hybr2d Theory of Hybrid Systems Informatik 2 | RWTH AACHEN UNIVERSITY

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

BACHEOR OF SCIENCE THESIS

AN FMPLEX-INSPIRED SIMPLEX HEURISTICS

Kai Hilgers

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Marco Lübbecke

Additional Advisor:
Jasper Nalbach

Aachen, 01.03.2022

Abstract

The purpose of a Simplex heuristic is to reduce the need for costly pivot steps. However, most heuristics only consider local criteria - they ignore the previous steps. This paper investigates a pivoting rule, that limits which variables can be selected based on their bounds, effectively reducing the dimensions of a problem when only a few bounds in one direction exist. Our approach is based on the FMplex algorithm [Kob] but implemented as an adapted Simplex heuristic. We have built a prototype and experimentally evaluated our performance using the QF_LRA benchmarks from the SMT-LIB library [SMT]. Our results are comparable to Bland's Rule [Bla] both in number of problems solved and performance. Given the prototypical implementation, this suggests, that it is worth while to keep track of previous steps, as well as eliminating opposite bounds, when applicable, to speed up the Simplex algorithm.

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.

Kai Hilgers
Aachen, den 28. März 2022

Contents

1 Introduction **9**
- 1.1 Motivation 9
- 1.2 Related work 10
- 1.3 Own contribution 10

2 Preliminaries **11**
- 2.1 Boolean satisfiability problem SAT 11
- 2.2 Satisfiability modulo theories SMT 11
- 2.3 Linear real arithmetic LRA 11
- 2.4 Fourier Motzkin (FM) variable elimination 12
- 2.5 FMplex 12
- 2.6 Simplex 13

3 FMplex Simplex heuristic **17**
- 3.1 Considering only one direction for each original variable 17
- 3.2 Required steps for the algorithm 18
- 3.3 Construction 19
- 3.4 Preventing dead ends 22
- 3.5 Entering rule 23
- 3.6 Exit rule 24
- 3.7 Soundness 24
- 3.8 Observation of required pivoting steps 27
- 3.9 Results 27

4 Conclusion **31**
- 4.1 Summary 31
- 4.2 Discussion 31
- 4.3 Future work 32

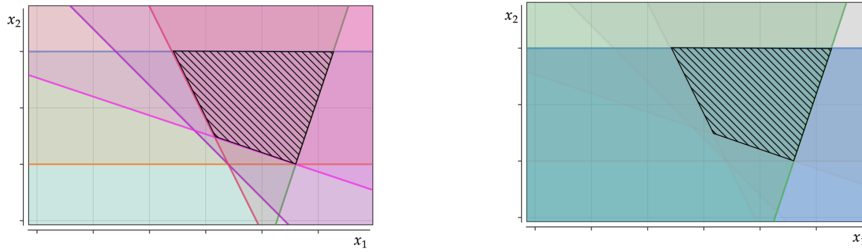
Bibliography **33**

Chapter 1

Introduction

1.1 Motivation

Many real world problems can be expressed as Boolean combinations of inequations between linear polynomials. These problems can be grouped together into the quantifier free linear real arithmetic (QF_LRA). We are mostly interested in whether or not a solution in these problems exists or if the problem is unsatisfiable. This process is called satisfiability (SAT) checking. Simplex is one such satisfiability checking method for QF_LRA and is broadly used. The Simplex algorithm maintains a tableau representation of the problem and a variable assignment. When searching for solutions in the Simplex context, we encounter bounds that are violated by the current assignment. In this case, Simplex applies a pivot step, which adapts the assignment to satisfy the previously violated bound. Most of the time these violated bounds will not be the tightest bound in a specific direction. This results in pivoting with the given variable and later pivoting with another variable that was the actual largest bound. Especially in larger problems, it is unlikely that we pivot with the correct bound in the beginning. If we could know, for each variable, which is the most restricting bound beforehand, we could solve the problem in linear time, in the number of original variables. However, we would need to compute the restrictiveness of a variable for each of the bounds. Additionally, the restrictiveness depends on the previously selected bounds. So we would need to compute a lot of different possibilities. Instead of doing these computations, in this work we guess that a constraint is the most restrictive and proceed forward. If we encounter a contradicting constraint to our previously assumed largest bounds, we can backtrack and assume that this constraint is now the most restrictive. This approach was used in a modified Fourier Motzkin (FM) variable elimination algorithm [Kob], where iteratively guessing and backtracking reduced the complexity of the problem. In this previous work, a static variable ordering obtained from this idea was proposed and tested. While it did not surpass Bland's Rule in performance, we believe that the idea is still worth iterating on. Additionally, the FM approach currently does not support incrementality. We attempt to transform this attempt into a Simplex heuristic, where we restrict which variables we can pivot with in respect to a direction. Our goal is to essentially reduce the dimensionality of the problem in each step and greatly reduce the number of pivot steps required. We aim to support the incremental approach used in modern SMT solvers.



(a) Complex problem with many constraints (b) Reduced problem with only upper bounds on either x_1 or x_2

Figure 1.1: Demonstration of our approach. By essentially ignoring some type of bounds (in this case the lower bounds on x_1 and x_2) we can reduce the complexity of the problem.

1.2 Related work

Our work is based on the Simplex algorithm [Dan] however, we aim to restrict which variables are suitable for pivoting even further, in an attempt to reduce the amount of combinations that need to be computed. This idea is based on the approach used in [Kob]. Instead of eagerly computing each upper and lower bound in the Fourier Motzkin algorithm, only one upper or lower bound was chosen and assumed to be the most restrictive. This reduced the complexity from doubly exponential to being singly exponential. Our goal for this thesis is to reproduce the steps that FMplex takes in a Simplex heuristic.

1.3 Own contribution

Our contributions in this thesis are

- Converting the FMplex algorithm into a Simplex heuristic. We explain how to obtain a Simplex heuristic and provide an implementation to the SMT-RAT [CKJ$^+$] codebase.
- Comparing differences that arise in the Simplex context. Introducing assignments from the Simplex algorithm, that were not present in FM, lead to an issue that we resolved.
- Experimentally evaluating our method when compared to other heuristics.
- Obtaining a deeper understanding about the structure of SMT solving to inspire future research.

Chapter 2

Preliminaries

In the following chapter we will introduce all the preliminaries that our work is based on. We explain the Simplex algorithm and show the steps that are needed to form a heuristic.

2.1 Boolean satisfiability problem SAT

In the SAT problem we receive as input a boolean formula and need to determine, whether a satisfying assignment to the variables exists. This is an NP-complete problem as shown by Cook and Levin independently [Coo]. Hence finding efficient solutions is of great interest as all NP problems can be expressed as a SAT problem.

2.2 Satisfiability modulo theories SMT

One can extend the SAT problem with theories such as bit-vectors, lists and real numbers. Where in the SAT problem we have literals such as x and y, in the SMT problem we could, for example, have uninterpreted functions f. Valid problems could then be $f(f(x,y),y) = f(x,y)$ where f is one such function. An SMT solver now uses the SAT solver to find possible solutions in the boolean structure of the problem and then checks whether these solutions are consistent with the underlying theory. In this paper we focus on solving (quantifier free) linear real arithmetic (LRA) problems.

2.3 Linear real arithmetic LRA

Definition 2.3.1 (LRA)**.** *Linear Real Arithmetic is a first order theory using the signature of* $(\mathbb{R},0,1,+,<)$

Definition 2.3.2 (Constraint)**.** *Let* $x_1,\ldots,x_n$ *be variables and* $a_1,\ldots,a_n,b \in \mathbb{R}$ *constants.*
We define a linear constraint as $\sum_{i=1}^{n}(a_i \cdot x_i) \bowtie b$*, where* $\bowtie \in \{<, \leq, =, >, \geq\}$*.*

Definition 2.3.3 (Most restrictive constraint)**.** *Assume variables* $x_1,\ldots,x_n$*, constraints* c_1,c_2*, assignment* $\alpha : X \mapsto \mathbb{R}$ *and an index set* $I \subseteq \{1,\ldots,n\}$*. Let* c_1' *and* c_2' *results from* c_1 *respectively* c_2 *by substituting* $\alpha(x_i)$ *for each* x_i *with* $i \in I$*. We define* $c_1 \leq_{\alpha,I} c_2$ *iff* c_2' *implies* c_1'*, i.e. if all assignments that satisfy* c_1' *also satisfy* c_2'*.*

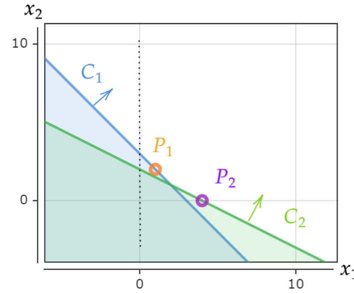


Figure 2.1: Most restrictive constraint visualization

Example 2.3.1. *Let consider $C_1 := x_1 + x_2 \leq 3$ and $C_2 := x_1 + 2x_2 \leq 4$ (see Figure 2.1). Now initially with an empty α and empty I neither $C_1 <_{\alpha,I} C_2$ nor $C_2 <_{\alpha,I} C_1$ holds, since $P_1 = (1{,}2)$ is contradicting to $C_1 <_{\alpha,I} C_2$ and $P_2 = (4{,}0)$ is contradicting $C_2 <_{\alpha,I} C_1$. However when we have $I = \{x_1\}$ and $\alpha : x_1 \mapsto 0$ it holds that $C_1 <_{\alpha,I} C_2$, since C_1 simplifies to $x_2 \leq 3$ and C_2 to $x_2 \leq 2$. We can now see that C_2 is more restrictive than C_1 since every solution to C_2 is a solution in C_1. The visual intuition behind this is that on the dotted line - where $x = 0$, the green area of C_2 is a sub-area of the blue one of C_1.*

2.4 Fourier Motzkin (FM) variable elimination

The Fourier Motzkin [Fou] variable elimination method can be used to solve LRA systems of inequations. While we do not apply the computation in this paper, it is still useful to understand the procedure, since our heuristic works in a similar manner. The elimination works in two steps. First, a variable is chosen. Then each inequation is rewritten as either a lower or an upper bound to this variable, or no bound at all. In the second step, new equations are generated by combining every lower bound with every upper bound. Therefore, in every step, one variable is eliminated. While this is a straight-forward algorithm, it is not efficient in practice, since combining every lower and upper bound leads to a combinatorial blow-up.

2.5 FMplex

The FMplex algorithm [Kob], that our heuristic is based on, aims to reduce the combinatorial blow-up in the FM algorithm by assuming that one opposite bound is the largest bound. Instead of combining every lower with every upper bound, every lower bound only gets combined with this one upper bound. If one finds a SAT solution, then it is of course valid. However, in the UNSAT case, one has to find out, whether the system is UNSAT as a whole, or if the wrong bound was chosen. One can backtrack to the position where the conflict originated from, and select a different upper bound. If no such bound can be chosen, the conflict has to occur even prior to this selection. Finally, one can find the conflict in the first step and prove, that the system is UNSAT.

We propose a Simplex heuristic based on this approach and aim to take the exact same steps, that FMplex would take.

2.6 Simplex

The Simplex algorithm proposed by Dantzig [Dan] in 1990 improved upon the previous Fourier Motzkin variable elimination approach. The input is a set of QF_LRA constraints over some variable set X. We now transform these inequalities into equalities as follows. For each constraint c_j of the form $\sum_{i=1}^{n}(a_i \cdot x_i) \bowtie b$ we introduce a new so called *slack variable* s_j to encode the non-constant part of the inequation, setting $\sum_{i=1}^{n}(a_i \cdot x_i) = s_j$. To compensate for the lost lower and upper bounds, we introduce the bound $s_j \bowtie b$. Notice that when we find values for our so-called *original* variables X and our slack variables S that satisfy both the equalities as well as the bounds, we will have a satisfying *assignment* to our original problem. When combining our original variables as well as the slack variables, we obtain $\mathcal{V} = X \cup S$. We partition this set into *basic* variables $\mathcal{B} := S$ as well as *nonbasic* variables $\mathcal{N} := X$. The constraints can now be rewritten as a $|\mathcal{V}| \times |\mathcal{V}|$ matrix called *tableau* T. We denote by $t_{i,j}$ the value of the tableau in the i-th row and j-th column.

For every variable $v_i \in \mathcal{V}$, if v_i is a slack variable, the i-th row contains the factors in the equation for s_i , and -1 at $t_{i,i}$. Otherwise v_i is an original variable and the row contains zeroes only. Additionally we require the *lower* and *upper* bounds $l,u \in \mathbb{R}^{|\mathcal{V}|}$ where $l_i = -\infty$ and $u_i = \infty$ except where $s_i \bowtie b$ in this case we insert the respective lower or upper bound b.

We now need to assure that $T \cdot \alpha(\mathcal{V}) = 0$ and $l \leq \alpha(\mathcal{V}) \leq u$ holds to find a satisfying solution to the original problem. Here, $\alpha(\mathcal{V})$ assigns each variable in $\mathcal{V}$ a specific value from $\mathbb{R}$. There are two invariants that hold during the Simplex algorithm.

1. $T \cdot \alpha(\mathcal{V}) = 0$
2. Every nonbasic variable satisfies its lower and upper bound.

In the beginning this holds, since all variables are assigned the value 0 and each of the nonbasic variables is an original variable - that does not have a bound in this transformed form. It can, however, happen that a basic variable violates its bound. In this case, the Simplex algorithm uses a pivot method to update the assignment of the variables.

VOID UPDATE (int j, float δ)
1 $\alpha(v_j) \leftarrow \alpha(v_j) + \delta$
2 **for all** $i \in [1, \ldots, |\mathcal{V}|]$ **with** $t_{i,j} \neq 0$
3 **do** $\alpha(v_i) \leftarrow \alpha(v_i) + t_{i,j} \cdot \delta$
[KBD]

Algorithm 1: Update the assignment of the *nonbasic* variable v_j by δ

VOID PIVOT (int i, int j)
1 $r_j \leftarrow r_j - \frac{1}{t_{i,j}} \cdot r_i$
2 **for all** k with $v_k \in \mathcal{B} \wedge k \neq j$
3 **do** $r_k \leftarrow r_k + t_{k,j} \cdot r_j$
4 $\mathcal{B} \leftarrow (\mathcal{B} - \{v_i\}) \cup \{v_j\}$
5 $\mathcal{N} \leftarrow \mathcal{V} \backslash \mathcal{B}$
[KBD]

Algorithm 2: Pivot the *basic* variable v_i with the *nonbasic* variable v_j where $t_{i,j} \neq 0$

void UpdateAndPivot (int i, int j, float δ)
1 UPDATE(j, δ)
2 **if** $i \neq j$
3 **then** PIVOT(i, j)
[KBD]

Algorithm 3: Update the assignment for the basic variable v_j by δ and pivot v_i with v_j

During the UpdateAndPivot method both, invariants are maintained. One can now repeatedly use this method to resolve every conflict until a solution is found or no variable is *suitable for pivoting.*

2.6.1 Suitable for pivoting

When a basic variable $x_i \in \mathcal{B}$ violates any of its bounds, x_i is *violating.* Assume that $l(x_i) > \alpha(x_i)$ i.e. the lower bound is greater than the assignment. We can now consider a nonbasic variable $x_j \in \mathcal{N}$. There exist three cases.

$t_{i,j} > 0$ The factor of x_j is positive. Increasing the assignment of $t_{i,j}$ will increase x_i. If x_j is not already at its upper bound $\alpha(x_j) < u(x_j)$ we call x_j suitable for pivoting.

$t_{i,j} = 0$ Changing the assignment of x_j will not change the assignment of x_i. We do not need to consider this pair.

$t_{i,j} < 0$ The factor of x_j is negative. Decreasing the assignment of t_{i_j} will increase x_i. If x_j is not already at its lower bound $l(x_j) < \alpha(x_j)$ we call x_j suitable for pivoting.

In practice, multiple basic variables will be violating with multiple nonbasic variables being suitable for pivoting. We need a rule for selecting which basic variable to choose and then which nonbasic variable to pivot with. These rules are called *entering rule* and *exit rule* respectively.

2.6.2 Entering rule

At every pivot step, some variable will become nonbasic. Selecting which variable to choose has a significant impact on performance. Consider, for example, the case where we have multiple lower bounds that each subsume another.

Example 2.6.1. *Assume $s_1 = y, s_1 \leq 3$, $s_2 = y, s_2 \leq 2$, $s_3 = y, s_3 \leq 1$ When we now pivot with s_1 first, s_2 and s_3 will still be violating, and therefore, we will need an additional pivot step as opposed to pivoting with s_3 immediately - since it subsumes the other bounds.*

2.6.3 Exit rule

After selecting a variable according to the entering rule, we also need a variable to become basic and leave the nonbasic variables. This variable has to be suitable for pivoting. This selection will again have performance implications. One should aim to keep the most restrictive bounds in the basis to ensure that the problem reduces in dimensionality.

2.6.4 Heuristic

Such rules for entering and leaving the basis form a *heuristic.* Several such heuristics exist and all of them aim to reduce the number of pivot steps required. Whereas correctness does not depend on the heuristics, to ensure that the Simplex algorithm is still complete, the heuristic needs to be *terminating.* To ensure termination in the Simplex algorithm, it is sufficient to ensure, that no pivoting cycles exist. Some heuristics are not terminating themselves, but use a complete heuristic such as Bland's Rule [Bla] as a fallback after a given amount of pivot steps. In Bland's Rule, variables are ordered and when a choice needs to be made, the smallest possible variable is taken, with ties being broken by selecting the smallest index.

Chapter 3

FMplex Simplex heuristic

3.1 Considering only one direction for each original variable

Our hypotheses is that it is valid to only consider either the lower or the upper bounds for a given original variable but not both. If we do not find a solution under this relaxed set of bounds, no solution exists.

We show this with an inductive proof. Consider the one-dimensional case. When there exist no upper and lower bounds, the problem is unbounded, and it is of course valid to only consider lower or upper bounds - since no bounds exist.

In case the problem is unbounded in one direction, we will have to ensure that the other direction is selected as the bounds that we chose. Then the opposite set is empty, just like in the solution.

Otherwise, if there is a solution then at this point, one of the lower bounds will be the largest and one of the upper bounds will be the smallest.

Now consider that we have an n-dimensional problem that we know has a solution where we consider only one direction for each dimension. When we now consider the $n + 1$-dimensional case, we can project the original solution space onto the new axis. The projection will again be an interval with lower and upper bounds since the solutions are always convex. We will again be able to choose whether to consider the upper and lower bounds - for each of them a solution exists.

Example 3.1.1. *Consider the cube in Figure 3.1. For this example we only consider the upper bounds on each original variable x,y,z. On the cube the three upper bounds are marked in a dark shade of red.*

In the first step we project the cube onto the x-y plane. Now we have only two upper bounds, one for each dimension. We choose the upper bound on y and continue. Finally, we arrive at the last dimension where we only consider the closed interval [1,2]. Since we only consider the upper bounds, we choose 2 (point A") and arrive at a solution to the problem. Substituting back into the previous projections, you can see how we obtained A" from A', which we got from A. Different selection of lower and upper bounds, with different variable orderings can lead us to selecting any 8 of the cubes vertices.

Of course, we still have to consider that is has to be possible to reach these upper bounds in the Simplex context.

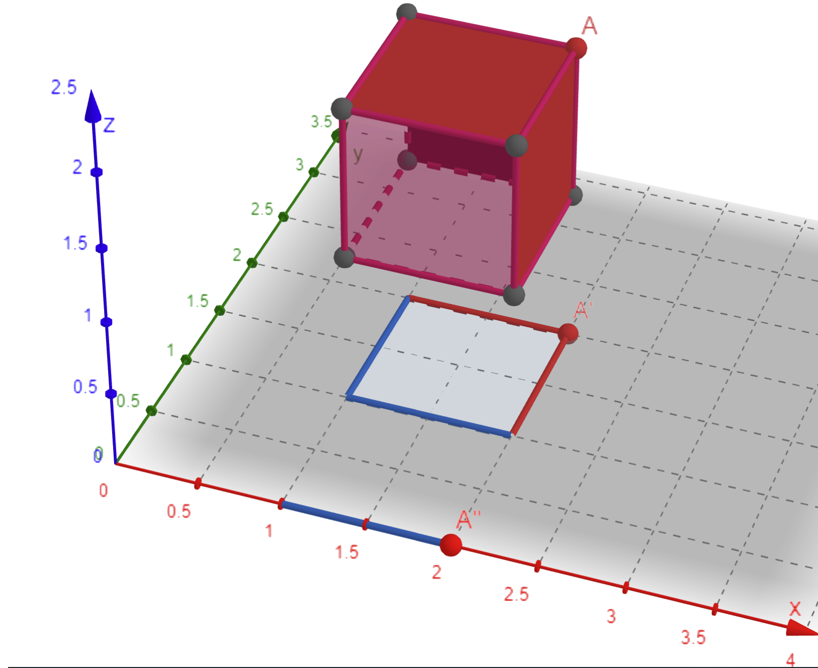


Figure 3.1: Projection of a cube solution to each underlying dimension

3.2 Required steps for the algorithm

- For each original variable compute which constraints are lower or upper bounds for this variable.
- Introduce a data structure for our bookkeeping efforts.
- Form a complete pivoting heuristic, consisting of an entering and leaving rule as well as updating our data structure.
- Work incrementally and generate infeasible subsets. While this step is implemented in the provided source code, the introduction of incrementality would exceed the limitations of this thesis.

3.2.1 Classification into lower and upper bounds

We consider a slack variable s_i. s_i can have one of three bounds.

$l \leq s_i$ **:** s_i is a lower bound

$s_i \leq u$ **:** s_i is an upper bound

$l \leq s_i \leq u$ **:** s_i has both bounds (we use this to represent equalities)

Now it also holds before doing any pivot steps that $s_i = \sum_{j=1}^{n} \alpha_j \cdot x_j$

We can classify the pair (s_i,α_j) to be the bound that we obtain from Table 3.1

	s_i is lower	s_i is upper	s_i is both
$\alpha_j > 0$	lower	upper	both
$\alpha_j < 0$	upper	lower	both

Table 3.1: Classification of the bound for the pair (s_i, α_j)

Example 3.2.1. *Let us consider the following constraint:*
$S_1 = -2 \cdot x_1 + 4 \cdot x_2$ *where* $S_1 \leq 3$. *We notice that* S_1 *is an upper bound.*
Now for each of the original constraints we obtain from Table 3.1 in the "S_1 *is upper" column.*

(S_1, x_1) *is a lower bound, since -2 is negative.*

(S_1, x_2) *is an upper bound, since 4 is positive.*

To speed up future lookups, we construct a mapping where we map each slack variable and bound pair to the list of original dependencies.

3.3 Construction

To store information about the steps that we have already taken, we construct the following list of *containers*. Each container contains the following four information:

Original variable x_i The original variable selected at this step.

Original bound b_i Whether we collect lower, upper or both bounds for this original variable.

Selected s_{selected} The current selected variable for this original variable and bound.

Tested S_{tested} The previous tested variables for this combination of original variable and bound.

There are now a few properties that we will have to ensure to hold at every step of the algorithm.

I $s_{\text{selected}} \in S_{\text{tested}}$

II $|S_{\text{tested}}|$ monotonically increases - we do not allow any tested variables to be removed.

III $s_{\text{selected}} \leftarrow s'_{\text{selected}} \implies s'_{\text{selected}} \notin S_{\text{tested}}$ if we select a variable, we have not yet tested it.

3.3.1 Method backtrackingLength

Assume a constraint s_i that violates its bound b. We can now compute the backtracking length required to insert this constraint into our list.

We begin by collecting all original dependencies of this constraint-bound combination as defined in Table 3.1. We then iterate over our list backwards and keep track of the length until the following condition is met.

The original variable of the container equals one of the original dependencies of our constraint.

The bound in the container equals the bound of our constraint.

The constraint is not already in the tested set.

We can then return the length to the first container that fulfills all of these conditions. For an easier understanding of this algorithm, we instead calculate the distance for each original dependency and bound and return the minimum - in practice this is inefficient.

```
INT BACKTRACKINGLENGTH ( Variable original, Variable constraint, Bound bound)
1  i ← 0
2  for container ∈ reversed(Containers)
3  do i ← i + 1
4      if container.originalVariable = original
5         ∧   container.bound = bound
6         ∧   constraint ∉ container.tested
7         then return i
```

Algorithm 4: Calculate the backtracking length of a given original variable, constraint and bound b

3.3.2 Method insert

When we insert a constraint s_i with its original dependence x_j and bound b, we start from the end of our list.

If the x_j is not already in our list of original variables, we can add a new container to the list and set its original variable to be x_j, the bound to be b and selected and tested to be s_i.

Otherwise we might require backtracking. While the original dependency differs from that of the list, we remove this container from the list. We repeat this until we arrive at the container with the correct original variable.

We then update the selected variable to be s_i and add s_i to the tested set.

```
VOID INSERT ( Variable originalVariable, Variable constraint , Bound bound)
 1  if originalVariable ∉ list.allOriginalVariables
 2    then
 3          list.append(constraint,originalVariable,bound)
 4          return
 5
 6  while container := Containers.popback()
 7  do if container.originalVariable = originalVariable
 8        ∧  container.bound == bound
 9        ∧  constraint ∉ container.tested
10        then container.tested.add(constraint)
11             container.selected = constraint
12             Containers.append(container)
13             return
14
```

Algorithm 5: Insert the constraint into our list

Notice that this ensures that at most one container for each original variables exists.

3.3.3 Dead ends

We initially believed that it could be sufficient to always only consider one direction for each variable.

One might assume that either the lower bounds are smaller than the upper bounds or that the system is unsatisfiable. This is incorrect though, since even in satisfiable systems there might exist assignments where the most restrictive lower bound is greater than the most restrictive upper bound. We will demonstrate this in an example shortly.

This issue did not appear in the FM implementation that this heuristic is based on, since no concrete variable assignments were required. Instead, only symbolic representations are used.

The following example will show that it is sometimes necessary to change the direction of the bounds.

Example 3.3.1. *Consider the example in Figure 3.2 our constraints are*
$s_1 = x_1 + x_2$, $2 \leq s_1$ *and* $s_2 = x_1$, $s_2 \leq 1$

In the beginning, only s_1 *is violated. We pivot with the first variable that is suitable, which is* x_1*. After inserting this into our list, we arrive at a conflict.The bound of the basis variable* s_2 *is now violated, but we can not pivot, because* s_2 *is an upper bound on* x_1*, but we chose to only consider lower bounds for* x_1*.*

This is a fundamental issue in the algorithm, and we identified three ways to address this.

In our current implementation we allow the switching of bounds, should such a problem occur. We can notice the issue when a basic variable is violated, but no original variable is suitable or requires backtracking. When we encounter such a

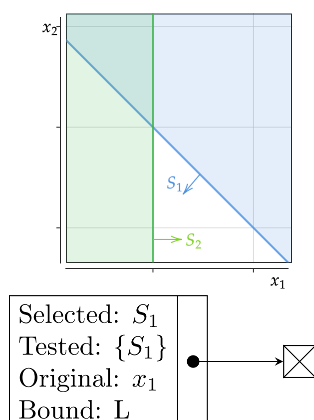


Figure 3.2: A visualization where bound switching is necessary

variable, we choose the original dependency with the shortest backtracking length and change the direction to now include the opposite bounds.

Obviously this results in basically reversing our previous efforts of only considering one direction, as well as removing all progress that we have already achieved in the previous direction.

Another option would be to essentially move the conflicting selected variable up through the list, until it is suitable for insertion. In our example we would move the conflicting selection S_1 up to its other possibility of a lower bound on x_2. This way we would free up the x_1 to now represent the upper bound S_2. Due to the limitations of this thesis, this approach is not yet implemented nor proven to be correct.

Instead of switching the direction of the bound, we believe that it is actually possible to prevent these cases from appearing in the first place.

3.4 Preventing dead ends

This section discusses ideas that have not yet been fully validated, but rather are intended to serve as a basis for future work.

At the moment we greedily insert the first original variable and bound for the variable, that we encounter. This results in two main issues.

First of all, we are more likely to encounter a bound when there are many of these bounds. However, we would prefer selecting the bound with the fewest constraints. Therefore, reducing our options the most, while still ensuring correctness.

Additionally, we can encounter the dead ends described in the previous section.

We believe that there is a way to resolve both of these issues with a bit of pre-computation.

We begin by iterating over all of our basic variables and classifying them into x_{i_L} and x_{i_U} as we are already doing. Instead of using these sets just for our lookups, we aim to arrive at an ordering that respects the following rule.

Let x_{i_B} be the selected bounds for x_i, $x_{i_B} = x_{i_L}$ or x_{i_U}.

Let now x_{i_s} be one variable in x_{i_B}. It must now hold that the $x_{j_B} \nsubseteq \{x_{i_s} | i < j\}$ for all possible selection of each x_{i_s}. The intuition behind this is as follows. If x_{j_B} would be a subset, then it would be possible to arrive at a point where we do not have a

basic variable to pivot into our basis for this original variable, effectively reducing the dimension of the problem which is not equisatisfiable as we have seen in the example. However this rule is not correct for some edge cases, consider for example the case where $s_1 = x + y$ is the only bound for x and y respectively, then with this rule no ordering would be possible, even though the system with s_1 and $s_2 = z$, $s_1 \leq 1, s_2 \leq 1$ is satisfiable.

Future work will form a concise rule that hopefully does not need to compute each of the possible orderings, as this would lead to combinatorial blow-up while aiming to select the smallest sets possible to reduce the amount of pivoting steps. We can still use the intuition behind this idea, to solve our previous example.

Example 3.4.1. *Assume* $s_1 = x_1 + x_2$*,* $2 \leq s_1$ *and* $s_2 = x_1$*,* $s_2 \leq 1$*.*

We compute $x_{1_L} = \{s_1\}, x_{1_U} = \{s_2\}$ *and* $x_{2_L} = \{s_1\}, x_{2_U} = \{\}$*.*

From the previous rule we can see that selecting x_{1_L} *is not a valid choice, since then both* x_{2_L} *and* x_{2_U} *are subsets of* x_{1_L} *so we exclude this possibility. We now consider selecting* x_{1_U}*, now* x_{2_L} *is valid, since* s_1 *is not in* x_{1_U}*. We arrive at the valid ordering* $x_{1_U} < x_{2_L}$*. Notice that we could also change the order to arrive at* $x_{2_L} < x_{1_U}$ *which would also be valid. Though this symmetry does not always hold.*

Using this fixed ordering in our algorithm would lead to a satisfying solution, because we would pivot s_1 with x_2 instead and arrive at a solution, since s_2 would still satisfy its bound. Additionally, future work should consider cases where x_{i_L} or x_{i_U} is empty because in this case, we would not need to pivot with any constraint and could instead just increase or decrease the value of x_i until we find a solution. The best case example for this is a system with many lower bounds on a large set of variables but no upper bounds on any of them, in this case setting all variables to large values will be satisfying.

3.5 Entering rule

When selecting which variable to become nonbasic, we have to consider the backtracking length defined earlier. We need to consider all possibilities for extending the list first, before we backtrack to a previous original variable. We check each basic variable that violates its bound and compute the backtracking length for every original dependency of this variable. We then return the variable with the shortest backtracking length.

```
VARIABLE ENTERINGRULE ()
 1  bestCandidate ← None
 2  shortestBacktrackingDistance ← Infinity
 3  for basicVariable ∈ B
 4  do lowerViolated ← α(basicVariable) < lower(basicVariable)
 5      upperViolated ← α(basicVariable) > upper(basicVariable)
 6      if ¬lowerViolated ∧ ¬upperViolated
 7         then continue
 8      for original,bound ∈ originalDependencies[basicVariable]
 9      do backtrackingDistance ← list.getBacktrackingLength(original,basicVariable,bound)
10          if backtrackingDistance < shortestBacktrackingDistance
11             then bestCandidate = basicVariable
12                  shortestBacktrackingDistance = backtrackingDistance
13  return bestCandidate
14
```

Algorithm 6: Entering rule

3.6 Exit rule

We begin by computing the set of original variables already used in the list and call it X_L. We then use our pre-computed list of original dependencies to gather all bounds for the entering variable and violated bound and call it X_E.

Now there are three possibilities.

$X_E \nsubseteq X_L$ **suitable** In this case there exists an original variable $x \in X_E$ that is still part of the basis $x \in \mathcal{N}$ and not yet used in the list. We can simply return x.

$X_E \subseteq X_L$ **backtracking** For this case, all original dependencies in X_E are already in our list. However, there exists at least one original variable $x \in X_L$ with the same bound as the violated leaving bound in X_E. Starting backwards at the end of the list, we return the currently selected variable for such an original variable x.

$X_E \subseteq X_L$ **bound switching** In this case, all original dependencies have the wrong bound in the list. We encountered a dead end. In this case we again start at the end of the list and backtrack until we find an original variable that is in X_E. We then switch the bound to the correct one.

This step is problematic in practice and a performance hit, so future work will focus on eliminating this issue.

3.7 Soundness

To show that our adapted algorithm is still sound and complete, we have to proof termination as well as correctness.

3.7.1 Proof of termination

For our heuristic to be non-terminating there would need to exist a cycle, where we keep pivoting with the same variables over and over again without making any progress.

We show that for every list there exist only finitely many valid continuations. Thus, we show that the algorithm will not enter a cycle and hence terminates. Since we have a finite tableau, there are finitely many constraints. Each of these constraints will at most be a bound for n original variables. We call a container *full* when it already contains each of the constraints in its tested set, that is an original bound for this original variable and bound. When we backtrack to position i, the tested set for the container i will increase by one. We also delete the trail starting at container $i+1$. It now follows that we can backtrack to i at most n times until container i is full.

An induction will now show, that only finitely many continuations of the list exist. For a given list there exist only finitely many continuations when backtracking up to length i. Consider $i = 0$. When we do not backtrack at all, only the values in the last container can change. At most n variables can be selected until the tested set is full. Therefore, at most n continuations exist.

We now consider that the hypotheses holds for a given i. When we now backtrack up to length n, the container $n-(i+1)$ will again only accept up to n values in its tested set. For each of these values, we know that the trail of backtracking length i is finite. Therefore, we know, that for backtracking length $i+1$, only finitely many continuations exist.

Since each step of the algorithm corresponds to exactly one of these lists, we know that the algorithm terminates.

3.7.2 Proof of correctness

We use the terminating property of the algorithm to show that it is also correct.

Due to our construction in each step, our selected variables $\eta := (S_0, \ldots, S_k)$ are a subset of the nonbasic variables $\mathcal{N}$.

Additionally, since we prevent already tested variables to be selected again, it holds that at each step we encounter a new set η'. The only times where we return SAT is if η is a satisfying basis for the input.

On the other hand, we only return UNSAT when η contains a conflict.

Lets consider x_i is violating. Assume that at some point a variable x_j is suitable for pivot and will at some point lead to a solution (otherwise the problem is UNSAT), but our list prevents it from being selected. This happens when we have either already tested this variable [1] or when another variable x_k has a shorter backtracking length [2].

The first case is a contradiction, since we would have already found this solution when we selected x_j previously.

The second case might be an issue at first, however, if pivoting with x_k does not lead to a solution, then at some point we will have to select x_j when all other backtracking lengths are larger.

Therefore, at each step, at some point the "correct" variable that will be part of a SAT basis will finally have a chance to be selected. Additionally, we will show that the algorithm will never explore all possibilities and exit without returning SAT or UNSAT.

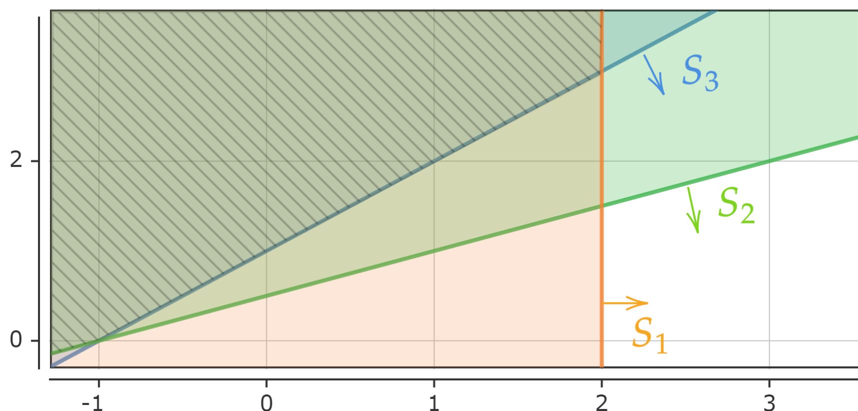
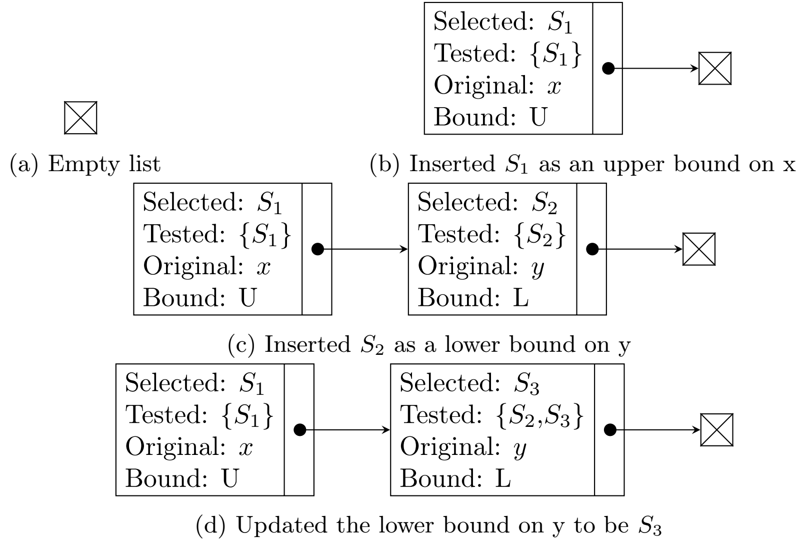


Figure 3.3: Visualization of Example 3.7.1

Since each step corresponds to a unique basis, when the algorithm would have checked all of the basis, we tried each possibility. However, since the tableau that we receive can be partitioned into SAT and UNSAT, we are sure to reach at either a SAT or UNSAT basis before terminating.

Example 3.7.1 (Example run of our heuristic)**.** *Assume three constraints* $S_1 = x$, $S_2 = -0.5x + y$, $S_3 = -x + y$ *with their bounds* $S_1 \leq 2$, $0.5 \leq S_2$, $1 \leq S_3$ *visualized in Figure 3.3.*
For our first step we need to determine for each constraint original variables it bounds.
We obtain: $X_L = \{\}$, $X_U = \{S_1,S_2,S_3\}$, $Y_L = \{S_2,S_3\}$, $Y_U = \{\}$
Starting with the assignment $x = 0, y = 0$*, we notice that* S_1 *violates its bound. Only* x *is suitable for pivoting. We check for our leaving rule and receive* x*, since the list is currently empty see Figure 3.4a.*
We now pivot with x *and insert (*S_1*,*x*,upper) into our list Figure 3.4b.*
Now at $x = 2, y = 0$ S_2 *violates its lower bound and* y *is suitable for pivoting.*
Our bounds on S_2 *are upper on* x *and lower on* y*.*
We again check our leaving rule and receive y *since it is not currently represented in the list.*
We insert (S_2*,*y*,lower) into our list and pivoting with* y *Figure 3.4c.*
At $x = 2, y = 1.5$ S_3 *violates its lower bound.* S_1 *and* S_2 *are suitable for pivoting.*
We again collect its original dependencies and obtain an upper bound on x *and a lower bound on* y*.*
In our leaving rule we now consider S_1 *and* S_2*. We notice that* y *has a shorter back-tracking distance in our list, so we return* S_2 *Figure 3.4d.*
We update the last container and pivoting with S_2*.*
Finally no variable is violated, so we can return SAT.



(a) Empty list

(b) Inserted S_1 as an upper bound on x

(c) Inserted S_2 as a lower bound on y

(d) Updated the lower bound on y to be S_3

3.8 Observation of required pivoting steps

From the induction we can also derive at an upper bound to the amount of pivoting steps.

When we consider an actual case, each original variable x_i will have x_{i_L} many lower bounds and x_{i_U} upper bounds.

One can arrive at the conclusion that we can pivot at most $\prod_{i=1}^{n} max(x_{i_L}, x_{i_U})$.

In the worst case, this will be n^m. Where m is the number of constraints in the system. However, in practice we can reduce this number by a large factor. The max function in the product already gives a hint on how we can reduce this number. For each of the original variables we should try to minimize the amount of constraints with that bound. Since we can choose which bound (lower or upper) to select at this point, we should aim to take constraints with the lower count.

In this work we simply chose the first bound that we encounter and stick to this selection. This is, however, inefficient in practice since when we have many lower bounds, we are more likely to encounter a lower bound, and therefore, might have to check all of them, instead of just considering the fewer upper bounds. Selecting the right ordering to pivot can lead to drastically reduced pivoting steps.

3.9 Results

To analyze the runtime of our heuristic, we ran the QF_LRA benchmarks [SMT] with a timeout of 5 minutes and a memory limit of 4Gigabyte. As a reference, we used our implementation using Bland's Rule [Bla]. The results are presented in Table 3.2 we additionally computed the virtual best (VB) when taking the best runs of each algorithm. Our algorithm never exceeds the memory limit, compared to the reference, that reaches a memout 60 times. This is, however, not an advantage as one can see in the VB, since only 2 of the 60 memouts could actually be solved by the FMplex heuristic. Which means, that these memouts are timeouts in our implementation. In Figure 3.5b one can see that the run-times of both heuristics are very similar. In

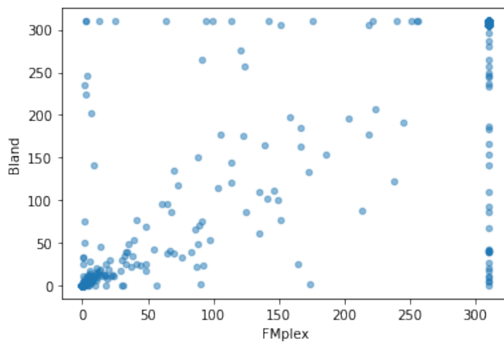
	FMplex	Bland	VB
SAT	514	524	534
UNSAT	377	385	392
TIMEOUT	757	679	664
MEMOUT	0	60	58
SOLVED	891	909	926

Table 3.2: Results from the QF_LRA benchmark [SMT]

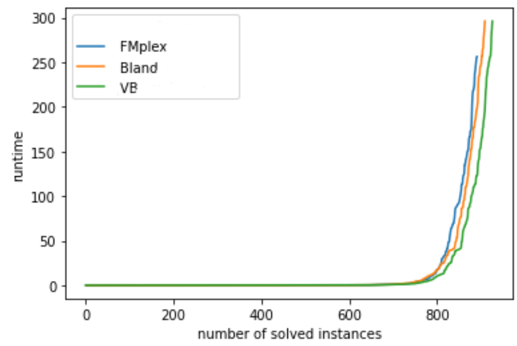
Figure 3.5a the run-times of each problem is plotted. It is clear that both heuristics have their advantages and disadvantages. Our heuristic solves 18 problems less in total. Interesting to note is the cluster of problems that our heuristic timed out on, that could be solved by Bland's Rule almost immediately. This might be because of a bug in one of the modules, since our heuristic gets interrupted by a very long run of the underlying SAT module. While this could be explained by different infeasible subsets and therefore different runs in the SAT module, it is unlikely that this is the whole explanation. In total 16 problems timed out, where Bland's Rule could solve the problem in less than a minute. On the other hand only 4 problems timed out in Bland's Rule, that were solved by our heuristic in less than a minute. Making it unlikely that the different paths of the algorithm are the complete explanation for this.

Finally, we can compare our runtime to that of the previous FMplex iteration, using a static variable ordering as seen in Figure 3.5c. The runtime of our implementation is an improvement to that of the Min. Sign heuristic that was previously proposed.

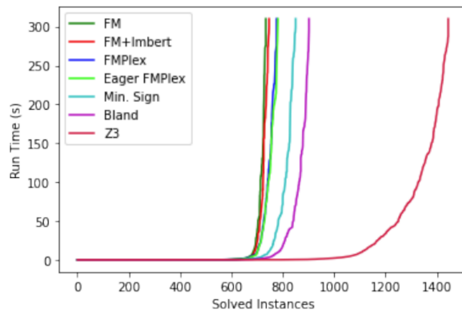
In conclusion, we note that extra bookkeeping effort in our heuristic seems to be worth the effort, since the run-times are very similar. It seems feasible that improvements to the bookkeeping effort could lead to a better performance than Bland's Rule.



(a) Scatter plot of the run-times of our heuristic and our reference implementation using Bland's Rule



(b) Runtime analysis between our heuristic, Bland's Rule and the virtual best (VB)



(c) Reference runtimes from the previous FMplex paper [Kob]

Chapter 4

Conclusion

4.1 Summary

We introduced the FMplex simplex heuristic and showed that it is a sound heuristic. By restricting pivoting to certain directions, we can effectively reduce the computational effort required to solve the system. Additionally, we discovered a problem that is not preset in the FM variation of the heuristic and introduced the bound switching technique to solve it. While this decreases performance by a large factor, the other proposed solution would have exceeded the limitations of this paper. Future work will, however, inspect the possibility of precomputed variable orderings. The sourcecode generated for this work was provided to SMT-RAT [CKJ+].

4.2 Discussion

While the results of the algorithm are comparable to a regular Simplex algorithm using Bland's Rule, we still believe that this is an idea worth investigating. The current implementation should be considered an investigative prototype and multiple improvements can be made. Including but not limited to

1. Introducing a heuristic for which original variable to choose, when multiple are an original bound for the constraint and not already present in the list. Currently we choose the first one, that we encounter.

2. Using a hashmap to reduce the lookup process when updating a container or computing the backtracking length

3. Caching the results of our entry rule and update them with our new list when needed.

4. Restructuring the tableau datastructure. The current implementation uses a quadruple chained list. This results in iteration over the whole list when we already know, which candidate we want to pivot with. One could store the columns of the tableau in our list to reduce the lookup to a constant time.

There is one particular issue that arises from the current simplex variation. If there are only a few upper bounds, it would be ideal to consider them first. However,

initially, when we assign each variable to be 0, these bounds are likely to be satisfied. Therefore, preventing them from being inserted into our list in the first place. This is a large decrease in performance on problems that might not have many upper bounds. One solution to this might be to begin by pivoting with some constraints that are not violated, but where we only have a few competing constraints for that specific original variable and bound. This would force the heuristic to keep searching in this direction.

Additionally, the problems that we encountered in the algorithm lead to an exciting new idea where we consider the precomputed bounds to form the variable ordering instead of computing it dynamically.

4.3 Future work

While the sourcecode implemented in SMT-RAT [CKJ^{+}] does support incrementality, the proofs and ideas in this paper are based on a non-incremental version. Future work should investigate if our current implementation can be improved upon, when considering incrementality in more detail.

The dead end problem discovered in this paper is the main area where further research is needed. This problem was not present in the FMplex version that this work is based on. We proposed the naive bound switching technique - while easy to compute, it has proven to be ineffective in reducing the number of pivot steps required.

The backtracking approach introduced might solve this problem - by backtracking the conflicting selection up the list to free up a container for our basic variable, we might be able to eliminate the dead ends. This might resolve the problem; however we currently do not obtain a proof of correctness for this resolution.

Finally, future work will investigate the pre-computation of the list. This has several advantages that might speed up the heuristic drastically. First of all, it would eliminate the dead end problem that we encountered, since dead ends would be eliminated in pre-computation. Secondly, it would, in a best case scenario, reduce our constraints that we consider to a minimum, or at least close to a minimum. Additionally, it would allow pivoting with variables that are currently not violated but that we are sure are connected to a solution or where we can prove that no solution can exist. This would greatly reduce the complexity of the problem when there are only a few bounds for one direction. Finally, it might be possible to resolve conflicts in the Simplex tableau with an edge sliding technique. Instead of performing costly pivots, we might be able to change assignments to slide along one constraint when we are sure that a solution will have to exist alongside this edge. It might be possible to adapt the Sum of Infeasiblities introduced in [KBD] to find these solutions.

Bibliography

[Bla] R. Bland. New finite pivoting rules for the simplex method.

[CKJ+] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving.

[Coo] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158. Association for Computing Machinery. event-place: New York, NY, USA.

[Dan] George B. Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. Association for Computing Machinery.

[Fou] Joseph Fourier. Analyse des travaux de l'académie royale des sciences pendant l'année 1824, partie mathématique, 1827. engl. transl. (partially) in: D.A. Kohler, translation of a report by Fourier on his work on linear inequalities, opsearch10 (1973) 38–42.

[KBD] Tim King, Clark Barrett, and Bruno Dutertre. Simplex with sum of infeasibilities for SMT. In *2013 Formal Methods in Computer-Aided Design*, pages 189–196.

[Kob] Thesis Kobialka. Connecting Simplex and Fourier-Motzkin into a novel quantifier elimination method for linear algebra.

[SMT] LIB SMT. SMT-LIB the satisfiability modulo theories library https://smtlib.cs.uiowa.edu/benchmarks.shtml.