

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

BACHELOR OF SCIENCE THESIS

ADAPTING CYLINDRICAL ALGEBRAIC COVERINGS FOR NON-LINEAR OPTIMIZATION

Jan Hebisch

Examiners:

Prof. Dr. Erika Ábrahám

Prof. Dr. Jürgen Giesl

Additional Advisor:

Valentin Promies

Aachen, February 14, 2025

Abstract

The Cylindrical Algebraic Coverings (CAIC) algorithm provides a method for solving Satisfiability Modulo Theories (SMT) on the domain of non-linear real arithmetic (NRA). This thesis aims to reshape the algorithm to fit optimization problems, which includes the treatment of unbounded or open solution intervals, where the optimum is realized from a found cell. An implementation is presented and tested on datasets differing in objective functions. Additionally, a subsequent procedure is used to verify that a concluded result is in fact optimal.

Contents

1	Introduction	7
2	Preliminaries	9
2.1	Satisfiability Problems	9
2.2	The Coverings Algorithm	11
3	Optimization	23
3.1	Nonlinear Optimization	23
3.2	CAIC for Optimization	23
3.3	Example	29
4	Implementation and Benchmarking	35
4.1	Implementation in SMT-RAT	35
4.2	Benchmark Procedure	35
4.3	Benchmark Results	37
5	Conclusion	45
5.1	Summary	45
5.2	Future work	45
	Bibliography	49

Chapter 1

Introduction

Cylindrical Algebraic Coverings (CAIC), first presented in [KÁED21], have emerged as a fresh contender to the established Cylindrical Algebraic Decompositions (CAD), introduced in [Col98], in solving non-linear real arithmetic formulae. Suggested advantages opposed to the long-standing CAD algorithm are a lower number of projections, polynomials and samples. A key limitation of the CAD algorithm is the excessive number of resultants computed between polynomials that could be avoided if, rather than decomposing the real line, one were to cover it. In certain examples, the effect of disregarding certain resultants can be significant, depending on the complexity of the constraints involved. This paper aims to transform the CAIC algorithm into an optimization variant, either minimizing or maximizing a function over variables subject to non-linear real arithmetic constraints comprised in a formula. Since the optimization approach will largely consist of solving different satisfiability problems, the benefits of using the CAIC algorithm, as opposed to simply parsing these subtasks to a CAD implementation, are effective on the optimization variant as well.

In chapter 2, the underlying technical foundations for SMT-solving are introduced. Following that, the CAIC algorithm as well as its implemented variant for quantified formulae are presented. Thereafter, in chapter 3, the optimization problem is formulated and an implementation of an optimization variant of the CAIC algorithm is introduced, consisting of not only finding an optimal target function value, but also efficiently inferring a full optimal solution based on previous, buffered findings. Chapter 4 briefly outlines the framework within which the implementation is made. Afterwards, due to the lack of existing fitting benchmark sets, it is described how optimization benchmark sets are inferred from a single satisfiability benchmark set. Subsequently, a procedure is introduced to verify the optimality of the solutions found for the benchmark sets. Furthermore, the results are evaluated with regard to the effects of different optimization functions on runtime as well as connections between properties of certain benchmark instances and performance. Finally, chapter 5 summarizes the results and provides an outlook on further potential extensions.

Chapter 2

Preliminaries

2.1 Satisfiability Problems

The satisfiability problem is to decide whether a formula φ defined over variables $x_1, \dots, x_n$ can be satisfied. Given an assignment (also called interpretation), mapping each variable to a value within a given domain, φ either evaluates to `true` or `false`. If there exists an assignment such that φ , evaluated under this assignment, yields `true`, the formula is called satisfiable (SAT), and otherwise unsatisfiable (UNSAT). One kind of formula, restricted to the Boolean domain only, is defined by *Propositional Logic*.

Definition 2.1.1 (Propositional Logic Formula [BT18]). A propositional logic formula φ is evaluated with respect to a set of atomic propositions AP , which are to be evaluated as Boolean values. It is constructed with Boolean functions of atomic propositions and other well formed propositional logic formulae:

$$\varphi := a \mid \neg\varphi \mid \varphi \wedge \varphi$$

where $a \in AP$. With negation and conjunction composing a functionally complete system, arbitrary Boolean functions can be formulated through combination of those stated.

Despite their restricted expressiveness, propositional logic formulae can play a crucial part in the solving of more complex formulae, as will be discussed later in this chapter. Firstly, to increase said expressiveness of our formulae, we are interested in expanding the set of atomic propositions AP beyond Boolean constants.

Definition 2.1.2 (First-Order Logic Formula [BT18]). A first-order logic formula is any formula constructed as follows:

$$\varphi := p(x_1, \dots, x_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi \mid \forall x.\varphi$$

where $x_1, \dots, x_n$ are variables, p is a predicate symbol mapping an assignment for a set of variables to either `true` or `false` and the quantifiers $\exists$ and $\forall$ allow generalizing variables over their domain. An assignment maps each variable to a value within its domain.

First-order logic expands propositional logic by extending our set of atomic Boolean variables AP with propositions $p(x_1, \dots, x_n)$. This implies we are no longer restricted to Boolean input variables and can now evaluate arbitrary inputs, given fitting predicates. Such propositions will be referred to as *constraints*, as they can restrict the set of possible assignments for the decision variables.

Definition 2.1.3 (Polynomial Constraints). We define a polynomial constraint as

$$c := \sum_{i=0}^m a_i \prod_{j=0}^n x_j^{e_{i,j}} \sigma 0$$

where for $0 \leq i \leq m$ and $0 \leq j \leq n$, $e_{i,j} \in \mathbb{N}_0$ are exponents, $a_i \in \mathbb{Q}$ are coefficients, and x_j are the variables. We establish an ordering $x_1 \prec x_2 \prec \dots \prec x_n$ among these variables according to the ordering of their labels. σ is a comparator from the set $\{<, \leq, >, \geq, \neq, =\}$.

For the purpose of this paper, the scope of our formulae is narrowed down to polynomial constraints only, which is the field known as *non-linear real arithmetic*, often referred to as *Real Algebra*.

Definition 2.1.4 (Non-linear Real Arithmetic Formula [KN22]). *Non-linear real arithmetic* is a first-order logic theory, where predicates are polynomial constraints and the variable domain are the real numbers:

$$\varphi := c \mid \neg \varphi \mid \varphi \wedge \varphi \mid \exists x. \varphi \mid \forall x. \varphi$$

with c being a polynomial constraint over variables $\{x_1, \dots, x_n\}$ with each x_i being over the domain $\mathbb{R}$ for $0 \leq i \leq n$.

2.1.1 SMT-Solving for Non-Linear Real Arithmetic

The satisfiability of a non-linear real arithmetic formula falls under the category of *Satisfiability Modulo Theory solving (SMT-solving)*. SMT-solving means determining the satisfiability of a first-order formula with respect (modulo) to a certain theory, in our case being defined by a signature $\Sigma = (\{+, -, *\}, \{<, \leq, >, \geq, \neq, =\})$ and the real numbers $\mathbb{R}$ [BT18]. The field of SMT-Solving can be divided into the categories *eager* and *lazy*. The *eager* approach consists of inferring a propositional logic formula from the first-order logic formula, which can then be solved using a SAT-solver. The *lazy* approach in contrast begins by using a SAT-solver on a *Boolean abstraction* and treats the theory component afterwards, in the so-called *theory search*.

Definition 2.1.5 (Boolean Abstraction). Given a non-linear real arithmetic formula φ , we obtain its *Boolean abstraction* φ' by replacing each of its constraints c_i entirely with a new variable $a_i \in AP$. φ' is a propositional logic formula.

After obtaining the Boolean abstraction of the input formula, the SAT-solver either directly concludes that it is unsatisfiable, in which case we can ignore the theory and conclusively report UNSAT, or it evaluates it to SAT. In the latter case, the theory solver now receives a set C of constraints, namely those constraints c_i (or $\neg c_i$) of which the corresponding abstraction variable a_i was assigned `true` (or `false`) in the solution. The theory solver now searches for a satisfying assignment of the variables of the original formula, that is, an assignment that satisfies each of the constraints

in C . If we conclude SAT, the entire algorithm terminates, yielding the satisfying assignment. In the case of a constraint set for which the conjunction yields UNSAT, the theory solver has the additional task of providing an explanation to the SAT-solver such that it can reduce its search space for satisfying assignments. This explanation will be a preferably minimal infeasible subset of C , which the SAT-solver can use to extend the Boolean abstraction in such a way that this subset's constraints will no longer be assigned to be entirely `true`.

2.2 The Coverings Algorithm

Having iterated the basic foundations of our problem domain, we now move to the actual algorithm that solves NLRA-formulae. The Cylindrical Algebraic Covering (CAlC) algorithm, first introduced in [KÁED21], can be applied to solve Boolean combinations of non-linear real arithmetic constraints. While initially presented for quantifier-free formulae only (i.e. existential quantifiers remain allowed, as, in the scope of satisfiability, they can be seen equivalent to the absence of a quantifier), it can be generalized to formulae with both existential and universal quantifiers, as shown in [KN22].

Although this thesis expands on the coverings algorithm, we begin by introducing cylindrical algebraic decompositions, which act as the groundwork for the coverings algorithm.

2.2.1 Cylindrical Algebraic Decompositions

The Cylindrical Algebraic Decompositions (CAD) algorithm, introduced in [Col98], tackles the problem of satisfiability by *decomposing* the search space into regions, so-called *cells*, for each of which all of the polynomials in the input formula remain sign-invariant, i.e. the polynomial constraints remain satisfiability-invariant. Then, satisfiability (and, if the problem is satisfiable, a solution as well) can be inferred by sampling each of these regions exactly once and evaluating each constraint with each sample.

Definition 2.2.1 (Cell [KÁED21]).

1. For a space $\mathbb{R}^n$, a *cell* is a non-empty connected subset of $\mathbb{R}^n$.
2. A cell C is *algebraic* if it can be described as the solution set of a conjunction of polynomial constraints, restricted to the comparators $\{=, >, <\}$.
3. A cell C from $\mathbb{R}^n$ is *sign-invariant* for a polynomial $p(x_1, \dots, x_n)$ if and only if one of these conditions holds:
 - (a) $\forall x \in C. p(x) > 0$
 - (b) $\forall x \in C. p(x) < 0$
 - (c) $\forall x \in C. p(x) = 0$

Definition 2.2.2 (Decomposition [KÁED21]).

1. A *decomposition* D of $\mathbb{R}^n$ partitions the space into finitely many pairwise disjoint cells:

$$D = \{C_1, \dots, C_h\}$$

with $\mathbb{R}^n = \cup_{i=1}^h C_i$ for pairwise disjoint C_i

2. A decomposition is *algebraic* if each of its cells is algebraic.
3. A decomposition D can be *sampled* using a function assigning a point $s \in C$ to each cell $C \in D$.
4. A *cylindrical decomposition* D of $\mathbb{R}^n$ over a decomposition D' of $\mathbb{R}^m$ with $m < n$ describes a decomposition for which each cell's projection onto $\mathbb{R}^m$ yields a cell of D' . In consequence, the projections are either identical or disjoint, and not overlapping.

Definition 2.2.3 (Delineability [Áb22]). We denote the substitution of a value a for a variable x in a polynomial p as $p(a, x)$. A set $P \subseteq \mathbb{Q}[x_1, \dots, x_n]$ of polynomials is delineable on a region $R \subseteq \mathbb{R}^{n-1}$ if for each $1 \leq i, j \leq n$ with $i \neq j$, for all $a \in R$, the following three conditions hold:

1. The number of roots of $p_i(a, x_n)$ is constant.
2. The number of different roots of $p_i(a, x_n)$ is constant.
3. The number of common roots of $p_i(a, x_n)$ and $p_j(a, x_n)$ is constant.

The aim is now to deduce a cylindrical decomposition that is algebraic in the sense that it can be described as a conjunction of polynomial constraints, namely a conjunction containing each polynomial constraint by itself or its negation. CAD achieves this in three phases.

1. **Projection Phase:** In the first phase, we iteratively eliminate main variables from the polynomials, creating one set $P_i \subseteq \mathbb{Q}[x_1, \dots, x_i]$ for each of the n dimensions, starting with P_n , the polynomials present in the original constraint set. P_{i-1} is then created using the polynomials from P_i , and consists of each polynomial's discriminants, leading coefficients, and resultants of combinations of polynomials.
2. **Base phase:** The real roots of the projection phase's last calculated polynomials, namely those from P_1 , are calculated. We then gather a set Z_1 of samples by taking all of these roots, then one sample from each of the spaces in between, one below the smallest and one above the largest roots.
3. **Lifting Phase:** In the final phase, we extend the one-dimensional samples in Z_1 to a final set Z_n of n -dimensional samples, one dimension at a time. Z_{i+1} is inferred from Z_i by taking each polynomial from P_{i+1} , evaluating it for each sample $z \in Z_i$ and computing the roots. From (and around) these roots, samples are taken analogously to the base phase and make up Z_{i+1} .

The motivation behind the projection phase is delineability. For a given sample in the i -th lifting phase, we know that the zeros of these characterizing polynomials on the real line for x_i define the positions at which the polynomials on the next higher dimension might violate delineability. This happens because either their roots cross (detected through resultants), or they diverge (detected through leading coefficients), or the number and order of roots of just a single polynomial changes without divergence (detected through discriminants).

2.2.2 Cylindrical Algebraic Coverings in a Theory-Last Approach

The CAIC algorithm from [KÁED21] differs from CAD in two major aspects: Firstly, while CAD is chronologically divided into projection, base and lifting phase, CAIC is of recursive nature, both extending and reducing sample points in dimensions. Depending on the implementation, this can already imply having to determine fewer cells and samples. Secondly, by using coverings instead of decompositions, we can reduce the number of polynomials based on which we characterize satisfiability-invariant regions. Due to its nature of solving only a conjunction of constraints, this foundational algorithm requires a *lazy* SMT-solving architecture that first extracts such conjunctions from a formula.

Given a set of polynomial constraints, determined by a preceding SAT solver, the original covering algorithm carries out the theory search through partial sampling of solutions. With respect to a predefined variable ordering $x_1 \prec x_2 \prec \dots \prec x_n$, CAIC begins with sampling a value for the lowest dimension, i.e. that of x_1 . For this sample to be potentially satisfying, unsatisfying intervals for the current variable with respect to the previous sample (which in this first case is the empty sample) are computed with polynomial constraints that, evaluated with the sample, are univariate. Then, a sample outside of these intervals is picked and we reiterate the process of finding unsatisfying intervals. We continue until either a satisfying solution is found, or a dimension is reached for which no value can be sampled, due to the entire real line being covered by unsatisfiable intervals. In this case, we pick an ideally as small as possible set of polynomials, so that such an UNSAT covering of the real line can be concluded from just these polynomials alone with the current sample. In the next step, we want to infer an interval around the sample point on the previous dimension which we can exclude from our further search. Stemming from the Cylindrical Algebraic Decomposition (CAD) algorithm, CAIC now takes use of CAD-technology to generalize this unsatisfying sample and the corresponding polynomials to a larger interval. Through recursive applying of these steps and with the help of bookkeeping for UNSAT intervals and their responsible polynomials, either a satisfying solution or an infeasible subset of constraints can be found.

Definition 2.2.4 (Covering [KÁED21]).

1. A *covering* D of $\mathbb{R}^n$ decomposes the space into finitely many cells:

$$D = \{C_1, \dots, C_h\}$$

with $\mathbb{R}^n = \bigcup_{i=1}^h C_i$. The key difference to a decomposition is that the cells do not need to be pairwise disjoint, i.e. they can overlap.

2. A covering of $\mathbb{R}^n$ is *algebraic* if each of its cells is algebraic.
3. A covering D can be *sampled* using a function assigning a point $s \in C$ to each cell $C \in D$.
4. A *cylindrical covering* D of $\mathbb{R}^n$ over a covering D' of $\mathbb{R}^m$ with $m < n$ describes a decomposition for which each cell's projection onto $\mathbb{R}^m$ yields a cell of D' .

CAIC produces coverings once an UNSAT extension of a sample has been found. While we could now simply determine all sign-invariant regions for the set of polynomials at our current dimension, we instead focus only on a subset of constraints and their sign-invariance, namely those representing the potentially overlapping cells which make up the covering of the real line. If the extended sample trivially violates a single constraint, said subset will only consist of that single constraint. If e.g. further recursion yields a more complex UNSAT cover, multiple intervals represented by multiple polynomials must be considered. To keep track of these cells and their representing polynomials, we must introduce a fitting data structure.

Definition 2.2.5 (Algebraic Intervals [KÁED21, KN22]). An *algebraic interval* over an $(i - 1)$ -dimensional sample point s is a tuple $I = (I_\ell, I_u, I_L, I_U, I_{P_i}, I_\perp)$ containing:

- $\ell \in \mathbb{R}$ the lower bound
- $u \in \mathbb{R}$ the upper bound
- I_L the set of polynomials defining the lower bound
- I_U the set of polynomials defining the upper bound
- I_{P_i} a set of multivariate polynomials with main variable x_i
- $I_\perp$ a set of multivariate polynomials with main variable of lower dimension than that of x_i

Precisely, the polynomials in I_L and I_U have the main variable x_i and have a root for the previous sample, which defines the corresponding lower or upper interval border of the enclosing interval. The polynomials from I_{P_i} and $I_\perp$ define the unsatisfiability of the generalized interval in the way that any perturbation of an i -dimensional sample point that does not lead to a sign change in any of these polynomials will not cause a new SAT region opening up within the interval. Thereby we can ensure that I is entirely UNSAT for the current partial sample.

Algebraic intervals are what we refer to as our bookkeeping structure, implying they serve a further purpose than just to keep track of UNSAT intervals on the current real line $s \times \mathbb{R}$. Considering the base algorithm, this of course is extended by polynomials being passed down from one interval to the next lower-dimensional one. In Chapter 3.2.3 however, these structures will play a crucial role in efficiently finding an optimal solution with respect to an interval.

2.2.3 Constructing UNSAT Intervals

Now, a method is needed to generalize an unsatisfying partial sample to an interval, with the aim of covering the real line of the partial sample with UNSAT intervals.

Whenever the algorithm finds a covering of the real line at dimension $i + 1$ for a sample $s \in \mathbb{R}^i$, represented by a set of algebraic intervals $\mathbb{I}$, the algorithm now constructs a *characterization* from each of those structures I ([KÁED21], Algorithm 4). Since order-invariance of the polynomials in P_{i+1} must be secured among each other as well as in regard to I_L and I_U , resultants, discriminants and leading coefficients are all included in this characterization in a similar manner to CAD. Polynomials with lower-dimensional main variables $I_\perp$ are also included. These polynomials skip at least one dimension but could restrict intervals at lower dimensions. Finally, the

characterization is expanded by the resultants of those polynomials defining upper and lower borders of neighboring intervals, from left to right. This ensures that the interval borders do not cross, no matter what value from within the generalizing UNSAT interval at dimension i is picked, such that the whole real line on dimension $i + 1$ remains covered by UNSAT intervals.

Now, this characterization needs to be converted into a single resulting algebraic interval I ([KÁED21], Algorithm 5). Therefore, the real line is partitioned by the roots of all characterizing polynomials with main variable x_i , since we need a region where all those polynomials are sign-invariant w.r.t. our current dimension, implying order-invariance (and thereby completeness, with no gaps) of the previously inferred covering. Then, we pick those roots closest to our UNSAT (partial) sample as the new interval's lower and upper bound, and choosing the corresponding polynomials for I_L and I_U respectively. I_{P_i} and $I_{\perp}$ consist of the entire characterization, categorized according to their main variable.

2.2.4 CAIC for Quantified Formulae: A Theory-First Approach

The discussed coverings algorithm so far only deals with quantifier-free formulae. Therefore, in [KN22], a more generalized variant is introduced. Quantified input formulae firstly are to be transformed into prenex normal form. On our domain of satisfiability, the algorithm treats each variable as quantified, adding an existential quantifier for each previously unquantified variable which is exactly what is implied by the task of finding a satisfying assignment. The resulting formula will be of the form:

$$\varphi = \exists x_1 \dots \exists x_k Q_{k+1} x_{k+1} \dots Q_n x_n \bar{\varphi}(x_1, \dots, x_n)$$

with quantifiers $Q_i \in \{\exists, \forall\}$ and originally quantified variables x_i , $i \in \{k+1, \dots, n\}$, originally unquantified variables x_j , $j \in \{1, \dots, k\}$, and a quantifier-free formula $\bar{\varphi}$ called *matrix*. We call $\exists x_1 \dots \exists x_k Q_{k+1} x_{k+1} \dots Q_n x_n$ the *prefix*.

Algorithm 1: user_call() [KN22]

Data: Global prefix $Q_1 x_1 \dots Q_n x_n$ and matrix $\bar{\varphi}$.

Result: Either SAT or UNSAT.

$(f, O) := \text{recurse}(());$

// Alg. 2

return f ;

Algorithm 2: recurse(s) [KN22]

Data: Global prefix $Q_1 x_1 \dots Q_n x_n$ and matrix $\bar{\varphi}$.

Input: Sample point $s = (s_1, \dots, s_{i-1}) \in \mathbb{R}^{i-1}$ such that $\bar{\varphi}[s] \neq \text{false}$.

Result: (SAT, I) or (UNSAT, I) where $s \times I$ can or can not be extended to a model for any $s_i \in I$. In both cases, the algebraic information attached to I describes how s can be generalized.

if $Q_i = \exists$ **then**

 | **return** exists(s);

// Alg. 3

end

else

 | **return** forall(s);

// Alg. 4

end

Algorithm 3: exists(s) [KN22]

Data: Global prefix $Q_1x_1...Q_nx_n$ and matrix $\bar{\varphi}$.
Input: Sample point $s = (s_1, ..., s_{i-1}) \in \mathbb{R}^{i-1}$ such that $\bar{\varphi}[s] \neq \text{false}$.
Result: see Alg. 2.
 $\mathbb{I}_{\text{unsat}} := \emptyset$;
while $\bigcup_{I \in \mathbb{I}_{\text{unsat}}} I \neq \mathbb{R}$ **do**
 $s_i := \text{sample_outside}(\mathbb{I}_{\text{unsat}})$;
 if $\bar{\varphi}[s \times s_i] = \text{false}$ **then**
 $(f, O) := (\text{UNSAT}, \text{get_enclosing_interval}(s, s_i))$; // Alg. 5
 else if $\bar{\varphi}[s \times s_i] = \text{true}$ **then**
 $(f, O) := (\text{SAT}, \text{get_enclosing_interval}(s, s_i))$; // Alg. 5
 else
 $(f, O) := \text{recurse}(s \times s_i)$; // Alg. 2
 end
 if $f = \text{SAT}$ **then**
 $R := \text{characterize_interval}(s, O)$; // Alg. 6
 $I := \text{interval_from_characterization}((s_1, ..., s_{i-2}), s_{i-1}, R)$;
 // [KÁED21], Alg.5
 return (SAT, I) ;
 else if $f = \text{UNSAT}$ **then**
 $\mathbb{I}_{\text{unsat}} := \mathbb{I}_{\text{unsat}} \cup \{O\}$;
end
 $R := \text{characterize_covering}(s, \mathbb{I}_{\text{unsat}})$; // Alg. 7
 $I := \text{interval_from_characterization}((s_1, ..., s_{i-2}), s_{i-1}, R)$;
// [KÁED21], Alg.5
return (UNSAT, I) ;

In algorithm 2, we now iterate over the prefix, arranged by the predefined variable ordering. In case of an existential quantifier, the routine in algorithm 3 is now equivalent to the previous routine: We first sample until we either satisfy or violate our formula and then infer an enclosing interval with algorithm 5. We then collect UNSAT intervals until either the real line is covered, yielding UNSAT, or a SAT interval is found, yielding SAT, and continue this step throughout dimensions, inferring intervals through characterizations whenever a result is found at a higher dimension. The case of a universal quantifier requires the inverse procedure: In algorithm 4, SAT intervals are collected until either the real line is covered, yielding SAT, or an UNSAT interval is found, yielding UNSAT.

Algorithm 4: forall(s) [KN22]

Data: Global prefix $Q_1x_1...Q_nx_n$ and matrix $\bar{\varphi}$.
Input: Sample point $s = (s_1, \dots, s_{i-1}) \in \mathbb{R}^{i-1}$ such that $\bar{\varphi}[s] \neq \text{false}$.
Result: see Alg. 2.

$\mathbb{I}_{sat} := \emptyset;$
while $\bigcup_{I \in \mathbb{I}_{sat}} I \neq \mathbb{R}$ **do**
 $s_i := \text{sample_outside}(\mathbb{I}_{sat});$
 if $\bar{\varphi}[s \times s_i] = \text{false}$ **then**
 $(f, O) := (\text{UNSAT}, \text{get_enclosing_interval}(s, s_i));$ // Alg. 5
 else if $\bar{\varphi}[s \times s_i] = \text{true}$ **then**
 $(f, O) := (\text{SAT}, \text{get_enclosing_interval}(s, s_i));$ // Alg. 5
 else
 $(f, O) := \text{recurse}(s \times s_i);$ // Alg. 2
 end
 if $f = \text{SAT}$ **then**
 $\mathbb{I}_{sat} := \mathbb{I}_{sat} \cup \{O\};$
 else if $f = \text{UNSAT}$ **then**
 $R := \text{characterize_interval}(s, O);$ // Alg. 6
 $I := \text{interval_from_characterization}((s_1, \dots, s_{i-2}), s_{i-1}, R);$
 // [KÄED21], Alg.5
 return (UNSAT, I);
 end
 $R := \text{characterize_covering}(s, \mathbb{I}_{sat});$ // Alg. 7
 $I := \text{interval_from_characterization}((s_1, \dots, s_{i-2}), s_{i-1}, R);$
 // [KÄED21], Alg.5
return (SAT, I);

Algorithm 5: get_enclosing_interval(s, s_i) [KN22]

Data: Global matrix $\bar{\varphi}$
Input: Sample point $s \in \mathbb{R}^{i-1}$ and $s_i \in \mathbb{R}$ such that
 $\bar{\varphi}[s \times s_i] \in \{\text{false}, \text{true}\}.$
Result: A satisfiability-invariant algebraic interval I around s_i over s
 $P := \text{implicant_polynomials}(\bar{\varphi}, s \times s_i);$
Perform standard CAD simplifications to P ;
 $I := \text{interval_from_characterization}(s, s_i, P);$ // [KÄED21],
Alg.5
return I ;

Both subroutines 3 and 4 also perform the inference of a characterization and the subsequent generalization, which is either performed based on a covering or a single interval from the higher dimension. This step is performed not only for UNSAT, but also SAT solutions, which is necessary because universal quantifiers from lower dimensions need to be resolved using these generalizations.

This new approach also differs in the fact that the matrix is not just a conjunction of constraints derived by a SAT-solver from a Boolean abstraction. Considering this, algorithm 5 makes use of *implicants*, with which, given a sample that suffices to evaluate the matrix to either true or false instantly, a satisfiability invariant interval representation can be determined with fewer polynomials than using all

polynomials that are violated. This not only means less computational effort for the characterization, but also potentially smaller intervals as less polynomial zeros are considered for interval borders.

Definition 2.2.6 (Implicants [KN22]). Let ψ and φ be formulae, $s \in \mathbb{R}^i$ a (partial) sample, and let $\text{constraints}(\psi)$ and $\text{constraints}(\varphi)$ be the sets of constraints contained within ψ and φ respectively. We call ψ an *implicant* of φ with respect to s if one of the two following statements holds:

- $\varphi(s) = \text{true} \wedge \psi(s) = \text{true} \wedge (\psi \implies \varphi) \wedge \text{constraints}(\psi) \subseteq \text{constraints}(\varphi)$
- $\varphi(s) = \text{false} \wedge \psi(s) = \text{true} \wedge (\psi \implies \neg\varphi) \wedge \text{constraints}(\psi) \subseteq \text{constraints}(\varphi)$

For a *prime implicant* φ of ψ , it must hold that $\text{constraints}(\psi)$ is minimal w.r.t. all other implicants of ϕ , i.e. there exists no other implicant ψ' of ϕ with $\text{constraints}(\psi') \subset \text{constraints}(\psi)$

Algorithm 5 collects implicants of $\bar{\varphi}$ with respect to s using a helper-method `implicant_polynomials( $\bar{\varphi}, s \times s_i$ )`. Since we are dealing with an implicant from which we can conclude general satisfiability w.r.t. the sample, the sign-invariance of its characterization will ensure satisfiability-invariance within the cell nonetheless, making the implicant applicable for both SAT and UNSAT intervals.

Algorithm 6: `characterize_interval( $s, I$ )` [KN22]

Input: Sample point $s \in \mathbb{R}^i$ and a single interval I over s in dimension $i + 1$.

Result: Polynomials $R \subseteq \mathbb{Q}[x_1, \dots, x_i]$ characterizing a satisfiability-invariant region around s .

Extract $\ell = I_\ell$, $u = I_u$, $L = I_L$, $U = I_U$, $P_{i+1} = I_{P_{i+1}}$, $P_\perp = I_{P_\perp}$;
 $R := P_\perp \cup \text{disc}(P_{i+1}) \cup \{\text{required_coefficients}(p) \mid p \in P_{i+1}\}$;
 $R := R \cup \{\text{res}(p, q) \mid p \in L, q \in P_{i+1}, q(s \times \alpha) = 0 \text{ for some } \alpha \leq l\}$;
 $R := R \cup \{\text{res}(p, q) \mid p \in U, q \in P_{i+1}, q(s \times \alpha) = 0 \text{ for some } \alpha \geq u\}$;
 Perform standard CAD simplifications to R ;
return R ;

Algorithm 7: `characterize_covering( $s, \mathbb{I}$ )` [KN22]

Input: Sample point $s \in \mathbb{R}^i$ and a covering of algebraic intervals $\mathbb{I}$ over s in dimension $i + 1$.

Result: Polynomials $R \subseteq \mathbb{Q}[x_1, \dots, x_i]$ characterizing a satisfiability-invariant region around s .

$\mathbb{I} := \text{compute_cover}(\mathbb{I});$ // [KÁED21], Section 4.4.1
 $R := \bigcup_{I \in \mathbb{I}} \text{characterize_interval}(s, I);$ // Alg. 6
for $j \in \{1, \dots, |\mathbb{I}| - 1\}$ **do**
 | $R := R \cup \{\text{res}(p, q) \mid p \in U_j, q \in L_{j+1}\}$;
end
 Perform standard CAD simplifications to R ;
return R ;

Algorithm 6 and 7 compute the characterization in a similar manner to 2.2.3, the only noteworthy difference being that the characterized region may also be SAT, a difference with no direct effect on the two subroutines themselves. Once a sample s is reached, for which the matrix completely evaluates to either `true` or `false`, the goal is to extend that sample to an interval within which the formula retains its status

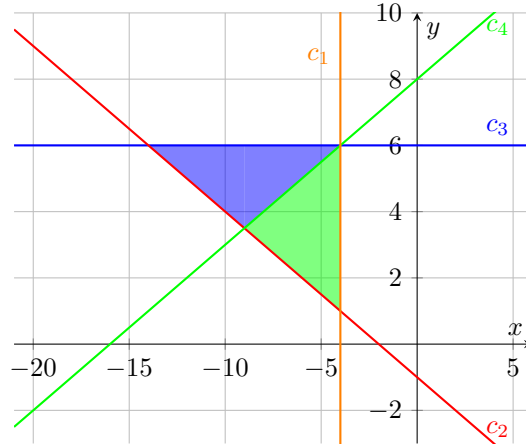


Figure 2.1: Feasible region for $\varphi = c_1 \wedge c_2 \wedge (c_3 \vee c_4)$ with constraints:

$$c_1 : 4 + x < 0$$

$$c_2 : -x - 2y - 2 < 0$$

$$c_3 : y - 6 < 0$$

$$c_4 : -0.5x + y - 8 < 0$$

In the green area, all constraints are fulfilled, while in the blue area, all but c_4 are.

(SAT or UNSAT). Through use of implicants, we only consider a potentially small set polynomials and then as usual take the closest zeros to the sample as interval borders. The use of implicants corresponds to the SAT-solving component of NRA, as here, the Boolean abstraction is considered. This theory-first approach allows us to solve the satisfiability-problem in only a single theory search, whereas the previous approach required a multitude, depending on the preceding assignment for the Boolean abstraction.

2.2.5 Example

To illustrate the basic process of the covering algorithm, we iterate through it with input φ from figure 2.1. The task is to determine satisfiability of this NLRA formula. To keep the example simple, it remains quantifier free. Still, as the example is not just a conjunction of constraints, it showcases that 2.2.4 generalizes the approach to all kinds of Boolean combinations, saving us the potential multiple iterations of a theory search.

Before we evaluate partial samples for the variables contained in φ , some pre-processing is required. Since our algorithm only treats quantified variables, we extend the quantified prefix with an existential quantifier for each previously free variable. Then, the *exists*-routine is called for the first variable x . Firstly, x is assigned an arbitrary sample $s = (0)$. We now evaluate the matrix and find that $\bar{\varphi}[s] = \text{false}$. Now, an implicant is inferred. In this case, this step is simple, as there is only one constraint (which is part of a simple conjunction) in the current dimension that we have documented so far, namely $c_1 : x + 4 < 0$. The subsequent implicant with respect to the false sample is represented by $x + 4 \geq 0$. This polynomial only crosses the axis at $x = -4$, so that the enclosing interval around our partial sample is $[-4, \infty)$. We collect this interval in our set of UNSAT intervals and continue our search outside of

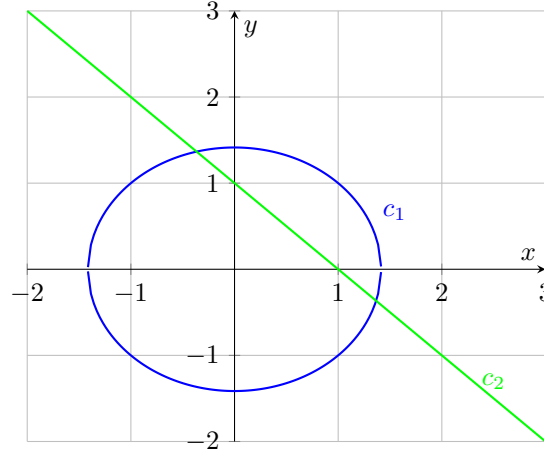


Figure 2.2: Satisfiability problem for $\varphi = c_1 \wedge c_2$ with constraints:
 $c_1 : x^2 + y^2 = 2$
 $c_2 : x + y = 1$

this region, sampling $s_1 = -5$. This time, the matrix is not violated because c_1 still holds under the partial assignment, with the other constraints not yet evaluating to a truth value. Therefore, we recursively step into the next dimension and extend the sample to $s = (-5, 0)$. The matrix evaluates to `false` because c_2 is violated. The only root of $c_2[(-5)]$ is 1.5 and we collect the UNSAT interval $(-\infty, 1.5)$. The next sample $(-5, 3)$ in contrast satisfies the matrix entirely, which means a satisfying solution has been found. Usually, the satisfying intervals would now need to be characterized, as the recursion might stem from a *forall*-segment, which needs characterized SAT cells to cover its real line with SAT intervals. Similarly, in case of a preceding *exists*-segment, a *forall*-segment would need to characterize UNSAT intervals as well. Therefore, in order to still avoid unnecessary characterizations, the implementation passes this information forward through the recursion in the form of Boolean parameters.

2.2.6 Example: Characterization of an UNSAT Interval

To showcase the characterization of an UNSAT interval, we consider a new example in figure 2.2. With both constraints being equality constraints, after sampling 0 for x , the real line for y is covered in UNSAT intervals as the two constraints do not cross on the y -axis. Subsequently, in figure 2.3, the covering is characterized by two discriminants and two resultants between the two covering polynomials. The discriminant and resultant polynomials, as a characterization, are now needed to be sign-invariant in order to infer an UNSAT region. We take the interval around our current sample $s = (0)$, as marked in red and exclude it from the search space.

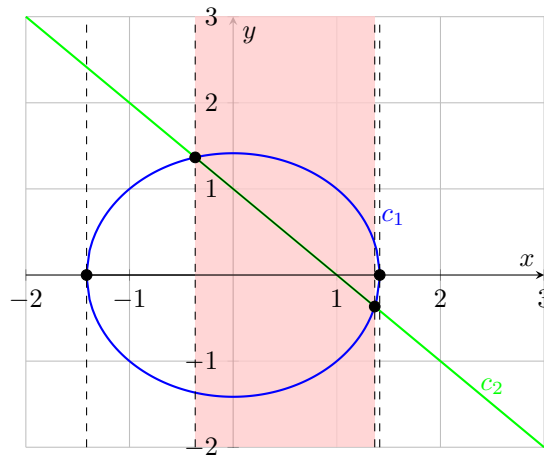


Figure 2.3: Subdivision of x -axis into delineable regions for the constraints in the covering. The leftmost and rightmost cuts are characterized by discriminants, the two inner cuts by resultants between the two polynomials. The red area represents the concluded UNSAT interval for x .

Chapter 3

Optimization

3.1 Nonlinear Optimization

The previously described cylindrical algebraic covering algorithms so far have been applicable for satisfiability and quantifier elimination. This paper aims to transfer the algorithm from the domain of satisfiability checking nonlinear real arithmetic formulae to nonlinear optimization, while maintaining the advantages of the covering algorithm over other alternatives.

Definition 3.1.1 (Single-Objective Optimization Problem [IYA12]). Given a set C of k polynomial constraints $c_i(x)$ over a vector of variables $x = (x_1, \dots, x_n) \in \mathbb{R}^n$ and a real-valued function $f(x)$, find an assignment which satisfies all constraints from C and minimizes f :

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{s.t.} & c_1(x), \dots, c_k(x) \end{array}$$

We call $f(x)$ the *objective function* and $x_1, \dots, x_n$ *decision variables*. As a maximization problem is equivalent to minimization of the negated objective function, this definition can easily be generalized.

This optimization problem now needs to be adapted in such a way that the objective function commands the search of the optimization algorithm, tying the assignments of variables to that of a chosen objective value. Therefore, a symbolic variable t is introduced:

$$\begin{array}{ll} \text{minimize} & t \\ \text{s.t.} & (t = f(x)) \wedge \varphi(x) \end{array}$$

Here, 2.2.4 allows us to generalize and replace the conjunction of constraints $c_i(x)$ with an arbitrary Boolean combination $\varphi(x)$ over polynomial constraints.

3.2 CAIC for Optimization

Similarly to the simple formulation of the optimization problem, the CAIC algorithm only underlies sporadic changes for basic optimization. This is especially noticeable when it comes to purely obtaining an optimal target value, where the recursive elements can even remain unchanged.

3.2.1 Parsing and Restrictions

The use of universally quantified variables within the target function conflicts the aim of finding an optimal solution, since quantified variables do not occur outside of the scope of the quantifier. Adding a quantified variable into the target function is thereby impossible, as the target function lies outside all these scopes. Therefore, while the following implementation is based on the theory-first approach for quantified formulae in [KN22], we restrict ourselves to optimization problems where the variables appearing in the target function are not quantified, that is, to be treated in the same manner as the existentially quantified variables. We denote the absence of a quantifier by listing a *free* quantifier $\emptyset$ and also assume a pre-processing step that arranges all free variables at the start of the variable ordering. The implementation still allows for input formulae with quantified variables that do not appear in the target formula.

3.2.2 Obtaining the Optimal Target Value

After adding the equality constraint $t = f(x)$ to the formula as in 3.1, the target variable remains to be integrated into the variable ordering. We put t in front, so that we can fix the target value at a given iteration and then deploy the remaining recursion, deciding whether the target value can be reached under the constraints or not.

Algorithm 8: `find_optimum( $\epsilon, u\_val$ )`

Data: Global prefix $Q_1x_1, \dots, Q_nx_n$ ($Q_i \in \{\exists, \forall, \emptyset\}$, $1 \leq i \leq n$, $\emptyset$ denoting the absence of a quantifier. Unquantified variables precede all others.) and global matrix $\overline{\varphi}$, maximal padding value $\epsilon > 0$, sufficiently small value $u_val \in \mathbb{R}$ representing $-\infty$.

Result: (UNSAT, *null*, *null*) or (SAT, *opt*, *ass*) with optimal value *opt* $\in \mathbb{R}$ and assignment *ass* = (*ass*₀, ..., *ass*_{*m*}) with *ass*_{*j*} $\in \mathbb{R}$ for $1 \leq j \leq m$.

Declare I_{sat} as an empty interval;

$\mathbb{I}_{exclude} := \emptyset$;

while $\bigcup_{I \in \mathbb{I}_{exclude} \cup \{I_{sat}\}} I \neq \mathbb{R}$ **do**

$s := \text{sample_outside_and_below}(\mathbb{I}_{exclude}, I_{sat})$;

$(f, I) := \text{free}(s)$; // Algorithm 10

if $f = \text{SAT}$ **then**

$I_{sat} := I$;

 Let $I = (\ell, u, I_L, I_U, I_{P_i}, I_{\perp})$;

$\mathbb{I}_{exclude} := \mathbb{I}_{exclude} \cup (\ell, \infty, I_L, I_U, I_{P_i}, I_{\perp})$;

else

$\mathbb{I}_{exclude} := \mathbb{I}_{exclude} \cup \{I\}$;

end

end

if I_{sat} is an empty interval **then**

return (UNSAT, *null*);

else

$val_{opt} := \text{get_optimum_from_interval}(I_{sat}, \epsilon)$; // Algorithm 9

return (SAT, $\text{find_solution}(val_{opt})$); // Algorithm 11

end

Algorithm 8 therefore collects intervals over the real line with respect to the optimization variable. These intervals are generalizations from the deployed recur-

sions. We store the most recently found SAT interval as I_{sat} . The helper method `sample_outside_and_below( $\mathbb{I}_{exclude}, I_{sat}$ )` is designed to sample a value outside of all intervals in $\mathbb{I}_{exclude}$ and below I_{sat} , guaranteeing that every interval we find and refresh I_{sat} with is in fact superior to the previously stored one, if there exists one. To prevent unnecessary covering of target values that no longer need to be considered due to sub-optimality, we add an interval ranging from the left border of the optimal interval up to infinity into $\mathbb{I}_{exclude}$. Still, the UNSAT intervals also need to be collected in $\mathbb{I}_{exclude}$ to ensure that the real line is covered (at least to the left of the lowest SAT interval) and there is no smaller potential target value. Now, in each step, we choose a sample, call the base algorithm with that sample and store the resulting enclosing interval, until the real line is covered in intervals. The method `free( $s$ )` represents a call of the base algorithm subject to some adjustments in order to infer a solution later on and will be discussed in 3.2.3. To prevent the search for a larger target value than the current smallest solution, we additionally exclude every value above a found SAT-interval from the search space. We achieve this by modifying a found SAT-interval to range all the way up to infinity and including it in $\mathbb{I}_{exclude}$. If in the end there are no SAT intervals, UNSAT is returned, meaning that regardless of the target function, the original formula is unsatisfiable. Otherwise, the optimal target value will be the smallest possible value from the leftmost interval.

Algorithm 9: `get_optimum_from_interval( $I_{opt}, \epsilon, u\_val$ )`

Input: (I_{opt}, ϵ, u_val) with interval I_{opt} , maximal padding value $\epsilon > 0, \epsilon \in \mathbb{R}$ and sufficiently small value $u_val \in \mathbb{R}$ representing $-\infty$

Result: Optimal target value $\ell \in \mathbb{R}$

Let $I_{opt} = (\ell, u, I_L, I_U, I_{P_i}, I_\perp)$;

if I_{opt} is unbounded to the left **then**

if I_{opt} is bounded to the right by u **then**

return $u - |u_val|$;

end

return u_val ;

end

if I_{opt} has strict left border **then**

$\ell := \min(\ell + \epsilon, \text{sample_inside}(I_{opt}))$;

end

return ℓ ;

However, in many cases, the leftmost interval border cannot simply be chosen. Only in case of a closed interval, the true optimal value will be the exact left border ℓ . For an interval that is strict to the left, solving the constraints with this target value will be infeasible, as we thereby leave the SAT cell and enter an UNSAT cell, and we need to choose $\ell + \epsilon$ with some sufficiently small $\epsilon > 0$ instead. Algorithm 9 therefore takes such an input ϵ , which acts as an upper limit as to how inaccurate the optimal value is allowed to be. To ensure that a feasible target value is chosen nonetheless, we choose a sample from the optimal interval in case it is smaller than $\ell + \epsilon$, which is achieved via a helper method `sample_inside( $I_{opt}$ )`. For intervals that are unbounded towards the left, we use a sufficiently small value u_val that represents $-\infty$. If the interval is unbounded to the right as well, all target values will be feasible and we simply return u_val . Otherwise, $u - u_val$ is chosen to make sure to remain within the current SAT interval. While the exact choice of how to treat these cases can vary, this implementation shows how to avoid issues that lead to errors.

3.2.3 Obtaining a Full Solution

We now have obtained an approximate optimal value for the target function. The most simplistic approach to finding a full solution, i.e. an assignment of all variables that yields this optimal target value, requires an entirely new execution of the CAIC algorithm for satisfiability, inserting the optimal value for the target value and treating the resulting formula as a satisfiability problem. However, depending on the input, this approach could considerably increase runtime.

Algorithm 10: `free(s)`

Data: Global prefix $Q_1x_0...Q_nx_n$ ($Q \in \{\exists, \forall, \emptyset\}$, $\emptyset$ denoting the absence of a quantifier. Unquantified variables precede all others.), matrix $\bar{\varphi}$, global (mutable) mapping `variable_map` from variables to algebraic intervals (we consider x_0 our already sampled target variable)

Input: Sample point $s = (s_1, \dots, s_{i-1}) \in \mathbb{R}^{i-1}$ such that $\bar{\varphi}[s] \neq false$

Result: (SAT, I) or $(UNSAT, I)$ where $s \times I$ can or can not be extended to a model for any $s_i \in I$. In both cases, the algebraic information attached to I describes how s can be generalized

```

if  $Q_i \neq \emptyset$  then
   $(f, I) := \text{recurse}(s);$                                      // [KN22], Alg.2
  return  $(f, I);$ 
end
 $\mathbb{I}_{unsat} := \emptyset;$ 
while  $\bigcup_{I \in \mathbb{I}_{unsat}} I \neq \mathbb{R}$  do
   $s_i := \text{sample\_outside}(\mathbb{I}_{unsat});$ 
  if  $\bar{\varphi}[s \times s_i] = false$  then
     $(f, I_i) := (UNSAT, \text{get\_enclosing\_interval}(s, s_i));$  // [KN22], Alg.5
  else if  $\bar{\varphi}[s \times s_i] = true$  then
    Clear variable_map;
     $(f, I_i) := (SAT, \text{get\_enclosing\_interval}(s, s_i));$  // [KN22], Alg.5
  else
     $(f, I_i) := \text{free}(s \times s_i);$  // Algorithm 10, recursive call
  end
  if  $f = SAT$  then
    variable_map $[x_i] := I_i;$ 
     $R := \text{characterize\_interval}(s, I_i);$  // [KN22], Alg.6
     $I_{i-1} := \text{interval\_from\_characterization}((s_1, \dots, s_{i-2}), s_{i-1}, R);$ 
    // [KÁED21], Alg.5
    return  $(SAT, I_{i-1});$ 
  else if  $f = UNSAT$  then
     $\mathbb{I}_{unsat} := \mathbb{I}_{unsat} \cup \{I_i\};$ 
  end
end
 $R := \text{characterize\_covering}(s, \mathbb{I}_{unsat});$  // [KN22], Alg.7
 $I := \text{interval\_from\_characterization}((s_1, \dots, s_{i-2}), s_{i-1}, R);$ 
// [KÁED21], Alg.5
return  $(UNSAT, I);$ 

```

Instead, we take use of our bookkeeping structure, the algebraic intervals. Algorithm 10 therefore represents an altered version of algorithm 3, the subroutine called for existentially quantified variables. Algorithm 10 instead focuses on the free variables, which we had assumed to precede all quantified variables in our variable ordering. These are the variables whose assignments make up the final solution. Fortunately, the already existing implementation of the algebraic intervals provides sufficient context to infer values for variables that correspond to any partial sample within the current cell. This is ensured by the polynomials in I_L and I_U , whose roots generalize the borders of the SAT-interval in the current dimension to be dependent on whatever samples were made in the previous dimensions. Once a satisfying sample has been found, the cell is generalized for each dimension downwards one after another. These intervals correspond exactly to the satisfiability-invariant cell that contains the sample. We save the enclosing interval for each variable in a mapping until the bottom level is reached, which is that of the optimization variable. If this cell later becomes the minimal one, meaning we do not find any satisfying cell afterwards which would imply a smaller optimum, we will infer a solution from this algebraic information. If we do find a better cell, the interval mapping will be overwritten. While the overwriting itself is practical, we still need to clear the mapping beforehand. For the case that we did not, suppose we have a mapping for the free variables in our ordering $x_1 \prec x_2 \prec x_3$. In a different iteration, we now find a solution after only sampling x_1 and x_2 . Then, the information stored for x_3 would not correspond to the current cell and lead to undefined results later on.

Algorithm 11: `find_solution(res, valopt)`

Data: Global (mutable) mapping variable `_map` from variables to a algebraic intervals (we consider x_0 our already sampled target variable)

Input: Optimal target value $val_{opt} \in \mathbb{R}$

Result: Optimal Model vector $ass \in \mathbb{R}^{n+1}$

$ass_0 := val_{opt};$

for $i = 1, \dots, n$ **do**

$I := \text{variable_map}[x_i];$

$ass_i := \text{get_value_from_interval}(I, ass);$ // Algorithm 12

end

return $(ass_0, \dots, ass_n);$

Algorithm 11 now takes the interval mapping and uses the value of the current dimension, starting with the target value, to infer the next one from the correspondingly mapped interval with algorithm 12. We evaluate the root of a polynomial from I_L or I_U with the existing partial assignment, which is always possible since these polynomials had previously vanished for a sample at the same dimension, meaning their main variable cannot be of higher dimension than the current. A challenge algorithm 12 faces is that, just like the optimal interval, the intervals might not be simple bounded intervals with non-strict boundaries. If a bound to either side does not exist, neither do characterizing roots. If a bound is strict, we once again cannot choose the exact root of a polynomial since it also lies just outside the actual interval.

One change that becomes noticeable in algorithm 12, where polynomials are evaluated, is that the implementation uses a slightly altered definition of algebraic intervals. Polynomials in I_L , I_U are no longer just stored as polynomials, but as *indexed roots*. An indexed root contains not only the polynomial itself (or a reference to it), but also the index of the root which marks the border of our cell. We denote

an indexed root as a tuple (p, i) , with p the polynomial and i the chosen index of the root of the polynomial, representing the i -th smallest root. Without this index, we would have to take every root of a given polynomial into account, insert the existing assignments, and then take the closest root in the desired direction. Due to delineability ensuring the order invariance of the roots, this would be the exact same root, meaning we would not infer a different cell. Still, we would waste computations by fully evaluating and comparing all roots.

Algorithm 12: `get_value_from_interval( $I, ass$ )`

Data: Maximal padding value $\epsilon > 0$

Input: Algebraic interval I , previous assignments $ass \in \mathbb{R}^i$

Result: Value $v \in \mathbb{R}$

Let $I = (\ell, u, I_L, I_U, I_{P_i}, I_\perp)$;

if I is bounded to the left **then**

 Pick (p_ℓ, i) as first indexed root from I_L ;

 Substitute variables in p_ℓ with assignments from ass ;

$lower_val := i$ -th root of p_ℓ ;

if I has strict left border **then**

if I is bounded to the right **then**

 Pick (p_u, j) as first indexed root from I_U ;

 Substitute variables in p_u with assignments from ass ;

$upper_val := j$ -th root of p_u ;

$sampler_val := \text{sample_between}(lower_val, upper_val)$;

return $sampler_val$;

else

return $\lfloor lower_val \rfloor + 1$;

end

else

return $lower_val$

end

else if I is bounded to the right **then**

 Pick (p_u, i) as first indexed root from I_U ;

 Substitute variables in p_u with assignments from ass ;

$upper_val := i$ -th root of p_u ;

if I has strict right border **then**

return $\lceil upper_val \rceil - 1$

else

return $upper_val$

end

else

return 0;

end

The case-distinction in algorithm 12 can be schematically represented as follows:

1. $(a, b), (a, b]$: We pick the root of a polynomial characterizing the left border and add ϵ in order to avoid picking a value outside of the interval. We pick a root of the right border to pick a value within the interval via sampling in between these borders, but excluding the borders to remain within the interval.
2. (a, ∞) : We pick the root of a polynomial characterizing the left border and

return the next higher integer in order to avoid picking a value outside of the interval.

3. $[a, b]$, $[a, b)$, $[a, \infty)$: We pick the root of a polynomial characterizing the left border, as it will lie within the interval.
4. $(-\infty, b)$: Analogous to 2.
5. $(-\infty, b]$: Analogous to 3.
6. $(-\infty, \infty)$: The feasibility of the formula subject to optimality is entirely independent of the current variable, thus we arbitrarily pick 0.

Our implementation always picks the first element from I_L or I_U . In reality, any polynomial from either set could be picked. For a previous sample, all polynomials in I_L for instance had a root at the exact lower border of the interval. In algorithm 6, the resultants of all polynomials that vanish below or exactly at the lower interval border (with respect to the current sample) were added into the characterization. This also includes the resultants between the polynomials in I_L . In consequence, since we remain within the cell, not changing the signs of the resultants (among the other characterization polynomials), their roots remain order-invariant. Since their roots coincided before, they will again even for a different sample within the cell. This means that whichever polynomial we choose from I_L , it will always yield the same interval border for partial samples from within the cell and we can thus pick any singular polynomial, and the same holds for I_U .

To exactly describe the properties of a cell, we also extend the algebraic interval to contain information about whether a border is strict, unbounded or neither. We therefore extend the tuple representation of the algebraic interval by a tuple $B = (lower_strict, upper_strict, lower_unbounded, upper_unbounded)$ of boolean values to $I = (\ell, u, I_L, I_U, I_{P_i}, I_{\perp}, B)$

3.3 Example

Figure 3.1 shows the previous example from 2.1 extended by a simple target function $f = x + y$, which we aim to minimize. The target function is parsed as an additional equality constraint, and the variable ordering $f \prec x \prec y$ is declared. We subdivide the optimization process into a *solving phase* and a *substitution phase*, as visualized in 3.2.

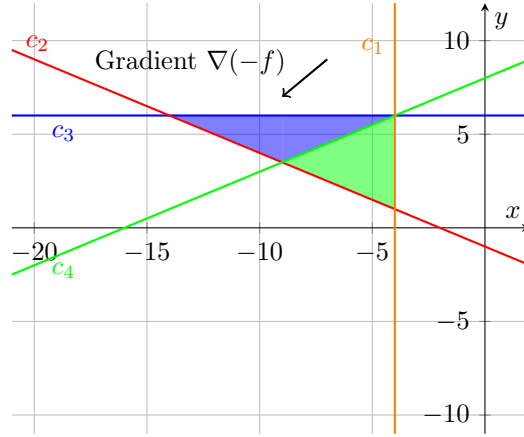


Figure 3.1: Feasible Region for $\varphi = c_1 \wedge c_2 \wedge (c_3 \vee c_4)$ with constraints:
 $c_1 : 4 + x < 0$, $c_2 : -x - 2y - 2 < 0$, $c_3 : y - 6 < 0$, $c_4 : -0.5x + y - 8 < 0$, Optimization
 Target: $\min f = x + y$

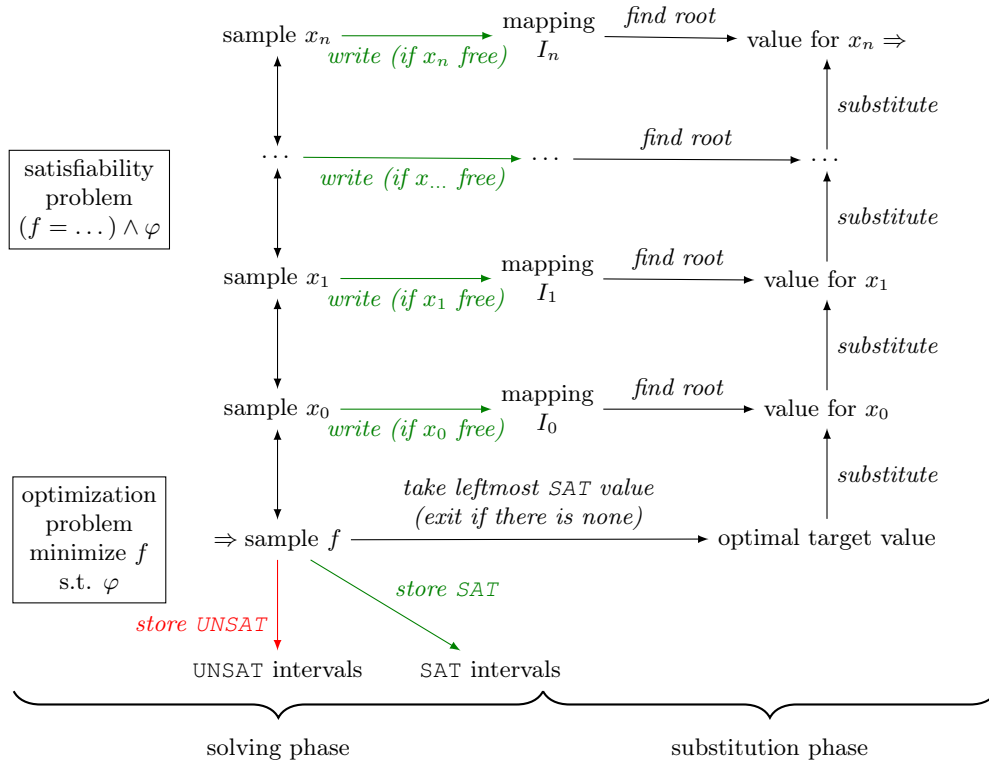


Figure 3.2: The optimization process. A green arrow indicates that the current sample is satisfying, a red arrow indicates that it is unsatisfying. The process starts on the bottom left at "sample f " and exits at the top right "value for x_n ", if not before due to unsatisfiability.

3.3.1 Solving Phase

$\text{free}(s = ())$: We begin our routine with an empty sample, aiming to cover the real line all the way up to a feasible target value by sampling values for f .

$\text{free}(s = (0))$: After sampling ($f \mapsto 0$), satisfiability is still unknown, so we sample x .

$\text{free}(s = (0, 0))$: The assignment $x \mapsto 0$ violates $c_1 : 4 + x < 0$. We find the implicant $4 + x \geq 0$ and exclude $[-4, \infty)$ from the search space

$\text{free}(s = (0, -5))$: After sampling ($x \mapsto -5$), satisfiability is still unknown, so we sample y .

$\text{free}(s = (0, -5, 0))$: The sample yields UNSAT, we generalize to an UNSAT interval $(-\infty, 1.5)$ with respect to partial sample $s = (0, -5)$

$\text{free}(s = (0, -5, 3))$: The sample yields UNSAT, we generalize to an UNSAT interval $(1.5, 5)$ with respect to partial sample $s = (0, -5)$

$\text{free}(s = (0, -5, 6))$: The sample yields UNSAT, we generalize to an UNSAT interval $(6, \infty)$ with respect to partial sample $s = (0, -5)$

$\text{free}(s = (0, -5, -5))$: After a few UNSAT samples for y we find a satisfying one. We clear the variable mapping and map y to the cell containing -5, determined by

```
get_enclosing_interval(s = (0, -5, -5)) :
    y ↦ Iy = (ℓ, u, IL, IU, IP2, I⊥, B)
    with ℓ = 5, u = 5, IL = {(y + x - f, 1)},
        IU = {(y + x - f, 1)},
        B = (false, false, false, false)
```

$\text{free}(s = (0, -5))$: We map x to the cell containing -5, determined by

```
characterize_interval(Iy) :
    x ↦ Ix = (ℓ, u, IL, IU, IP1, I⊥, B)
    with ℓ = -6, u = -4, IL = {(x - f + 6, 1)},
        IU = {(x + 4, 1)},
        B = (true, true, false, false)
```

$\text{free}(s = (0))$: We map f to the cell containing 0, determined by

```
characterize_interval(Ix) :
    f ↦ If = (ℓ, u, IL, IU, IP0, I⊥, B)
    with ℓ = -3, u = 2, IL = {(f + 3, 1)},
        IU = {(f - 2, 1)},
        B = (true, true, false, false)
```

I_f , as seen in figure 3.3, is our first SAT-interval for f , reducing our search space only to target values below the interval. After repeating the sampling and solving step multiple times, we obtain a partial covering of the real line for f , expanding

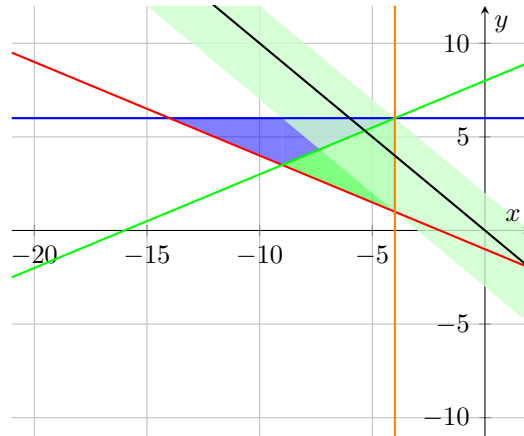


Figure 3.3: The optimization problem after the first iteration of the base algorithm: On the black line lie solutions that lead to the initial sampled target value $f \mapsto 0$. The green area shows a generalization of this sample within which for each sample of f a satisfying solution can certainly be found.

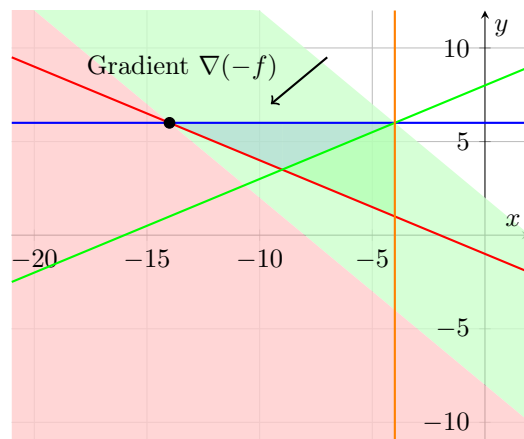


Figure 3.4: The final partial covering of the search space for f . The green area marks that there exists a solution for the corresponding target value, the red area marks the absence of such a solution. The black dot marks an optimal solution.

our satisfying region all the way down to a strict lower border of -8 , which can be seen in 3.4. It now remains to find a full solution, i.e. to infer where exactly on the interval border the entire solution lies. Since we have a strict border, we search for an approximate solution right above the border.

3.3.2 Substitution Phase

The final SAT-interval found in the solving phase overwrites the mapping to the following:

$$\begin{aligned}
 f &\mapsto I_f = (\ell^f, u^f, I_L^f, I_U^f, I_{P_0}^f, I_{\perp}^f, B^f) \text{ with } \ell^f = -8, \quad u^f = -3, \\
 I_L^f &= (\{f + 8\}, 1), \quad I_U^f = (\{f + 3\}, 1), B^f = (\text{true}, \text{true}, \text{false}, \text{false}) \\
 x &\mapsto I_x = (\ell^x, u^x, I_L^x, I_U^x, I_{P_1}^x, I_{\perp}^x, B^x) \text{ with} \\
 I_L^x &= (\{x - f + 6\}, 1), \quad I_U^x = (\{x - 2f - 2\}, 1), B^x = (\text{true}, \text{true}, \text{false}, \text{false}) \\
 y &\mapsto I_y = (\ell^y, u^y, I_L^y, I_U^y, I_{P_2}^y, I_{\perp}^y, B^y) \text{ with} \\
 I_L^y &= (\{y + x - f\}, 1), \quad I_U^y = (\{y + x - f\}, 1), B^y = (\text{false}, \text{false}, \text{false}, \text{false})
 \end{aligned}$$

We pick $\ell^f = -8$ as our optimum, but add $\epsilon := 0.001$, since the left interval border is strict and -7.999 still remains within I_f .

For x , we take the first and only element from I_L^x , substituting f with -7.999 into the first root of $x - f + 6$, resulting in the assignment of $lower_val := -13.999$. Since the lower border is strict and the interval is bounded to the right, we also evaluate the upper border polynomial $x - 2f - 2$, assigning $upper_val := -13.998$ and then sample $sampled_val = -13.9985$ from in between.

For y , as the lower border of I_y is not strict, we simply evaluate the first root of $y + x - f$ under the current assignments, yielding 5.9995 .

The final approximate optimal assignment is thus:

$$f \mapsto -7.999, x \mapsto -13.9985, y \mapsto 5.9995$$

Chapter 4

Implementation and Benchmarking

4.1 Implementation in SMT-RAT

We implement our algorithm in the Satisfiability-Modulo-Theories Real Arithmetic Toolbox (SMT-RAT), introduced in [CKJ⁺15]. SMT-RAT is a C++-library divided into multiple modules representing different solving procedures, one of them being the `CoveringNG` module, which is the current implementation of the presented algorithm from [KN22] and the one this thesis extends. In SMT-RAT, the SMT-solving process can be composed with strategies, combining multiple modules. A module is called with parameters from another module as a *backend*. In many cases for instance, a propositional SAT-solving component precedes an actual theory-search procedure, as is the case for the previously implemented `NewCovering` module.

The `CoveringNG` module can stand for itself, as there no longer is a preceding SAT-solver necessary. We only add a backend in 4.2.2, where we require the satisfiability-check of certain formulae.

4.2 Benchmark Procedure

To test our implementation, we use the `QF_NRA` benchmark set from SMT-LIB [BFT16]. It contains various folders of satisfiability problems all within our domain of quantifier-free nonlinear real arithmetic. We run our computations on an Intel Xeon 8468 processor, assigning a maximum of 4GB RAM as well as one minute of processing time for each file.

There are two major considerations for benchmarking the optimization algorithm. Firstly, due to a lack of existing optimization benchmarks, we need to convert the `QF_NRA` files into optimization benchmarks. Secondly, we want to develop an approach to check not just correctness of our results, but also optimality.

4.2.1 Creating Optimization Problems from existing Satisfiability Problems

Our benchmark files are of the SMT-LIB2-format. In a header, the logic is declared, which in our case is always `QF_NRA`, for quantifier-free nonlinear real arithmetic. Then in the body of the file, real variables are declared with `(declare-fun x () Real)` or `(declare-const x () Real)`, where `x` is a unique variable name. Constraints are denoted by `(assert ( $\sigma$   $f_1$   $f_2$ ))`, with σ a comparator, where f_1 and f_2 are terms written in polish notation, just like the surrounding comparison itself. We systematically adjust the body by extracting the variables and adding a function over the variables with either `(minimize  $f$ )` or `(maximize  $f$ )`, with f again being a term in polish notation. In the case of maximization, SMT-RAT will automatically convert the problem into an equivalent minimization problem before our algorithm starts. Finally, we extend the footer, consisting of `(check-sat)` and `(exit)`, with the commands `(get-objectives)` and `(get-model)` in between. In a preliminary run, we check the runtime of the satisfiability benchmarks to exclude complex cases that already overshoot our runtime or memory limits even without any additional objective function. This reduces the number of files from 12154 to 9633. We thereby obtain a multitude of different benchmark sets, each consisting of 9633 files, where each benchmark set differs in the kind of objective function inserted into each file.

4.2.2 Checking Solutions for Optimality

For the purpose of confirming that we fulfilled the task of optimality with a given solution, we need to append a check to the end of our implementation. The implementation should already be correct by construction. Nonetheless, we use this method to assert the correctness of the implementation. As this check will be of significant computational effort, we double the maximal processing time.

Our optimality check relies on an existing module for satisfiability-checking in SMT-RAT, which we use as a backend after our algorithm is finished. We choose the existing implementation of the CAIC algorithm for satisfiability problems, the reason being that we not only want to allow quantifiers for our input, but will additionally take use of them for our optimality check. Other implementations often lack the treatment of quantifiers. In a first simple check of an optimal solution, we always confirm that a found optimum opt is valid by parsing the formula φ in conjunction with the constraint that the optimization variable is assigned opt , i.e. we check for satisfiability:

$$\varphi \wedge (f = opt)$$

Once we have asserted that the backend returns *SAT* on this input, it is to be checked if there exists a smaller target value that still yields a satisfying solution. The idea is to constrain the formula to yield only smaller target values and then prove it to be UNSAT. Assuming the leftmost found SAT-interval has a non-strict left border, we can simply parse:

$$\varphi \wedge (f < opt)$$

For a strict left border, yielding an optimum of $opt = \ell + \epsilon, \epsilon > 0$, with ℓ being the lower border from the last stored SAT-interval. we instead assert to be UNSAT:

$$\varphi \wedge (f \leq \ell)$$

In this case, we also formulate the addition of such an ϵ in a third backend check, in which we assert the following formula to be satisfiable:

$$\forall x \exists y ((x > 0) \rightarrow (\varphi \wedge f = \ell + y \wedge y < x))$$

Our example from 3.3 for instance underlies the following checks: After inferring $\ell = -8$, $opt = -7.999$, we first verify the feasibility of such an optimal target value by SAT-checking $\varphi \wedge (f = -7.999)$. After receiving *SAT* from this backend, the second backend run checks $\varphi \wedge (f \leq -8)$ for a smaller target value and returns UNSAT as expected, confirming that there is no target value feasible that is smaller than -8 or exactly -8 . The third backend run, asserting that $\forall x \exists y ((x > 0) \rightarrow (\varphi \wedge f = -8 + y \wedge y < x))$ is satisfiable, now ensures that every optimal target value that is close enough to -8 , but still slightly larger than -8 , is feasible.

4.2.3 Treating Interval Representations

The lower border of our final SAT-interval in SMT-RAT is represented by the data type *Real Algebraic Number* (RAN). In the best case, this RAN is *numeric*, i.e. a rational number that is simply represented by a numerator and a denominator. In many cases however, since the interval borders are derived from roots of polynomials, a RAN must represent an irrational number. Since rounding up or down to a valid rational representation would introduce inaccuracy, the most appropriate way to represent such numbers is to save the polynomial $p(x)$ itself and additionally two rational numbers a and b that closely surround this root. a and b serve the purpose of specifying which one of the roots of the polynomial is given. In this case, we need to encode this *non-numeric* representation into our formula with additional constraints. We therefore introduce a variable ℓ which we want to represent the interval border. We fix it to be the exact root by adding the constraint $p(\ell) = 0$ via conjunction. a and b are incorporated by a conjunction of two constraints $\ell > a$ and $\ell < b$. Now we can treat ℓ as the exact border value and, depending on what we are checking, constrain the target variable to be equal to, or less than ℓ . The resulting additional formula is of the form

$$p(\ell) = 0 \wedge \ell > a \wedge \ell < b \wedge t \sigma \ell$$

with $\sigma \in \{<, \leq, =\}$.

4.3 Benchmark Results

We now pick different types of target functions to compare the effects of different target functions on runtime and memory usage. For each of the given benchmark files with variables $x_1, \dots, x_n$, we add four objectives. As a baseline, we minimize just a single variable with $\min x_1$. We then minimize a sum of multiple variables to assess performance differences between *single-dimensional* target functions:

$$\min x_1$$

and *multi-dimensional* target functions:

$$\min(x_1 + x_2 + x_3 + x_4 + x_5)$$

In the next step, we observe performance on both a *quadratic* objective:

$$\min(x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2)$$

and a *cubic* objective:

$$\min(x_1^3 + x_2^3 + x_3^3 + x_4^3 + x_5^3)$$

For the quadratic function, we additionally investigate a benchmark set with a *maximization* objective:

$$\max(x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2)$$

Lastly, we pick a *non-convex* objective:

$$\min(x_1^3 - x_1^2 + x_2^3 - x_2^2 + x_3^3 - x_3^2 + x_4^3 - x_4^2 + x_5^3 - x_5^2)$$

which introduces the additional challenge of local minima/maxima. We always pick the same variables from each instance, namely the first five variables that are declared. We only include results that did not exceed our memory limit and time limit, which we increased to two minutes accounting for the routine to check for optimality of each result. We evaluate a separate timer for the optimization itself, and sort out results for which it exceeds the original timeout of one minute, so that the optimization algorithm itself is allotted exactly as much runtime as the satisfiability algorithm which did not underlie any additional checks and was thus evaluated under the previous time constraint of one minute. We find that none of our benchmark runs produce any segmentation faults that are not correlated with the exceeding of the memory limit. It follows that all of our optimality checks indicate that the found target value is in fact optimal, since otherwise a failed assertion would provoke a segmentation fault. Figure 4.1 shows the ordered runtimes of the optimization algorithm on each of the four variants of the 9633 input files.

Notably, the single-variable objective optimization behaves quite similarly to the satisfiability variant, with performance only deteriorating slightly and for few inputs. With an increased number of variables involved however, like in the sum-objective, the complexity of the problem seems to drastically increase for a larger subset of problems. Naturally, since the effect of the objective depends on which of the variables are arbitrarily included in the objective, some inputs are barely affected by the addition of their target function, while others soar in complexity.

The squared variant of our benchmark set, despite seemingly introducing a more complex target function, performs slightly better than the sum variant. A possible reason for this is the fact that a sum of squared variables restricts the actual search space for the target variable to $[0, \infty)$. While the sampling algorithm disregards this circumstance, the algorithm is quick to reason that $(-\infty, 0)$ is to be excluded from the real line.

For the maximization variant, this advantage ceases to exist, as the optimum, depending on the constraints, can range from $-\infty$ to ∞ again. The resulting performance is now slightly worse than that of the simple sum objective.

For the cubic variant, the weight of the higher complexity of the target function and consequently that of the characterizations becomes noticeable with less than 70% of input files solved, as seen in table 4.1.

The non-convex variant of our benchmark introduces another significant decrease in performance for a multitude of input files. With the introduction of a large number

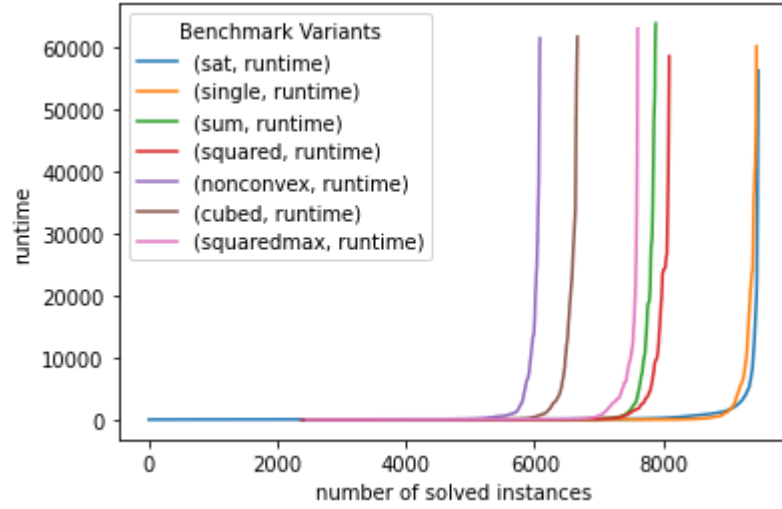


Figure 4.1: Ordered runtimes [ms] of the optimization algorithm on the four different transformed benchmark datasets.

	Solved portion	Average solving time [ms]	Average substituting time [μ s]
Single variable	98.55%	833.43	295.04
Sum of variables	82.10%	1151.12	51.57
Squared sum (min)	84.23%	1114.70	25.33
Squared sum (max)	79.01%	830.21	7.14
Cubic sum	69.47%	1492.76	403.39
Non-convex sum	63.21%	1283.55	107.03

Table 4.1: Average times for solving and substitution phase per dataset. For the average solving and substituting time, only elements below the runtime and memory limit are considered.

of local minima, the convex function seems to cause a stark rise in the number of small delineable regions for the target variable. This can be seen in table 4.2, as for the average solved and satisfiable instance from the non-convex benchmark, the algorithm infers over 10 SAT-intervals that are not optimal before finding the optimal one, whereas in the previous sets with minimization objectives, it always took less than half of this number of SAT intervals on average. Figures 4.2 and 4.3, comparing the amount of found SAT intervals per instance, underline this finding: From the single-variable to the cubed benchmark set, a slight tendency for an increased amount of found SAT intervals can be noted. The non-convexity of the final benchmark set however is what causes the most significant increase of all.

The average number of satisfying target intervals mostly correlates with the general performance and runtime of the algorithm. Only for the maximization variant of the squared sum, this appears not be the case. In comparison, the percentage of solved instances for the cubic variant is lower by almost 10 points, yet almost twice as many satisfying intervals at target variable level are inferred for the maximization of

	Average number of satisfying target intervals found among solved SAT instances only
Single variable	1.77
Sum of variables	2.96
Squared sum (min)	2.33
Squared sum (max)	5.39
Cubic sum	2.70
Non-convex sum	11.12

Table 4.2: The average number of sat target intervals. Only results where an optimum could be inferred were considered, i.e. UNSAT solutions were excluded as well as those remaining unsolved.

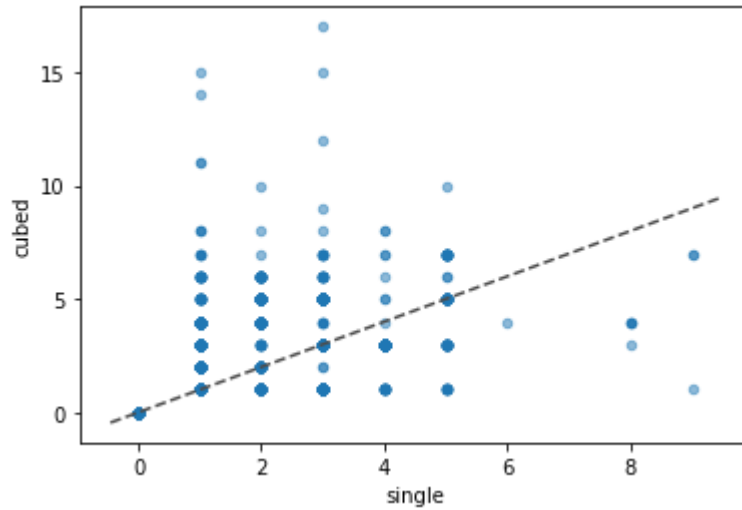


Figure 4.2: Comparison between the number of sat target intervals found for the single variable target instances and cubed sum target instances. A dot on the line marks instances with equal amounts of target level SAT intervals. More opaque dots indicate a larger number of instances located at the given position

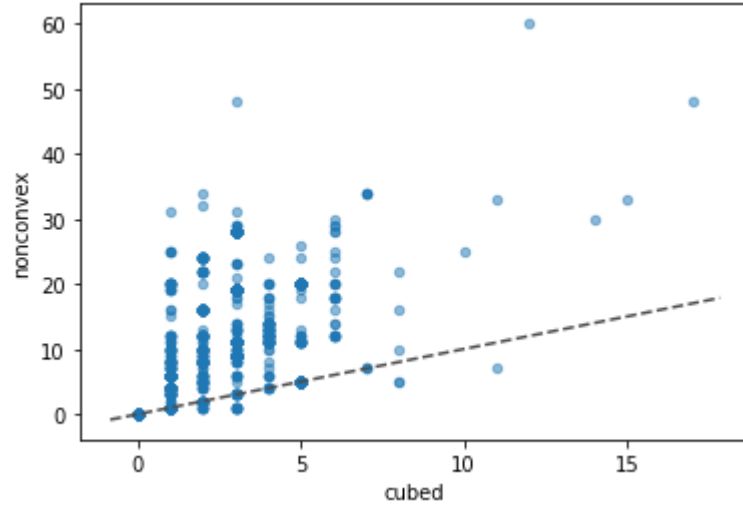


Figure 4.3: Comparison between the number of sat target intervals found for the cubed sum target instances and the non-convex sum target instances. A dot on the line marks instances with equal amounts of target level SAT intervals. More opaque dots indicate a larger number of instances located at the given position.

the squared sum. This observation can be explained by the nature of the sampling algorithm. SMT-RAT parses the objective into a minimization of a sum of negated squares. Resulting target values will thereby be negative, and $(0, \infty)$ can be excluded quickly. This time however, compared to the minimization variant where $(-\infty, 0)$ can be excluded, unbounded and arbitrarily small solutions are possible. The sampling algorithm starts at a target value 0 and tends to sample relatively close to the existing partial covering. Therefore, if a very small solution exists or it is unbounded, the sampling algorithm will likely have found a large portion of the sub-optimal but feasible target value intervals.

In contrast, no clear correlation can be found between runtime and the average number of intervals accumulated over all levels and cells. For instances for which an optimum could be inferred within time and memory constraints, the average number of found intervals remains relatively constant throughout the different benchmark sets, as seen in table 4.3. If all instances are considered, the average number of intervals does vary significantly between benchmark sets, but the trends do not coincide with the cubic sum actually yielding about half as many intervals on average as both squared sum instances each. While it could be concluded that other factors such as the complexity of the average characterization by far outweigh the quantity of intervals to be inferred in terms of runtime and memory usage, it has to be considered that both statistics in table 4.3 underlie a bias. The first column only describes the solved instances, far less of which are accounted for in the non-convex sum. While one could erase all instances from the benchmark that are not successfully solved in at least one instance, such a run still yielded similar results, possibly because the remaining instances will still be those that the relatively complex non-convex target function has only minor effect on. Considering all instances as in the second column, although thereby including found intervals even for unsolved sets, we still obtain a skewed result.

	Average number of intervals found on all levels among solved instances only	Average number of intervals found on all levels among all instances
Single variable	1101.68	1737.55
Sum of variables	1592.60	3010.91
Squared sum (min)	1973.11	3913.91
Squared sum (max)	2071.40	4141.41
Cubic sum	1239.55	2099.49
Non-convex sum	1361.62	2320.30

Table 4.3: The average number of intervals at any level, firstly only among those instances the solver terminated for without running out of time or memory, secondly for all instances.

For a given instance for which the algorithm overshot the time or memory limit, it is unclear just how many intervals the algorithm still would have inferred if there were no limitations.

We introduce a second comparison, in which two benchmark sets can be compared at a time. We only consider benchmark instances for which the algorithm terminated in both runs, and subdivide the instances by whether the result is UNSAT or SAT. While we are forced to eliminate all instances that timed out in at least one of those runs, which unfortunately removes potential outliers with lots of intervals and runtime, we now at least have a fair comparison for benchmark instances of lower complexity. Figure 4.4 shows such a comparison between the single variable benchmark set and the squared (minimization) benchmark set. No clear indication can be found that the number of intervals inferred increases for either of the two variants compared to the single variable variant. This underlines the hypothesis that the complexity is significantly influenced by other factors than just interval quantity, as despite causing the largest number of timeouts, no significant increase in the number of intervals can be observed in this direct comparison.

Comparing the average runtime of the solving phase to that of the substituting phase gives the insight that the runtime of the substitution phase, consisting of mere microseconds, can be disregarded. This is not unexpected, as there are only up to two substitutions per variable to be made with no recursions whatsoever, and thus the complexity remains linear. The solving phase on the other hand strongly relies on the runtime of the recursion. For optimizing runtime, the focus could either lay on the implementation of the base algorithm itself, or optimizing the sampling routine with hopes of calling the base algorithm less overall.

While observing those benchmark instances which caused a timeout, it becomes unmistakably apparent that runtime increases with more complex target functions, but only only for satisfiable instances. As seen in table 4.4, unsatisfiable instances are almost entirely solved for each benchmark set. The number of satisfiable instances solved on the other hand appears to strongly depend on the complexity of the target function, with only a quarter solved for the non-convex sum dataset. It should be noted that only very few benchmark instances exceeded the memory limit rather than the runtime limit, which is why we generally attribute a large runtime as cause to unsolved instances.

This observation can be explained by the process of the algorithm. Once a satisfying

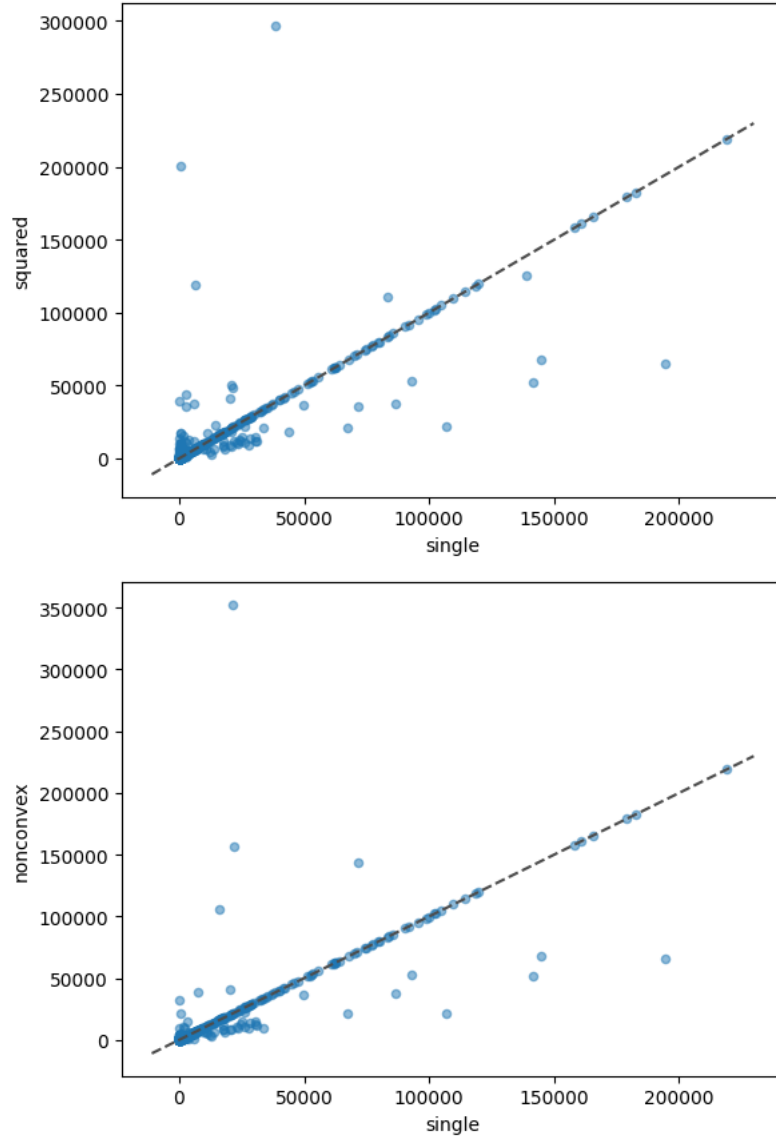


Figure 4.4: Comparing the average number of intervals found in total per instance between the single variable benchmark set and the (minimizing) squared/non-convex benchmark set. Only runs are included for which the instance was solved within time and memory limits for both benchmark sets involved at a time.

solution is found, it is to be characterized. Since we have picked a distinct target value before, once we reach the final variable that is included in the target function, it lies directly on the root of the target function. This correlates with how shifting this variable either way would result in a different target value than the one sampled before. The enclosing interval is therefore defined by a point interval of this root itself, and the potentially complex target function will play into the characterization, ultimately

	Portion of <i>UNSAT</i> instances solved	Portion of <i>SAT</i> instances solved
Single variable	99.80%	87.86%
Sum of variables	97.01%	60.68%
Squared sum (min)	96.77%	64.61%
Squared sum (max)	96.81%	55.03%
Cubic sum	95.60%	37.86%
Non-convex sum	97.01%	25.37%

Table 4.4: Percentages of SAT and UNSAT problem instances solved within the given time and memory restrictions per benchmark set.

increasing runtime. If there is no satisfying solution to begin with however, the target function's polynomial will define fewer enclosing intervals. When a (partial) solution is detected to be false, the implicant is likely to consist of other polynomials and not of the target function's polynomial.

Chapter 5

Conclusion

5.1 Summary

This thesis aimed to transform the Cylindrical Algebraic Coverings algorithm, introduced in [KÁED21], from a satisfiability-checking algorithm to an optimization algorithm. The implementation of this algorithm in SMT-RAT, which this thesis expands, is based upon a quantifier-permitting theory-first approach from [KN22]. Therefore, the recursive structure of the algorithm was taken advantage of, as the optimization target could be transformed to form just a single further recursive layer and an additional constraint. The procedure can then be described as "trial and error", with the optimization task converted into no more than a finite number of satisfiability checks for different target values.

Using a large number of satisfiability checks, the runtime and memory usage of such an algorithm drastically increases compared to the sole satisfiability variant. To assess what further effect the choice of the target function has, multiple different target functions were tested on the same underlying satisfiability problems. It was found that the complexity of the target function significantly impacts the performance of the algorithm. For instance, increasing the number of variables included in the target functions from one to five, but also generally introducing nonlinear and non-convex functions all further decreased performance for satisfying instances by increasing the number of satisfying intervals on the target level and also causing more complex characterizations to be computed. In contrast, for the number of intervals found in total, no clear correlation with certain target functions could be concluded.

Additionally, the underlying quantifier-permitting architecture of SMT-RAT allowed the checking of solutions for optimality, by abstracting the optimality of a solution into numerous formulae, depending on the properties of that solution, and checking satisfiability or unsatisfiability of those formulae.

5.2 Future work

5.2.1 Solutions for Multiple Objectives

This thesis only focuses on problems with single target functions. For multiple target functions, the objective of Pareto-optimality can be treated.

Definition 5.2.1 (Pareto-Optimality [Cen77]). Assume we have a problem consisting of targets $\min f_1(x), \min f_2(x), \dots, \min f_n(x)$ and a constraint formula $\varphi(x)$. We call a point x_0 *Pareto-optimal* for the problem if it satisfies φ and there is no other satisfying solution $x \neq x_0$ with $f_i(x) \leq f_i(x_0)$ for all $1 \leq i \leq n$ and $f_j(x) < f_j(x_0)$ for some $1 \leq j \leq n$. A solution is therefore called Pareto-optimal if there is no other solution that improves at least one target function value without worsening any of the other target function values.

While there are potentially multiple Pareto-optimal solutions, one goal could be to find a Pareto-optimal solution that optimizes targets in a strict order of priority. After normalizing our set of targets to minimization problems, we declare an ordering $f_1(x) \prec f_2(x) \prec \dots \prec f_n(x)$. Similarly to the single-target optimization approach, we now introduce one variable f_i per target function $f_i(x)$. We initialize a formula $\varphi_0(x) := \varphi(x)$, where $\varphi(x)$ represents the original input formula. For each of the target functions, descending through their ordering, we do the following steps: We add the current target variable to the front of the original variable ordering to obtain the ordering $f_i \prec x_1 \prec \dots \prec x_m$, conjunct the constraint $f_i = f_i(x)$ to $\varphi_{i-1}(x)$ to obtain $\varphi_i(x) \wedge f_i = f_i(x)$ and then perform the single-variable optimization algorithm for f_i , but only up to the point in algorithm 8 where the optimum val_opt is inferred. Afterwards, we substitute val_opt for f_i in $\varphi_i(x)$, remove f_i from the variable ordering and repeat for f_{i+1} .

This method optimizes each target variable subject to the optimality of its predecessors, which again is constrained by the optimality of their predecessors, all the way up to f_1 which is even globally optimal. The Pareto-optimality can be proven by inductivity, given the optimization algorithm successfully finds such an optimum. We refer to an assignment as Pareto-optimal if it can be extended to a Pareto-optimal solution.

For f_1 , Pareto-optimality holds because optimality itself holds. Once we have found a Pareto-optimal assignment for variables $f_1, \dots, f_k$, our algorithm infers an assignment for f_{k+1} . Assume that our assignment for $f_1, \dots, f_{k+1}$ is not Pareto-optimal. Since $f_1, \dots, f_{k+1}$ does not violate $\varphi_{k+1}(x)$, and therefore neither $\varphi(x)$, the violation of Pareto-optimality must be due to the existence of some satisfying assignment that reduces at least one f_j , $1 \leq j \leq k+1$, without increasing any other f_i , $1 \leq i \leq k+1$, $i \neq j$. For $j < k+1$, such an assignment cannot exist as $f_1, \dots, f_k$ would then not be Pareto-optimal. For $j = k+1$, since the algorithm ensures that f_{k+1} is minimized given the optimal assignments for $f_1, \dots, f_k$, no alternative assignment can exist that improves f_{k+1} without changing and worsening any of the higher-priority functions.

As shown, the single-objective optimization algorithm can be used to tackle problems with multiple objectives. The natural next step would be to implement this lexicographic approach in practice and evaluate it with benchmarks that include an ordered set of functions according to their priority. Another approach to optimization regarding multiple objectives for which our algorithm could be applied is the aggregation of those objectives into a single composite function. This transformation assumes that the target functions can be weighted relative to each other. Once the trade-off is formulated by the composite function, the basic optimization algorithm can be applied.

5.2.2 Symbolic Evaluation and Direction Vectors

At the end of the solving phase, if the leftmost SAT-interval for the target variable has a strict left bound, our algorithm only yields an approximate solution with an error of up to some pre-selected ϵ . If the interval is unbounded to the left, we even only return and work with a value deemed sufficiently small. In both cases, our solution would be more accurate if we allowed symbolic optima $\ell + \epsilon$, $\ell \in \mathbb{R}$ and $-\infty$ respectively. For an optimum $\ell + \epsilon$, an implementation requires a framework that allows symbolic evaluation of polynomials, such that at each level, a term dependent on a symbol ϵ can be substituted into the border polynomials. For an optimum $-\infty$, the focus would lay on evaluating limits of polynomials and then supplying an additional direction vector that indicates a direction for the assignment which decreases the objective further.

Bibliography

- [BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [BT18] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.
- [Cen77] Yair Censor. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization*, 4(1):41–59, 1977.
- [CKJ⁺15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: an open source c++ toolbox for strategic and parallel smt solving. In *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24–27, 2015, Proceedings 18*, pages 360–368. Springer, 2015.
- [Col98] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 85–121, Vienna, 1998. Springer Vienna.
- [IYA12] Hidenao Iwane, Hitoshi Yanami, and Hirokazu Anai. An effective implementation of a symbolic-numeric cylindrical algebraic decomposition for optimization problems. In *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation*, pages 168–177, 2012.
- [KÁED21] Gereon Kremer, Erika Ábrahám, Matthew England, and James H Davenport. On the implementation of cylindrical algebraic coverings for satisfiability modulo theories solving. In *2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 37–39. IEEE, 2021.
- [KN22] Gereon Kremer and Jasper Nalbach. Cylindrical algebraic coverings for quantifiers. In *Proceedings of the 7th International Workshop on Satisfiability Checking and Symbolic Computation (SC² 2022)*, 2022.
- [Áb22] Erika Ábrahám. Satisfiability checking: The cylindrical algebraic decomposition method II, 2022. Lecture slides, RWTH Aachen University, WS 22/23.