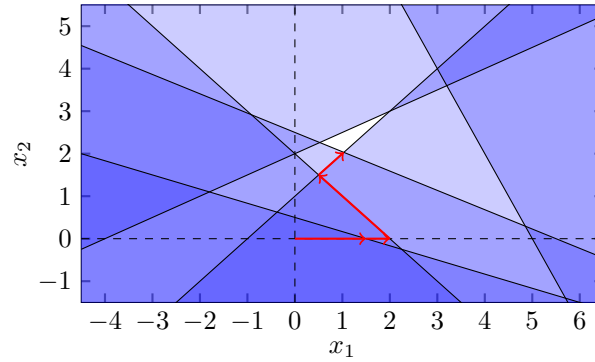


Figure 5.7: Solving process for the third theory call of the second example when using *Bland's rule*.

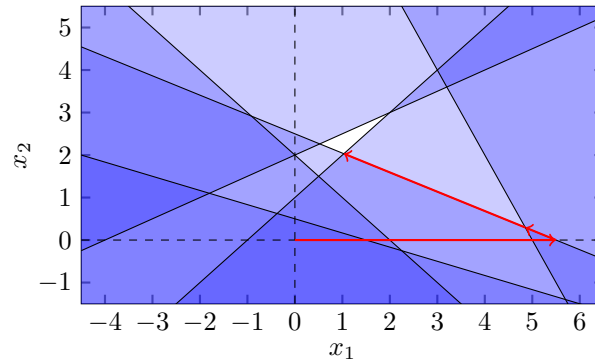


Figure 5.8: Solving process for the third theory call of the second example when using the *conflict-based* heuristic.

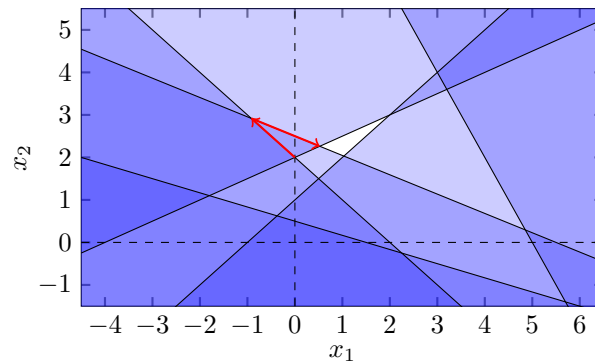


Figure 5.9: Solving process for the third theory call of the second example when using the *difference-based* heuristic.

	Bland	difference	conflict	soi	virtual best
sat	546	544	542	414	560
unsat	394	406	387	309	410
timeout	731	721	742	950	
memout	82	82	82	80	
total solved	940	950	929	723	970
total unsolved	813	803	824	1030	783

Figure 5.10: Number of instances solved with time limit of 5 minutes.

5.4 Going into detail

Following our initial experiments, we took a closer look at the solving process of our heuristics. For this, we performed another round of benchmarking where we additionally collected a bunch of statistics about the solving process. We then generated several profiles and scatterplots and examined the collected statistics. Our most important findings are presented in this section.

5.4.1 The setting

For this round of benchmarking we raised the timeout to 5 minutes. We used the same version of the heuristics as in our first tests, that is *Bland's rule*, the *difference-based* heuristic and the *conflict-based* heuristic only check the row of the leaving variable for conflicts instead of checking all rows. Figure 5.10 shows the number of solved instances with the new time limit. Raising the timeout from 1 minute to 5 minutes didn't cause any major changes (e.g. the *virtual best* still only solves 2-4% more instances than the first three heuristics).

5.4.2 The FPPModule

The FPPModule is the first module called on the input and used to simplify the formula (see Section 2.5). Since the SATModule and the LRAModule are only called after the execution of the FPPModule is complete, the performance on all instances on which the FPPModule either times out or solves the formula is unaffected by the LRAModule and therefore the chosen heuristic. The exact number of these instances varies slightly for each round of benchmarks taken, but is always roughly 30% of the total instances. Roughly 40% of these instances are solved by the FPPModule, on the other 60% the FPPModule times out. Figure 5.11 shows this distribution in a pie diagram.

For the remaining tests, we only looked at the instances where the LRAModule was called.

5.4.3 Performance profile and scatterplots

Next, we generated the performance profile as well as all scatterplots for our heuristics. Figure 5.12 and 5.13 show the results.

The performance profiles of *Bland's rule*, the *difference-based* heuristic and the *conflict-based* heuristic are again very similar to each other. In the full performance profile, the difference between these three only becomes clearly visible from around

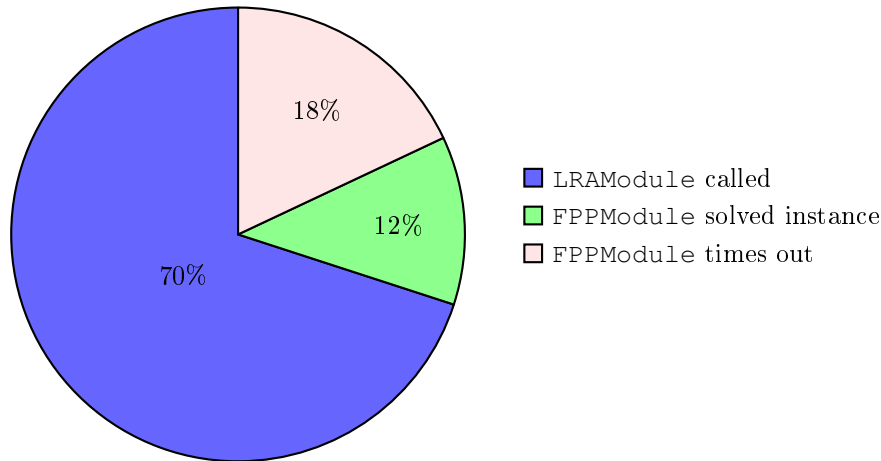


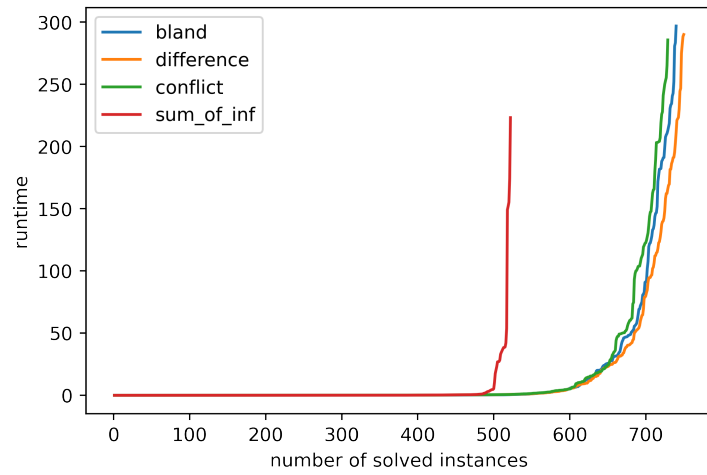
Figure 5.11: Pie diagram depicting the number of instances that the *FPPModule* solves and the number of instances where the *FPPModule* times out in relation to the total number of instances.

650 instances forward. From there on, the *difference-based* heuristic performs slightly better than *Bland's rule* while the *conflict-based* heuristic performs slightly worse. *Sum of infeasibilities* is an outlier, performing very similar to the other heuristics up until around 400 to 450 instances but performing significantly worse from there on.

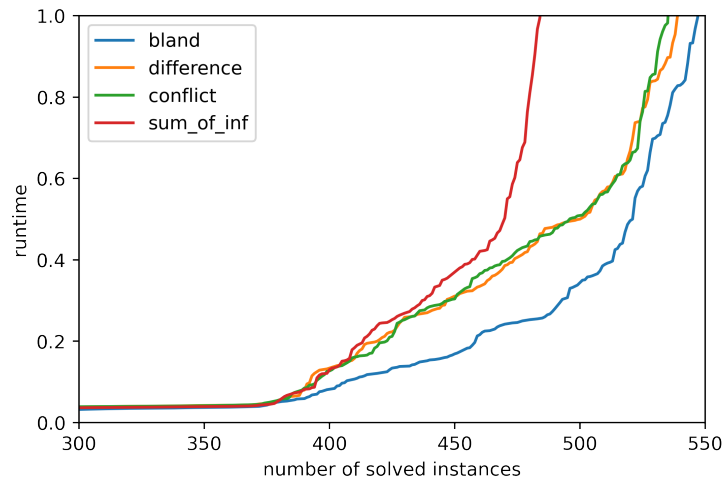
Probably the most important thing to note however is that all four heuristic solve the majority of their instances extremely fast and the number of additional instances solved starts to shrink rapidly with higher runtimes. The enlarged section of the performance profile shows that almost all instances solved by *sum of infeasibilities* are solved in less than 1 second. While this ratio is a bit lower for the other heuristics, they still solve more than 2/3 of their solved instances in less than 1 second. It should also be noted that even in the enlarged section of the performance profile (with the *y*-axis only ranging from 0 to 1 second), the heuristics are still indistinguishable until about 360 instances.

When looking at the scatterplots comparing the *Bland's rule*, the *difference-based* heuristic and the *conflict-based* heuristic in Figure 5.13, we note that there are in fact a number of instances where one heuristic significantly outperforms the other (we will call these instances *outliers*). However, there are a few things to note here. First of all, even though we filtered all instances where the *LRAModule* was not called, there are still more than 1200 instances remaining. Secondly, there are a lot of instances tightly grouped together near the bottom left corner. These groups are most likely composed of a lot more instances than apparent at first glance. Similarly, the singular point in the top right corner is in fact caused by more than 700 instances stacked on top of each other. In comparison, the number of *outliers* is rather small. Nonetheless, it should be noted that some *outliers* do in fact exist.

When comparing *sum of infeasibilities* to the other heuristics, we note that *sum of infeasibilities* outperforms the other heuristics on a few instances. However, there are always a lot of instances where *sum of infeasibilities* times out while the other heuristic solves them, including a large group of instances solved by the other heuristic in only a couple of seconds.

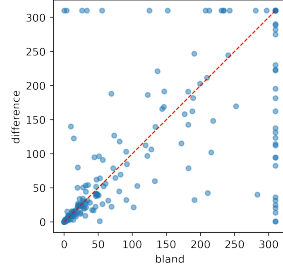


(a) Full profile.

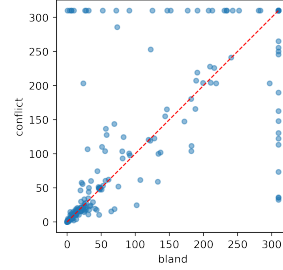


(b) Enlarged section.

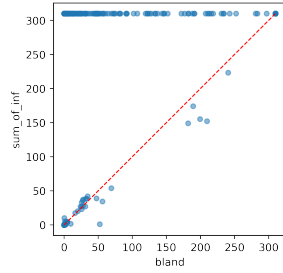
Figure 5.12: Performance profile for all heuristics. Depicts the number of instances that are solved in at most y seconds where y is the value of the y -axis.



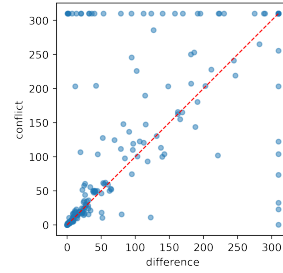
(a) *Bland's rule and difference-based heuristic.*



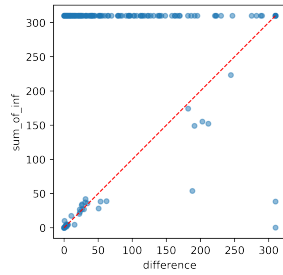
(b) *Bland's rule and conflict-based heuristic.*



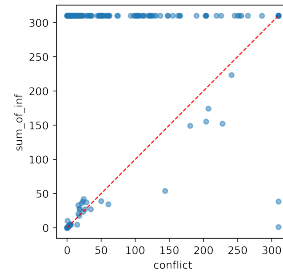
(c) *Bland's rule and sum of infeasibilities.*



(d) *Conflict-based and difference-based heuristic.*



(e) *Difference-based heuristic and sum of infeasibilities.*



(f) *Conflict-based heuristic and sum of infeasibilities.*

Figure 5.13: Scatterplots for all possible combinations of two heuristics.

average number of pivots	Bland	difference	conflict	soi
total average	70	502	132	48567
average unsolved	149	1268	301	83648

Figure 5.14: Average number of pivots per theory call over all instances and over the unsolved instances only.

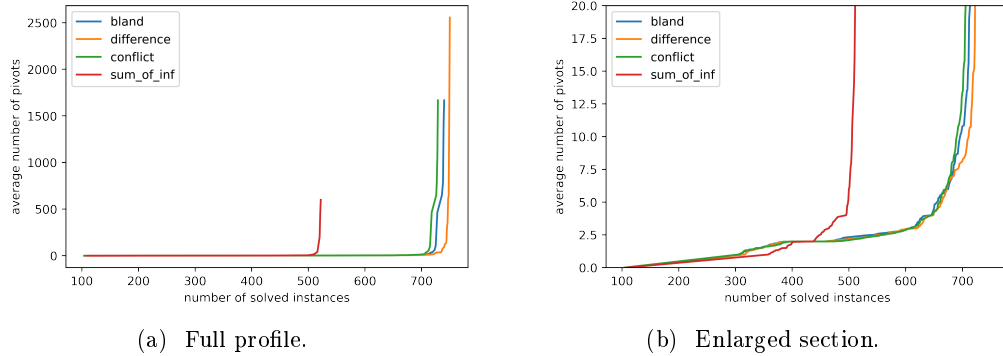


Figure 5.15: Performance profile for the *average number of pivots*. Note that only solved instances are considered here.

5.4.4 Collected Statistics

Most of the collected statistics were already implemented in the `LRAModuleStatistics` class of the `LRAModule`. We fixed some bugs causing the existing statistics to output distorted results in certain cases and implemented a few timers.

Average number of pivots

The *average number of pivots* is obtained by taking the *total number of pivot operations* the solver performed when executed on the instance and dividing it by the *total number of theory calls* performed. Since the *average number of pivots* is the main parameter Simplex heuristics try to optimize, we took a close look at this statistic. We collected the *average number of pivots* for each instance and computed the average over all these values. We then repeated this computation, but instead of computing the average over all instances, we only considered the instances which the heuristic could not solve. Figure 5.14 shows the results.

The first thing to note is that the *average number of pivots* for *sum of infeasibilities* is several order of magnitudes higher than those for the other heuristics. This is again inconsistent with the statements made in [KBD13]. Secondly, even though the *difference-based* heuristic performed better than *Bland's rule* and the *conflict-based* heuristic, its *average number of pivots* is significantly higher. To get a better understanding of these numbers, we generated the performance profile for the *average number of pivots*, which is shown in Figure 5.15.

The performance profile in Figure 5.15 has a shape similar to the one in Figure 5.12. The *average number of pivots* shows largely the same behavior for all heuristics: For

the vast majority of the solved instances, the *average number of pivots* is extremely small even though there are a few solved instances with a high *average number of pivots*. More precisely, all heuristics have an average number of less than 5 pivots per theory call on around 80-90% of their solved instances. This leaves little room for optimization in terms of heuristics. Unsurprisingly, all heuristics have basically the same *average number of pivots* for these instances, with the exception of *sum of infeasibilities* having far fewer such instances but also having a slightly lower *average number of pivots* for a bunch of them. The similarity of the performance profiles from Figure 5.12 and Figure 5.15 indicates that the instances with a low *runtime* and those with a low *average number of pivots* are mostly identical. For these instances, optimization in terms of heuristics is likely not feasible, leaving only the few instances solved with a high *average number of pivots* and the unsolved instances. At this point, the reader may notice that the number of instances where optimization in terms of heuristics is possible is already beginning to drop rapidly. First, roughly 30% of the 1753 instances were proven to be irrelevant due to the preprocessing performed by the `FPPModule` and now optimization is indicated to be infeasible for another 600-700 instances.

On a side note, Figure 5.15 also shows that, contrary to what Figure 5.14 seems to indicate at first glance, the *difference-based* heuristic does not seem to need more pivot operations per theory call. The higher values for this heuristic are probably due to the *difference-based* heuristic being designed to not only decrease the *average number of pivots* but also the time required for one *pivot step* (see Section 3), thus allowing the heuristic to perform more pivots in the same time.

Timer

While observing the behavior of the *average number of pivots* already provided some helpful insight, there were still some open questions. For example, judging the optimization potential of *Bland's rule* with an average of 149 pivot operations per theory call on the unsolved instances was still difficult without knowing how much time the pivoting process took in relation to the total runtime of the solver. Because of this, we implemented three timers:

1. a timer that measures the time from the first theory call to the return of the last theory call, which we will refer to as *first-to-last timer*,
2. a timer that measures the total duration of all theory calls together, which we will refer to as *theory call timer* and
3. a timer that measures the total time spent pivoting (pivot selection + pivot step), which we will refer to as *pivoting timer*.

It is important to note that the times measured by the timers are subsets of each other, that is the time measured by the *pivoting timer* is contained in the time measured by the *theory call timer* which is in turn contained in the time measured by the *first-to-last timer*. We were interested in the percentage of the measured times in relation to the total runtime rather than the measured times itself. In the following, we will discuss the insights gained from these percentages.

In the previous subsection about the *average number of pivots*, we explained that optimization is most likely infeasible for 600-700 instances because the *average number of pivots* per theory call needed by all heuristics (including something as simple as

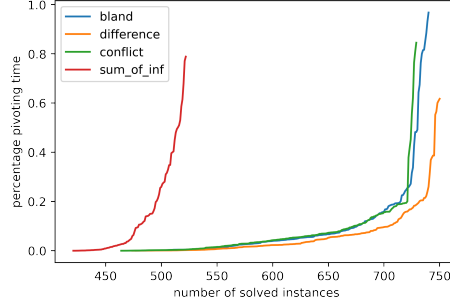


Figure 5.16: Performance profile for the statistic *percentage pivoting time*. The statistic is computed by taking the time measured by the *pivoting timer* and dividing it by the total *runtime*.

Bland's rule) is already extremely small. However, as mentioned before, a heuristic can also try to minimize the cost of a pivot step instead of or in addition to minimizing the number of pivots. After the implementation of the *pivoting timer* we were able to gauge the feasibility of this option by taking a look at the performance profile depicted in Figure 5.16. The profile shows that around 650 instances are solved with the *pivoting time* making up only 10% or less of the total runtime (again this is true for all heuristics except *sum of infeasibilities*). It is reasonable to assume that given the shape of the profile, which is again similar to the shape of the previous profiles, these instances are mostly identical with the instances where the *average number of pivots* was extremely small. We conclude that optimizing the cost of the pivot step is unlikely to lead to major performance improvements on these instances.

Looking at the performance profiles in Figure 5.17 and comparing them to the one in Figure 5.16, we note that even on the instances solved by *sum of infeasibilities*, the theory solver spends most of its time pivoting. Additionally, we note two things about the other heuristics:

- There is a significant number of instances where the time measured by the *first-to-last timer* is only a fraction of the total runtime. This is probably due to the `FPPModule` taking a lot of time for preprocessing.
- On the other hand, there is also a significant number of instances where both the time measured by the *first-to-last timer* and the time measured by the *theory call* timer is notably higher than the pivoting time. Therefore, even though there seems to be little to gain by optimizing the used heuristic, optimizing other parts of the Simplex algorithm implemented in the `LRAModule` may lead to some measure of improvement.

Like we did with the *average number of pivots*, we also computed the average of these percentages both over all instances and over the unsolved instances only. Figure 5.18 shows the results. We were especially interested in the averages computed over the unsolved instances, since these are the instances we know the least about so far.

Looking at the percentages for the *pivoting time*, we note that while the values are a bit higher for the unsolved instances, they are still pretty small for the first three heuristics (the *conflict-based* heuristic having the highest percentage which is

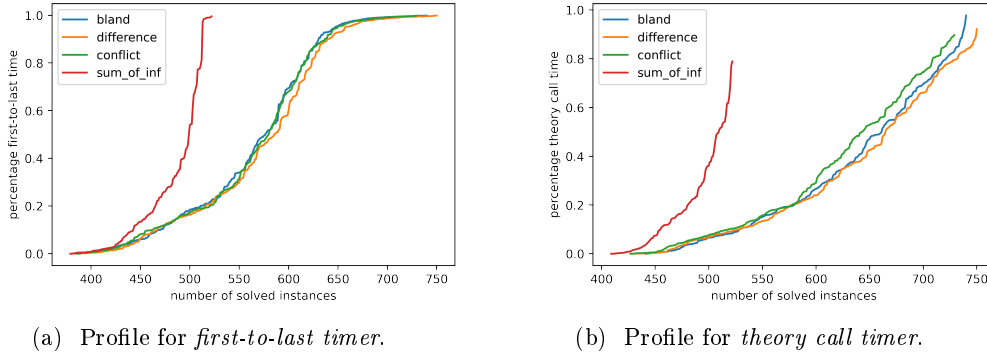


Figure 5.17: Performance profiles for the *first-to-last timer* and the *theory call timer*. The statistic is computed by taking the time measured by the timer and dividing it by the total runtime.

	Bland	difference	conflict	soi
% pivoting				
total average	9.17	8.22	10.50	54.05
average unsolved	10.20	10.91	15.94	88.35
% theory call				
total average	26.02	27.47	28.22	55.29
average unsolved	38.20	43.10	44.22	89.42
% first-to-last				
total average	43.09	43.53	43.95	56.99
average unsolved	68.46	71.23	71.62	91.27

Figure 5.18: Average values of the percentages derived from the measured times over all instances and over the unsolved instances only.

still below 16%). Therefore, we come to the conclusion that, like with the solved instances, optimizing the heuristic is unlikely to lead to major performance improvements. However, looking back at Figure 5.14, the *average number of pivots* for the unsolved instances (ranging from 149 to 1268 for the first three heuristics) is high enough that reduction by optimization of the heuristic may be possible, even though the performance gains will, as stated, likely be relatively minor.

The average value for the *first-to-last timer* over the unsolved instances is fairly high across the board, making up roughly 70% of the runtime for the first three heuristics and roughly 90% for *sum of infeasibilities*. That means that the time taken by the FPPModule preprocessing the input is fairly small (although we need to keep in mind that the instances where the FPPModule timed out are already excluded here). Combined with the values for the *theory call timer*, this leaves some possibility for improving performance by optimizing the theory calls and the LRAModule in general.

Figure 5.19 visualizes the relation between the three different times measured and the total runtime for *Bland's rule*, since it is both the most basic heuristic and the

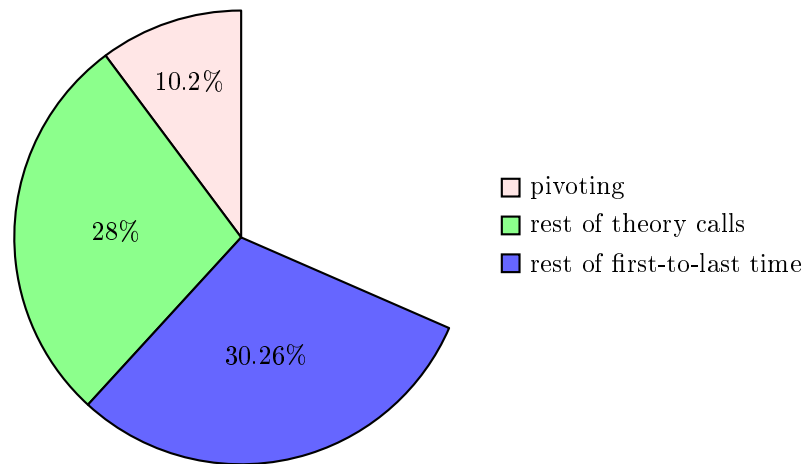


Figure 5.19: Pie diagram depicting the relation between the three different times measured and the total runtime for *Bland's rule*. The diagram is based on the average percentages over the unsolved instances.

one initially used in the *LRA*Module.

Chapter 6

Conclusion

6.1 Summary

We explained four heuristics for the pivot selection in general Simplex, taken from various sources. We gave a proof of termination for *Bland’s rule* that is a more in-depth version of the proof from [DdM06]. We implemented the four heuristics as well as a small framework that enables modular switching of the used heuristic during runtime and simplifies changes to existing heuristics and the addition of new ones. We discovered that the first three heuristics showed a surprisingly similar performance when benchmarked, which is inconsistent with the results described in [Gri09, KBD13]. We then performed extensive testing to find the cause for this. The tests showed that the preprocessing performed by the `FPPModule` rendered the theory solver completely irrelevant for roughly 30% of the instances and that even on the remaining 70%, minimizing the *pivoting time* can only lead to very small performance gains. On the other hand, the tests indicated some potential for optimizing the theory solver in general.

6.2 Discussion

We believe that the divergence between the performance of the *difference-based* heuristic and *sum of infeasibilities* observed in [Gri09, KBD13] and their performance observed in this thesis is at least partly due to the different environments in which they were tested. While we implemented the two heuristics in the `LRAModule` of `SMT-RAT`, [Gri09] presents an implementation of the *difference-based* heuristic for the SMT solver `MATHSAT` and [KBD13] presents an implementation of *sum of infeasibilities* for the SMT solver `CVC4`. Additionally, the solvers were most likely run with different configurations. Therefore, the divergent results could be caused by differences in the preprocessing, the active extensions of the SAT solver and the theory solver, the underlying data structures, etc.

There are also some things to note about the two heuristics themselves. Firstly, while the *difference-based* heuristic is very simple and straightforward, *sum of infeasibilities* is far more complex and therefore our implementation is likely to differ from the implementation in [KBD13]. Secondly, we believe that the high *average number of pivots* observed in Section 5.4 is the result of *sum of infeasibility* being particularly prone to *cycling* and that this is also one of the main causes for the significantly worse

performance of the heuristic in our tests.

Regarding the similarities between *Bland’s rule*, the *difference-based* heuristic and the *conflict-based* heuristic, we believe the `FPPModule` to be one of the major causes for those. Our hypothesis is that, apart from rendering the theory solver irrelevant for 30% of the instances, it also tends to bring the input formula in a form that leaves little room for divergence in terms of heuristics.

6.3 Future work

We do not believe that introducing new heuristics to SMT-RAT holds much promise unless the setup differs from the one used in this thesis. Future work on Simplex therefore requires finding a setup for SMT-RAT where the chosen Simplex heuristic has a significantly higher effect on the overall performance of the solver. If our hypothesis about the `FPPModule` turns out to be correct, deactivating it could solve the problem. However, depending on how much performance is lost when preprocessing is deactivated, this might not be desirable. Alternatively, one could either modify the `FPPModule` or introduce an entirely different preprocessing module which increases performance while still leaving some optimization potential in terms of Simplex heuristics.

For the sake of completeness we also want to mention that there is still one important parameter influenced by the Simplex heuristic that isn’t thoroughly tested yet. In our tests, we focused on the effect of the chosen heuristic on the *average number of pivots* and the *pivoting time* in general. However, there is a third important parameter in this regard, the *conflict* returned in case the passed formula is unsatisfiable. We didn’t test this parameter thoroughly because since the returned *conflict* influences the behaviour of the SAT solver (including subsequent theory calls), the effects of changes to this parameter are very hard to measure. Additionally, since the modified conflict check implemented in this thesis had little effect on the performance (contrary to the observations in [KBD13]), we don’t think this approach holds much promise either.

Finally, future work could also focus on optimizing the theory solver in general instead of the Simplex heuristic.

Bibliography

- [AC78] D. Avis and V. Chvátal. Notes on Bland’s pivoting rule. In *Polyhedral Combinatorics*, pages 24–34. Springer, Berlin, Heidelberg, 1978.
- [Ali20] F. Alieff. Simplex heuristics in SMT solving. https://ths.rwth-aachen.de/wp-content/uploads/sites/4/alieff_bachelor.pdf, 2020. Bachelor Thesis. RWTH Aachen University.
- [BFT16] C. Barret, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [Bla77] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of operations Research*, 2(2):103–107, 1977.
- [CKJ+15] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám. SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 360–368. Springer, Berlin, Heidelberg, 2015.
- [Dan63] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- [DdM06] B. Dutertre and L. de Moura. Integrating simplex with DPLL(T). Technical Report SRI-CSL-06-01, Computer Science Laboratory, SRI International, 2006.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518. Springer, Berlin, Heidelberg, 2003.
- [GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *16th International Conference on Computer Aided Verification (CAV)*, pages 175–188. Springer, Berlin, Heidelberg, 2004.
- [Gri09] A. Griggio. *An Effective SMT Engine for Formal Verification*. PhD thesis, University of Trento, 2009.
- [KBD13] T. King, C. Barrett, and B. Dutertre. Simplex with sum of infeasibilities for SMT. In *2013 Formal Methods in Computer-Aided Design (FMCAD)*, pages 189–196. IEEE, 2013.

- [KS16] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, Berlin, Heidelberg, 2016.
- [MB11] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.
- [MSS96] J. P. Marques-Silva and K. 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.
- [Nal20] J. Nalbach. A novel adaption of the simplex algorithm for linear real arithmetic. Master’s thesis, RWTH Aachen University, 2020.
- [NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
- [Tse83] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pages 466–483. Springer, Berlin, Heidelberg, 1983.