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

BACHELOR OF SCIENCE THESIS

SIMPLEX HEURISTICS IN SMT SOLVING
SIMPLEX-HEURISTIKEN IN SMT SOLVING

Fabian Alieff

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

Additional Advisor:
Jasper Nalbach, M.Sc.

Aachen, 18.11.2020

Abstract

The standard in Satisfiability Modulo Theory solving is the Simplex for DPLL(T) algorithm, proposed by Dutertre and de Moura. The SMT-solver SMT-RAT being developed at the Theory of Hybrid Systems Research Group also uses this algorithm as a basis for their LRA-module. In this work different heuristics for choosing variables during the pivoting operation of the simplex method are researched and implemented in this module. Then all chosen heuristics are tested on the QFLRA benchmarks from the SMT-LIB library and their performance is compared to the performance of the original module. While one heuristic could not be tested due to errors in the implementation which lead to wrong results on the benchmarks, the other heuristics underperformed the LRA-module in many of the benchmarks and overall produced more timeouts.

Eidesstattliche Versicherung

Alieff, Fabian	355167
Name, Vorname	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Simplex-heuristiken in SMT Solving

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift

Contents

List of Figures

Chapter 1

Introduction

Satisfiability Modulo Theories (SMT) solving is the process of deciding on the satisfiability of a given logical formula with respect to some first order theories. The theory important for this work is quantifier free linear real arithmetic, the first order theory over $\mathbb{R}$ with signature $\{0,1,+,<\}$. Most modern Satisfiability Modulo Theories (SMT) solvers use the Simplex algorithm introduced by [DDM06] for linear arithmetic reasoning in a DPLL(T) framework ([BCD$^+$11][CGSS13][DMB08][CHN12][BPST10]). Solvers using the simplex algorithm rely on variable choosing heuristics for the pivoting operation, to either prove unsatisfiability or find a satisfying assignment for a given subset of formulas. The most basic heuristic being Bland's rule, which chooses variables according to a fixed ordering. While guaranteeing termination, the runtime of Bland's rule is in many cases suboptimal, as it converges slowly [KBD13]. For this reason different heuristics taking different local or global criteria into account are created.
At the *Theory of Hybrid Systems Research Group*, part of the Chair of Computer Science I2 of RWTH Aachen University, SMT-RAT, an SMT-solver which implements different solvers in modules is being developed.

SMT-RAT includes an LRA-MODULE, also based on [DDM06] for linear arithmetic reasoning. All work in this thesis was done in this module.
The goal of this work is the implementation, testing and comparison of different variable ordering heuristics for the simplex algorithm in the context of SMT-solving.

1.1 Outline

Section 2 gives background information about the simplex algorithm in general and the usage of the simplex algorithm in the context of LRA-solving which are needed for a better understanding of this work. Section 3 gives a brief explanation of SMT-RAT, in which the heuristics were implemented. Section 4 gives a detailed explanation of the theory behind the implemented heuristics and why they were chosen. Section 5 describes the implementation of those heuristics inside the LRA-MODULE of SMT-RAT. An evaluation of the different strategies, which were tested on the cluster provided by the *Theory of Hybrid Systems Research Group* is shown in section 6 and section 7 concludes the thesis, giving a quick summary, plus an outlook on how this work could be extended upon in the future.

Chapter 2

Background

2.1 SMT-Solving

The basic focus of SMT-Solving lies in finding out if a formula is satisfiable (with respect to some first order theory T). The DPLL(T)-SOLVER is the main architecture used by modern SMT-SOLVERS. It implements a SAT-SOLVER, which is used to search for a satisfying assignment of propositional formulas, in cooperation with one or more theory-solvers(T-solvers). For this purpose, a given quantifier-free formula Θ is transformed into a propositional formula Θ', by replacing it's atoms θ_i with propositions p_i. The SAT-solver then searches for a satisfiable assignment of Θ'. If a satisfying assignment is found, the implemented T-solvers check the corresponding set of atoms for consistency. In case of an inconsistency, the solver needs to return an explanation for the found conflict by providing any infeasible subset of the atoms given by the SAT-solver. In an incremental SMT-solver, instead of solving the complete formula Θ' at once, the SAT-SOLVER searches for partial solutions for Θ' and check this partial solution with the theory solver, incrementally extending the partial solution, if the theory solver returns SAT or backtracking if it returns $UNSAT$. This work focuses on an LRA-SOLVER, which uses the simplex method for the satisfiability checking of $QFLRA$ formulas.

2.2 Simplex method

The Simplex method was designed mainly to solve linear programming problems by finding an optimal solution to an objective function with respect to a set of linear real arithmetic constraints.
A feasible solution is any vector $x \in \mathbb{R}$, for which the constraints are satisfied. A feasible solution is optimal, if the vector x also maximizes the objective function. The Simplex method works in two phases, in the first the algorithm searches for any feasible solution. If a feasible solution is found, it is used to search for an optimal feasible solution in the second phase.
In the context of satisfiability checking, it is enough to consider only the first phase. For that purpose a given set of constraints of the form $\sum_{j=1}^{n} a_{ij}x_j \bowtie c_j$ is transformed into a set of constraints of the form $\sum_{j=1}^{n} a_{ij}x_j - s_j = 0$ with $s_j \bowtie b_j$, where $\bowtie \in \{=, \leq, \geq\}$. s_1,...,s_m are called slack variables. The resulting constraints are then used

to derive lower bounds l_i and upper bounds u_i for each $x_i \in V$, as well as a Simplex tableau with rows of the form $b_i = \sum_{n_j \in N} a_{ij} n_j,\ b_i \in B$, where B is the set of basic variables with a size of m and N the set of non-basic variables with a size of n, with $B, N \subset V$. Initially N consists of all original variables and B consists of all slack variables.

Each state of the simplex method includes an assignment (initially set to: $\alpha(x_i) = 0\ \forall i \in \{1,.....,n+m\}$) for each variable and a tableau, for which the invariants $Ax = 0$ and $\forall n_j \in N \quad \alpha(n_j) > l_j \wedge \alpha(n_j) < u_j$ always hold [DOW+55]. The method then works as follows:

If a basic variable violates it's bounds, if possible, pivoting is performed, until either no more basic variables violate their bounds or no pivoting step is possible in which case a conflict is found which means, that the problem is unsatisfiable. If no basic variable violates it's bounds, the problem is satisfiable. Pivoting consists of the following steps :

1. Choose a basic variable, that violates it's bounds, either $\alpha(b_i) < l_i$ or $\alpha(b_i) > u_i$
2. If possible, choose a non-basic variable n_j with
   - If $\alpha(b_i) < l_i$:
     - either $a_{ij} > 0$ and $\alpha(n_j) < u_j$
     - or $a_{ij} < 0$ and $\alpha(n_j) > l_j$
   - If $\alpha(b_i) > u_i$:
     - either $a_{ij} < 0$ and $\alpha(n_j) < u_j$
     - or $a_{ij} > 0$ and $\alpha(n_j) > l_j$
3. If a suitable variable n_j for pivoting is found, perform the pivoting step, else return unsatisfiable:
   (a) Solve equation $b_i = a_{ij} n_j + \sum_{k \neq j} a_{ik} n_k$ for n_j to $n_j = \frac{b_i}{a_{ij}} - \sum_{k \neq j} \frac{a_{ik}}{a_{ij}} x_k$
   (b) Swap n_j and b_i, such that b_i becomes non-basic and n_j becomes basic
   (c) Update rows of the tableau accordingly, the ith row by changing $[a_{i1}...a_{ij}...a_{in}]$ to $[\frac{-a_{i1}}{a_{ij}}...\frac{1}{a_{ij}}...\frac{-a_{in}}{a_{ij}}]$ and every other row by replacing n_j with it's equivalent computed in 3a .
   (d) Update assignment of n_j: $\alpha(n_j) = \alpha(n_j) + \frac{l_i - \alpha(b_i)}{a_{ij}}$
   (e) Update all assignments of variables depending on n_j
4. Go back to step 1.

To ensure termination a selection heuristic for the pivoting row/column needs to be used, the most basic heuristic for that is Bland's rule which establishes a variable order, from which always the first variable feasible for a pivoting step is chosen. A heuristic for choosing the basic variable is called the *entering rule* and a heuristic for choosing the non-basic variable for pivoting is called the *leaving rule*. A more detailed explanation of different heuristics will be discussed later.

Figure 2.1: Basic check procedure [DDM06]

```
1  procedure Check()
2   loop
3    select the smallest basic variable $b_i$ such that
4    $\alpha(b_i) < l_i$ or $\alpha(b_i) < u_i$
5     if there is no such $b_i$ then return satisfiable
6     if $\alpha(b_i) < l_i$ then
7          select the smallest non-basic variable $n_j$ such that
8           ($a_{ij} > 0$ and $\alpha(n_j) < u_j$)or($a_{ij} < 0$ and $\alpha(n_j) > l_j$)
9      if there is no such $n_j$ then return unsatisfiable
10      pivotAndUpdate($b_i$,$n_j$,$l_i$)
11    if $\alpha(b_i) > u_i$ then
12         select the smallest non-basic variable $n_j$ such that
13          ($a_{ij} < 0$ and $\alpha(n_j) < u_j$) or ($a_{ij} > 0$ and $\alpha(n_j) > l_j$)
14     if there is no such $n_j$ then return unsatisfiable
15      pivotAndUpdate($b_i$,$n_j$,$u_i$)
16  end loop
```

2.3 Simplex in SMT-Solving

QFLRA(quantifier-free linear real arithmetic) formulas are first order formulas containing only atoms which are either propositional variables or of the form :

$$a_1x_1 + ... + a_nx_n \bowtie c,$$
$$where \bowtie \in \{= , \neq , < , > , \leq , \geq\}$$

The atoms given to a LRA-solver create a set of such constraints. It is assumed, that, for the purpose of performing the simplex method in an LRA-solver, the tableau $b_i = \sum_{n_j \in N} a_{ij} n_j \quad b_i \in B$, is derived from the constraint matrix A, B is the set of basic variables and N the set of non-basic variables. The solver then checks a given set of constraints for consistency using the simplex method. Figure 2.1 shows a basic version of such a check procedure, here `pivotAndUpdate` performs a pivoting operation on the tableau, where b_i and n_j are swapped and the assignment of n_j gets updated by v.

If `check` finds that the current set of assignments is consistent, a checkpoint for the current state of the tableau and assignments is created to which can be backtracked in case of an inconsistent set of assignments later. In case the `check` procedure returns unsatisfiable, an explanation is generated by computing $N^+ = \{x_j \in N \mid a_{ij} > 0\}$ and $N^- = \{x_j \in N \mid a_{ij} < 0\}$. Then an explanation for a conflict occurring at line 8 of Figure 2.1 is $\Gamma = \{x_j \leq u_j \mid x_j \in N^+\} \cup \{x_j \geq l_j \mid x_j \in N^-\} \cup \{x_i \geq l_i\}$. If it occurs at line 13, an explanation is $\Gamma = \{x_j \leq u_j \mid x_j \in N^+\} \cup \{x_j \geq l_j \mid x_j \in N^-\} \cup \{x_i \neq u_i\}$.

To assert a new atom γ, an `assert` procedure is implemented, which uses subprocedures, as shown in Figure 2.2. It updates the bounds l_i and u_i and checks for consistency. In case of inconsistency, *UNSAT* is returned. The basic algorithm works as follows: as long, as the assert procedure returns SAT, new atoms are asserted. If it returns *UNSAT*, the check procedure is invoked, until either all variables satisfy their bounds, in which case more atoms can be asserted, or a conflict is detected.

2.4 SMT-RAT

The implementation and testing of Simplex heuristics was done in SMT-RAT, an *Open Source C++ Toolbox for Strategic and Parallel SMT Solving* [CKJ+15]. The focus of SMT-RAT lies in it's modularity and ability to compose different solving techniques. For that purpose, each solving technique is implemented using the `Module` class.

The composition of different modules can be done via a strategy. Strategies represent a tree structure of modules and are derived from the `Manager` class. In a strategy, the root module will be called on any given input file. Modules can call their child nodes on a passed formula using `runBackends`. For example in the context of this work, a SAT-Solver module with a LRA-Solver module as it's child node is used to solve formulas in quantifier-free real linear arithmetic. It is possible to allow for parallel execution of backends, in which case the first backend to terminate is used, as well as restricted execution of backends using conditions under which they are to be used. A more detailed explanation of SMT-RAT and it's software design, including advanced solving techniques implemented as individual components in the frontend, is provided in the manual on *http://smtrat.github.io/*.

Figure 2.2: Basic assert procedure [DDM06]

```
procedure AssertUpper($b_i \leq c_i$)
  if $c_i \geq u_i$ then return satisfiable
  if $c_i < l_i$ then return unsatisfiable
  $u_i := c_i$
  if $b_i$ is a non−basic variable and $\alpha(b_i) > c_i$ then update($b_i$,$c_i$)
```

```
procedure AssertLower($b_i \geq c_i$)
  if $c_i \leq l_i$ then return satisfiable
  if $c_i > u_i$ then return unsatisfiable
  $u_i := c_i$
  if $b_i$ is a non−basic variable and $\alpha(b_i) < c_i$ then update($b_i$,$c_i$)
```

Chapter 3

Heuristics

This Section will explain the basics of the implemented strategies for deciding on a pair of variables for pivoting, and why they were chosen. A detailed explanation of how they are implemented will follow in Section 4. All strategies switch to Bland's rule for variable selection after a fixed number of pivoting steps.

3.1 Bland's rule

Bland's rule uses indices $\{1,\ldots,i\}$ with $x_1,\ldots,x_i \in V$ and chooses the basic variable b_k violating one of it's bounds such that $k < l$ for all other basic variables b_l violating one of their bounds, with $l \neq k$, $l,k \in \{1,\ldots,i\}$ and analogous a suitable non-basic variable n_j, such that $j < l$ for all other non-basic variables n_l suitable for pivoting with $l \neq j$, $l,j \in \{1,\ldots,i\}$. This is done to make sure that the search for satisfiability terminates [AC78]. In the following, indices $\{1,\ldots,i\}$ with $x_1,\ldots,x_i \in V$ are assumed.

3.2 Activity-Based

The activity-based strategy tracks for each variable x_k their pivoting activity $activitybasic_k$ for the amount of times it has left the basis and $activitynonbasic_k$ for the amount of times it has entered the basis. For a pivoting operation, a basic variable b_k violating one of it's bounds is chosen with $activitybasic_k \leq activitybasic_l$ for all b_l violating one of their bounds with $l \neq j$, $l,k \in \{1,\ldots,i\}$ and analogous a suitable non-basic variable n_j, such that $activitynonbasic_j \leq activitynonbasic_l$ for all n_l suitable for pivoting with $l \neq j$, $l,j \in \{1,\ldots,i\}$. Ties are broken by the variable with smaller index.
The idea behind this strategy is, to choose variables which would be overlooked in other strategies e.g. Bland's, as to not miss a variable which could potentially reduce the number of pivoting steps, if selected for a pivoting operation.

3.3 Difference-Based

The difference-based strategy considers for each basic variable b_k, violating one of it's bounds, a $\theta_k = \mid l(x_k) - alpha(x_k) \mid$, if the lower bound is violated or $\theta_k = \mid$

$u(x_k) - alpha(x_k) \mid$, if the upper bound is violated. The variable b_k is chosen, so that for all basic variables b_l violating their bounds, $\theta_k \leq \theta_l$ with $l \neq k$, $l,k \in \{1,\ldots,i\}$. From all suitable non-basic variables, a variable n_j is chosen based on the difference $\theta_j = \mid u(n_j) - \alpha(n_j) \mid$, if the resulting pivoting operation would increase $\alpha(n_j)$ or $\theta_j = \mid l(n_j) - \alpha(n_j) \mid$, if the resulting pivoting operation would decrease $\alpha(n_j)$, such that $\theta_j \geq \theta_l$ with $l \neq j$, $l,j \in \{1,\ldots,i\}$. Ties are broken by the variable with the smaller index.

When the assignment of a non-basic variable gets changed to violate on of it's bounds during a pivoting step, the assignment has to be corrected later, often through an extra pivoting step. The reasoning behind selecting the basic variable with the smallest difference from assignment to violated bound is, so that the assignment of the corresponding chosen non-basic variable needs to change the minimum amount for the pivoting operation which would optimally result in the changed assignment of the non-basic variable not causing it to violate it's bounds so that there does not have to be another pivoting operation later to correct the assignment, or if it causes a bound to be violated, the correction only needs to change the assignment a minimal amount. Choosing a non-basic variable with the biggest difference to it's corresponding bound follows the same objective.

3.4 Priority-Based

The priority-based strategy assigns a non-basic and a basic priority to each variable. For a pivoting step, the strategy prioritizes basic variables b_k violating their bounds, which have a non-basic priority. For suitable non-basic variables n_j, the strategy prioritizes ones with basic priority. The priorities for each variable are updated, each time the check procedure returns *UNSAT* to the SAT-solver. Basic variables gain basic priority and non-basic variables gain non-basic priority. When deciding between basic variables with non-basic priority and non-basic variables with basic priority, different sub-heuristics were tested:

- Bland's rule: the variable with the lowest index is chosen.
- Activity-based sub-heuristic: the variable is chosen as described in Section 3.2.
- Difference-based sub-heuristic: the variable is chosen as described in Section 3.3.
- Mixed sub-heuristic: the variable is mainly chosen as described in ection 3.2, in case of variables having the same activity and priority, the variable is chosen as described in Section 3.3.

The strategy tries to return the tableau to a state, where in a previous test, a conflict was detected, done under the assumption, that this will increase the chance of quickly encountering a conflict again. This would reduce the amount of pivoting steps needed, before returning a result to the SAT-SOLVER and, as most times many conflicts are detected before a feasible set of constraints is found, subsequently reduce the overall runtime.

3.5 Sum of Infeasibilities

The *Sum of Infeasibilities*[KBD13] heuristic works differently from the other heuristics presented, in that instead of trying to locally improve the pivoting operation, a global criterion in the form of a function which needs to be minimized, is used. This function is the *Sum of Infeasibilities*, which is defined as $Vio(V) = \sum_{x \in V} Vio(x)$ with:

$$Vio(x) = \begin{cases} l(x) - \alpha(x) & \alpha(x) < l(x) \\ \alpha(x) - u(x) & \alpha(x) > u(x) \\ 0 & otherwise \end{cases}$$

Minimizing this function will prove satisfiability, if the minimum is 0, and unsatisfiability otherwise. This function can be written as $Vio_F(V) = \sum_{x \in V} Vio_F(x)$, Vio_F being the result of replacing $\alpha(x)$ by x in $Vio(x)$. As it is only piecewise linear, it cannot be represented in the tableau. Therefore a linear approximation of Vio_F is used, which, to ensure correctness, still depends on the current assignment:

$$f = \sum_{n_j \in N} (\sum_{x_i \in V} d_i \cdot entry_{i,j}) \cdot n_j)$$

with:

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

The coefficients for each non-basic n_j variable need to get updated each time the assignment of a variable x_i gets updated and d_i changes to some d'_i, by adding $(d'_i - d_i) \cdot entry_{i,j}$ to it. $Vio(V)$ or any of it's approximations do not have to be explicitly computed for the algorithm, it is enough to only consider the coefficients of each variable.

3.5.1 Algorithm

$Flex(f)$ denotes the set of non-basic variables n_j, which enable the function to decrease, by changing their assignment. $\Delta Vio(\delta,j)$ denotes the amount $Vio(V)$ would change, if the assignment of n_j is changed by δ. How $\Delta Vio(\delta,j)$ is computed will be described later. The strategy then works by first checking, if there are any conflicts. If there are none, a non-basic variable $n_j \in Flex(f)$ together with a corresponding basic variable b_k , which does not necessarily have to violate it's bounds, are chosen, if $\Delta Vio(\delta,j) < \Delta Vio(\delta,l) \quad \forall\ \delta, \forall\ l \in \{1,\dots,i\}$. Here δ represents a change of the assignment of n_j, which would result in the corresponding basic variable b_k to be set either to it's upper or it's lower bound. If $Flex(f) = \emptyset$ and there are no conflicts, the strategy returns satisfiable.

3.5.2 Computing $\Delta Vio(\delta,j)$

The function Vio_F is linear between breakpoints δ,which can be used to compute $\Delta Vio(\delta_i,j)$ [KBD13]:
It is assumed, that all breakpoints δ_i are sorted into a decreasing list of the negative values and an increasing list of the positive values, with a $\delta_0 = 0$ included in both.

With $\Delta Vio(0,j) = 0$, $\beta_0 = 0$ and K_i being the set of basic variables b_k for which d_k changes to some d'_k at a given breakpoint δ_i, $\Delta Vio(\delta_i,j)$ can be computed as follows:

$$\begin{aligned} & \Delta Vio(\delta_i,j) = \Delta Vio(\delta_{i-1},j) + \beta_{i-1} \cdot (\delta_i - \delta_{i-1}) \\ with \quad & \beta_i = \beta_{i-1} + \sum_{k \in K_i} (d'_k - d) \cdot entry_{k,j} \end{aligned}$$

This is done separately for the set of negative breakpoints and the set of positive breakpoints. For each $x_j \in Flex(f)$, there exists a pair (δ,k) such that $\Delta Vio(\delta,j) \leq 0$[KBD13].

Chapter 4

Implementation

For the purpose of testing different simplex heuristics, the LRA-MODULE of the SMT-RAT project was edited and renamed to SIMPLEXHEURISTICSLRAMODULE. In the following this will be referenced as the LRA-MODULE. This section will first give a brief explanation of the basic functions of the LRA-module, then a more detailed explanation of the Implementation of the different simplex-heuristics and lastly will mention difficulties encountered while implementing and testing the heuristics, as well as problems still existing.

4.1 LRA-Module

To process the constraints given to it by the SAT-MODULE, the LRA-MODULE implements the following basic interfaces:

```
bool LRAModule::informCore(const Formula& _constraint)
```

Informs the module about a new constraint before it will be added, by adding it to the set `mLinearConstraints`, if it is linear and initializing corresponding bounds. If the constraint is consistent, `true` is returned, else `false`

```
bool LRAModule::deinformCore(const Formula& _constraint)
```

Removes the given constraint from `mLinearConstraints` and corresponding bounds from the tableau.

```
bool LRAModule::addCore(const ModuleInput::const_iterator)}
```

Adds the the formula at the given position to the next satisfiability check, by activating all corresponding bounds.

```
void LRAModule::removeCore(const ModuleInput::const_iterator)}
```

Removes the the formula at the given position from the next satisfiability check, by deactivating all corresponding bounds.

```
Answer LRAModule::checkCore(bool)
```

Performs a satisfiability check on the current set of constraints, by calling the function `nextPivotingElement` or if the *Sum of Infeasibilities* heuristic is used function `nextPivotingElementSof` implemented in the tableau class, both returning a pair `std::pair<EntryID,bool>`. The `EntryID` returned represents the entry of the tableau, which is chosen for the pivoting step, if the boolean value returned is `true`. If it is `LAST_ENTRY_ID`, the current assignment is satisfying and `checkCore` returns `SAT`. If the boolean value returned is `false`, the current assignment is unsatisfiable and `checkCore` creates an infeasible subset with the `EntryID` returned by `nextPivotingElement` representing the begin of a conflicting row, and returns *UNSAT*.

4.2 Code Structure

Switching the heuristic used can be done via the `TableauSettings` class. As the *Sum of Infeasibilities* heuristic has a significantly different approach to selecting variables for the pivoting step, first considering the leaving rule and then the entering rule, it was implemented in it's own function for pivoting-element selection. The other heuristics could all be implemented in the same function, by switching the criterion for variable comparison. In the following, these functions will be explained separately. Priority, activity and the coefficient for the *Sum of Infeasibilities* function have all been implemented in the variable class, so they can be easily accessed.

4.2.1 Basic Implementation

Figure 4.1: Basic procedure for finding the next element for pivoting

```
nextPivotingElement():
  for ($b_j$ |($b_j$ is basic and violates one of its bounds)):
    if($b_j$ worse than bestBasicVar according to chosen heuristic):
      continue loop
    else:
      entry = getTableauEntryForPivot()
      if(entry == LAST_ENTRY_ID):
        return(beginOfFirstConflictRow, false)
      else:
        update bestBasicVar and bestEntry
  if (bestEntry == LAST_ENTRY_ID):
    return(LAST_ENTRY_ID, true)
  else:
    return(bestEntry, true)
```

Figure 4.1 shows the basic implementation for all heuristics except for *Sum of Infeasibilities*. The function iterates through all basic variables, which violate one of their bounds. It then checks, if the variable is worse for pivoting than a previously selected 'best' variable, according to the chosen heuristic. If so, the function skips to the next loop. This is done to not waste time on computing the best non-basic variable for a basic variable, which will not be chosen anyway. If the

variable is better, a suitable tableau-entry for pivoting is computed by the function `getTableauEntryForPivot`.
`getTableauEntryForPivot` works, as shown in Figure 4.2. It iterates through entries of the row of the given basic variable b_j. If the non-basic variable corresponding to an entry is viable for pivoting, the function compares it, according to the chosen heuristic, to a previously computed 'best' non-basic Variable. If it is better, the `bestEntry` and `bestNonBasicVar` get updated.
`nextPivotingElement` checks, if the best entry found for a basic variable violating a bound is `LAST_ENTRY_ID`, in which case a pair `(LAST_ENTRY_ID,false)` is returned, indicating a conflict to `checkCore`.
If no conflict is detected, a pair `(bestEntry,true)` is returned, indicating a satisfiable assignment, if `bestEntry == LAST_ENTRY_ID`, otherwise indicating the next element for pivoting to `checkCore`.

Figure 4.2: Basic procedure for finding the best tableau-entry for pivoting a given basic variable

```
getTableauentryForPivot(Variable $b_j$):
    for ($entry_l$ |($entry_l$ is in row of $b_j$ and corresponding
    non-basic variable is viable for pivoting)):
        if (nonBasicVar corresponding to $entry_l$
        is better than bestNonBasicVar according to chosen heuristic):
            update bestEntry and bestNonBasicVar
    return bestEntry
```

4.2.2 Sum of Infeasibilities Implementation

Figure 4.3: Basic procedure for finding the next element for pivoting when using the *Sum of Infeasibilities heuristic*

```
nextPivotingElementSoF():
    checkForConflicts()
    if (there is a conflict):
        return (beginOfFirstConflictRow, false)
    for $n_j$ | ($n_j \in Flex(f)$)
        entry = getEntrySof($n_j$)
        if($sgn(\Delta Vio(\delta, j)) \cdot |coeff_j|$ is smaller than minValue)
            update bestEntry and minValue
    if (bestEntry == LAST_ENTRY_ID):
        return(LAST_ENTRY_ID, true)
    else:
        return(bestEntry, true)
```

Figure 4.3 shows the basic implemented procedure, for choosing the next pivoting element, when the *Sum of Infeasibilities* heuristic is chosen.
It iterates over the non-basic Variables in $Flex(f)$, if there is no conflict in the current

state of the tableau. If a conflict is detected, a pair `(LAST_ENTRY_ID,false)` is returned, indicating a conflict to `checkCore`.
A tableau-entry corresponding to each $n_j \in \mathit{Flex}(f)$ is computed separately by `getEntrySoF`(Figure 4.4), by creating a set S consisting of pairs (θ^+,entry_l) and θ^-,entry_l for each non-zero tableau-entry in the column of the given non-basic variable n_j and θ^+ and θ^- setting the corresponding basic variable to it's upper or lower bound. `getImpactSof()` computes ΔVio, as described in section 3.5.2 for each of these pairs and returns a tuple containing the smallest ΔVio as well as the corresponding θ and entry.
`nextPivotingElementSoF` chooses the entry minimizing $sgn(\Delta Vio(\delta,j)) \cdot \mid coeff_j \mid$ for pivoting and returns `(bestEntry,true)`, indicating the next pivoting element to `checkCore`, or `(LAST_ENTRY_ID,true)`, in the case of *Flex(f)* being empty, indicating a satisfying state of the tableau.

Figure 4.4: Basic procedure for finding the best tableau-entry for pivoting, when using the *Sum of Infeasibilities* heuristic

```
getEntrySoF(Variable $n_j$):
    S = $\emptyset$
    for($entry_l$ |($entry_l$ is in column of $n_j$ and $entry_l$ != 0)
        S ← {$\theta^+$,$entry_l$}
        S ← {$\theta^-$,$entry_l$}
    bestTuple = getImpactSoF(S,$n_j$)
    return bestTuple
```

4.3 Difficulties

The main difficulty implementing the *Sum of Infeasibilities* heuristic lied in the implementation of the optimization function as part of the tableau. As mentioned in section 4.2 the function has been implemented as part of the variable class, assigning each variable a coefficient. The initialization for these had to be done during each `checkCore` run, which would increasing runtime. The overall performance of the *Sum of Infeasibilities* strategy however could not be tested, as the final Implementation could not solve the vast majority of the benchmarks correctly. The reason for these errors could not be determined.

Chapter 5

Evaluation

The implemented strategies were tested on the *QFLRA* benchmarks of the SMT-LIB library [BST10], using the BENCHMAX tool provided with SMT-RAT. All tests were run on the cluster provided by the *Theory of Hybrid Systems Research Group*, using 4 x 2.1 GHz AMD Opteron with 12 Cores each and 192 GB of RAM.

5.1 Runtimes

In the following, all tests are represented through a scatterplot, comparing the runtimes of the tested strategy to the performance of the original LRA-MODULE with standard Settings and a second graph, showing for a given runtime (in min) the number of benchmarks which were solved in it, as well as the amount of benchmarks on which the solver did not time out. The graph of the reference can be seen in Figure 5.1. From 1648 tests, 973 were completed before a timeout. For testing purposes, all tests which took more than ten minutes were considered a timeout.

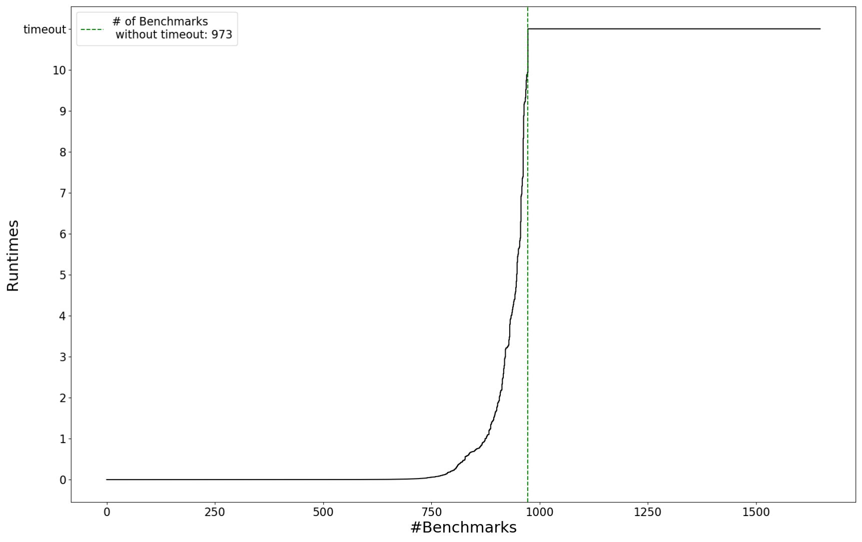
5.1.1 Difference-based results

Figure 5.2 shows the results of the tests done with the difference-based simplex heuristic. The second graph shows, that 928 of the benchmarks were completed before a timeout. The scatterplot shows, that the majority of tests, which did not cause a timeout were completed faster using the standard LRA-MODULE. Especially some tests, which were completed within a few seconds using the LRA-MODULE, fared way worse with the difference-based strategy, causing a timeout. Notably the difference-based approach timed out on more benchmarks of the `sal` [HMP16] family.

5.1.2 Activity-based results

Figure 5.3 shows the results of the tests done with the activity-based simplex heuristic. The second graph shows , that 922 of the benchmarks were completed before a timeout. The scatterplot shows, that the majority of tests, which did not cause a timeout, similar to the difference-based test results, were completed faster using the standard LRA-MODULE. While solving 3 benchmarks of the `miplib` family [GHG+19] and 6 of the `sc` family [BP06], which the original solver timed out on, the activity-based strategy performed, among others, notably worse on the `metitarsky`

Figure 5.1: Results of the tests for the original LRA-solver



family of benchmarks, which are generated by the *Meti-Tarski* tool [AP10], timing out on 16 of them, which could be solved by the original LRA-SOLVER

5.1.3 Priority-based results

Figure 5.4 shows the results of the tests done with the priority-based simplex heuristic using Bland's rule as a sub-heuristic, Figure 5.5 shows the results when using the difference-based sub-heuristic, Figure 5.6 when using the activity-based sub-heuristic and Figure 5.7 shows the results when using the mixed approach for the sub-heuristic. The second graphs show, that respectively 920, 918, 920 and 919 of the tests were completed before a timeout. For all used sub-heuristics, with exception of a few outliers, the scatterplots consistently show worse runtimes than the LRA-MODULE tests produced. Notably all versions of the priority-based approach timed out on more benchmarks of the `sc` and `sal` families.

Figure 5.2: Results of the tests for the difference-based heuristic

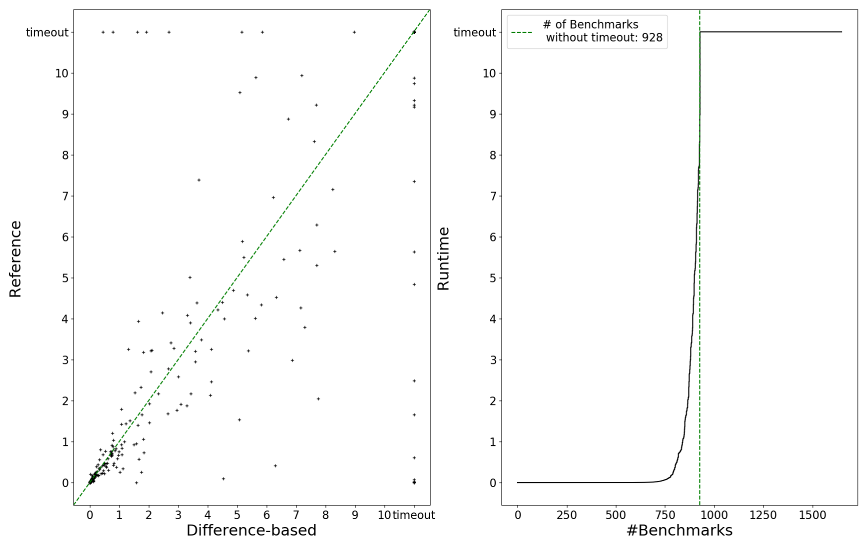


Figure 5.3: Results of the tests for the activity-based heuristic

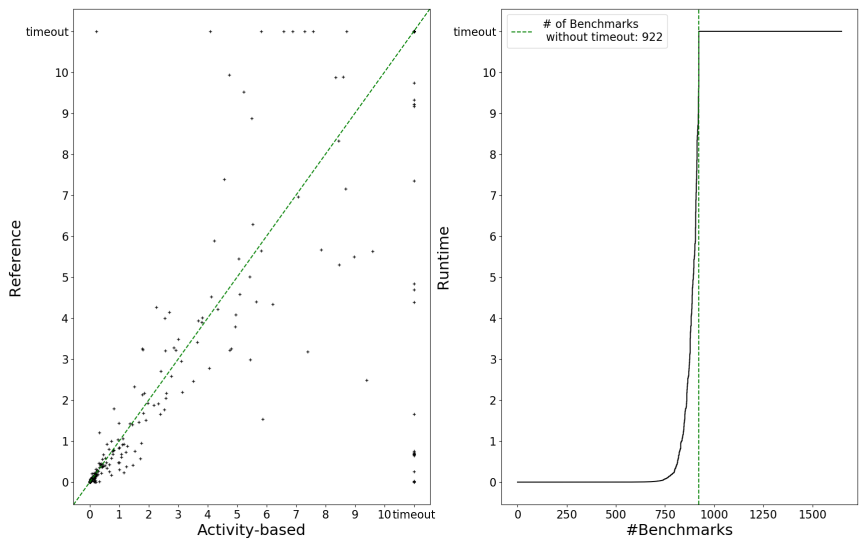


Figure 5.4: Results of the tests for the priority-based heuristic, using Bland's rule as a sub-heuritsic

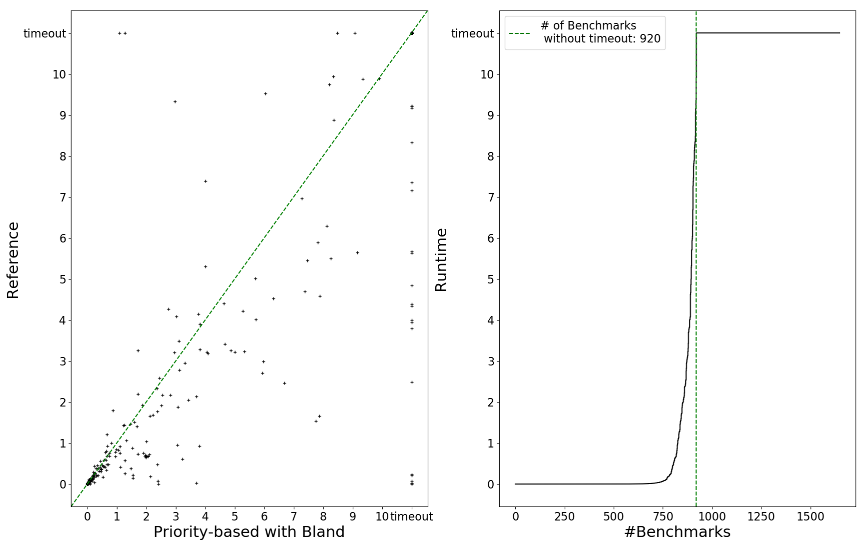


Figure 5.5: Results of the tests for the priority-based heuristic, using the difference-based approach as a sub-heuritsic

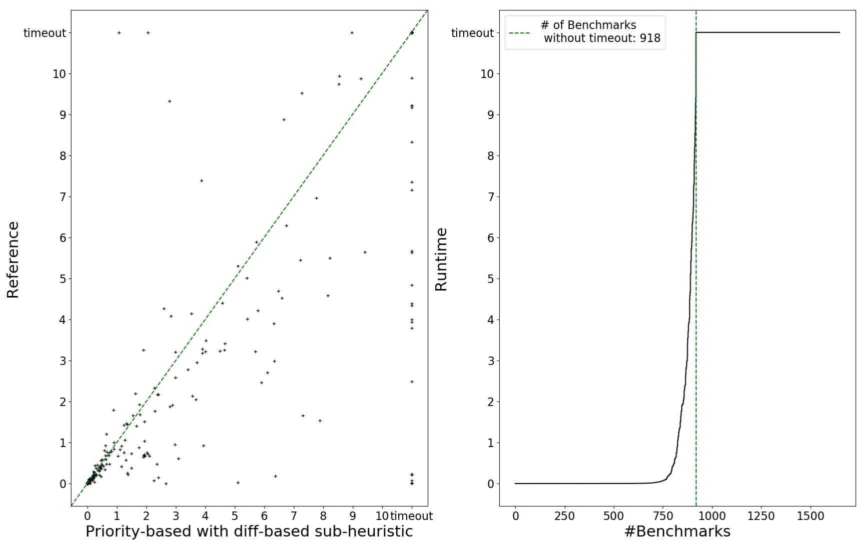


Figure 5.6: Results of the tests for the priority-based heuristic, using the activity-based approach as a sub-heuritsic

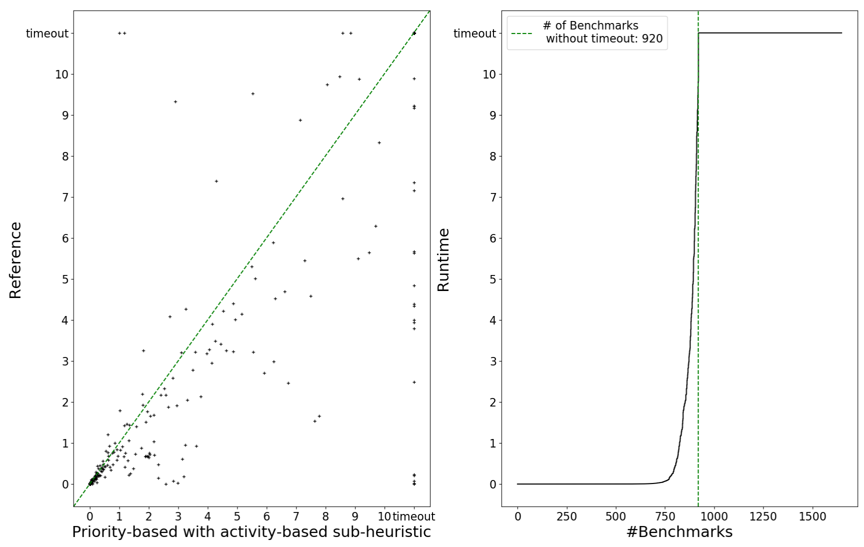
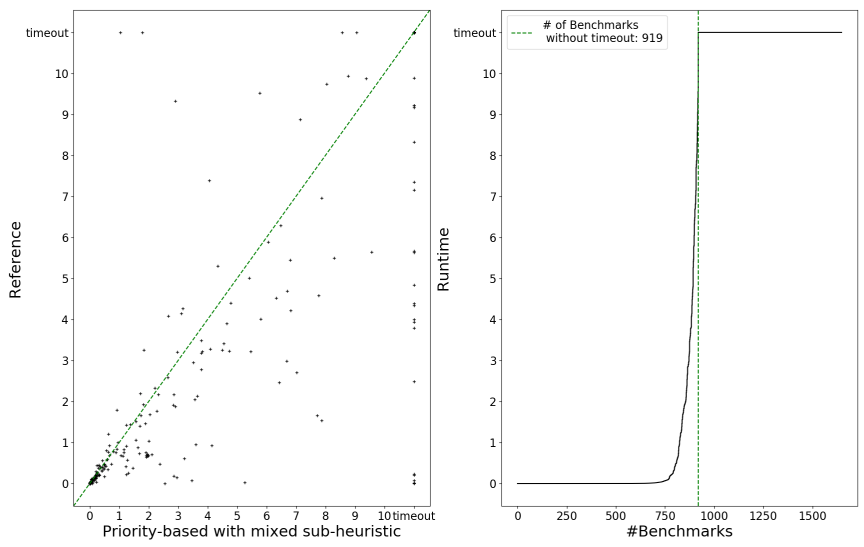


Figure 5.7: Results of the tests for the priority-based heuristic, using the mixed approach as a sub-heuritsic



Chapter 6

Conclusion

6.1 Summary

This work researched and implemented variable selection heuristics for pivoting in the context of the simplex algorithm in SMT-solving. The main concepts for those heuristics were based on the difference from assignment to the bounds of the variables, their pivoting activity, a previously assigned priority for either being a basic or a non-basic variable and an approach based on the *Sum of Infeasibilities* of all variables violating their bounds [KBD13]. The heuristics were implemented in SMT-RAT, a SMT-solver developed at the *Theory of Hybrid Systems Research Group*, however the Implementation of the heuristic based on the *Sum of Infeasibilities* could not be tested, as it still contained errors. The other heuristics were tested on the *QFLRA* benchmarks of the SMT-LIB library and compared to the original LRA-MODULE of SMT-RAT as a reference. While producing correct results, the overall performances of the tested strategies were worse than the performance of the original module, regarding number of timeouts and most individual runtimes.

It should be noted, that while it could not be implemented correctly in this work, the *Sum of Infeasibilities strategy* showed promising results in the tests done by [KBD13] and therefore should be considered for future work.

Bibliography

[AC78] David Avis and Vasek Chvátal. Notes on bland's pivoting rule. In *Polyhedral Combinatorics*, pages 24–34. Springer, 1978.

[AP10] Behzad Akbarpour and Lawrence Charles Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

[Bar71] Richard H Bartels. A stabilization of the simplex method. *Numerische Mathematik*, 16(5):414–434, 1971.

[BCD$^+$11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BP06] Geoffrey M Brown and Lee Pike. Easy" parameterized verificaton of cross clock domain protocols. In *Seventh International Workshop on Designing Correct Circuits DCC: Participants' Proceedings*. Citeseer, 2006.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 150–153. Springer, 2010.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). www. *SMT-LIB. org*, 15:18–52, 2010.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating smt solver. In *International SPIN Workshop on Model Checking of Software*, pages 248–254. Springer, 2012.

[CKJ$^+$15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT : an Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Theory and Applications of Satisfiability Testing : SAT 2015 ; 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings / Marijn Heule ; Sean Weaver*

[Hrsg.], volume 9340 of *Lecture Notes in Computer Science*, pages 360–368, Cham, Sep 2015. International Conference on Theory and Applications of Satisfiability Testing, Austin, Tex. (USA), 24 Sep 2015 - 27 Sep 2015, Springer International Publishing.

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

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DOW$^+$55] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[GHG$^+$19] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. Miplib 2017: Data-driven compilation of the 6th mixed-integer programming library. *Optimization online preprint: http://www. optimization-online. org/DB_HTML/2019/07/7285. html. Submitted to Mathematical Programming Computation*, 2019.

[HMP16] Andrew Healy, Rosemary Monahan, and James F Power. Evaluating the use of a general-purpose benchmark suite for domain-specific smt-solving. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1558–1561, 2016.

[KBD13] 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. IEEE, 2013.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.