Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

BACHELOR OF SCIENCE THESIS

A LEVEL-WISE VARIANT OF SINGLE CELL CONSTRUCTION IN CYLINDRICAL ALGEBRAIC DECOMPOSITION

Philippe Specht

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Additional Advisor:
Jasper Nalbach M.Sc.

Aachen, September 29, 2020

Acknowledgment

I would first like to thank Professor Erika Ábrahám for teaching me the necessary theoretical background and for advising which I do not take as granted.

Additionally, I would like to thank Jasper Nalbach who advised me in theoretical and practical regard, and who answered my every question in detail.

Furthermore, I would like to thank Professor Christopher W. Brown for revising the presented algorithm and for breaking down complex knowledge for me.

In addition, I would like to thank Professor Jürgen Giesl for examining this thesis.

Last but not least, I would like to thank my family and friends, especially my parents, for supporting me in every way possible.

Abstract

This thesis presents a novel, level-wise approach to single cell construction in Cylindrical Algebraic Decomposition (CAD). It can be used in Satisfiability Modulo Theories solvers for Quantifier Free Non-Linear Real Arithmetic like for example *nlsat* [JdM12]. For one such solver, testing shows that the usage of the level-wise approach over a recursive approach [BK15] decreases the mean solving time by 10.38%.

Given a set of multivariate polynomials with rational coefficients and a point, a single cell construction algorithm returns a cylindrical algebraic cell around the point. In general, this cell is a superset of the cell that would be constructed as a part of a CAD.

The idea of the level-wise approach is to construct the cell top to bottom. This means initializing the cell as $\mathbb{R}^i$ for the appropriate i and then determining the correct bounds one component after the other, starting in the i-th dimension down to the first. Brown-McCallum's projection operator [Bro01][BK15] which is a reduction of McCallum's projection operator [McC98] is used in the process.

Contents

Chapter 1

Introduction

Satisfiability Modulo Theories (SMT) solving deals with the checking of satisfiability of *First Order Logic* (FOL) formulas over specific FOL theories. One such theory for which specific solvers exist is the *Quantifier Free Non-Linear Real Arithmetic* (QFNRA). This theory's literals are essentially equalities and inequalities, each comparing a multivariate polynomial with rational coefficients to zero. The first remotely efficient SMT solver for QFNRA was presented in 1975, creating a *Cylindrical Algebraic Decomposition* (CAD) [Col75]. To check satisfiability for a formula, this technique receives a set containing all polynomials that are in the formula as an input. It then partitions the possible solution space into a finite number of regions over which the sign of every polynomial is constant. Thus, the (in)equalities that the polynomials are embedded in evaluate to either true or false over the entire region. Checking satisfiability is then only a question of checking a sample for each of the regions.

A more recent algorithm to check satisfiability of a QFNRA formula called *nlsat* was published in 2012 [JdM12]. nlsat essentially searches over the entire $\mathbb{R}^n$ for a satisfying theory assignment $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$. Thereby, n is the maximal number of variables in a polynomial in the given formula. This assignment α is constructed dimension by dimension. At some point, the algorithm might run into a situation where there is a partially constructed sample $\alpha^i = (\alpha_1, \ldots, \alpha_i)$ with $1 \leq i \leq n-1$, that cannot be extended because the insertion of α^i into the formula makes it unsatisfiable. In this case, the algorithm wants to find a sign-invariant region around α^i, so that it can be excluded from the search space. One way of doing so is the *single cell construction*. This means constructing a description of the cell of a CAD in which α^i lies or if possible even a superset of this cell.

The main contribution of this thesis is a level-wise single cell construction algorithm. Thereby, the "level" refers to the *level of a polynomial* p which can be seen as an extension of the term of the dimension of a space to polynomials. Given a variable order, the level of p is defined as the index of the greatest variable appearing in p. Generally, a single cell construction algorithm receives a point $\alpha = (\alpha_1, \ldots, \alpha_i) \in \mathbb{R}^i$ and a set containing polynomials of maximally level i. The level-wise algorithm works as follows on this input. First, the cell is initialized as $\mathbb{R}^i$. Then, the cell's bounds are determined level by level, iterating down the variable order and thus the dimensions of the cell. On each level, after determining the bounds, the polynomials on the current level are projected using an optimized version of McCallum's projection operator. One part of this optimization is derived from the projection that is used in

another, *recursive single cell construction* algorithm [BK15].

Additionally, level-wise single cell construction has been implemented in SMT-RAT, an Open Source C++ Toolbox for Strategic and Parallel SMT Solving [smt] [CKJ+15], where recursive single cell construction is already implemented [Neu18]. These implementations are compared in the use-case of an SMT solver for QFNRA that is similar to nlsat. The results indicate that the level-wise algorithm is correct and that it is computationally more efficient than the recursive algorithm.

The remainder of this thesis is structured as follows. In Chapter 2, an intuitive explanation of nlsat is given as a use case for single cell construction. The following Chapter 3, gives some preliminaries to CAD, again in a mostly intuitive manner. In Chapter 4, a primitive but already level-wise approach to single cell construction is presented. An optimization to this approach is then given in Chapter 5. The main part of the thesis is then concluded with the evaluation in Chapter 6. This evaluation is a comparison of two implementations of SMT solvers for QFNRA, one using recursive single cell construction as in [BK15] and one using optimized level-wise single cell construction. Finally, in Chapter 7, the thesis is summarized and possible future work is presented.

Chapter 2

nlsat: A Decision Procedure for Non-Linear Arithmetic

In 2013, Leonardo de Moura and Dejan Jovanović [dMJ13] proposed *Model Constructing Satisfiability Calculus* (*mcSat*) as a novel, abstract decision procedure for checking *Satisfiability Modulo Theories* (SMT) for certain theories. One first order theory for which they specified this abstract procedure is the theory of *Quantifier Free Non-Linear Real Arithmetic* (QFNRA). They called this specified algorithm *nlsat* [JdM12].

As the name suggests, *mcSat* works by constructing a model for a given formula, meaning that it searches for a Boolean assignment to satisfy the Boolean abstraction but also for a theory assignment for each theory variable to satisfy the underlying literals in parallel. If the search runs into a conflicting theory assignment, theory conflict resolution is done with regard to the current, possibly partial assignment. Specified to *nlsat*, this allows for an optimized resolution approach like single cell construction, in contrast to the computation of a whole CAD. Furthermore, resolving theory conflicts is only done for a small set of constraints, a *conflicting core*. Also, theory conflict resolution for QFNRA is usually based on CAD which is very computationally heavy. Therefore, it is computationally the most expensive part of the procedure. Only considering a conflicting core and having a partial assignment leads to a conservative use of CAD which is an advantage of nlsat.

This advantage's effect can be seen in testing. An implementation of nlsat performs consistently better than other (at that point in time) modern SMT solvers for QFNRA [JdM12]. It is thus in our interest to further optimize this procedure. Therefore, after introducing some definitions, we will give an intuitive explanation to nlsat as an application of single cell construction.

2.1 Preliminaries

Since we only give an intuitive insight on nlsat, we will not give nlsat specific definitions, but we will clarify our language used for talking about QFNRA since this will be used throughout this thesis.

Definition 2.1.1 (Level of a polynomial)**.** *Given an variable ordering $x_1 \prec \ldots \prec x_n$, the level of a polynomial $p \in R[x_1, \ldots, x_n]$ for a ring R is the greatest i such that*

$deg_{x_i}(p) > 0$ if p is non-constant or 0 if p is constant.

Throughout this thesis, we will always use variables $x_1, \ldots, x_n$ in polynomials and the variable ordering $x_1 \prec \ldots \prec x_n$ for some $n \in \mathbb{N}$.

Definition 2.1.2 (Constraint)**.** *A constraint F over $R[x_1, \ldots, x_n]$ is of the form $p \nabla 0$ where $p \in R[x_1, \ldots, x_n]$ for a ring $R \in \{\mathbb{Z}, \mathbb{Q}\}$ and $\nabla \in \{=, \neq, \geq, \leq, >, <\}$. Also, the level of a constraint $p \nabla 0$ is the level of p.*

Note that the multiplication of a constraint over $\mathbb{Q}[x_1, \ldots, x_n]$ with the product of the denominators of all its coefficients results in an equivalent constraint over $\mathbb{Z}[x_1, \ldots, x_n]$. Throughout this thesis, we will therefor consider polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$ in the input of procedures, even though polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$ could be used equivalently.

Definition 2.1.3 (Formula, Clause)**.** *A (QFNRA) formula $\mathcal{F}$ in CNF is of the form $\bigwedge_{i \in N} \bigvee_{j \in M} F_{ij}$ where N and M are finite index sets and F_{ij} is a constraint or, generalized to first order logic terms, a literal. Then for each $i \in N$, $\bigvee_{j \in M} F_{ij}$ is a clause. $\mathcal{F}$ can be represented in clausal form as $\{\bigcup_{j \in M} \{l_{ij}\} \mid i \in N\}$.*

Example 2.1.1. *$\mathcal{F}_1$ is a QFNRA formula in CNF in constraints over $\mathbb{Z}[x_1, x_2]$*

$$\mathcal{F}_1 = (\underbrace{x_1^2 x_2 + x_1^3 - 3 > 0}_{\text{a literal/constraint}} \vee x_1^2 - x_2 \leq 0) \wedge \underbrace{(x_1^4 x_2^7 - x_2 + 7 \neq 0 \vee x_1^2 x_2^7 - x_1 x_2 - 6 \geq 0)}_{\text{a clause}}.$$

In clausal form

$$\mathcal{F}_1 = \{\{x_1^2 x_2 + x_1^3 - 3 > 0, x_1^2 - x_2 \leq 0\}, \{x_1^4 x_2^7 - x_2 + 7 \neq 0, x_1^2 x_2^7 - x_1 x_2 - 6 \geq 0\}\}.$$

2.2 nlsat Algorithm Intuition

Generally, nlsat is used to check the satisfiability of a QFNRA Formula $\mathcal{F}$ in CNF consisting of constraints over $\mathbb{Z}[x_1, \ldots, x_n]$. For this, $\mathcal{F}$ is decomposed into $\mathcal{F}_0 \dot{\cup} \ldots \dot{\cup} \mathcal{F}_n$ where $\mathcal{F}_i$ is a set containing all clauses of $\mathcal{F}$ where i is the maximal level of a constraint in such clauses. $\mathcal{F}_0$ can be dealt with on a Boolean level since the constraints in the clauses in $\mathcal{F}_0$ contain no variables and thus all evaluate to true or false. Then, nlsat searches for a Boolean assignment for the literals, as well as for an assignment for the theory variables contained in said literals level by level.

More precisely, nlsat first look into the clauses contained in $\mathcal{F}_1$. The algorithm tries to find a Boolean assignment for the literals which satisfies each clause while still keeping the theory compatible. This means that a literal is only assumed to be true if it has at least one common, satisfying assignment to the theory variable x_1 with all the previously assumed to be true literals. Such an assignment to x_1 is called *feasible*. If a Boolean assignment is found which satisfies all clauses in $\mathcal{F}_1$ while still maintaining a non-empty set of feasible candidates for x_1, one such candidate is picked as a for the time being fixed assignment for x_1.

Then, $\mathcal{F}_2$ can be examined in the same way since the constraints in the clauses in $\mathcal{F}_2$ becomes univariate through the assignment of x_1 on the previous level. This is continued level by level. If level $n + 1$ would be reached, nlsat would have found an assignment for $x_1, \ldots, x_n$ which satisfies literals in such a way that overall the formula $\mathcal{F}$ would be satisfied. Therefore, the procedure would return that $\mathcal{F}$ is *satisfiable*.

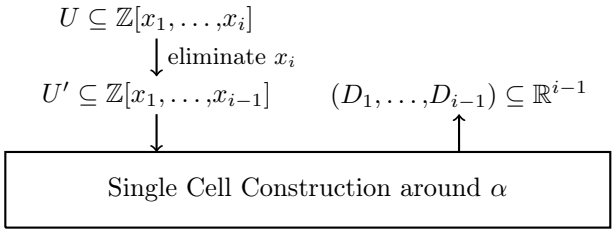


Figure 2.1: Use of Single Cell Construction in explain function of *mcSat* as black-box

Nevertheless, it can also happen that while examining $\mathcal{F}_i$ for some $i \in \{1,\ldots,n\}$, a Boolean assignment which is not satisfying or leaves no feasible candidate for x_i is reached. The first case is a *Boolean conflict*, the latter a *theory conflict*. In both cases, nlsat will employ conflict resolution and backtrack. Boolean conflicts are dealt with by Boolean resolution. For a theory conflict on the other hand, nlsat needs to determine why there is no possible assignment for x_i. It therefor isolates a minimal subset of the constraints which causes a possible theory assignment to fail, an *unsatisfiable core*, and creates an *explanation* to why that is.

Creating this explanation is where single cell construction can be utilized. In the current state, there is a small set of conflicting constraints U of at most level i and a theory assignment for $x_1,\ldots,x_{i-1}$ which is represented as the point $\alpha = (\alpha_1,\ldots,\alpha_{i-1})$. Doing a full CAD-projection for the i-th level (more on projection in Chapter 3), results in a sample point α in $i-1$ components and a set U' containing constraints of maximal level $i-1$. This is exactly the input necessary for the construction of a single algebraic cell around α. Executing single cell construction then results in a cell $(D_1,\ldots,D_{i-1}) \subseteq \mathbb{R}^{i-1}$ that can be exclude from the search space for the theory assignment in the nlsat algorithm. This is summarized and displayed in Figure 2.1.

Now, formulating into a QFNRA formula that for each $j \in \{1,\ldots,i-1\}$, x_j can not be in D_j, is an explanation that can be merged into $\mathcal{F}$. This concludes the conflict resolution and nlsat can *backtrack* to a lower level where a Boolean assignment that leaves a feasible theory assignment exists. If there is no such level, the algorithm essentially backtracks to level 0, thus concluding that the given formula is *unsatisfiable*.

Chapter 3

Cylindrical Algebraic Decomposition

Alfred Tarski published a decision procedure to check satisfiability modulo theories for the theory of QFNRA in 1948 [Tar98] which proved the problem's decidability. His approach was computationally very inefficient so that later in 1975, most notably George E. Collins introduced the *Cylindrical Algebraic Decomposition* (CAD) [Col75] as a far more efficient, yet still doubly exponentially complex, approach.

To decide satisfiability for a QFNRA Formula $\mathcal{F}$ in constraints over $\mathbb{Z}[x_1, \ldots, x_n]$, a CAD can be computed. Let Q be the set containing all the constraints in $\mathcal{F}$. A CAD is a partition of $\mathbb{R}^n$ into finite connected regions. For all these regions $R \subseteq \mathbb{R}^n$, each polynomial p of a constraint $F \in Q$ does not change its sign over R, meaning that for all $a \in R$ either $p(a) = 0$, $p(a) < 0$ or $p(a) > 0$. Thus, each constraint is either satisfied or unsatisfied over the entire region which results in $\mathcal{F}$ also being either satisfied or unsatisfied over the entire region. Therefore, only one sample point from each of the finite regions has to be checked for satisfaction of $\mathcal{F}$. If one such point satisfies $\mathcal{F}$, the formula is satisfiable, otherwise it is not. After introducing some definitions, we will further investigate how to compute a CAD.

3.1 Preliminaries

First, we will substantiate some of the terms that were already used in the introduction of this chapter. Then, some more concepts which will be of relevance in the more specific explanation of the CAD are defined. Definitions 3.1.1 to 3.1.4 are copied from [ÁHK20] and Definitions 3.1.5 is copied from [BK15].

Definition 3.1.1 (Region)**.** *A region of $\mathbb{R}^n$ is a non-empty, connected subset of $\mathbb{R}^n$.*

Thus follows that every real, open or closed interval $(a,b) \subseteq \mathbb{R}$ is a Region of $\mathbb{R}$ for $a,b \in \mathbb{R}$. Also, for every two regions $R, R' \subseteq \mathbb{R}$, $R \times R' \subseteq \mathbb{R}^2$ is a region on $\mathbb{R}^2$.

Definition 3.1.2 (sign of a polynomial)**.** *Given a polynomial $p \in \mathbb{Z}[x_1, \ldots, x_n]$,*

$a \in \mathbb{R}^n$, the sign of p at a is

$$sgn(p(a)) := \begin{cases} -1, & \text{if } p(a) < 0, \\ 0, & \text{if } p(a) = 0, \\ 1, & \text{if } p(a) > 0. \end{cases}$$

Definition 3.1.3 (Sign-invariant Region)**.** *Given a finite, non-empty set $P \subseteq \mathbb{Z}[x_1,\ldots,x_n]$, a region $R \subseteq \mathbb{R}^n$ is (P-)sign-invariant if for all $p \in P$ and $a,b \in R$, $sgn(p(a)) = sgn(p(b))$.*

Let us have a look at some sign invariant regions defined by the polynomials

$$p_1 = (x_1 - 4)^2 + (x_2 - 4)^2 - 9 \quad \text{and} \quad p_2 = x_1 - x_2 - 3 \tag{3.1}$$

over $\mathbb{Z}[x_1,x_2]$. For this, we consider the plots of the implicit equations $p_1 \stackrel{!}{=} 0$ and $p_2 \stackrel{!}{=} 0$. These can be seen in Figure 3.1. The blue (•) regions displayed in (a), (b) and (c) are each maximal sign-invariant regions. They are maximal in a sense that there exists no superset of the region which is also sign-invariant.

In Figure 3.1(a), a sign-invariant region in the form of a two dimensional plane is displayed. For each point $(a,b) \in \mathbb{R}^2$ of said plane, it holds that $p_1(a,b) < 0$ and $p_1(a,b) < 0$. Figure 3.1(b) shows a sign-invariant region being a one dimensional line so that for each point $(a,b) \in \mathbb{R}^2$ on the line, it holds that $p_1(a,b) < 0$ and $p_1(a,b) = 0$. Finally, in Figure 3.1(c), there is a zero dimensional, sign-invariant region displayed as a point in (7,4). For this point, it holds that $p_1(7,4) = p_2(7,4) = 0$.

Before continuing with the definition of delineability, we would like to create an intuition for this term along the previous example of p_1 and p_2. Delineability of a set of polynomials P of level n over a region R of dimension $n-1$ means that the number and order of the real roots of P over R is fixed.

Figure 3.2 visualizes this. We again consider the plot of the implicit equations $p_1 \stackrel{!}{=} 0$ and $p_2 \stackrel{!}{=} 0$ but this time with the addition of regions $R_1 = (3,7) \subseteq \mathbb{R}_1$ and $R_2 = (4,7) \subseteq \mathbb{R}_1$. Figure 3.2(a) shows region R_1 on which $\{p_1,p_2\}$ is not delineable since the number of roots over $4 \in R_1$ is equal to 2 and is thereby smaller than the number of roots over $5 \in R_1$ which is equal to 3. Said roots are marked in blue (•). Figure 3.2(b) on the other hand shows region R_2 on which $\{p_1,p_2\}$ is delineable. The number of roots over each point in R_2 is fixed at 3 and the order is also fixed as informally speaking *root of p_1 → root of p_2 → root of p_1*.

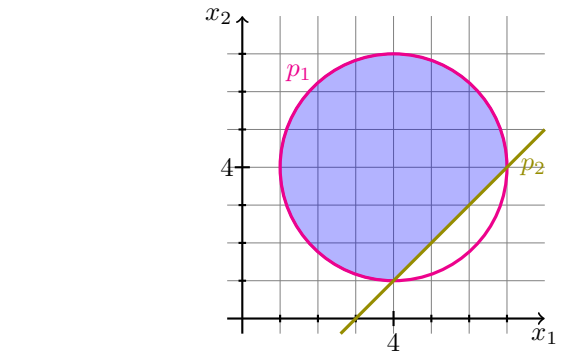
Definition 3.1.4 (Delineability)**.** *Given a region $R \subseteq \mathbb{R}^{n-1}$ and finite, non-empty $P \subseteq \mathbb{Z}[x_1,\ldots,x_n]$ where $n \geq 2$. P is delineable on R if for all $p \in P$, $a \in R$:*

1.) the number of roots of $p(a)$ is constant,

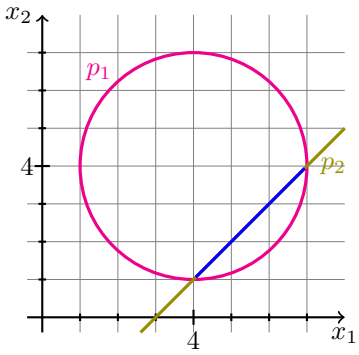
2.) the number of different roots of $p(a)$ is constant,

3.) for all $q \in P$, $q \neq p$, the number of common roots of $p(a)$ and $q(a)$ is constant.

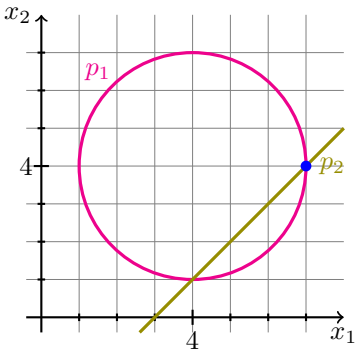
Last of all, we assume that we have a procedure to isolate the roots of a univariate polynomial $p \in \mathbb{Z}[x]$, for example one of the algorithms presented in [Joh98].



(a) 2 dimensional sign-inv. region

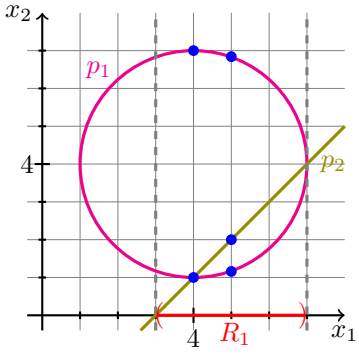


(b) 1 dimensional sign-inv. region

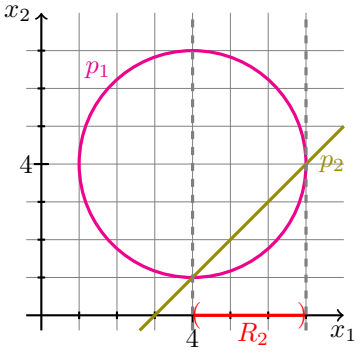


(c) 0 dimensional sign-inv. region

Figure 3.1: Plot of the implicit equations $p_1 \overset{!}{=} 0$ and $p_2 \overset{!}{=} 0$ with a sign-invariant region each marked in blue



(a) $\{p_1,p_2\}$ is not delineable over R_1



(b) $\{p_1,p_2\}$ is not delineable over R_2

Figure 3.2: Plot of the implicit equations of $p_1 = 0$ and $p_2 = 0$ with highlighted regions

3.2 Procedure for CAD Computation

Before presenting a procedure which can be used to calculate a CAD, let us formally define this term (adapted[1] from [ÁHK20], originally defined in [Col75]):

Definition 3.2.1 (Cylindrical Algebraic Decomposition)**.** *A finite $\mathcal{C} \subseteq \mathcal{P}(\mathbb{R}^n)$ is called a Cylindrical Algebraic Decomposition (CAD) of $\mathbb{R}^n$ for a finite, non-empty $P \subseteq \mathbb{Z}[x_1, \ldots, x_n]$, $n \geq 1$ and an element $C \in \mathcal{C}$ is called a (CAD-)cell, if the following holds:*

1. $\bigcup_{C \in \mathcal{C}} C = \mathbb{R}^n$,
2. $C \cap C' = \emptyset$ *for all* $C, C' \in \mathcal{C}$ *with* $C \neq C'$,
3. *Every* $C \in \mathcal{C}$ *is a* P*-sign-invariant region,*
4. *Every* $C \in \mathcal{C}$ *can be represented as the solution of a set of constraints restricted to operators* $=$, $>$ *and* $<$ *over* $\mathbb{Q}[x_1, \ldots, x_n]$,
5. *If* $n > 1$ *then there exists a CAD* $\mathcal{C}'$ *of* $\mathbb{R}^{n-1}$ *such that for every* $C \in \mathcal{C}$ *there is a* $C' \in \mathcal{C}'$ *such that the projection of* C *to the first* $n-1$ *dimensions is* C'.

Point one to three state that a CAD is a partition, or rather *decomposition* of $\mathbb{R}^n$ into sign-invariant regions. The fourth point states that the decomposition is *algebraic*. And the last point states that the decomposition is *cylindrical*, meaning that each cell C of dimension n forms a cylinder over the cell that is C's projection to its first $n-1$ dimensions.

We now present a procedure to calculate a CAD, or to be more precise, a finite set $S \subseteq \mathbb{R}^n$ which contains one point from each cell of the CAD since we need just that for the purpose of satisfiability checking. We call this procedure COMPUTE_CAD. As input, it takes a set of polynomials $P \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ which for checking satisfiability of a Formula $\mathcal{F}$ were taken from the Constraints in $\mathcal{F}$. Note that only the polynomials are taken from $\mathcal{F}$, ignoring the operators comparing them to 0 as well as the logical operators connecting the constraints. This is due to a CAD being a more fine-grained partition of $\mathbb{R}^n$ than theoretically necessary, namely into sign-invariant regions regarding the polynomials rather than "satisfiability-invariant" regions regarding the constraints.

Generally, COMPUTE_CAD consists of two main phases, the *projection* phase and the *lifting* phase. In the projection phase, a CAD-projection operator (definition adapted from [ÁHK20]) is used.

Definition 3.2.2 ((CAD-)projection operator)**.** *Given finite, non-empty $P \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ where $n \geq 2$, a mapping*

$$pr : \mathcal{P}(\mathbb{Z}[x_1, \ldots, x_n]) \to \mathcal{P}(\mathbb{Z}[x_1, \ldots, x_{n-1}])$$

is a projection operator. We call pr a CAD-projection operator, if any region $R \subseteq \mathbb{R}^{n-1}$ is $pr(P)$-sign invariant iff R is P-delineable.

The closure of P under pr is defined as the set of polynomials

$$pr^*(P) = P \cup \bigcup_{k=1}^{i-1} pr^k(P)$$

[1] By "adapted" we always mean structurally copied but modified in regard to application to our purposes

where $pr^{\,1}(P) := pr\,(P)$ *and* $pr^{\,n+1}(P) := pr\,(pr^{\,n}(P))$ *for* $n \geq 1$.

Such a projection operator is recursively applied to the starting polynomials, projecting polynomials to lower levels along the given variable order until univariate polynomials are reached. This is depicted on the left side of Figure 3.3. A concrete projection operator is McCallum's Projection Operator (originally defined by Scott McCallum [McC98]).

Definition 3.2.3 (McCallum's projection operator)**.** *Given a set* P *of polynomials of level* $\leq i$*, McCallum's projection operator is defined as*

$$\begin{aligned} proj\,(P) = &\bigcup_{\substack{p\in P\\ p \text{ of level } i}} coeff_{x_i}(p) \quad \cup \bigcup_{\substack{p\in P\\ p \text{ of level } i}} \{disc_{x_i}(p)\} \quad \cup \\ &\bigcup_{\substack{p,q\in P,\ p\neq q\\ p,q \text{ of level } i}} \{res_{x_i}(p,q)\} \quad \cup \bigcup_{\substack{p\in P\\ \text{level of } p\, <i}} \{p\} \end{aligned}$$

where for $p, q \in \mathbb{Q}[x_1, \ldots, x_{i-1}][x_i]$ *of level* i*, considered as univariate in* x_i

- $coeff_{x_i}(p) \subseteq \mathbb{Q}[x_1, \ldots, x_{i-1}]$ *is the set containing the coefficients of* p*,*
- $disc_{x_i}(p) \in \mathbb{Q}[x_1, \ldots, x_{i-1}]$ *is the discriminant of* p*,*
- $res_{x_i}(p,q) \in \mathbb{Q}[x_1, \ldots, x_{i-1}]$ *is the resultant of* p *and* q*.*

Note that McCallum's projection operator is incomplete. This means that there are some inputs for which the operator fails which can be detected before applying it. In practical application, these inputs are rare; 15.42% or 3.57% in our testing of the level-wise algorithm, depending on the context (see Table 6.1). Therefore, we ignore them in this chapter. For the single cell construction later on, this will lead to there being a case distinction where *FAIL* might be returned.

McCallum's projection operator clearly is a projection operator since its components - coefficients, discriminants and resultants of polynomials of level i - are of level less than i. Harder to see is how it is an incomplete CAD-projection operator. Let us therefore for some polynomials Q look at the roots of the polynomials in $proj(Q)$ since these roots define the $proj(Q)$-sign-invariant regions that are Q-delineable as stated in Definition 3.2.2. Generally speaking, the polynomials in $proj(Q)$ are coefficients, discriminants and resultants. We will inspect where these have their roots with respect to the polynomial(s) that they are derived from. For the purposes of demonstration, we consider 2-dimensional polynomials as examples because higher dimensional cases are essentially based on these.

First, consider the *discriminant*. It covers *turning points* in the root plot since at these points the number in roots changes. To visualize this, Figure 3.4a shows $p_1 \overset{!}{=} 0$ with p_1 as in Equation (3.1). Also, the roots of $disc_{x_2}(p_1) \in \mathbb{Q}[x_1]$ are depicted in blue (•). The gray points (•) mark the turning points which caused the roots. Additionally, the cylinder walls which would be implied by these roots in a CAD are depicted as gray, dashed lines. It can be seen that over the sign-invariant regions $(-\infty,1)$, [1,1], (1,7), [7,7], $(7,\infty) \subseteq \mathbb{R}_1$ which are implied by the roots of $disc_{x_2}(p_1)$, the number and order of the roots of p_1 is fixed, meaning that these regions are all already $\{p_1\}$-delineable in this example.

The second component of McCallum's operator is the *resultant*. It projects *intersections* of the roots of two polynomials to the roots of a polynomial on a lower level.

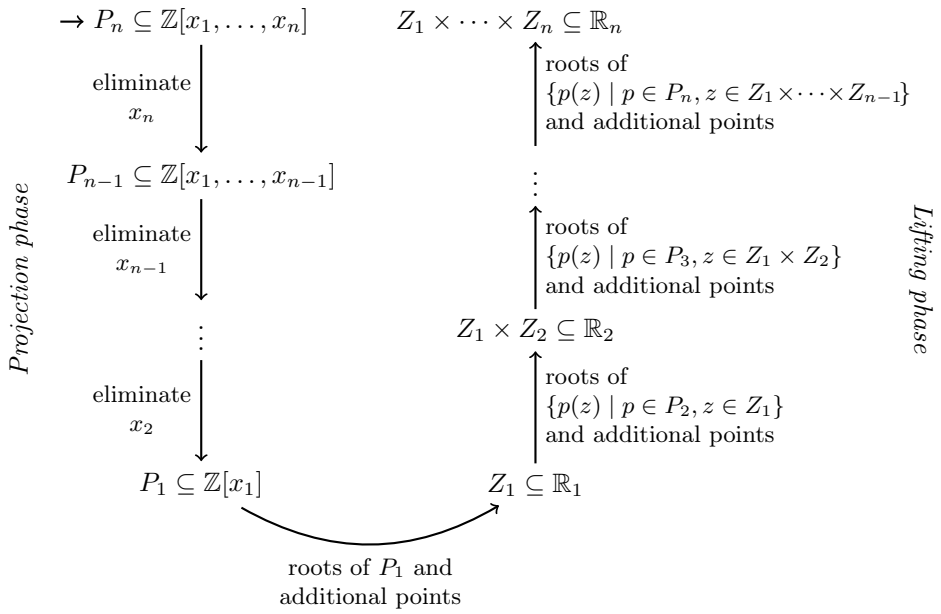
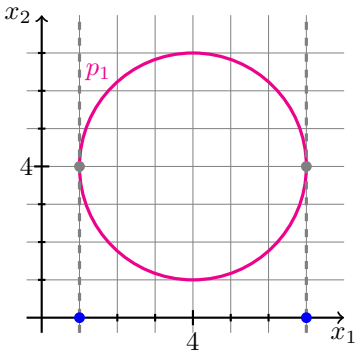


Figure 3.3: Sketch of COMPUTE_CAD procedure (adapted from [ÁHK20])

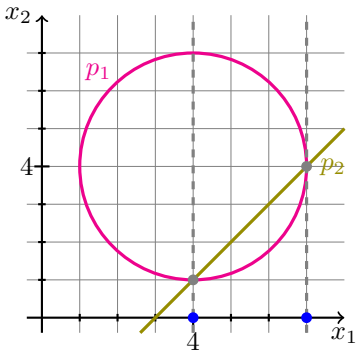
We again consider the example of p_1 and p_2 as in Equation (3.1). In Figure 3.4b, $p_1 \overset{!}{=} 0$ and $p_2 \overset{!}{=} 0$ are depicted. Along this, the roots of $res_{x_2}(p_1,p_2) \in \mathbb{Q}[x_1]$ can be seen in blue (•) with the intersections that cause them marked in gray (•). Also, the gray, dashed lines display the cylinder walls which would be implied by these roots in a CAD.

Last of all, consider the *coefficients*. These project where the roots of the polynomial they are taken from are *tending towards infinity* to roots on a lower level. Since this does not happen for p_1 and p_2, we consider a new example, namely $p_3 = (x_1 - 2)x_2 - 1$. The plot of the implicit equation $p_3 \overset{!}{=} 0$ with the root of $coeff_{x_2}(p_3) \in \mathbb{Q}[x_1]$ in blue (•) is depicted in 3.4b.

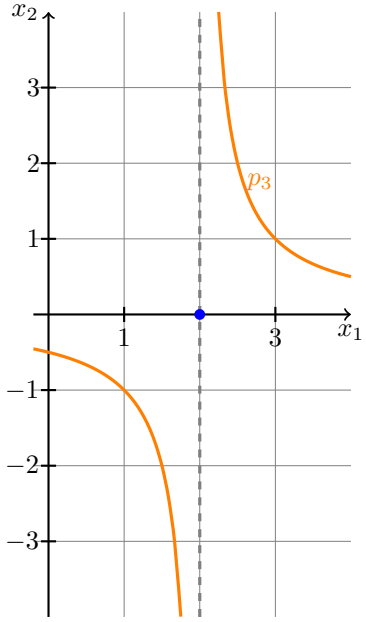
Having roots at these points is essential for the *lifting* to work properly. Let us refocus on the bigger picture again. We just saw that a projection operator can be applied iteratively, level by level, along the variable order, calculating the closure under the projection operator. The polynomials in the closure are divided by level into sets $P_1, \ldots, P_n$ where P_i contains the polynomials of the closure that are of level i. With this calculated, the *lifting* starts. For this, the roots of the univariate polynomials in P_1 are isolated. Adding a point from in between each pair of neighboring roots, as well as a point above the greatest and one below the smallest root, results in a set of 1-dimensional points which is the CAD for $\mathbb{R}_1$. A CAD for $\mathbb{R}_1$ is then expanded to a CAD for $\mathbb{R}_2$ and so on until level n is reached. For such a step from a CAD for $\mathbb{R}_i$ to a CAD for $\mathbb{R}_{i+1}$, the polynomials in P_i are evaluated at each point from the CAD for $\mathbb{R}_i$. These polynomials are again univariate. Thus, the roots and the additional points can be calculated as before to receive a set of 1-dimensional points. Taking the Cartesian product of this set with the i-dimensional set of points which is the CAD for $\mathbb{R}_i$ is then the CAD for $\mathbb{R}_{i+1}$. The lifting phase is shown on the right in Figure 3.3.



(a) Plot of the implicit equations $p_1 \stackrel{!}{=} 0$ and $disc_{x_2}(p_1) \stackrel{!}{=} 0$



(b) Plot of the implicit equations $p_1 \stackrel{!}{=} 0$, $p_2 \stackrel{!}{=} 0$ and $res_{x_2}(p_1, p_2) \stackrel{!}{=} 0$



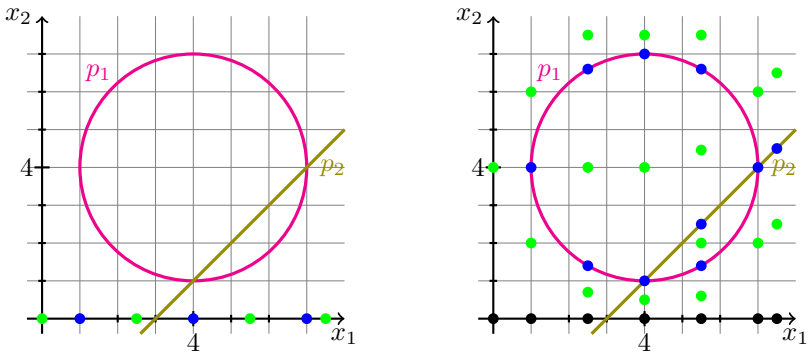
(c) Plot of the implicit equation $p_3 \stackrel{!}{=} 0$ with roots of polynomials in $coeff_{x_2}(p_3)$

Figure 3.4: Plots of implicit equation(s) with highlighted roots of different components of McCallum's projection operator

3.3 Exemplary Computation of a CAD

To clarify how COMPUTE_CAD using McCallum's projection operator works, we visualize it on our example of $P = \{p_1, p_2\}$ as given in equations (3.1). Again, the implicit equations $p_1 \overset{!}{=} 0$ and $p_2 \overset{!}{=} 0$ are considered. First, the closure under McCallum's projection operator is calculated which in this case is just the computation of the polynomials that result from projection of p_1 and p_2 on level 1. Then, the lifting of the CAD to level 1 starts. For this, the roots of the polynomials that resulted from projection are considered. These roots of the polynomials in $proj(P) \subseteq \mathbb{Q}[x_1]$ are $\{1{,}4{,}7\} \subseteq \mathbb{R}_1$ (see Figure 3.4). They are depicted in blue (•) in Figure 3.5a. The additional points between roots, above the greatest and below the smallest root are marked in green (•).

The next step is already the lifting to level 2 which can be seen in Figure 3.5b. The points over which we lift are now black and the roots of p_1 and p_2 over each of these points are marked in blue (•). The additional points are then marked in green (•) again. Note that a few points below the x_1-axis are not depicted. Still, Figure 3.5b (almost) shows a possible set of testing candidates for every sign-invariant region in blue and green. It can also be seen that some regions have more than one sample point constructed for it. A minimal set of sample points that represents a CAD is depicted in Figure 3.6.



(a) After lifting to level 1 (b) After lifting to level 2

Figure 3.5: Plot of the implicit equations $p_1 \overset{!}{=} 0$ and $p_2 \overset{!}{=} 0$ during the COMPUTE_CAD procedure

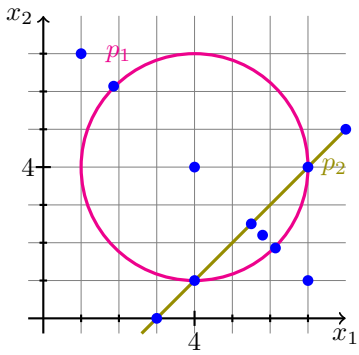


Figure 3.6: Plot of the implicit equations $p_1 \overset{!}{=} 0$ and $p_2 \overset{!}{=} 0$ with a minimal set of sample points that represent a CAD

Chapter 4

Single Cell Construction

In the terms of what is achievable with COMPUTE_CAD, we now do not want construct a CAD $\mathcal{C}$ for $P \subseteq \mathbb{Z}[x_1, \ldots, x_n]$, but instead with the additional input of a point $\alpha \in \mathbb{R}^n$ construct only the cell in $\mathcal{C}$ which contains α (if possible even a superset). Note that in this case, we actually want to create a description of the cell. First, we take a brief look at the existing work on this topic.

4.1 Related Work: Recursive Single Cell Construction

Christopher W. Brown and Marek Košta in [BK15] presented an algorithm for single cell construction that works recursively in a *Depth First Search*-style. More precisely, for a set Q of polynomials and a point α, their algorithm calls a merge procedure for each polynomial $q \in Q$. Let q be of level i. The merge procedure projects q to polynomials of lower levels which are then each merged recursively, unless they are already of level 1. When the recursive call returns to q, the bounds of the so far constructed cell in the i-th dimension are updated through the consideration of the roots of $p(\alpha_1, \ldots, \alpha_{i-1}, x_1)$ as bounds for the cell in the i-th dimension. To be precise, if one such root lies closer to α_i than the current upper or lower bound, then the respective bound is updated as this root.

This approach has been implemented by Malte Neuss [Neu18] in SMT-RAT, an Open Source C++ Toolbox for Strategic and Parallel SMT Solving [smt][CKJ+15]. We implemented the later on presented novel algorithm in SMT-RAT as well and will test the implementation against the recursive one. The results can be found in Chapter 6. Before the implemented algorithm is presented, we present a simpler algorithm. In order to do so, we need a few more definitions.

4.2 Preliminaries

We first introduce a theoretical and then a practical representation of roots. The following definitions are adapted from [BK15].

Definition 4.2.1 (Indexed root expression)**.** *For $p \in \mathbb{Q}[x_1, \ldots, x_i]$ of level i and non-negative integer j, we define the indexed root expression $root(p, j, x_i)$ at the point*

$\alpha = (\alpha_1, \ldots, \alpha_{i-1})$ as the j-th distinct real root of $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ (ordered smallest to largest). If polynomial $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ is the zero polynomial or if it has fewer than j distinct real roots, the expression has value undef.

We also allow the special indexed root expressions root$(+\infty, 1, x_i)$ and root$(-\infty, 1, x_i)$ to represent positive and negative infinity, respectively. We will not do any arithmetic with these expressions, so those semantics will not be addressed.

We formally extend the usual relational operators $\{<, >, \leq, \geq, =, \neq\}$ to be false if either the left or right hand side is undef. We allow the expressions root$(+\infty, 1, x_i)$ and root$(-\infty, 1, x_i)$ on the left and right hand side of the relational operators with the obvious semantics.

Definition 4.2.2 (RealAlgNum)**.** *A RealAlgNum is a pair (p, I), where p is a square-free univariate polynomial whose coefficients are rational numbers or RealAlgNum's. I is an isolating interval for a distinct real root of p. If A is a RealAlgNum, we adopt the notation A.p and A.I to refer to the two components of the pair. We refer to the real algebraic number that is the distinct real root of A.p in A.I as val(A).*

To simplify the presentation below, we will use A to denote both the pair (p, I) and the real algebraic number val(A), which the pair represents. This will always be clear from context. Furthermore, we allow A.p to be $+\infty$ or $-\infty$, in which case val(A) is $+\infty$ or $-\infty$, respectively.

RealAlgNums are used in the following algorithms to represent roots. They are useful since we are generally not able to calculate roots of univariate polynomials but to isolate them in an arbitrarily small interval. Also, they are used in the following data structure that will represent the cell throughout the algorithm and as a result.

A OneCell data structure is a tuple $(P,D) = ((P_n, \ldots, P_1), (D_1, \ldots, D_n))$. In P, the initial polynomials and the polynomials that result from projection throughout the presented algorithms are inserted and stored. A polynomial of level i can be found in P_i. D represents the actual cell with reference to polynomials in P. Each dimension of D is either a *section* or a *sector*.

A section of level i is described by a single root r of a univariate polynomial $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ where p is of level i. For this root, it must hold that $r = \alpha_i$ to be a section.

A sector on level i is described by two roots r_1, r_2 of univariate polynomials $p_1(\alpha_1, \ldots, \alpha_{i-1}, x_1), p_2(\alpha_1, \ldots, \alpha_{i-1}, x_1)$ where p_1 and p_2 are each of level i. They surround the i-th component of the given point, meaning that $r_1 < \alpha_i < r_2$. In the case of a sector, it is also possible that one or both of the bounds are not existent, meaning that $r_1 = -\infty$ and/or $r_2 = \infty$. A more formal definition follows.

Definition 4.2.3 (OneCell Data Structure)**.** *A OneCell data structure containing point $\alpha = (\alpha_1, \ldots, \alpha_n)$ is a pair (P, D), where $P = (P_n, \ldots, P_1)$ is a tuple of sets of integer polynomials, and $D = (D_1, \ldots, D_n)$. D_n is either $(sector, l, L, u, U)$ or $(section, e, E)$.*

If D_n is a section, then $D_n.e = (p, j)$ is a pair where $p \in \mathbb{Z}[x_1, \ldots, x_n]$ is a polynomial of level n and $j \in \mathbb{N}$. $D_n.E$ is a RealAlgNum. If D_n is a sector, then:

- *Either $D_n.l$ is a pair (p, j) where $p \in \mathbb{Z}[x_1, \ldots, x_n]$ is a polynomial of level n and $j \in \mathbb{N}$, and $D_n.L$ is a RealAlgNum, or $D_n.l = (-\infty, 1)$.*
- *Either $D_n.u$ is a pair (p, j) where $p \in \mathbb{Z}[x_1, \ldots, x_n]$ is a polynomial of level n and $j \in \mathbb{N}$, and $D_n.U$ is a RealAlgNum, or $D_n.u = (+\infty, 1)$.*

Furthermore, the following conditions hold:

1. *If* $n > 1$, $((P_{n-1},\ldots,P_1),(D_1,\ldots,D_{n-1}))$ *is a OneCell data structure containing the point* $(\alpha_1,\ldots,\alpha_{n-1})$.
2. *If* D_n *is a section, then* $D_n.E.p$ *is a non-constant univariate polynomial in variable* x_n, *which divides* $D_n.e.p(\alpha_1,\ldots,\alpha_{n-1},x_n)$.
3. *If* D_n *is a sector, then either* $D_n.l = (-\infty,1)$ *and* $D_n.L = (-\infty,[0,0])$, *or the following conditions hold:*
   (i) $D_n.L.p$ *is a non-constant univariate polynomial in variable* x_n, *which divides* $D_n.l.p(\alpha_1,\ldots,\alpha_{n-1},x_n)$.
   (ii) $D_n.L.p$ *has no real root in the open interval* $(val(D_n.L),val(D_n.U))$.
4. *If* D_n *is a sector, then either* $D_n.u = (-\infty,1)$ *and* $D_n.U = (+\infty,[0,0])$, *or the following conditions hold:*
   (i) $D_n.U.p$ *is a non-constant univariate polynomial in variable* x_n, *which divides* $D_n.u.p(\alpha_1,\ldots,\alpha_{n-1},x_n)$.
   (ii) $D_n.U.p$ *has no real root in the open interval* $(val(D_n.L),val(D_n.U))$.
5. *If* D_n *is a section, the following formula is true at point* α:
$$\alpha_n = root(D_n.e.p,D_n.e.j,x_n)$$
6. *If* D_n *is a sector, the following formula is true at point* α:
$$root(D_n.l.p,D_n.l.j,x_n) < \alpha_n < root(D_n.u.p,D_n.u.j,x_n)$$
7. $S(D_1,\ldots,D_n)$ *(see Definition 4.2.4) is a cylindrical subset of* $\mathbb{R}^n$.
8. *The polynomials in* $bpoly(D_n) \subseteq P$ *(see Definition 4.2.5) are delineable on* $S(D_1,\ldots,D_{n-1})$.
9. *Let* S *be the maximal connected region of* $\mathbb{R}^n$ *containing* α, *such that the polynomials from* P *of level at most* n *are tag-invariant in* S *(see Definition 5.1.2). Then it holds that* $S = S(D_1,\ldots,D_n)$.

Definition 4.2.4. *Let* $D = (D_1,\ldots,D_i)$ *be as in Definition 4.2.3 and* $\alpha = (\alpha_1,\ldots,\alpha_n) \in \mathbb{R}^n$, *where* $n \geq i$. *We define* $F(D)$ *to be a formula which is true when* $i = 0$. *If* $i > 0$, *and* D_i *is a section then we define* $F(D)$ *to be:*
$$F(D_1,\ldots,D_{i-1}) \wedge x_i = root(D_i.e.p,D_i.e.j,x_i).$$
If $i > 0$, *and* D_i *is a sector then we define* $F(D)$ *to be:*
$$root(D_i.l.p,D_i.l.j,x_i) < \alpha_n < root(D_i.u.p,D_i.u.j,x_i).$$
We define $S(D)$ *as* $\mathbb{R}^0$ *if* $i = 0$, *and otherwise*
$$S(D) = \{\gamma \in \mathbb{R}^i \mid F(D) \text{ is true at } \gamma\}.$$
We define $S_\alpha(D)$ *as*
$$S_\alpha(D) = \{(\alpha_1,\ldots,\alpha_{i-1},\gamma_i) \in \mathbb{R}^i \mid F(D) \text{ is true at } (\alpha_1,\ldots,\alpha_{i-1},\gamma_i)\}.$$

Definition 4.2.5. *Let* $((P_n,\dots,P_1),(D_1,\dots,D_n))$ *be a OneCell data structure. For* $1 \leq i \leq n$ *we define* $bpoly(D_i)$ *to be the set of bounding polynomials occurring in* D_i*. If* D_i *is a section, then* $bpoly(D_i) = \{D_i.e.p\}$*. If* D_i *is a sector then* $bpoly(D_i) = \{D_i.l.p, D_i.u.p\}\backslash\{-\infty,+\infty\}$*.*

For $m \leq n$ *we define* $bpoly(D_1,\dots,D_m) = \bigcup_{i=1}^{m} bpoly(D_i)$*.*

4.3 Primitive level-wise Single Cell Construction

The following algorithm (Alg. 1) constructs a single algebraic cell around a point α. We call it primitive since it does a full projection with McCallum's projection operator (see l.2) which we will later on improve upon. On the other hand, this yields the advantage of the projection operator being easily exchangeable. First, the full projection is stored in P and the cell D is initialized (see ll.2-5). Then, the levels are iterated from top to bottom, making it a level-wise approach.

For each level i, the algorithm distinguishes whether D_i is a *section* or a *sector*. In the *section* case, the algorithm first checks for failure (see ll.9-12). Failure occurs when a polynomial p is *nullified* over $(\alpha_1,\dots,\alpha_{i-1})$, meaning that $p(\alpha_1,\dots,\alpha_{i-1},x_1)$ is constantly 0. After that, the section simply gets defined through a polynomial that vanishes at $(\alpha_1,\dots,\alpha_i)$ and the according root as a RealAlgNum (see ll.13-15).

In the *sector* case, all polynomials of level i, evaluated at $(\alpha_1,\dots,\alpha_{i-1})$, are considered. The roots of all these univariate polynomials in x_i are then isolated. From all these roots, the two which are the closest above and below α_i are searched for. These two roots as RealAlgNums, with the polynomials from which they are derived, then define the sector (see ll.17-26).

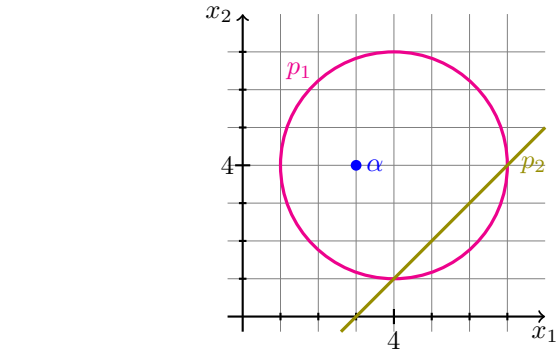
Algorithm 1: Primitive level-wise Single Cell Construction

Input : $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ where α_k is a RealAlgNum for $k \in \{1, \ldots, n\}$
$Q = \{q_1, \ldots, q_m\} \subseteq \mathbb{Z}[x_1, \ldots, x_n]$

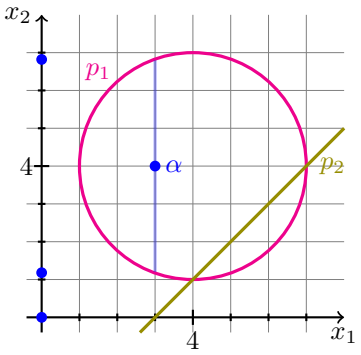
Output: a OneCell data structure $((P_n,..,P_1),(D_1, \ldots, D_n))$ containing α, such that $\{q_1, \ldots, q_m\} \subseteq (P_n \dot{\cup} \ldots \dot{\cup} P_1) \subseteq proj^*(Q)$
or (FAIL,f,i). In that case the following hold:
1. f is an irreducible integer polynomial of level $2 \leq i \leq n$
2. $f \in proj^*(Q)$
3. $f(\alpha_1, \ldots, \alpha_{i-1}, x_i) = 0$

set $P = (P_n, \ldots, P_1)$ where $P_i = \emptyset$
for each $p \in proj^*(Q)$ **do**
  for each *non-constant irreducible factor* f *of* p **do**
    let i be the level of f
    set $P_i = P_i \cup \{f\}$
set $D = (D_1, \ldots, D_n)$ where $D_i = (sector,(-\infty,1),(-\infty,[0,0]),(+\infty,1),(+\infty,[0,0]))$
for $i := n$ *to* 1 **do**
  if *for some* $p \in P_i$, $p(\alpha_1, \ldots, \alpha_i) = 0$ **then**
    /* D_i is a section */
    if $i \neq 1$ **then**
      /* Check for failure caused by projection */
      for each $q \in P_i$ **do**
        if $q(\alpha_1, \ldots, \alpha_{i-1}, x_i) = 0$ **then**
          return (FAIL,q,i)
      set $\overline{p}$ to be the square-free part of the univariate polynomial $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$
      let j be the index of the root α_i of $\overline{p}$ and let I be an interval so that α_i is the only root of $\overline{p}$ in I
      set $D_i = (section,(p,j),(\overline{p},I))$
  else
    /* D_i is a sector */
    for each $p \in P_i$ **do**
      set $\overline{p}$ to the square-free part of the univariate polynomial $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$
      if $\overline{p}$ *has a root between* $D_i.L$ *and* α_i **then**
        let the RealAlgNum B be the root of $\overline{p}$ between $D_i.L$ and α_i which is closest to α_i
        let j be the index of the root B of $\overline{p}$
        set $D_i.l = (p,j)$ and $D_i.L = B$
      if $\overline{p}$ *has a root between* α_i *and* $D_i.U$ **then**
        let the RealAlgNum B be the root of $\overline{p}$ between α_i and $D_i.R$ which is closest to α_i
        let j be the index of the root B of $\overline{p}$
        set $D_i.u = (p,j)$ and $D_i.U = B$
return (P,D)

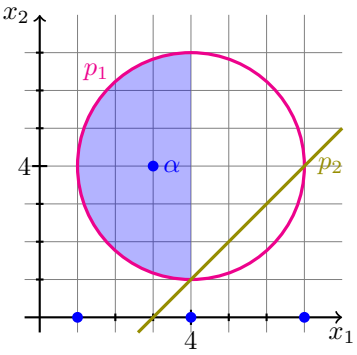
In the following, this procedure is visualized on the previously seen example of the polynomials p_1 and p_2 as in equations (3.1). The point around which the cell will be constructed is $\alpha = (3,4)$. This initialization can be seen in Figure 4.1a.



(a) Initial state



(b) After the first iteration



(c) After the second iteration

Figure 4.1: Plot of the implicit equations $p_1 \overset{!}{=} 0$ and $p_2 \overset{!}{=} 0$ throughout Alg. 1 with $\alpha = (3,4)$

In the first iteration of the algorithm, the roots of $p_1(3,x_2)$ and $p_2(3,x_2)$ are considered. The roots are marked as points in blue (•) on the x_2-axis in Figure 4.1b. They give the bounds for α_2 that can also be seen in light blue

$$root(p_1,1,x_2) < \alpha_2 < root(p_1,2,x_2).$$

In the second iteration, the roots of the resultant of p_1 and p_2 as well as the roots of the discriminant of p_1 are considered. These can be seen as blue (•) points on the x_2-axis in Figure 4.1c. Note that the the discriminant of p_2 and the coefficients of p_1 and p_2 are in the projection but have no roots and are thus not relevant here. From these roots, the bounds for α_1 are derived

$$root(disc_{x_2}(p_1),1,x_1) < \alpha_1 < root(res_{x_2}(p_1,p_2),1,x_1).$$

These two bounds combined result in the cell which can be seen in blue in Figure 4.1c.

The effort of the algorithm is similar to the of COMPUTE_CAD since the computationally most expensive part in COMPUTE_CAD is the projection which Alg. 1 does in a full manner. The algorithm presented in the next Chapter improves upon this.

Chapter 5

Optimized Level-wise Single Cell Construction

We propose a more optimized approach to level-wise Single Cell Construction. This optimization for once comes through a reduction of McCallum's projection operator found by Christopher W. Brown [Bro01]. We call this reduced operator Brown-McCallum's projection operator. With this optimization, only the leading coefficient with the addition of another coefficient in some cases is considered instead of all coefficients. Additionally, we reduce the projection with the incorporation of the knowledge that only a single cell is constructed. This leads to a general reduction of the amount of considered resultants.

5.1 Preliminaries

Before getting into the algorithm, we need two last definitions (Definition 5.1.2 copied from [BK15]).

Definition 5.1.1 (order-invariance)**.** *A polynomial p is order-invariant over a region R iff p is sign-invariant over R and if the sign over R is zero (all $a \in R$ are already roots of p), then the multiplicity of any two roots in R is the same.*

Another way to phrase having constant multiplicity of roots is that the *order* of p is constant over the region. A more formal definition can be found in McCallum [McC98].

Definition 5.1.2 (tagged polynomial)**.** *A* tagged polynomial *of level i is a pair (t,p), where $t \in \{oi,si\}$ and $p \in \mathbb{R}[x_1,...,x_n]$ is a polynomial of level i. A tagged polynomial (t,p) is tag-invariant on a connected region $R \subseteq \mathbb{R}^i$ if $t = oi$ and p is order-invariant on R, or if $t = si$ and p is sign-invariant on R.*

We will sometimes refer to a tagged polynomial (t,p) as p when the tag is not of relevance in a situation. This will be clear from context. In the same spirit, we use the OneCell data structure (Definition 4.2.3) with tagged polynomials in the upcoming algorithm.

With McCallum's projection operator, all polynomials are assumed to be order-invariant. Brown then showed that this property that is stronger than sign-invariance

is not necessary for the coefficients [Bro01][BK15]. Therefore, projected coefficients will always be tagged with *si* and all other polynomials with *oi* during their initialization.

5.2 Algorithm for Optimized Level-wise Single Cell Construction

The optimized algorithm is a variant of Alg. 1. Essentially, a full projection as seen in lines 2-4 is not done, but after determining the section or sector on the current level, the polynomials on the current level are projected. In the case of a section, this projection is inserted after line 15 and for the sector case after the loop in lines 17-26 of Alg. 1. The complete optimized algorithm can be seen in Alg. 2-6.

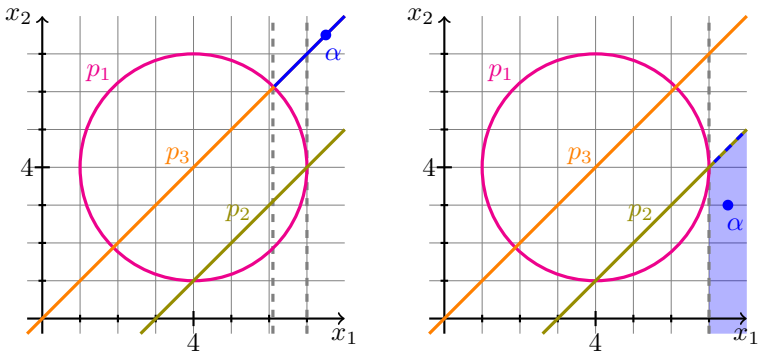
First of all, *Append* (Alg. 3) is a helper method to add each polynomial on its correct level in P. It is used for the initialization of P and for storing projection results in P. In the algorithm, the tag of the to be appended polynomial is considered. Since order-invariance implies sign-invariance (but not the other way around), the algorithm just updates the tag of a polynomial p from *si* to *oi* if p already was in P with *si* and is now appended with *oi*. Otherwise, it is simply appended on the correct level without duplication.

Construct (Alg. 2) is the main procedure that is structured similarly to Alg. 1. The addition of projection on the current level can be seen in lines 12-28 for the case of a section and in lines 31-32 with the call of *ProjectSector* (Alg. 6) for the case of a sector. With the exception of resultants, projection is done as in the algorithms *MergeRoot* and *MergeNotRoot* from [BK15] (converted to work iteratively). Thus also, the addition of *CoeffNonNull* (Alg. 4) which is an adaptation of *RefNonNull* in [BK15]. This algorithm possibly returns an additional coefficient of a given polynomial that is added to the projection. The additional coefficient can potentially shrink the cell below the level of the polynomial to prevent nullification of other projection components over a part of the otherwise bigger cell.

Also, in contrast to the algorithm in [BK15], the level-wise algorithm covers one case less to prevent nullification and thus failure that it can not easily deal with due to it being level-wise. When the recursive algorithm [BK15] reaches a polynomial that is nullified, it calls the algorithm *MergeNull* which does not return fail if the so far constructed cell *below* is already a point. Since the level-wise algorithm constructs the cell entirely top to bottom, this check is not easily possible.

Overall, using Brown-McCallum's operator reduces the amount of coefficients that are calculated to at most two. Additionally, in the case of a section fewer discriminants are calculated. More precisely, the discriminant is only calculated for the bounding polynomial and for polynomials that vanish at $(\alpha_1, \ldots, \alpha_i)$ and are tagged as order-invariant.

As to resultants, an optimization is done that is possible due to the iterative nature of the algorithm. Because of this, the bound(s) of the cell on level i are known when projecting the polynomials of level i. Therefore, in the section case, only the resultants between the polynomial defining the section on level i and every other polynomial are calculated. Let us visualize exemplary why resultants between two polynomials, of which neither is defining the section, are unnecessary in the projection. In Figure 5.1, the root plot of the implicit equations $p_1 \overset{!}{=} 0$ and $p_2 \overset{!}{=} 0$ as in Equation (3.1) with the



(a) In the case of a section (b) In the case of a sector

Figure 5.1: Plot of the implicit equations $p_1 \stackrel{!}{=} 0$, $p_2 \stackrel{!}{=} 0$ and $p_3 \stackrel{!}{=} 0$ showing exemplary the unnecessity of some resultants in projection in single cell construction

addition of the implicit equation $p_3 \stackrel{!}{=} 0$ where $p_3 = x_1 - x_2$ are depicted. Additionally, the point $\alpha = (7.5,7.5)$ and the cell it is contained in are marked in blue (•). This cell is defined by a section in its second dimension because $p_3(\alpha) = 0$. Therefore, when the algorithm reaches the projection part in the section case, the resultant of p_1 and p_2 is *not* calculated and appended as neither of the polynomials define the section. And this is clearly useful; since resultants project intersections of two polynomials in their roots, taking the resultant would cut the cell shorter than necessary, loosing the part between the gray, dashed lines. This generalizes, common roots between polynomials other than the one defining the cell on level i do not influence the cell.

In the sector case, resultants are calculated between the upper bound and the lower bound (if neither is $(-)\infty$). Additionally, resultants between the upper bound and each polynomial p for which $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ has a root above α_i as well as between the lower bound and polynomials p for which $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ has a root below α_i are calculated. These polynomials above and below the bounds that will be used in resultant calculation are filtered out in *DetermineBounds* (Alg. 5) and are put in the sets *up* and *down*. This is the only thing differentiating *DetermineBounds* from Alg. 1, lines 17-26.

An example for resultants that are not calculated in the sector case can be seen in Figure 5.1b. The same situation as in Figure 5.1a is depicted, only that α is now at (7.5,3), making the second dimension of the cell a sector. In this case, the algorithm would solely calculate the resultant between p_2 and p_3. Even though in this example this resultant is superfluous, if the line depicting the roots of p_3 was tilted differently so that it cut the roots of p_2 somewhere after 7.5 in the x_1-direction, not having this resultant would calculate a cell that is bigger than would be correct. On the other hand, not calculating the resultant between p_1 and p_3 is fine since their intersection in roots cannot interfere with the upper bound which is defined by p_2. The same holds for the resultant of p_2 and p_3; the intersection in their roots falls together with a turning point in roots of p_1 so that the edge of the cell is already inserted through the discriminant. Note that the discriminant is in the sector case of the algorithm taken for each polynomial.

Generalized, if two polynomials of which neither are the bounds of a cell intersect,

they intersect outside of the cell so that their resultant is not of importance for the single cell construction. If one of the two polynomials is a bound but for the other polynomial p, $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ does not have a root over/under α_i, then the interaction of p with the cell is already covered through the consideration of its discriminant or coefficients that are computed.

Overall, less resultants are calculated in the algorithm in contrast to a full projection with McCallum's projection operator, which calculates the resultant of every two polynomials of level i.

After projecting, sometimes tags are updated in the algorithms (see Alg. 2, l.11+28 and Alg. 6, l.8). This is theoretically not necessary since after projection, the tag of a polynomial is never considered again but it gives useful meaning to the tags. Before being projected, the tag states which type of invariance is *required* for the polynomial over the cell. After being projected (and possibly changed), the tag states which type of invariance is *guaranteed* for the polynomial over the cell.

Algorithm 2: Iterative OneCell Construct (*Construct*)

Input : $\alpha = (\alpha_1,...,\alpha_n) \in \mathbb{R}^n$ where α_k is a RealAlgNum for $k \in \{1,...,n\}$
$Q = \{q_1,...,q_m\} \subseteq \mathbb{Z}[x_1,...,x_n]$

Output: a OneCell data structure $((P_n,..,P_1),(D_1,...,D_n))$ containing α, such that $\{(si,q_1),...,(si,q_m)\} \subseteq (P_n \dot\cup ... \dot\cup P_1) \subseteq proj^*(Q)$
or (FAIL,f,i). In that case the following hold:
1. f is an irreducible integer polynomial of level $2 \leq i \leq n$
2. $f \in proj^*(Q)$
3. $f(\alpha_1,...,\alpha_{i-1},x_i) = 0$

set $P = (P_n,...,P_1)$ where $P_i = \emptyset$
set $P = Append(P,\{(si,q) \mid q \in Q\})$
set $D = (D_1,...,D_n)$ where $D_i = (sector,(-\infty,1),(-\infty,[0,0]),(+\infty,1),(+\infty,[0,0]))$
for $i := n$ *to* 1 **do**
  if *for some* $(t,p) \in P_i$, $p(\alpha_1,...,\alpha_i) = 0$ **then**
    /* D_i is a section */
    set $\overline{p}$ to be the square-free part of the univariate polynomial $p(\alpha_1,...,\alpha_{i-1},x_i)$
    if $\overline{p} = 0$ *and* $i \neq 1$ **then**
      return (FAIL,p,i)
    let j be the index of the root α_i of $\overline{p}$ and let I be an interval so that α_i is the only root of $\overline{p}$ in I
    set $D_i = (section,(p,j),(\overline{p},I))$
    set $t = oi$ // modifies P_i
    if $i \neq 1$ **then**
      set $P = Append(P,\{(si,ldcf_{x_i}(p)),(oi,disc_{x_i}(p))\})$
      set $r = CoeffNonNull(\alpha,p,(P,D))$
      if $r \neq 0$ **then**
        $P = Append(P,\{(si,r)\})$
      for each $(t',q) \in P_i \backslash \{p\}$ **do**
        if $q(\alpha_1,...,\alpha_{i-1},x_i) = 0$ **then**
          return (FAIL,p,i)
        set $P = Append(P,\{(oi,res_{x_i}(p,q))\})$
        if $q(\alpha_1,...,\alpha_i) = 0$ **then**
          if $t' = oi$ **then**
            set $P = Append(P,\{(oi,disc_{x_i}(q))\})$
            set $r = CoeffNonNull(\alpha,q,(P,D))$
            if $r \neq 0$ **then**
              $P = Append(P,\{(si,r)\})$
          else
            set $t' = oi$ // modifies P_i
  else
    /* D_i is a sector */
    set $((P,D),down,up) = DetermineBounds(\alpha,i,(P,D))$
    if $i \neq 1$ **then**
      set $P = ProjectSector(\alpha,i,(P,D),down,up)$
return (P,D)

Algorithm 3: Append P by polynomials contained in Q (*Append*)

Input : a tuple of sets $(P_n,..,P_1)$
a set of tagged polynomials $Q \subseteq \mathbb{Z}[x_i,...,x_n]$
Output: a tuple of sets $(P'_n,..,P'_1)$

```
for each (t,q) ∈ Q do
    for each non-constant irreducible factor f of q do
        let i be the level of f
        if (t,f) ∉ P_i and (oi,f) ∉ P_i then
            set P_i = P_i ∪ {(t,f)}
            if t = oi then
                set P_i = P_i \ {(si,f)}
return P
```

Algorithm 4: Ensure non-nullification of a polynomial (*CoeffNonNull*)

Input : $\alpha = (\alpha_1,...,\alpha_n) \in \mathbb{R}^n$ where α_k is a RealAlgNum for $k \in \{1,...,n\}$
p is an irreducible integer polynomial of level $i \in \{1,...,n\}$
$((P_n,..,P_1),(D_1,...,D_n))$ is a OneCell data structure
Output: a polynomial $r \in \mathbb{Z}[x_1,...,x_{i-1}]$

```
let p(x_1,...,x_i) = c_m x_i^m + ... + c_1 x_i + c_0 where c_j is a polynomial of at most level i − 1
if there is a non-zero constant in {c_m,...,c_0} then
    return 0
if ldcf_{x_i}(p) ∈ P and ldcf_{x_i}(p)(α_1,...,α_{i−1}) ≠ 0 then
    return 0
if disc_{x_i}(p) ∈ P and disc_{x_i}(p)(α_1,...,α_{i−1}) ≠ 0 then
    return 0
if F(D_1,...,D_{i−1}) ∧ ⋀_{j=0}^{m} c_j = 0 is inconsistent then
    return 0
set r = c_j such that c_j(α_1,...,α_{i−1}) ≠ 0
return r
```

Algorithm 5: Determine bounds for a sector (*DetermineBounds*)

Input : $\alpha = (\alpha_1,...,\alpha_n) \in \mathbb{R}^n$ where α_k is a RealAlgNum for $k \in \{1,...,n\}$
the current level i of polynomials which are looked at
$((P_n,..,P_1),(D_1,...,D_n))$ is a OneCell data structure where
for all $p \in P_i$, $p(\alpha) \neq 0$

Output: a OneCell data structure $((P'_n,..,P'_1),(D'_1,...,D'_n))$ with updated bounds in D'_i
two sets *down* and *up*, containing all polynomials $p \in P_i$ for which $p(\alpha_1,...,\alpha_{i-1},x_i)$ has a root in $(-\infty,\alpha_i)$, respectively $(\alpha_i, +\infty)$

```
1  set down = ∅ and up = ∅
2  for each (t,p) ∈ P_i do
3      set p̄ to the square-free part of
         the univariate polynomial p(α_1,...,α_{i-1},x_i)
4      if p̄ has a root between −∞ and α_i then
5          set down = down ∪ {p}
6          if p̄ has a root between D_i.L and α_i then
7              let the RealAlgNum B be the root of p̄ between
                 D_i.L and α_i which is closest to α_i
8              let j be the index of the root B of p̄
9              set D_i.l = (p,j) and D_i.L = B
10     if p̄ has a root between α_i and +∞ then
11         set up = up ∪ {p}
12         if p̄ has a root between α_i and D_i.U then
13             let the RealAlgNum B be the root of p̄ between
                 α_i and D_i.R which is closest to α_i
14             let j be the index of the root B of p̄
15             set D_i.u = (p,j) and D_i.U = B
16 return ((P, D), down, up)
```

Algorithm 6: Project polynomials of level i for a Sector (*ProjectSector*)

Input : $\alpha = (\alpha_1,...,\alpha_n) \in \mathbb{R}^n$ where α_k is a RealAlgNum for $k \in \{1,...,n\}$
the current level i of polynomials which are looked at
$((P_n,..,P_1),(D_1,...,D_n))$ is a OneCell data structure where
for all $p \in P_i$, $p(\alpha) \neq 0$
two sets *down* and *up*, containing all polynomials $p \in P_i$ for which
$p(\alpha_1,...,\alpha_{i-1},x_i)$ has a root in $(-\infty,\alpha_i)$, respectively $(\alpha_i, +\infty)$

Output: a OneCell data structure $((P'_n,..,P'_1),(D'_1,...,D'_n))$ with an update of P
which includes the projected polynomials of level i

```
1  for each (t,p) ∈ P_i do
2      if D_i.l = −∞ or D_i.u = +∞ or some real root of p(α_1,...,α_{i−1},x_i) is equal to
         val(D_i.L) or val(D_i.U) then
3          set P = Append(P,{(si,ldcf_{x_i}(p))})
4      set P = Append(P,{(oi,disc_{x_i}(p))})
5      set r = CoeffNonNull(α,p,(P,D))
6      if r ≠ 0 then
7          set P = Append(P,{(si,r)})
8      set t = oi                                          // modifies P_i
9  if D_i.l.p ≠ −∞ then
10     for each p ∈ down do
11         if p ≠ D_i.l.p then
12             set P = Append(P,{(oi,res_{x_i}(p,D_i.l.p))})
13 if D_i.u.p ≠ +∞ then
14     for each p ∈ up do
15         if p ≠ D_i.u.p then
16             set P = Append(P,{(oi,res_{x_i}(p,D_i.u.p))})
17 if D_i.l ≠ −∞ and D_i.u ≠ +∞ then
18     set P = Append(P,{(oi,res_{x_i}(D_i.l.p,D_i.u.p)})
19 return P
```

Chapter 6

Comparison: Recursive vs Level-wise Single Cell Construction

The optimized level-wise single cell construction algorithm from Chapter 5 as well as the recursive single cell construction [BK15] are implemented in SMT-RAT [smt] [CKJ+15]. We will in the following test them against each other. For this comparison, we utilize the sets of benchmarks contained in the SMT-LIB-BENCHMARKS QF_NRA library [QFN][BST+10]. The contained benchmarks are QFNRA formulas which we will refer to as *problems*. Therefore, we actually test the implementations in the use-case of a SMT solver for QFNRA that is based on *mcSat* and is similar to *nlsat*. Still, since only the source of the explanation changes between tests, we can draw conclusions to the performance of the recursive and the level-wise approach in comparison.

The solver is implemented in SMT-RAT. In the following, we will refer to this solver using explanations resulting from recursive single cell construction as *OC* and using explanations resulting from level-wise single cell construction as *LW*. Since both of theses explanations can fail, there is a fallback procedure which is similar to Alg. 1 but using a complete projection operator. This fallback is slower than both approaches to single cell construction. Thus, possible performance differences between LW and OC must still correlate to the performance of level-wise and recursive single cell construction.

Additionally, we created two solvers which try to apply each *Fourier-Motzkin variable elimination* (FM) [JBDM13], *Interval Constraint Propagation* (ICP) [Kre19] and *Virtual Substitution* (VS) [ÁNK] in that order before resorting to recursive, respectively level-wise single cell construction for the creation of explanations. We call these solvers *OC+* and *LW+*. Since FM, ICP and VS only work for polynomials of low degrees but faster for these, OC+ and LW+ are generally faster than OC and LW and use single cell construction more rarely. Therefore, the + solvers will give less of an impression on the performance of recursive against level-wise approach but more on the difference it can make in practical application.

The machine used for testing has four 2.1 GHz AMD Opteron CPUs with 12 Cores each. In the created test series, each problem had 15 minutes for computations with

6 GB of RAM available.

6.1 Use Case Focused

Firstly, we do not prove the correctness of our algorithm in this thesis but since the correct result is given with each problem, we can compare the results with the solution to indicate correctness. And indeed, all 9262 of 11489 problems solved by LW and all 9688 problems solved by LW+ returned correct results.

The overall performance in the test series can be seen in Table 6.1. Statistics two to five show that LW(+) generally performs better than OC(+). The following four statistics show that this is also true if we restrict the test to the problems solved by both solvers. Additionally, the overlap between problems solved by at least LW(+) or OC(+), and problems solved by both is very high at 99.29% for the non+, and at 99.92% for the + solvers. This can be derived from comparing statistics one and six, and shows that there is not a very significant amount of problems that only one of the solvers can solve in 15 minutes.

For the last three statistics, we should differentiate between the + and non+ solvers since the numbers vary vastly between them. In the non+ case, at least 299672 explanations were called which is a lot considering that these were called in the run of 11489 problems. Also, the success rate of explanations is higher for OC in comparison to LW. This could be due to OC covering an additional case to prevent failure as mentioned in Section 5.2. In contrast to that, the + solvers only call for at most 3786 explanations. This indicates that there are a lot of problems with low degrees in our testing data and shows that *at most* 3786 out of 11489 could have used single cell construction. This gives reason to a more specific comparison that focuses on the problems that actually used single cell construction in its calculation for a better comparison of the different ways of single cell construction.

We also looked into the actual results of the computations, i.e. if the solved problems were either satisfiable or not, but the results were very similar across the board when comparing OC(+) and LW(+).

6.2 Single Cell Construction Focused

For this comparison, we consider the same test but try to get as close as possible to the difference in performance that comes from having different ways of creating explanations through single cell construction. Thus, we isolated only the problems that employed single cell construction at least once during their solving time. This performance evaluation can be seen in Table 6.2.

The first statistic alone further justifies this comparison since the previously derived upper bound of at most 3786 problems that used single cell construction with the + solvers is actually even lower at not more than 767 problems. Thus, for a comparison of solving time between the single cell construction approaches, 95.39%[1] of the problems solved by at least one solver have no value. The same can be seen for the non+ solvers which was not obvious from the previous comparison, even though the effect is not as strong with 51.21% of the solved samples not using single cell construction.

[1] $\frac{\#\text{ problems solved by at least one} - \#\text{ problems solved by at least one using single cell construction}}{\#\text{ problems solved by at least one}}$

As an effect, the statistics show more clearly that the usage of level-wise single cell construction outperforms the recursive approach since LW is faster than OC with an even greater difference than before. In this evaluation, the improvement can also be seen clearly for LW+ over OC+. Furthermore, the overlap in problems solved by at least OC(+) or LW(+), and problems solved by both is decreased by roughly 1% to 98.48% for the + and to 98.43% for the non+ solvers compared to the previous evaluation. This decrease was expected; we will look further into these problems outside of the overlap in Section 6.3.2.

To quantify the difference in performance, we can look at the decrease in time in

Performance measure	Solver			
	OC	LW	OC+	LW+
# problems solved by at least one	9294		9692	
# problems solved	9260	9262	9688	9688
Mean runtime of solved problems in s	6.784	6.080	7.400	7.227
Difference in means in s	0.704		0.173	
Decrease in time in %	10.38		2.34	
# problems both solved	9228		9684	
Mean runtime of problems both solved in s	6.196	5.547	7.170	7.164
Difference in means in s	0.650		0.007	
Decrease in time in %	10.49		0.01	
# explanations called	308633	299672	3761	3786
# explanations successful	267945	253468	3625	3651
Explanation success rate in %	86.82	84.58	96.38	96.43

Table 6.1: Overall performance comparison of QFNRA solvers with 15 minutes runtime and 6 GB RAM on 11489 problems

Performance measure	Solver			
	OC	LW	OC+	LW+
# problems	5888	5867	771	767
# problems solved by at least one	4331		447	
# solved problems	4297	4299	443	444
Mean runtime of solved problems in s	13.385	11.861	34.124	31.168
Difference in means in s	1.524		2.955	
Decrease in time in %	11.39		8.66	
# problems both solved	4265		440	
Mean runtime of problems both solved in s	12.163	10.753	31.267	29.986
Difference in means in s	1.410		1.281	
Decrease in time in %	11.59		4.10	

Table 6.2: Single cell construction focused performance comparison of QFNRA solvers with 15 minutes runtime and 6 GB RAM on overall 11489 problems. Each statistic is meant with the premise "using single cell construction at least once in the solving process"

percent. It shows that the mean solving time needed on all problems that *each* solver solved using single cell construction is decreased by 11.39% by using LW over OC and by 8.66% by using LW+ over OC+. When considering only the problems that *both* solver solved using single cell construction this decrease is 11.59%, respectively 4.10%. This is still not a direct comparison of the runtime of the two single cell construction algorithms but most likely as close as we can get to the difference in performance using this method of testing.

6.3 Graphical Comparison

We will now analyze the test results in a less generalizing way. For this, we first consider *performance profiles* and then *scatter plots*.

6.3.1 Performance Profiles

A performance profile is a plot of time against the number of problems solved in under (or equal) that time. The time axis is thereby scaled logarithmic. Performance profiles for all four solvers on all problems can be seen in Figure 6.1.

The plot underlines that LW is generally faster than OC, even though there are points in time where their number in solved problems matches. A new take-away is that the results from the previous comparisons are not just due to a well chosen time parameter since this continuous depiction shows that LW has the upper hand at most points in time or is at least evenly matched. As to LW+ and OC+, a difference in performance between is not clearly visible. This is on par with the previous observation that few problems rely on single cell construction which is the only differentiating factor for the two solvers.

Analogous to before, we therefore consider the performance profile for all problems that used single cell construction in one or the other form as quantified in statistic three of Table 6.2. Since the difference in solved problems between + and non+ solvers is very high, the profiles are plotted separately. In Figure 6.2, the profiles for the + solvers can be seen and in Figure 6.3 the profiles for the non+ solvers are depicted. The scales in Figure 6.1 and Figure 6.2 are the same size so that a direct, visual comparison is possible. This comparison confirms what has previously seen, namely that a restriction to problems using single cell construction sets LW a bit further apart from OC. Even though Figure 6.3 is not on the same scale as the other figures, it still shows that LW+ performs better than OC+ which was not clear from Figure 6.1.

6.3.2 Scatter Plots

The scatter plots in Figure 6.4 and Figure 6.5 display the time needed for problems with LW(+) against the time needed with OC(+). Timeouts and memouts are incorporated as having taken the maximum computation time of 15 minutes. The subset of problems we consider contains only problems which used single cell construction in the solving process for at least one of the two solvers. This is useful since problems that did not use single cell construction would cluster on the diagonal since the procedures are the same in that case. Also, problems that neither of the solvers solved are omitted since these would just be a dot in the top right corner. The displayed set of problems is the one quantified in statistic two of Table 6.2.

As for illustration, problems that returned satisfiable are displayed as green upward pointing triangles and problems that returned unsatisfiable are marked as red downward pointing triangles. Additionally, the opacity of the dots is dimmed down so that clustering is indicated by more saturated coloring. Furthermore, a gray diagonal is shown in both of the plots. Problems that lay on this took the same time with both solvers. Problems lying above it took less time using LW(+) and problems below took less time using OC(+).

Generally, it looks like more dots are above the diagonal in both plots, again supporting our argument of a better performance of LW(+) over OC(+). Most of the points seem to cluster in the lower left corner around the diagonal, showing that there are a lot of simple problems in the testing data. Otherwise, some problems are on the outer rim of the plot indicating that for both solvers there are a few problems that only one of them is capable of solving in under 15 minutes. A noticeable clustering or pattern of unsatisfiable or satisfiable problems does not seem to occur in the plots.

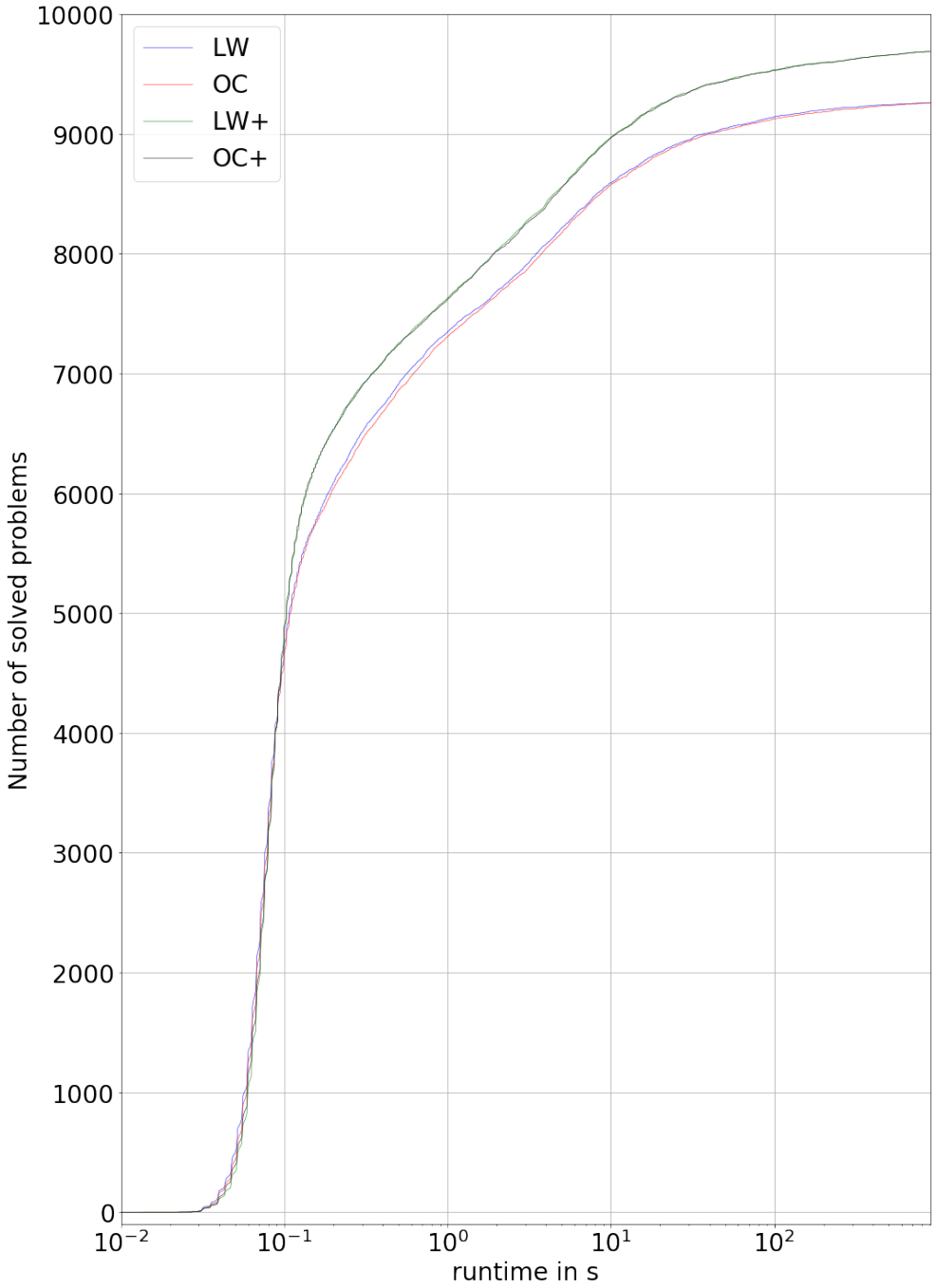


Figure 6.1: Performance profiles of QFNRA solvers with 15 minutes runtime and 6 GB RAM

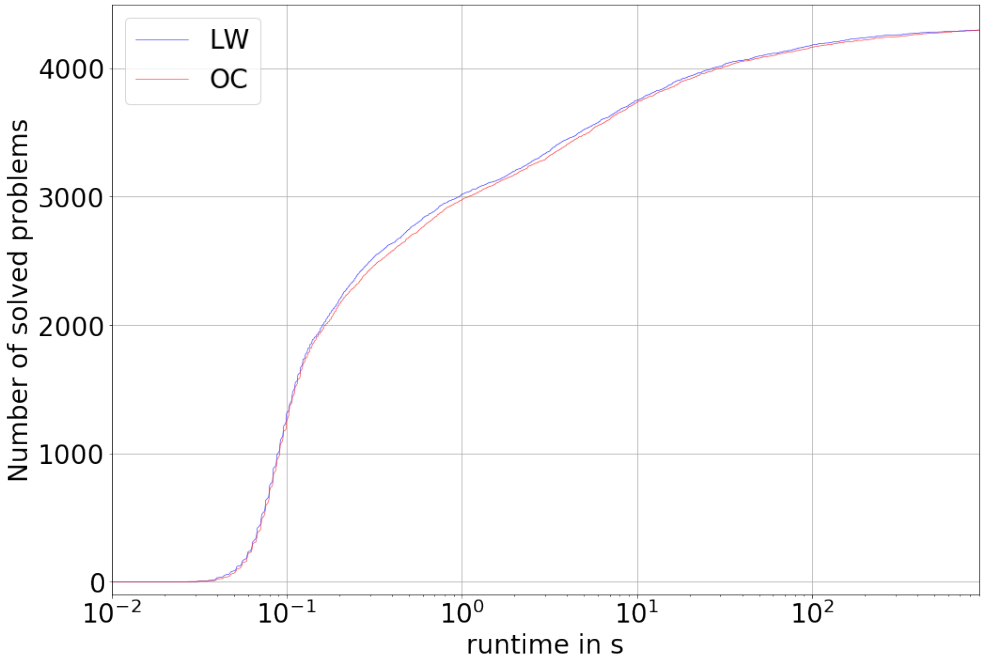


Figure 6.2: Performance profiles for LW and OC with 15 minutes runtime and 6 GB RAM restricted to problems solved with single cell construction

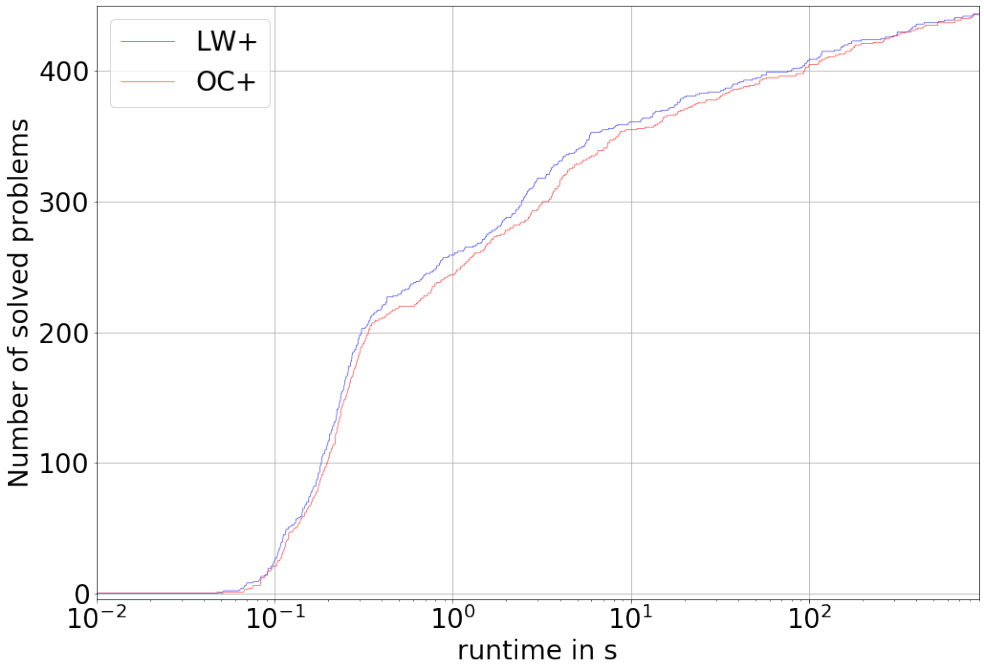


Figure 6.3: Performance profiles for LW+ and OC+ with 15 minutes runtime and 6 GB RAM restricted to problems solved with single cell construction

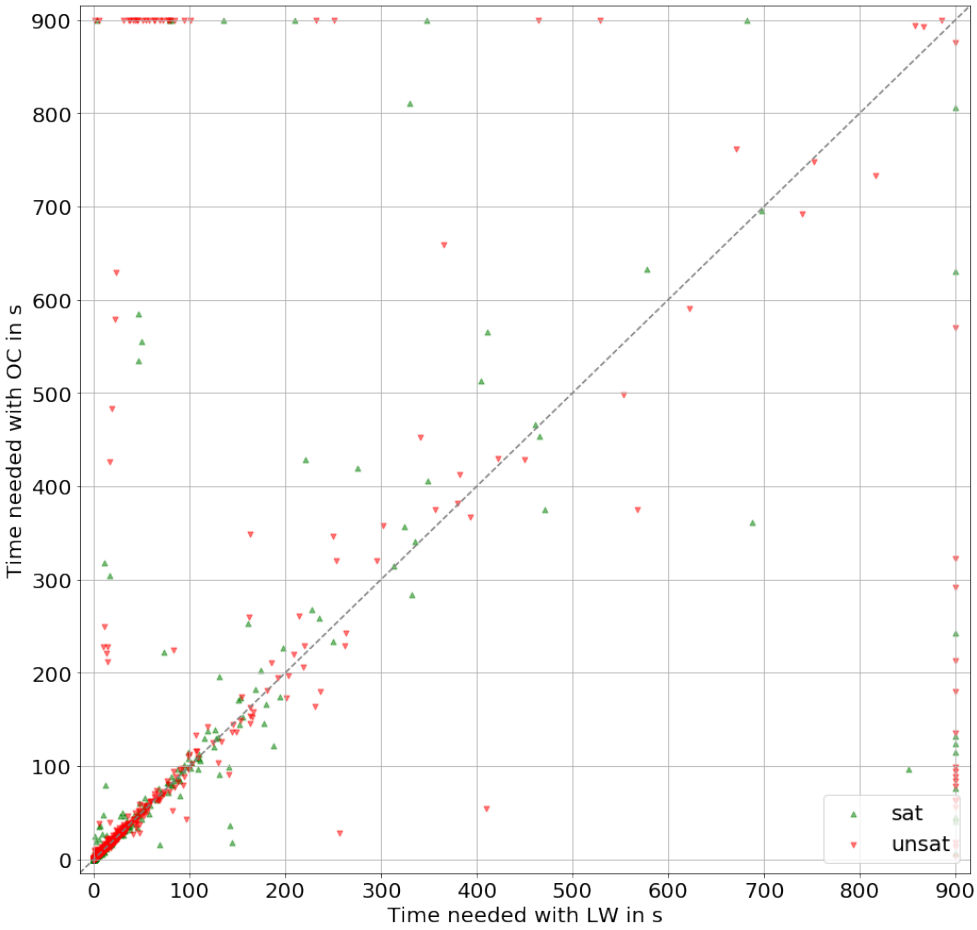


Figure 6.4: Scatter plot showing the runtime of LW vs OC with 15 minutes maximal runtime and 6 GB RAM each

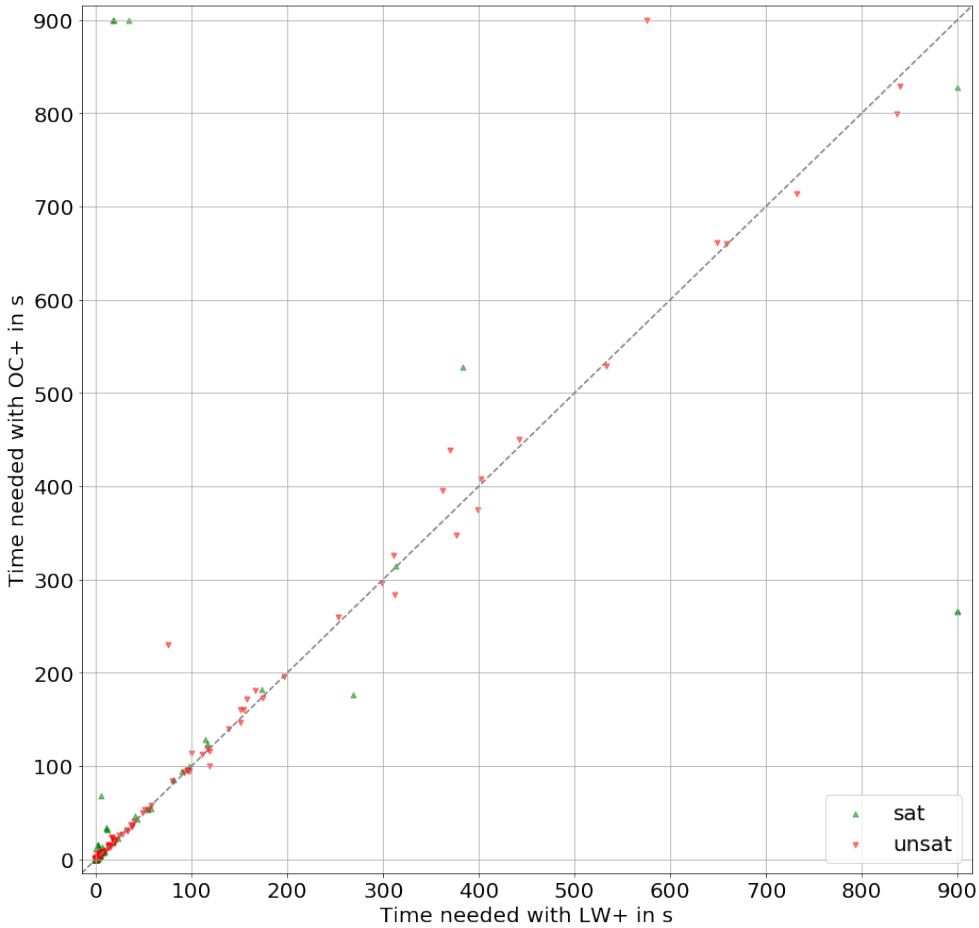


Figure 6.5: Scatter plot showing the runtime of LW+ vs OC+ with 15 minutes maximal runtime and 6 GB RAM each

Chapter 7

Conclusion

We conclude this thesis with a short summary and ideas for possible future work.

7.1 Summary

This thesis presented a novel approach to single cell construction that can be used in SMT solvers for QFNRA like for example *nlsat* [JdM12]. Single cell construction is essentially the construction of the cell of a CAD [Col75] in which a given point is contained. The presented, level-wise algorithm builds said cell level by level in contrast to the recursive algorithm [BK15] that merges polynomials as they come, thus refining bounds of the cell at different levels throughout the entire procedure. Both these algorithms use Brown-McCallum's projection operator [Bro01][McC98] with the level wise algorithm having some further optimization, reducing the amount of calculated resultants. Also, both are implemented in SMT-RAT, an Open Source C++ Toolbox for Strategic and Parallel SMT Solving [smt][CKJ+15].

With theses implementations, we compared the performance of the two approaches in the use-case of a solver for QFNRA similar to *nlsat*. As benchmarks, we chose the QFNRA formulas contained in the SMT-LIB-BENCHMARKS QF_NRA library [QFN][BST+10]. Running all benchmarks with 15 minutes maximal runtime and 6 GB RAM available each, showed that using the level-wise over the recursive approach in the solver resulted in a decrease in mean solving time of 10.38% (considering only solved formulas). With the help of performance profiles, we also verified that the found improvement is a continuous trend.

7.2 Future work

First of all, we did not proof the correctness of our algorithm. Proofing it would eliminate the marginal chance of there being an input for which the algorithm does not return the correct result.

Furthermore, McCallum's projection operator is not complete so that the algorithm can potentially return *FAIL*. Thus, a next step could be to use a complete operator like *Lazard's projection operator* [Laz94] to avoid the necessity of a fallback strategy. Still, in our testing, the proportion of failed single cell constructions is maximally 15.42% and in a more practical use-case only 3.57%. Therefore, it should be

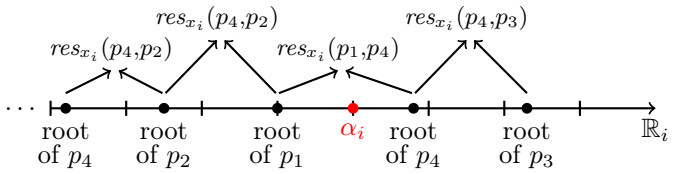


Figure 7.1: Alternative heuristic for resultant calculation in optimized level-wise single cell construction

kept in mind that a complete operator might increase the complexity so that using a Fallback strategy might still be better in performance.

Last of all, further heuristics for calculating resultants could be considered. Without proving correctness, we propose another heuristic for picking resultants to calculate in the section case of the optimized level-wise algorithm. Currently, the resultant between the upper/lower bound q and polynomials p with $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ having a root above/below α_i are calculated. Another idea is to sort the roots of all polynomials p with a root in $p(\alpha_1, \ldots, \alpha_{i-1}, x_i)$ and then taking the resultant between the polynomials that have neighboring roots. This is visualized in Figure 7.1. Furthermore, a hybrid variant of the currently implemented and the just presented heuristic is imaginable. If it works correctly, the hybrid variant could try to avoid very complex resultants or even try to calculate the least complex resultants. This is possible since the (asymptotic) complexity of the calculation of a resultant of two polynomials depends on the degrees of these polynomials which are accessible before calculating the resultant.

Bibliography

[ÁHK20] Erika Ábrahám, Rebecca Haehn, and Gereon Kremer. Lecture on satisfiability checking, Winter semester 2019/2020. https://ths.rwth-aachen.de/teaching/ws19/lecture-satisfiability-checking/.

[ÁNK] Erika Ábrahám, Jasper Nalbach, and Gereon Kremer. Embedding the virtual substitution method in the model constructing satisfiability calculus framework (work-in-progress paper).

[BK15] Christopher W. Brown and Marek Košta. Constructing a single cell in cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 70:14 – 48, 2015.

[Bro01] Christopher W. Brown. Improved projection for cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 32(5):447 – 465, 2001.

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

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368. Springer, 2015.

[Col75] George E Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 134–183. Springer Berlin Heidelberg, 1975.

[dMJ13] Leonardo de Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer Berlin Heidelberg, 2013.

[JBDM13] Dejan Jovanovic, Clark Barrett, and Leonardo De Moura. The design and implementation of the model constructing satisfiability calculus. In *2013 Formal Methods in Computer-Aided Design*, pages 173–180. IEEE, 2013.

[JdM12] Dejan Jovanović and Leonardo de Moura. Solving non-linear arithmetic. In *Automated Reasoning*, pages 339–354. Springer Berlin Heidelberg, 2012.

[Joh98] J. R. Johnson. Algorithms for polynomial real root isolation. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 269–299. Springer Vienna, 1998.

[Kre19] Gereon Kremer. *Cylindrical Algebraic Decomposition for Nonlinear Arithmetic Problems*. PhD dissertation, RWTH Aachen University, 2019.

[Laz94] D. Lazard. *An Improved Projection for Cylindrical Algebraic Decomposition*, pages 467–476. Springer New York, 1994.

[McC98] Scott McCallum. An improved projection operation for cylindrical algebraic decomposition. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 242–268. Springer, 1998.

[Neu18] Heinrich-Malte Neuss. Using Single CAD cells as explanations in MCSAT-style SMT solving. Master's thesis, RWTH Aachen University, 2018.

[QFN] Satisfiability modulo theories library for QFNRA. Available at, https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA.

[smt] SMT-RAT, a toolbox for strategic and parallel satisfiability modulo theories solving. Available at, https://github.com/smtrat/smtrat.

[Tar98] Alfred Tarski. A decision method for elementary algebra and geometry. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 24–84. Springer Vienna, 1998.