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

SPEEDING UP SINGLE CELL CONSTRUCTION VIA COMBINATORIAL OPTIMIZATION

Carsten Peter Perkampus

Communicated by
Prof. Dr. Erika Ábrahám

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

Additional Advisor:
Jasper Nalbach

Aachen, April 29, 2024

Abstract

In *Satisfiability Modulo Theories (SMT)* solving, the problem of computing a *Cylindrical Algebraic Decomposition (CAD)* is a crucial step in the process of solving formulas in the theory of real closed fields. The complexity of CAD computations grows doubly exponentially with the number of variables and the degree of the polynomials involved. This exponential growth in complexity poses a significant challenge for CAD-based SMT solvers, especially when dealing with large systems of equations and inequalities and has led to the development of new optimization techniques and algorithms like *levelwise single cell construction*. In this algorithm we compute only parts of the CAD to generalize certain properties of a set of polynomials that hold at a sample point to a connected set around it. As a part of this algorithm we need to find a suitable representation for generalizing orderings of real roots of some polynomials. This is a crucial step in the algorithm and in this thesis we aim to investigate how we can optimize the choice of the ordering using techniques from combinatorial optimization.

Contents

Chapter 1

Introduction

In the field of computational mathematics and computer science, the problem of finding solutions to formulas in the theory of Quantifier-Free Non-Linear Real Arithmetic (QFNRA) is a widely researched problem. The theory of QFNRA is of great importance because it allows us to reason about real-valued variables and their relationships using polynomial constraints. QFNRA thus serves as a fundamental building block for tackling complex real-world problems across various fields. At its core, QFNRA deals with formulas consisting of polynomial inequalities and equalities over real-valued variables without quantifiers. These formulas are used in many areas and applications including robotics and control systems to symbolic computation and formal verification. Solving these problems in a correct and time efficient manner is therefore of great importance. However, solving such formulas poses significant computational challenges due to the complex nature of non-linear relationships and the unbounded nature of the real numbers.

In this context, Satisfiability Modulo Theories (SMT) solving, which encompasses techniques for deciding the satisfiability of logical formulas over specific theories, emerges as a powerful tool for addressing the problems of QFNRA. One of the key techniques leveraged in the realm of QFNRA and SMT solving is Cylindrical Algebraic Decomposition (CAD). CAD, pioneered by George E. Collins in the 1970s [Col76], offers a systematic approach for quantifier elimination and for decomposing semi-algebraic sets into simpler cylindrical cells, therefore enabling the search for solutions to systems of polynomial equations and inequalities. Unfortunately, the complexity of CAD computations grows doubly exponentially with the number of variables and the degree of the polynomials involved. This exponential growth in complexity poses a significant challenge for CAD-based SMT solvers, especially when dealing with large systems of equations and inequalities.

Since its inception, CAD has been widely adopted and extended in the context of SMT solving with the development of new optimization techniques and algorithms. In more recent years, the levelwise single cell construction algorithm has emerged as a powerful tool for reducing the complexity of computing CADs [NÁS+24]. The authors note that the performance of their algorithm could potentially be improved in a number of ways. One such way is to look at combinatorial optimization techniques to reduce the impact of the combinatorial explosion that occurs when computing CADs. In this thesis, we aim to investigate the optimization problem of finding the optimal orderings of root functions of polynomials during levelwise single cell construction.

1.1 Research questions

As part of this thesis, we aim to answer the following research questions:

RQ1: How can we accurately estimate the cost of computing a resultant?

To optimize orderings of root functions of polynomials for the cost of resultants, we need to define and accurately estimate the cost of a resultant. This is a non-trivial task, as it is not clear what the cost of a resultant should represent. As such this question aims to define what the cost of a resultant should represent and how we can calculate it. The question also asks how different cost metrics impact the complexity and running time of the levelwise single cell construction algorithm.

RQ2: How can we optimize the ordering of root functions of polynomials for the cost of resultants?

Based on the cost metrics from RQ1, we want to investigate how we can optimize orderings of root functions for the cost of resultants. More generally, we want to know what complexity class the problem of finding the optimal ordering belongs to and if we can find an optimal indexed root ordering in polynomial time. By looking into this question, we aim to find out if we can improve the performance of the levelwise single cell construction algorithm by carefully selecting which resultants we will compute.

RQ3: How much time does it take to compute resultants in practice?

This question aims to investigate how much time it takes to compute resultants and how this time is distributed over all the calculated resultants. Specifically, we want to identify what parts of the algorithm are the most time-consuming.

RQ4: Can we also optimize orderings of root functions for the cost of discriminants?

Since discriminants are also used in the computation of resultants and are generally as expensive to compute as resultants, it is interesting to investigate if we can also optimize orderings of root functions for the cost of discriminants.

1.2 Thesis outline

The remainder of this thesis is structured as follows: We will first introduce the necessary background information in Chapter 2. In this chapter, we will first introduce the necessary background information needed to understand the basics of SMT solving, CAD, the levelwise single cell construction algorithm as well as more general concepts like optimization problems and graphs. The main part of this thesis is Chapter 3, where we will discuss the optimization problem of finding the optimal orderings of root functions along with different cost metrics. In Chapter 4, we will present the experimental results of our implementation of the discussed algorithms in SMT-RAT. Here we will also discuss the impact of resultant and discriminant computation times on the performance of the levelwise single cell construction algorithm more broadly. Finally, we will conclude this thesis in Chapter 5 by summarizing our findings and discussing possible future work.

Chapter 2

Preliminaries

We will start by introducing the theory of SMT solving and the theory of CADs. We will then introduce the levelwise algorithm and discuss some heuristics that can be used to improve the performance of the levelwise algorithm. Finally, we will introduce some graph theory concepts that will be used in the following chapters.

2.1 Satisfiability modulo theories

Satisfiability Modulo Theories (SMT) describes a field of computer science that deals with the decision problem of logical formulas. It essentially describes the task of deciding whether a given *formula* is satisfiable or not and extends the field of Boolean Satisfiability (SAT) by allowing formulas to contain variables of different types and predicates that are not part of the Boolean logic. In SMT solving we are given a formula in a specific *theory*, and we want to decide whether the formula is satisfiable or not. There are many theories that are able to express different kinds of problems. In this thesis we will focus on the theory of Quantifier-Free Non-Linear Real Arithmetic (QFNRA), which is a theory that allows us to express formulas that contain real-valued variables and polynomials.

We denote the set of natural numbers including 0 by $\mathbb{N}$, the set of rational numbers by $\mathbb{Q}$, and the set of real numbers by $\mathbb{R}$. Additionally, we denote the set of real numbers greater than or equal to 0 by $\mathbb{R}_{\geq 0}$. For two integers $i, j \in \mathbb{N}$ with $i < j$, we define $[i..j] = \{i, \dots, j\}$ and $[i] = [0..i]$ and for a vector $r \in \mathbb{R}^i$, we use r_j to denote the j-th component of r and $r_{[}j]$ to denote the vector $r_1, \dots, r_j$.

Definition 2.1.1 (Polynomial)**.** *A* polynomial *p in the variables $x_1, \dots, x_n$ is a finite sum of terms of the form $c \cdot x_1^{e_1} \cdot \dots \cdot x_n^{e_n}$ where $c \in \mathbb{Q}$ and $e_1, \dots, e_n \in \mathbb{N}$. The* set of all polynomials *in the variables $x_1, \dots, x_n$ is denoted by $\mathbb{Q}[x_1, \dots, x_n]$.*

For a vector $r \in \mathbb{R}^j$ with $j \in [1..i]$ and a polynomial $p \in \mathbb{Q}[x_1, \dots, x_i]$ we write $p(r, x_{j+1}, \dots, x_i)$ to define the polynomial after substituting $r_1, \dots, r_j$ for $x_1, \dots, x_j$ in p.

Definition 2.1.2 (Degree)**.** *For a polynomial p, we define the* degree *of p in the variable x as $deg_x(p) = k$, where k is the largest exponent of x in p such that the coefficient of x^k is non-zero.*

For the multivariate case we define the total degree *of a polynomial as the largest sum of exponents of the variables in the polynomial which has a non-zero coefficient.*

Definition 2.1.3 (QFNRA)**.** *Formulas in QFNRA consists of a set of polynomial constraints that are combined with logical connectives. We define atomic formulas in QFNRA as* $p \sim 0$ *where* p *is a polynomial and* $\sim \in \{<, =\}$*. More complex formulas can be constructed by combining atomic formulas with logical connectives such as "*$\neg$*" and "*$\wedge$*". Other relations like "*$\leq$*" and "*$\geq$*" can be expressed using the relations "*$<$*" and "*$=$*" and the logical connectives.*

Definition 2.1.4 (Conjunctive Normal Form (CNF))**.** *A formula is in CNF if it is a conjunction of clauses where each clause is a disjunction of literals. Literals are either a variable or the negation of a variable.*

2.2 Cylindrical algebraic decomposition algorithm

In SMT solving, there are multiple ways to decide the satisfiability of a given formula in the theory of QFNRA such as interval constraint propagation [TVKO17], virtual substitution [Wei97], subtropical satisfiability [FOSV17] and CAD. Out of those methods, CAD is the only practical algorithm that is complete for QFNRA, which is why the CAD algorithm is still widely used in SMT solving. In our case, we will use the CAD algorithm as part of the wider Model Constructing Satisfiability Calculus (MCSAT) framework [dJ13] to solve formulas in the theory of QFNRA.

2.2.1 CAD basics

To describe the CAD algorithm, we first need to define some basic concepts:

Definition 2.2.1 (Level [NÁS+24])**.** *Given the variables* $x_1, \ldots, x_n$ *under a fixed* variable ordering $x_1 \prec \cdots \prec x_n$ *and a polynomial* $p \in \mathbb{Q}[x_1, \ldots, x_n]$*, we define the* level *of a polynomial* p *as* $\text{level}(p) = n$*.*

Definition 2.2.2 (Sign)**.** *The* sign *of a number* $r \in \mathbb{R}$ *is defined as*

$$\text{sgn}(r) = \begin{cases} 1 & \text{if } r > 0 \\ 0 & \text{if } r = 0 \\ -1 & \text{if } r < 0 \end{cases}$$

Definition 2.2.3 (Sign-invariance)**.** *A polynomial* $p \in \mathbb{Q}[x_1, \ldots, x_n]$ *is* sign-invariant *on a set* $R \subseteq \mathbb{R}^n$ *if* $sgn(p(r)) = sgn(p(r'))$ *for all* $r, r' \in R$*. A set of polynomials* P *is sign-invariant on* R *if all polynomials in* P *are sign-invariant on* R*.*

Sign-invariance is an important property in the context of the CAD algorithm. If we know that the set of all polynomials appearing in a formula is sign-invariant on a cell, and we find that a sample point from the cell does not satisfy the formula, we can be sure that the cell does not contain any solutions to the formula. Conversely, if the sample point satisfies the formula, we can be sure that each element of the cell satisfies the formula.

Definition 2.2.4 (Decomposition)**.** *A* decomposition *of* $\mathbb{R}^n$ *is a partition of the set into a finite number of pairwise disjoint subsets* $\mathcal{C} = \{C_1, \ldots, C_k\}$ *such that* $\bigcup_{C \in \mathcal{C}} C = \mathbb{R}^n$*.*

Definition 2.2.5 (Semi-algebraic decomposition)**.** *A decomposition of* $\mathbb{R}^n$ *is called* semi-algebraic *if subset* $C \in \mathcal{C}$ *can be described by a finite number of polynomial (in-)equalities. That is, each subset* C *can be defined as*

$$C = \{x \in \mathbb{R}^n \mid p(x_1, \ldots, x_n) = 0\} \text{ or } C = \{x \in \mathbb{R}^n \mid p(x_1, \ldots, x_n) > 0\}$$

for a polynomial $p \in \mathbb{R}[x_1, \ldots, x_n]$ *or a union or intersection of such sets.*

Definition 2.2.6 (Cylindrical decomposition)**.** *A decomposition of* $\mathbb{R}^n$ *is called* cylindrical *if the set of projections of the cells onto the first* $n-1$ *variables is a cylindrical decomposition of* $\mathbb{R}^{n-1}$*. This means that the decomposition is cylindrical if the projected cells are still pairwise disjoint and cover* R^{n-1}*.*

Definition 2.2.7 (CAD [Col76])**.** *A CAD* $\mathcal{C}$ *is a set such that it is a semi-algebraic, cylindrical and finite decomposition of* $\mathbb{R}^n$*. The elements of this decomposition* $C \in \mathcal{C}$ *are called* cells*.*

The CAD for a set of polynomials $P \subset \mathbb{Q}[x_1, \ldots, x_n]$ *with* $n \geq 1$ *is a CAD of* $\mathbb{R}^n$ *whose cells are all* P*-sign-invariant.*

The idea behind a CAD is to decompose the space of the variables into cells such that each cell is P-sign-invariant. This allows us to decide satisfiability by testing a single sample point from each cell.

Definition 2.2.8 (Delineability [Col76])**.** *Let* $p \in \mathbb{Q}[x_1, \ldots, x_n]$ *be a polynomial and* $R \subseteq \mathbb{R}^{n-1}$ *be a cell. The polynomial* p *is* delineable *on* R *if there exists finitely many continuous functions* $\theta_1, \ldots, \theta_k : R \mapsto \mathbb{R}$ *(for* $k \geq 0$*) such that*

- $\theta_1 < \cdots < \theta_k$*,*
- *the set of real roots of the univariate polynomial* $p(r, x_{i+1})$ *is* $\{\theta_1(r), \ldots, \theta_k(r)\}$ *for all* $r \in R$*; and*
- *there exist constants* $m_1, \ldots, m_k \in \mathbb{N}_{>0}$ *such that for all* $r \in R$ *and all* $j \in [1..k]$*, the multiplicity of the root* $\theta_j(r)$ *of* $p(r, x_{i+1})$ *is* m_j*.*

The functions $\theta_1, \ldots, \theta_k$ *are called the* root functions *of* p *on* R*. A set of polynomials* P *is delineable on* $\mathbb{R}^n$ *if the product of every pair of polynomials in* P *is delineable on* R*.*

To guarantee sign-invariance of the polynomials in the formula, we need to ensure that the polynomials do not change sign within a cell. As such, we need to ensure that the input polynomials are delineable on the cells.

2.2.2 Projection operators

The CAD algorithm is split into two phases: the *projection phase* and the *construction phase*, which is also known as the *lifting phase*. In the projection phase, we eliminate variables from the given formula using the projection operator and in the construction phase, we construct the cells of the CAD by lifting the cells to higher levels.

Definition 2.2.9 (Resultant)**.** *Given two polynomials* $p_1, p_2 \in \mathbb{Q}[x_1, \ldots, x_i]$*, the* resultant *of the two polynomials is a polynomial that vanishes at those points where the two polynomials have a common root. We denote their resultant with respect to the variable* x_i *as* $res_{x_i}(p_1, p_2) \in \mathbb{Q}[x_1, \ldots, x_{i-1}]$*.*

We can use the resultant to identify "crossing-points" of two polynomials, which will be useful during the projection phase.

Definition 2.2.10 (Discriminant)**.** *Given a polynomial $p \in \mathbb{Q}[x_1, \ldots, x_i]$, the* discriminant *of a polynomial is a polynomial that vanishes at those points where the polynomial has a multiple root. We denote its discriminant with respect to x_i as* $\text{disc}_{x_i}(p) \in \mathbb{Q}[x_1, \ldots, x_{i-1}]$.

The discriminant is used to identify "turnarounds" in the graph of the polynomial.

Lastly, we use *coefficients* to identify divergence points of the polynomial. For a polynomial $p \in \mathbb{Q}[x_1, \ldots, x_i]$, the coefficients of the polynomial is denoted as $\text{coeff}_{x_i}(p) \in \mathbb{Q}[x_1, \ldots, x_{i-1}]$.

Definition 2.2.11 (Projection)**.** *A mapping $proj : 2^{\mathbb{Q}[x_1,\ldots,x_n]} \to 2^{\mathbb{Q}[x_1,\ldots,x_{n-1}]}$ is called a* CAD-Projection *if any proj(P)-sign-invariant region $R \subseteq \mathbb{R}^{n-1}$ is P-delineable.*

In the CAD algorithm we use CAD-Projections to eliminate variables from the given formula while also preserving the sign-invariance of the polynomials. Resultants, discriminants and coefficients are essential tools for the definition of projection operators.

Originally, the CAD algorithm used the projection operator from Collins [Col76] but since then many other projection operators have been developed, including the projection operator from McCallum.

Definition 2.2.12 (McCallum's projection operator [McC98])**.** *Given a set of polynomials $P \subset \mathbb{Q}[x_1, \ldots, x_n]$ and a variable ordering $x_1 < \cdots < x_n$, the projection operator from McCallum is defined as*

$$proj_{mc}(P) = \bigcup_{\substack{p,q \in P, p \neq q \\ \text{level}(p)=\text{level}(q)=i}} \{\text{res}_{x_i}(p, q)\} \;\cup\; \bigcup_{\substack{p \in P, \\ \text{level}(p)=i}} \{\text{disc}_{x_i}(p), \text{coeff}_{x_i}(p)\} \;\cup\; \bigcup_{\substack{p \in P, \\ \text{level}(p)<i}} \{p\}$$

The choice of projection operator is important because it affects the *completeness* of the CAD algorithm, and it can have a significant impact on the complexity of the algorithm since the projection operator is directly responsible for the number of resultants and discriminants that need to be computed.

Even though the projection operator from McCallum is not complete, we will use it in this thesis since the paper that describes the levelwise single cell construction algorithm uses this projection operator. This projection operator is chosen primarily because it is established and well-known in the field, is generally efficient and makes use of *equational constraints* [McC99, EBD20], which can significantly help in reducing the number of discriminants that need to be computed and therefore reduce the complexity of the algorithm.

The McCallum projection operator works by guaranteeing that the projected polynomials are *order-invariant*.

Definition 2.2.13 (Order [NÁS+24])**.** *For a polynomial $p \in \mathbb{Q}[x_1, \ldots, x_n]$ and a point $r \in \mathbb{R}^n$, the* order *of p at r is defined as*

$$\text{ord}_r(p) = \min\left(\left\{ k \in \mathbb{N} \;\middle|\; \begin{array}{l} \textit{some partial derivative of total order} \\ \textit{k of p does not vanish at r} \end{array} \right\} \cup \{\infty\}\right)$$

Definition 2.2.14 (Order-invariance [NÁS+24])**.** *A polynomial $p \in \mathbb{Q}[x_1, \ldots, x_n]$ is order-invariant on a set $R \subseteq \mathbb{R}^n$ if $\text{ord}_r(p) = \text{ord}_{r'}(p)$ for all $r, r' \in R$.*

Since order-invariance is a stronger property than sign-invariance, we can also maintain sign-invariance using the McCallum projection operator.

2.2.3 Equational constraints

Equational constraints can be used to reduce the projection size by reducing the number of discriminants that need to be computed. In the *section* case where any polynomial that is zero at the sample point is also zero in the entire cell we are trying to compute. By including the resultant of the section defining polynomial with another polynomial we can ensure that that polynomial is also sign-invariant. The work in [EBD20] shows that equational constraints are a powerful tool when trying to reduce the complexity of the algorithm. We will discuss how we can use equational constraints in Section 3.4.

2.2.4 Single cell construction

The CAD algorithm usually calculates all cells at once and takes samples from each cell to test each cell for satisfiability of the given formula. However, we often do not need all cells to decide satisfiability. Since each cell guarantees sign-invariance of the polynomials in the formula, we can decide satisfiability by testing a single sample point from each cell. This also means we should try to construct as few cells as possible and only construct cells that are necessary to decide satisfiability.

Instead of calculating all cells at once, we can alternatively guess a sample point from any cell and test it for satisfiability. If the chosen sample point satisfies the formula, we can stop the procedure and return sat. Otherwise, we can refine the cell containing the sample point and repeat the procedure. We accomplish this by calculating the boundary of the cell containing the sample point and taking this new information about the cell into account when choosing the next sample point.

This procedure is called *single cell construction*. In this thesis we will concentrate on the *levelwise single cell construction* algorithm as described in [NÁS+24]. There are other single cell construction algorithms that work polynomial-by-polynomial [BK15] instead of level-by-level, but we will not discuss them here.

For the levelwise single cell construction algorithm, we need to define some additional concepts that are required to describe the algorithm.

Definition 2.2.15 (Indexed root expressions [NÁS+24])**.** *Let $p \in \mathbb{Q}[x_1, \dots, x_{i+1}]$ with* $\text{level}(p) = i+1$*. For a given sample point $s \in \mathbb{R}^i$ we define the* indexed root expression $\text{root}_{x_{i+1}}[p, j] : \mathbb{R}^i \mapsto \mathbb{R} \cup \{\textit{undef}\}$ *as the j-th real root of p in x_{i+1}. The level of the indexed root expression is $i+1$.*

If p has less than j real roots, $\text{root}_{x_{i+1}}[p, j]$ *is undefined. For this thesis we will assume that any given indexed root expression are always defined for the given sample point. We will use the notation ξ to refer to the indexed root expression and $\xi.p$ to refer to the polynomial.*

Definition 2.2.16 (Symbolic intervals [NÁS+24])**.** Symbolic intervals *of level i are tuples of either the form $(\textit{sector}, l, u)$ or $(\textit{section}, b)$. In the sector case, l and u are indexed root expressions of level i that represent the lower and upper bounds of the interval. If the sector is unbounded in the lower or upper direction, the corresponding bounds are set to $-\infty$ and $+\infty$ respectively. For the section case, b is an indexed root expression of level i that represents the barrier of the section.*

The indexed root expressions that represent the lower and upper bounds of a sector are also called the sector-defining indexed root expressions. *The corresponding polynomials are called the* sector-defining polynomials. *Similarly, the indexed root expressions that represent the barrier of a section are called the* section-defining indexed root expressions *and the corresponding polynomials are called the* section-defining polynomial.

Symbolic intervals are used to represent the cells in the levelwise single cell construction algorithm. Each cell is represented by a sequence of symbolic intervals $R = (I_1, \dots, I_n)$, where each symbolic interval I_i bounds a variable x_i. The goal of single cell construction is thus to construct a sequence of symbolic intervals given a set of polynomials and a sample point as input.

Whether a symbolic interval is a sector or a section depends on the following condition: if there exists an indexed root expression ξ that lies on the sample point, the symbolic interval is a section, otherwise it is a sector.

2.2.5 Levelwise single cell construction

Using the definitions of indexed root expressions and symbolic intervals, we can now describe how we can construct a single cell instead of all cells at once. One simple way to achieve this is to simply project all cells as previously described and instead of lifting all cells at once, we only lift the cell containing the sample point. This is the naive approach to single cell construction as seen in [NÁS+24] and is not very efficient. It is still a good starting point to understand the levelwise single cell construction algorithm. In order to improve the efficiency of the algorithm, we need to reduce the size of the projection that we need to compute and to do this we need to carefully choose which resultants and discriminants we need to compute.

Indexed root orderings and representations

To maintain the sign-invariance of the polynomials over the cell, we need to guarantee that no polynomial crosses the cell's boundary and in order to do this we calculate the resultants of specific pairs of polynomials. When we calculate the resultants of the polynomials, and we find a common crossing point of the two polynomials (i.e. a root of the resultant), we can adjust the cell's boundary to ensure that the polynomials still remain delineable over the cell. One way to accomplish this is to calculate every possible resultant or the resultants of the sector-defining polynomials with every other polynomial. However, [NÁS+24] introduces the concept of *indexed root orderings* to give us more control over which resultants we need to calculate.

Definition 2.2.17 (Indexed root ordering [NÁS+24])**.** *Let $i \in \mathbb{N}$, and Ξ be a set of indexed root expressions of level $i+1$. An indexed root ordering on Ξ is a relation $\preceq \subseteq \Xi \times \Xi$ such that it's reflexive and transitive closure $\preceq^t$ is a partial order on Ξ. Indexed root orderings of this form are also called indexed root ordering of level $i+1$.*

We define the domain *of the indexed root ordering as the set of all indexed root expressions that appear in the ordering, i.e.* $\operatorname{dom}(\preceq) = \{\xi, \xi' \mid (\xi, \xi') \in \preceq\}$.

An indexed root ordering $\preceq$ of level $i+1$ matches a sample point s if and only if all indexed root expressions in Ξ appear in the ordering and the indexed root expressions are ordered according to the sample point, i.e. if $\xi \preceq^t \xi'$ then $\xi(s) \leq \xi'(s)$.

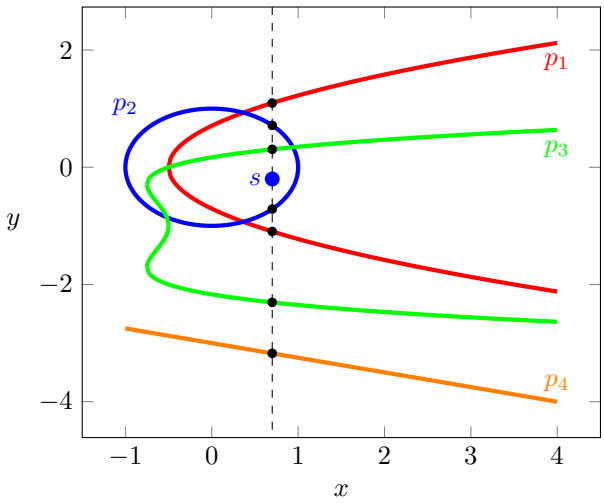


Figure 2.1: Example of a set of polynomials along with a sample point s. When computing the boundaries of the cell containing s we need to establish an ordering over the root functions of the polynomials as indicated by the dashed line and the marked roots of each polynomial on that line.

Definition 2.2.18 (Representation for Ξ [NÁS+24])**.** *A representation for Ξ with the sample point s is a tuple $(I, E, \preceq)$ where I is a symbolic interval of level i, $\preceq$ is an indexed root ordering of level i on Ξ, E is a set of equational constraints containing polynomials of level i. Representations must satisfy the following conditions:*

- *The symbolic interval I contains the sample point s,*
- *all indexed root expressions in Ξ are covered by the indexed root ordering $\preceq$ or their defining polynomials are in the set of equational constraints E,*
- *E must be the empty set if the symbolic interval I describes a sector and*
- *$\preceq$ matches $s_{[i-1]}$.*

It is important to note that when we add a polynomial to the equational constraints E we can not use this polynomial to guarantee the sign-invariance of any other polynomial, so if we include the polynomial $\xi.p$ in the set of equational constraints, we can not add any relation (ξ, ξ') or (ξ', ξ) in the indexed root ordering.

Example 2.2.1. *Suppose we want to compute the cell that contains the sample point s as shown in Figure 2.1. We must now compute the boundaries of the cell and for this to happen we decide to eliminate the variable y. To eliminate the variable y, we must compute the projection and in this case, we could compute the resultants* $\operatorname{res}_y(p_1, p_2), \operatorname{res}_y(p_1, p_3), \operatorname{res}_y(p_2, p_3)$ *and* $\operatorname{res}_y(p_3, p_4)$.

We should note that the choice of the resultants to compute is not unique For example, we could have also chosen to compute the resultants $\operatorname{res}_y(p_2, p_4)$ *instead of* $\operatorname{res}_y(p_3, p_4)$. *When running the algorithm to compute the cell, we would have a choice to make on which resultants to compute, and we can use this choice to optimize for any criteria we want.*

Connectedness

In order to maintain order-invariance we need to also maintain *connectedness* in the sector case. This is not necessary on the highest level, so we need to ensure connectedness in every projection except the first one.

To guarantee connectedness we can simply add $(I.l, I.u)$ to the indexed root ordering. However, there are cases where there are multiple possible choices for the sector-defining indexed root expressions. In [NÁS+24] this problem is solved by setting the sector-defining polynomials to be the polynomials with the lowest degrees. We will discuss this problem in Section 3.3.

The problem of choosing an optimal representation for a given set of indexed root expressions is not trivial. In many cases, there are multiple possible representations, but it is not clear which one is the best. However, the choice of indexed root ordering can have a significant impact on various aspects of the algorithm. For one, the indexed root ordering impacts the number of resultants that need to be computed and the degree of the resultants, is important because the degree of the projection grows doubly exponentially [BDE+16]. This in turn impacts the running time of the algorithm because the complexity of computing the resultants grows quadratically with the degree of the polynomials in the target variable [Duc00].

Additionally, the choice of indexed root ordering also impacts the size of the cells. Smaller cells mean that the solver may additionally check the remaining parts of the cell unnecessarily, thus wasting computation time constructing more cells as shown in [NÁS+24].

2.3 Heuristics for indexed root orderings

To optimize the running time of the algorithm we need to choose an indexed root ordering that minimizes the number of resultants and their degree while also minimizing the size of the cells. One such way to do this is to use *heuristics* to choose the indexed root ordering. The following definitions for heuristics are taken from [NÁS+24] and provide a good reference point for the later work in this thesis.

To define the indexed root orderings we only need to consider the set of indexed root expressions $\widetilde{\Xi}$ that contains for each polynomial only the indexed root expression that is closest to the sample point from below and above respectively [NÁS+24]. This is because the levelwise single cell construction algorithm already guarantees delineability of the polynomials over the cells and thus the root functions (see Definition 2.2.8) are already ordered over the current cell. Similar to [NÁS+24] we will implicitly assume that any set of indexed root expressions Ξ only contains the relevant indexed root expressions except when explicitly stated otherwise.

Definition 2.3.1 (Biggest cell representation [NÁS+24])**.** *Given a sample point s and a set of indexed root expressions Ξ with we define the indexed root ordering $\preceq_{biggest}$ on Ξ as follows:*

$$\begin{aligned}\preceq_{biggest} = &\{(\xi, \xi_{lo}) | \xi \in \widetilde{\Xi} \setminus \{\xi_{lo}\} \text{ s.t. } \xi(s) \leq \xi_{lo}(s)\} \cup \\ &\{(\xi_{up}, \xi) | \xi \in \widetilde{\Xi} \setminus \{\xi_{up}\} \text{ s.t. } \xi_{up}(s) \leq \xi(s)\}\end{aligned}$$

where ξ_{lo} and ξ_{up} are indexed root expressions that are closest to s. The set of equational constraints will be the empty set.

The idea behind the BIGGEST CELL heuristic is to choose the weakest-viable ordering such that the conditions of the indexed root ordering are still satisfied. This should in general reduce the impact of the indexed root ordering on the size of the cells, which should in turn maximize the size of the computed cells. As discussed earlier, this is important because bigger cells can potentially reduce the number of cells that have to be constructed by the algorithm and thus reduce the running time. However, as shown in [NÁS+24], when applying the BIGGEST CELL heuristic to benchmarks, the BIGGEST CELL on average calculates slightly more cells than the other heuristics. This suggests that the BIGGEST CELL heuristic does not always choose the ideal indexed root ordering to achieve the maximum cell size.

Specifically for the section case, we can use the EQUATIONAL CONSTRAINT heuristic instead of the BIGGEST CELL heuristic. For the EQUATIONAL CONSTRAINT heuristic, we simply apply the equational constraint rule to all polynomials in the set of polynomials except for the section defining polynomial. It is very similar to the BIGGEST CELL representation in that it also requires the calculation of the same set of resultants as the BIGGEST CELL representation. The difference is that the equational constraint representation is only applicable to the section case and can omit the calculation of most discriminants. This means that the equational constraint representation is strictly better than the biggest cell representation for the section case and should always be used instead.

Definition 2.3.2 (LOWEST DEGREE BARRIERS representation [NÁS+24])**.** *Given a sample point s and a set of indexed root expressions Ξ. For each $\xi \in \Xi$ we define $barrier(\xi) \in \Xi$ to be an indexed root expression ξ' such that the degree of $\xi'.p$ is minimal and $\xi(s) \leq \xi'(s) \leq s$ or $s \leq \xi' \leq \xi$ respectively. We use this to define the indexed root ordering $\preceq_{barriers}$ on Ξ as follows:*

$$\begin{aligned}\preceq_{barriers}=&\{(\xi, barrier(\xi)) \mid \xi \in \Xi \setminus \{\xi_{lo}\}, \xi(s) \leq s\} \cup \\ &\{(barrier(\xi), \xi) \mid \xi \in \Xi \setminus \{\xi_{up}\}, s \leq \xi(s)\}\end{aligned}$$

where ξ_{lo} and ξ_{up} are the indexed root expressions that are closest to s among all indexed root expressions that are below and above s respectively. The set of equational constraints will be the empty set.

The LOWEST DEGREE BARRIERS heuristic minimizes the degree of the resultants by choosing the indexed root ordering such that the resultants are of the lowest possible degree in the target variable. This is achieved by choosing a barrier between each indexed root expression and the sample point such that the degree of the resultant is minimized and adding the relation to the indexed root ordering. It is important to note that this heuristic does not guarantee overall lower degree resultants, but only lower degree resultants in the target variable for the current level. When running the LOWEST DEGREE BARRIERS heuristic on benchmarks, the heuristic on average produced higher degree resultants than the other heuristics [NÁS+24] despite aiming to minimize the degree of the resultants.

Definition 2.3.3 (CHAIN representation [NÁS+24])**.** *Given a sample point s and a set of indexed root expressions $\Xi = \{\xi_1, \ldots, \xi_n\}$. We define the indexed root ordering $\preceq_{chain}$ on Ξ according to*

$$\preceq_{chain}= \{(\xi_j, \xi_{j+1}) \mid j \in [1..k-1]\}$$

The set of equational constraints will be the empty set.

The CHAIN heuristic is a simple heuristic that chooses the indexed root ordering such that the transitive and reflexive closure of the ordering is a total order on the indexed root expressions.

Example 2.3.1. *Figure 2.2 shows a set of indexed root expressions and the indexed root ordering according to the BIGGEST CELL, CHAIN, and LOWEST DEGREE BARRIERS heuristics.*

The LOWEST DEGREE BARRIERS heuristic chooses the ordering based on the degree of the polynomials. For example, the polynomials p_1 and p_2 have a degree of 2 and 8 respectively in the target variable and thus the barrier of ξ_6 is chosen to be ξ_5. This is repeated for all indexed root expressions until we get the ordering that is shown in Figure 2.2c.

Notable is that the heurstics sometimes require us to compute resultants that are not necessary. For example, the CHAIN heuristic requires us to compute the resultant $\operatorname{res}_y(p_1, p_3)$ due to the relation $\xi_6 \preceq \xi_5$ even though we could replace this relation with $\xi_6 \preceq \xi_4$ and completely avoid the computation of this resultant.

2.4 Optimization problems and graph algorithms

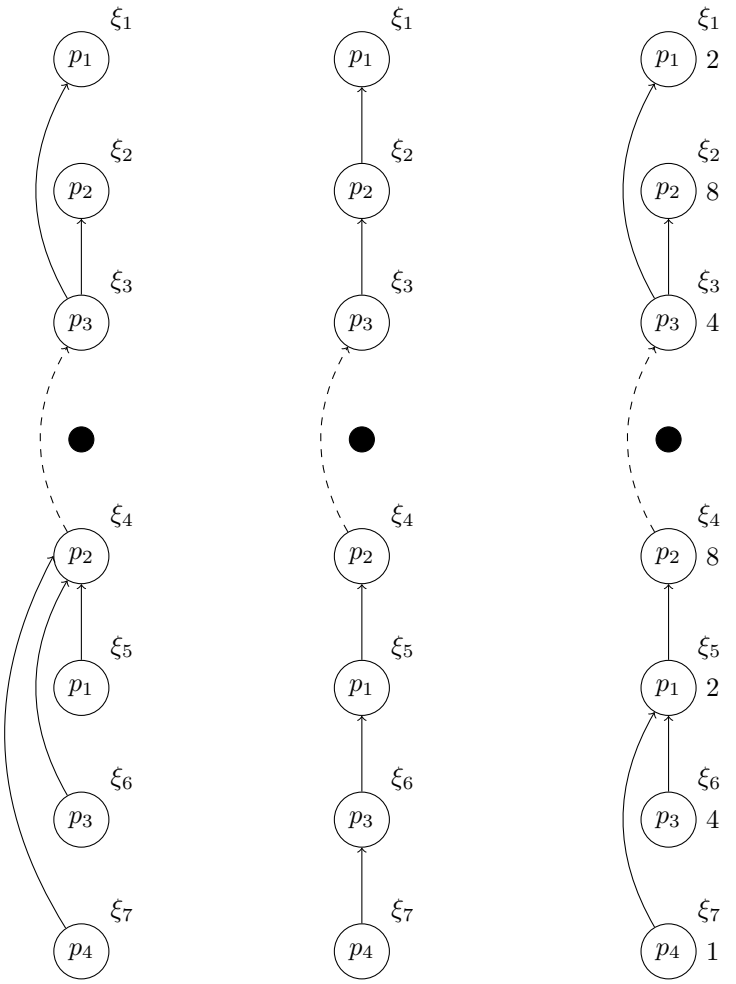
Optimization problems are mathematical challenges that involve finding the best solution from a set of possible solutions. In these problems, there is typically an *objective function* that needs to be either maximized or minimized, subject to a set of *constraints*. The objective function represents the quantity that should be optimized, such as profit, cost, time, efficiency, etc. Generally, the objective function is a mathematical function that takes the values of the *decision variables* as input and returns a real number as output. In our case, we want to find the optimal indexed root ordering that minimizes our objective function as described in Section 3.5. Decision Variables are the variables that can be adjusted or controlled to achieve the best outcome. The values of these variables influence the value of the objective function. The limitations or restrictions on the decision variables are called constraints. Constraints define the feasible region, the set of values the decision variables can take to satisfy the problem requirements. A solution that satisfies all the constraints is called a *feasible solution*. The *optimal solution* is the combination of values for the decision variables that either maximize or minimize the objective function while satisfying all the constraints. There may be one or more optimal solutions for a given optimization problem.

Combinatorial optimization is a branch of optimization that deals with problems where the solution space is discrete, and the goal is to find the best combination of elements from this finite set. Unlike continuous optimization problems, where decision variables can take any real value, combinatorial optimization involves selecting a subset or arrangement of discrete items to optimize an objective function, subject to certain constraints.

For our problem, we will model the problem as a graph problem. As such we will need to introduce some basic concepts of graph theory first.

Definition 2.4.1 (Graphs)**.** *A* graph $G = (V, E)$ *is a tuple of a set of vertices V and a set of edges $E \subseteq \{\{u, v\} \mid u, v \in V\}$.*

Graphs are a useful way to model problems that involve a set of objects and the relations between them. They are commonly used to model optimization problems in



(a) Biggest Cell heuristic (b) Chain heuristic (c) Lowest Degree Barriers heuristic

Figure 2.2: Example of the orderings of each heuristic when applied to the situation from Example 2.2.1. Each indexed root expression is represented by a node and the relations between the indexed root expressions are represented by edges. The sample point is represented by the black dot and the label on each node shows the polynomial that the indexed root expression is associated with. For the Lowest degree barriers heuristic, the degree of the polynomials is denoted next to their respective indexed root expressions.

computer science. Examples include the traveling salesman problem, the minimum spanning tree problem, the maximum flow problem, etc. In our case, we will use graphs to create a model inspired by the *minimum-weight edge cover problem* and the *minimum-weight perfect matching problem.*

Definition 2.4.2 (Minimum-weight edge cover)**.** *Given a graph $G = (V, E)$, an edge cover of G is a subset of edges $E' \subseteq E$ such that each vertex $v \in V$ is incident to at least one edge in E'.*

Given a weight function $w : E \mapsto \mathbb{R}_{\geq 0}$, we assign a weight to each edge in the graph and define the weight of an edge cover as the sum of the weights of all edges in the cover. An edge cover is called a minimum-weight edge cover if there is no other edge cover with a lower weight.

The problem of finding the minimum-weight edge cover is a well-known combinatorial optimization problem and closely related to the minimum-weight perfect matching problem.

Definition 2.4.3 (Minimum-weight perfect matching)**.** *Given a graph $G = (V, E)$, a perfect matching of G is a subset of edges $E' \subseteq E$ such that each vertex $v \in V$ is incident to exactly one edge in E'.*

Given a weight function $w : E \mapsto \mathbb{R}_{\geq 0}$, we assign a weight to each edge in the graph and define the weight of a perfect matching as the sum of the weights of all edges in the matching. A perfect matching is called a minimum-weight perfect matching if there is no other perfect matching with a lower weight.

There is a simple reduction from the minimum-weight edge cover problem to the minimum-weight perfect matching problem described by Schrijver [Sch02]: Given a graph $G = (V, E)$ and a weight function $w : E \mapsto \mathbb{R}_{\geq 0}$ we create a disjoint copy of the graph $\overline{G} = (\overline{V}, \overline{E})$. Let $G' = (V', E')$ be the new graph obtained by combining the two graphs G and $\overline{G}$. For each pair of corresponding vertices $v \in V$ and $\overline{v} \in \overline{V}$ we also add an edge $\{v, \overline{v}\} \in E'$ to the graph. We set the weight of that edge to twice the minimum weight of all edges that are incident to v in the original graph G. Given this reduction we can then find a minimum-weight perfect matching M for the newly created graph and recreate the minimum-weight edge cover from the original graph as follows: Remove all edges $M \cap \overline{E}$ and for each edge $(v, \overline{v}) \in M$ we will add an edge with the minimum weight that is incident to v in the original graph G to the edge cover. The resulting edge cover will be a minimum-weight edge cover for the original graph G with half the total weight of the minimum-weight perfect matching.

Example 2.4.1. *Let us consider the graph in Figure 2.3. The nodes v_1, v_2, and v_3 are the nodes of the original graph G and the nodes $\overline{v_1}$, $\overline{v_2}$, and $\overline{v_3}$ are the nodes of the disjoint copy $\overline{G}$. The edges between the nodes and their corresponding nodes in the disjoint copy are the edges that we added to the new graph G'.*

Since the minimum weight of all edges incident to v_1 is 3, we set the weight of the edge $\{v_1, \overline{v_1}\}$ to 6. Similarly, we set the weight of the edge $\{v_2, \overline{v_2}\}$ to 6 and the weight of the edge $\{v_3, \overline{v_3}\}$ to 10. After finding a minimum-weight perfect matching for the graph we can then recreate the minimum-weight edge cover for the original graph G: We remove the edges $\{v_1, \overline{v_1}\}$ and $\{v_3, \overline{v_3}\}$ from the set and add the edges $\{v_2, v_3\}$ to the edge cover because the edge $\{v_3, \overline{v_3}\}$ was part of the minimum-weight perfect matching.

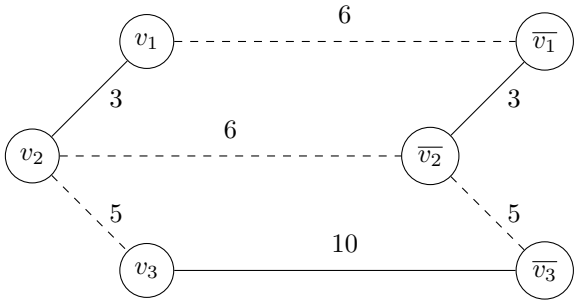


Figure 2.3: Constructed graph from the example. Dashed edges indicate that they are not in the minimum-weight perfect matching.

The problem of finding a minimum-weight perfect matching is a well-researched problem and there are algorithms that can solve it in polynomial time in general graphs [CR99]. If needed, there are also approximation algorithms that can find a minimum-weight perfect matching in polynomial time with a guaranteed approximation factor [DPS18].

Chapter 3

Optimization model

In this chapter, we will first introduce the optimization problem and present a model to solve the optimization problem. Specifically, we will examine three different cases:

1. The unrestricted set Ξ of indexed root expressions. This variant is NP-hard, and we will shortly describe why we will not consider it further.

2. The restricted set $\widetilde{\Xi}$ of indexed root expressions where there are no intersections between the indexed root expressions. This case can be solved in polynomial time using the model we present.

3. The restricted set $\widetilde{\Xi}$ where the indexed root expressions may intersect. For this case, we not yet know if the problem is NP-hard or not. We will show how we address this case in the optimization procedure and show some ways we explored to solve the problem.

We will then go over some things to consider for the sector and section cases and show how we address these in the optimization procedure. Finally, we will discuss a variety of cost functions that can be used to determine the cost of a resultant.

3.1 Modelling the optimization problem

Let Ξ be a set of indexed root expressions and $P \subset \mathbb{Q}[x_1, \ldots, x_n]$ along with a sample point s. To define the optimization problem we need to define a graph $G = (V, E)$ along with a weight function $w : E \to \mathbb{R}_{\geq 0}$. We define the set of vertices as follows:

$$V = \{v_\xi \mid \exists \xi' : \xi(s) < \xi'(s) \leq s \text{ or } s \leq \xi'(s) < \xi(s)\} \cup \{d, d'\}$$

Note that we only consider indexed root expressions $\xi \in \Xi$ in the model if there exists another indexed root expression ξ' that is between the sample point s and ξ. If there is no such indexed root expression we do not need to consider the indexed root expression in the model since that indexed root expression must be the sector-/section defining indexed root expression and is therefore trivially covered by the reflexive and transitive closure of the ordering. We also add two dummy vertices d and d' to the set of vertices to ensure that we can always construct a minimum-weight edge cover.

Based on the set of vertices we will define the set of edges E. To accomplish this we will first define some auxiliary sets:

$$\Xi_{p,q} = \{\xi \in \Xi \mid \xi.p \in \{p, q\}\}$$

The set $\Xi_{p,q}$ refers to indexed roots of the polynomials p and q. Additionally, we define the sets $\Xi^l_{p,q}$ and $\Xi^u_{p,q}$ as follows:

$$\Xi^l_{p,q} = \{\xi \in \Xi_{p,q} \mid \exists \xi' \in \Xi_{p,q} \setminus \{\xi\} : \xi(s) < \xi'(s) \leq s\}$$
$$\Xi^u_{p,q} = \{\xi \in \Xi_{p,q} \mid \exists \xi' \in \Xi_{p,q} \setminus \{\xi\} : s \leq \xi'(s) < \xi(s)\}$$

The set $\Xi^l_{p,q}$ refers to the outermost indexed roots of the polynomials p and q that are below the sample point s. This set only contains an indexed root ξ if there exists another indexed root ξ' that is between itself and s. The set $\Xi^u_{p,q}$ is defined similarly and refers to the indexed roots that are above the sample point s. Using these two sets we can then define the set of edges:

$$\begin{aligned} E = &\{\{v_\xi, v_{\xi'}\} \mid p, q \in P, p \neq q, \xi \in \Xi^l_{p,q}, \xi' \in \Xi^u_{p,q}\} \cup \\ &\{\{v_\xi, d\} \mid \xi \in \Xi^l_{p,q}, \nexists \xi' \in \Xi^u_{p,q}\} \cup \\ &\{\{d, v_\xi\} \mid \xi \in \Xi^u_{p,q}, \nexists \xi' \in \Xi^l_{p,q}\} \cup \\ &\{d, d'\} \end{aligned}$$

The definition of the edges is split into four subsets, where each subset has the following meaning:

1. The first set of edges describes the resultants $\text{res}_{x_i}(p, q)$ for two polynomials $p, q \in P$ with $p \neq q$ for which the indexed root expressions $\xi \in \Xi^l_{p,q}$ and $\xi' \in \Xi^u_{p,q}$ exist. This means that the resultant $\text{res}_{x_i}(p, q)$ can be used to cover the indexed root expression ξ below the sample point and the indexed root expression ξ' above the sample point.

   It is important to note that due to the definition of $\widetilde{\Xi}$ we know that each resultant can cover at most two indexed root expressions, so we can always find a suitable edge for each resultant.

2. In the case that there is no $\xi' \in \Xi^u_{p,q}$ we add an edge from $\xi \in \Xi^l_{p,q}$ to the dummy vertex d. This case indicates that the resultant $\text{res}_{x_i}(p, q)$ only protects the indexed root expression ξ below the sample point.

3. In the case that there is no $\xi \in \Xi^l_{p,q}$ we add an edge from the dummy vertex d to $\xi' \in \Xi^u_{p,q}$. This case indicates that the resultant $\text{res}_{x_i}(p, q)$ only protects the indexed root expression ξ' above the sample point.

4. Since the dummy vertex d must also be covered by the edge cover we add an edge from d to the other dummy vertex d'.

Lastly we need to define the weight function $w : E \to \mathbb{R}_{\geq 0}$ that assigns a real value to each edge in the graph. For this we will assume that we have a cost function $c : P \times P \to \mathbb{R}_{\geq 0}$ that assigns a cost to each pair of polynomials, which will correspond to the cost of the resultant of both polynomials. We define the weight function as:

$$w(e) = \begin{cases} c(p, q) & \text{if } e = \{v_\xi, v_{\xi'}\}, (p, q) \in \arg\min_{p,q \in P}\{c(p, q) \mid \xi \in \Xi^l_{p,q}, \xi' \in \Xi^u_{p,q}\} \\ c(p, q) & \text{if } e = \{v_\xi, d\}, (p, q) \in \arg\min_{p,q \in P}\{c(p, q) \mid \xi \in \Xi^l_{p,q} \text{ or } \xi \in \Xi^u_{p,q}\} \\ 0 & \text{otherwise} \end{cases}$$

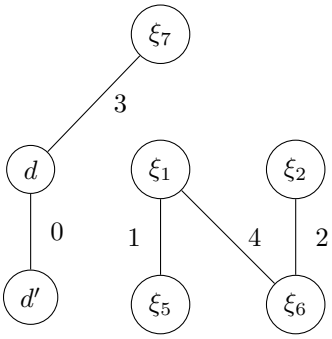


Figure 3.1: Example of model when applied to Example 2.2.1.

Since each edge in the graph corresponds to a resultant of two polynomials $p, q \in P$ we can simply use the cost function c to assign a weight to each edge. However, we need to keep in mind that two resultants can potentially cover the same indexed root expressions, and thus we will always pick the resultant with the lower cost. The only exception is the edge $\{d, d'\}$ since that edge is only needed to ensure correctness of the model and is intended to be free of cost since it does not correspond to a resultant.

Note that we do not place any requirements on the cost function c other than that the cost of a resultant should be non-negative. This is done to allow for the cost function to be defined in any way that is suitable for the specific problem at hand. For example, in the implementation we could use the cost function to assign a cost of 0 to resultants that have already been computed in the past. The cost function could potentially be restricted to find a better model for the optimization problem in future work.

Example 3.1.1. *When applying the model to the indexed root expressions as seen in Example 2.3.1 we get the graph as seen in Figure 3.1. In this example we only have to consider the indexed root expressions* $\xi_1, \xi_2, \xi_5, \xi_6, \xi_7$ *since the indexed root expressions* ξ_3 *and* ξ_4 *are the only possible choice for the sector-defining indexed root expressions and are therefore trivially covered by the reflexive and transitive closure of the ordering. To construct the edges we have to consider the following resultants:*

- $\mathrm{res}_y(p_1, p_2)$ *protects* ξ_1 *and* ξ_5*, and we will set the cost to 1.*
- $\mathrm{res}_y(p_1, p_3)$ *protects* ξ_1 *and* ξ_6*, and we will set the cost to 4.*
- $\mathrm{res}_y(p_2, p_3)$ *protects* ξ_2 *and* ξ_6*, and we will set the cost to 2.*
- $\mathrm{res}_y(p_2, p_4)$ *protects* ξ_7*, and we will set the cost to 3. For this indexed root expression we could consider multiple resultants, but we will assume that* $\mathrm{res}_y(p_2, p_4)$ *is the resultant with minimal cost.*

For each of these resultants we will add an edge to the graph with the corresponding cost. After solving the optimization problem we get the resultants $\mathrm{res}_y(p_1, p_2), \mathrm{res}_y(p_2, p_3)$ *and* $\mathrm{res}_y(p_2, p_4)$ *as the optimal solution. Using these resultants we can construct the indexed root ordering as* $\preceq = \{(\xi_7, \xi_4), (\xi_6, \xi_4), (\xi_5, \xi_4), (\xi_3, \xi_2), (\xi_2, \xi_1)\}$.

Note that the sets do not consider pairs of indexed roots that intersect even though

we need to consider them in order to get an optimal ordering. To solve this we have opted to simply try the different pairs of indexed roots in the implementation, i.e. we construct the graph assuming the cases $\xi(s) < \xi'(s)$ and $\xi'(s) < \xi(s)$ respectively, solve the minimum-weight edge cover and choose the one with the lower cost.

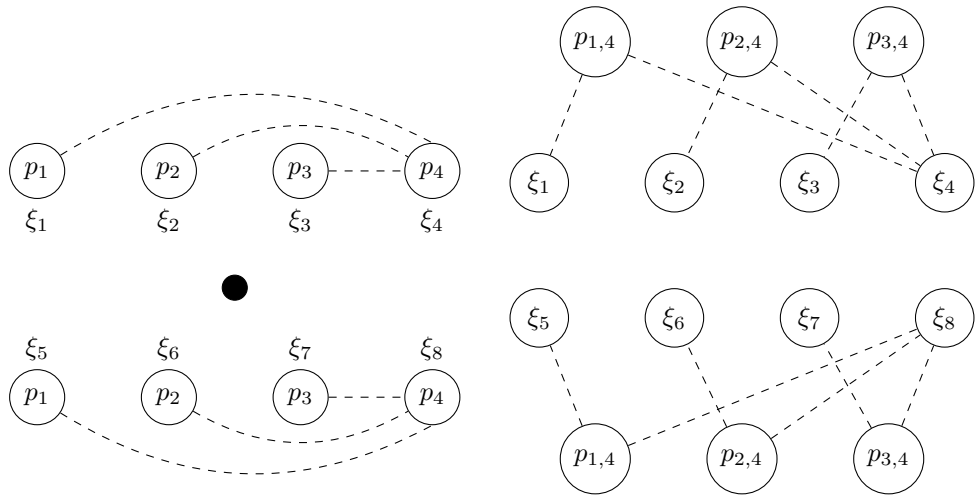
We have tried to find solutions by using a more sophisticated approach, but we have not been able to find a solution that is both correct and efficient. The problem is that when we consider intersecting indexed root expressions in the model we could potentially get an ordering, where two or more intersecting indexed root expressions are covered by the same resultant. One approach that we have tried is to first construct a minimum-weight directed spanning tree of the intersecting indexed roots below and above the sample point. Each edge would be interpreted as a resultant that could potentially be included in the optimal solution. We would then include these resultants in the reduction to the minimum-weight perfect matching as follows: Each resultant would add two vertices to the graph and connect the indexed root expressions that are covered by the resultant to the corresponding vertices. The idea behind this approach is that the minimum-weight directed spanning tree will filter out resultants that would not be included in the optimal solution anyway. Without these additional resultants we could construct orderings without loops in the ordering, i.e. situations where intersecting indexed root expressions cover each other circularly.

Unfortunately, this approach does not work in cases where two polynomials appear in the same layer on one side of the sample point and in different layers on the other side of the sample point since the minimum-weight directed spanning tree may filter out resultants that are needed for the optimal solution.

Example 3.1.2. *In this example we will consider the indexed root expressions and sample point as seen in Figure 3.2a. The graph in Figure 3.2b shows how the described approach would work on this instance. We first construct a minimum-weight directed spanning tree of the indexed roots and then add the resultants of that tree to the graph as shown in Figure 3.2a. Here the minimum-weight directed spanning tree is constructed by adding the edges* $\{\xi_1, \xi_4\}, \{\xi_2, \xi_4\}$ *and* $\{\xi_3, \xi_4\}$ *to the graph. This means that the resultants* $\operatorname{res}_{x_i}(p_1, p_4), \operatorname{res}_{x_i}(p_2, p_4)$ *and* $\operatorname{res}_{x_i}(p_3, p_4)$ *must also be considered in the optimization algorithm. For each resultant that we add additional vertices to the graph and each indexed root expression that would be covered by the resultant is connected to the corresponding vertices.*

Unfortunately, the example is rather simple, and this approach does not work in cases such as the one seen in Figure 3.3. Since the indexed root expressions of the polynomials are not intersecting with the indexed root expression of polynomial p_4 *below the sample point, we do not get the same directed minimum-weight spanning tree as above. Instead, we could, for example, get the resultants* $\operatorname{res}_{x_i}(p_2, p_1), \operatorname{res}_{x_i}(p_2, p_3)$ *and* $\operatorname{res}_{x_i}(p_1, p_4)$ *as the minimum-weight directed spanning tree. However, this would also mean that we again have to many resultants in the graph, and we could once again get a circular ordering if all resultants are included in the minimum-weight perfect matching.*

If we try each pair of indexed roots in the implementation we will get the correct result, but the runtime of the algorithm will be exponential in the number of indexed roots. In the evaluation (see Chapter 4) we will investigate how the naive approach performs in practice. For the future, it would be interesting to investigate if the problem has a polynomial time algorithm or if it is NP-hard.



(a) Example instance with indexed root expressions. The dashed edges between indexed root expressions show how the directed minimum-weight spanning tree could be constructed. Here the directions of the edges are omitted for simplicity.

(b) Example construction based on the instance shown in Figure 3.2a. This graph shows how the model would be modified in the minimum-weight perfect matching. Other parts of the reduction are omitted because they are not required to show how this approach works.

Figure 3.2: Example of an ideal case for the describe approach using minimum-weight directed spanning tree.

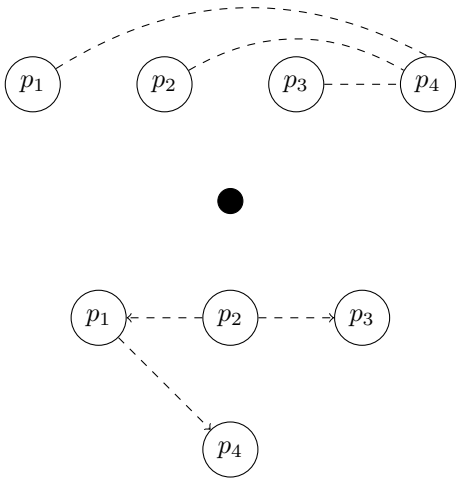


Figure 3.3: Example of a problematic set of indexed root expressions with multiple intersections on either side.

Lastly, we want to highlight that this model only works for the restricted set $\widetilde{\Xi}$ of indexed root expressions. If we instead consider the unrestricted set Ξ of indexed root expressions where each polynomial can appear multiple times above and below the sample point, the problem becomes NP-hard. This is because our model assumes that each resultant covers at most two different indexed root expressions, and we use this assumption to construct the edges of the graph. However, in the unrestricted set Ξ a resultant can cover any number of indexed root expressions. In this case we can prove that the problem is NP-hard by reducing the SAT problem to this problem similar to the reduction in Section 3.4.

As described in Section 2.3, we do not need to consider the unrestricted set Ξ because delineability ensures that the restricted set $\widetilde{\Xi}$ is sufficient to construct the ordering. For this reason we will not consider the unrestricted set Ξ further in this thesis and outline how this can be proven:

3.2 Correctness of the model

Given a set of indexed root expressions Ξ with respective polynomials P, a sample point s and a solution to the corresponding minimum-weight edge cover instance $F \subseteq E$, let R be the set of resultants that correspond to the edges in F. We now create an indexed root ordering as defined in Definition 2.2.18 based on F:

$$\begin{aligned}\preceq = \{(\xi, \xi') \mid res_{x+1}(p,q) \in R, \xi.p = p, \xi'.p = q, \xi(s) < \xi'(s) \leq s\} \cup \\ \{(\xi', \xi) \mid res_{x+1}(p,q) \in R, \xi'.p = p, \xi.p = q, s \leq \xi'(s) < \xi(s)\}\end{aligned}$$

To prove that this indexed root ordering is a valid solution we need to prove that it is a valid indexed root ordering:

Proof. To prove that each indexed root is protected by the ordering, we will use induction on the distance of the indexed root to the sample point s: Here we define the distance of an indexed root ξ to the sample point s as the number of root functions or intersections between itself and s. The base case are the indexed roots that are closest to s. These indexed roots are the boundaries of the constructed cell and are therefore trivially protected by the ordering. By the construction of the model, each vertex in the graph is incident to at least one edge and as such, for each indexed root, there exists a resultant $res_{x+1}(p,q)$ such that $\xi.p = p$ or $\xi.p = q$. There must also exist another indexed root ξ' such that $\xi'.p = p$ or $\xi'.p = q$. Depending on the ordering $\xi_1(s) < \cdots < \xi_n(s)$ of the indexed roots we have two cases:

- If $\xi(s) < \xi'(s) \leq s$: Due to the induction hypothesis we know that $\xi' \preceq^t I.l$ and because of $\xi \preceq \xi'$ we also know that $\xi \preceq^t I.l$.

- If $s \leq \xi'(s) < \xi(s)$: Due to the induction hypothesis we know that $I.u \preceq^t \xi'$ and because of $\xi' \preceq \xi$ we also know that $I.u \preceq^t \xi$.

In both cases we have shown that $\xi \preceq^t I.l$ and $I.u \preceq^t \xi$ and therefore ξ is covered by the ordering. This proof works similarly for the case where the given symbolic interval is a section instead of a sector. □

3.3 Maintaining connectedness

As described in Section 2.2.5, we sometimes need to ensure that the cells that are computed are connected. We have to ensure this for the lower levels, i.e. for every projection except the top most one. To ensure that the cells are connected, we will use a simple heuristic:

After computing the indexed root ordering for the current level, we will check if we need to maintain connectedness. If we do, we will first check if the model contains a resultant that we can use for this purpose and if it does, we can simply add the corresponding relation to the ordering. If we do not have a resultant that we can use, we go through each possible resultant that we can use and select the one that has the lowest cost. After selecting the resultant, we add the corresponding relation to the ordering. This heuristic is not guaranteed to find the optimal solution, but it should be able to find a solution that is good enough for most cases.

It is also possible to compute an optimal ordering that also guarantees connectedness in polynomial time since there are $O(P^2)$ possible resultants that we can use, and we could simply try all of them and then apply the optimization algorithm to find the optimal ordering. Since the optimization algorithm is solvable in polynomial time and the number of possible resultants is polynomial with respect to the number of indexed root expressions n we can also solve this problem in polynomial time. However, since this would require us to solve the optimization problem multiple times, it would be less efficient, and we will not use this approach in the implementation.

3.4 Integrating equational constraints

As discussed in Section 2.2.3, we can use equational constraints to further reduce the complexity of CAD procedure. As such, we want to investigate how we can integrate equational constraints into the optimization algorithm.

First, we want to show that the problem of finding the optimal indexed root ordering and equational constraints is NP-complete. To accomplish this, we will reduce the Satisfiability (SAT) problem to the Optimal-ordering-EC problem. We define the SAT problem as follows:

Satisfiability (SAT)

Input: A Boolean formula in CNF over the set of variables $X = \{x_1, \dots, x_n\}$ with the set of clauses $C = \{c_1, \dots, c_m\}$.

Question: Is there an interpretation $\alpha : AP \mapsto \{0, 1\}$ such that all clauses in C are satisfied?

To reduce the SAT problem to the Optimal-ordering-EC problem, we need to define the optimization problem as a decision problem:

Optimal-ordering-EC

Input: Set of indexed root expressions Ξ, set of polynomials P, sample point s, cost functions $c_{res} : P \times P \mapsto \mathbb{R}_{\geq 0}$ and $c_{disc} : P \mapsto \mathbb{R}_{\geq 0}$, cost limit $k \in \mathbb{N}$.

Question: Is there an indexed root ordering and a set of equational constraints that fulfill the constraints of Definition 2.2.18 such that the sum of the costs of all resultants and discriminants is $\leq k$?

Using these definitions we can now prove that Optimal-ordering-EC is NP-hard:

Proof. Given an instance of the SAT problem with a set of variables $X = \{x_1, \ldots, x_n\}$ and a set of clauses $C = \{c_1, \ldots, c_m\}$ we will construct an instance of the Optimal-ordering-EC problem. For each variable $x_i \in X$ we define the literal $\overline{x_i}$ as the negation of x_i. First we will define the set of indexed root expressions:

$$\Xi = \{\xi^u_{x_i}, \xi^u_{\overline{x_i}}, \xi^l_{x_i}, \xi^l_{\overline{x_i}}, \xi_{x_i} \mid x_i \in X\} \cup \{\xi_{c_j} \mid c_j \in C\} \cup \{\xi_s\}.$$

For each variable $x_i \in X$ we add the indexed root expressions $\xi^u_{x_i}, \xi^u_{\overline{x_i}}, \xi^l_{x_i}, \xi^l_{\overline{x_i}}$ and ξ_{x_i}. These indexed roots will be used to represent whether a variable will be assigned the value 0 or 1. We also add an indexed root expression ξ_{x_i} for each variable x_i to represent the variable itself and to ensure that we pick either x_i or $\overline{x_i}$. Furthermore, we add an indexed root expression ξ_{c_j} for each clause $c_j \in C$ to represent the clause itself. The last indexed root expression ξ_s is used to represent the section defining indexed root.

We also need to define the set of polynomials P and associate each indexed root expression with a polynomial:

$$P = \{p_{x_i}, p_{\overline{x_i}}, q_{x_i} \mid x_i \in X\} \cup \{p_{c_j} \mid c_j \in C\} \cup \{p_s\}.$$

For each variable $x_i \in X$ we assign the polynomial p_{x_i} to the indexed root expressions $\xi^u_{x_i}, \xi^l_{x_i}$, and the polynomial $p_{\overline{x_i}}$ to the indexed root expressions $\xi^u_{\overline{x_i}}, \xi^l_{\overline{x_i}}$. The polynomial q_{x_i} is assigned to the indexed root expression ξ_{x_i}.

The cost function $c_{res} : P \times P \to \mathbb{R}_{\geq 0}$ is defined as follows:

$$c_{res}(p, q) = \begin{cases} 0 & \text{if } p = p_{x_i} \text{ and } q = p_{c_j} \text{ and clause } c_j \text{ contains literal } x_i \\ 0 & \text{if } p = p_s \text{ and either } q = p_{x_i} \text{ or } q = p_{\overline{x_i}} \\ +\infty & \text{otherwise} \end{cases}$$

For each polynomial p we define the cost of the discriminant to be $c_{disc}(p) = 1$ the cost limit will be set to $k = n$.

Example 3.4.1. *Let $\varphi = (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3)$ be an instance of SAT. Based on this instance we can construct an instance of the Optimal-ordering-EC problem: The graph in Figure 3.4 shows an example indexed root ordering along with the set of equational constraints E. For each variable x_i we have to add either p_{x_i} or $p_{\overline{x_i}}$ to the set of equational constraints since otherwise we would have to add the costs of both discriminant $disc(p_{x_i})$ and $disc(p_{\overline{x_i}})$, which would exceed the cost limit. Whenever we add the polynomial of a literal to E we also have to add the polynomial of the opposite literal to the indexed root ordering because the ordering must also cover the indexed root expressions ξ_{x_i}.*

We will now show that the SAT instance is satisfiable if and only if the Optimal-ordering-EC instance has a solution.

"⇒": Let $\alpha : AP \mapsto \{0, 1\}$ be an interpretation that satisfies the SAT instance. Now we will construct a solution to the Optimal-ordering-EC instance based on the interpretation:

$$E = \{p_{x_i} \mid \alpha(x_i) = 0\}.$$

Based on E we can define the partial order $\preceq$ as follows:

$$\preceq = \{(\xi_s, \xi) \mid \xi.p \notin E\} \cup \{(\xi, \xi') \mid \xi.p \notin E, \xi(s) \leq \xi'(s)\}$$

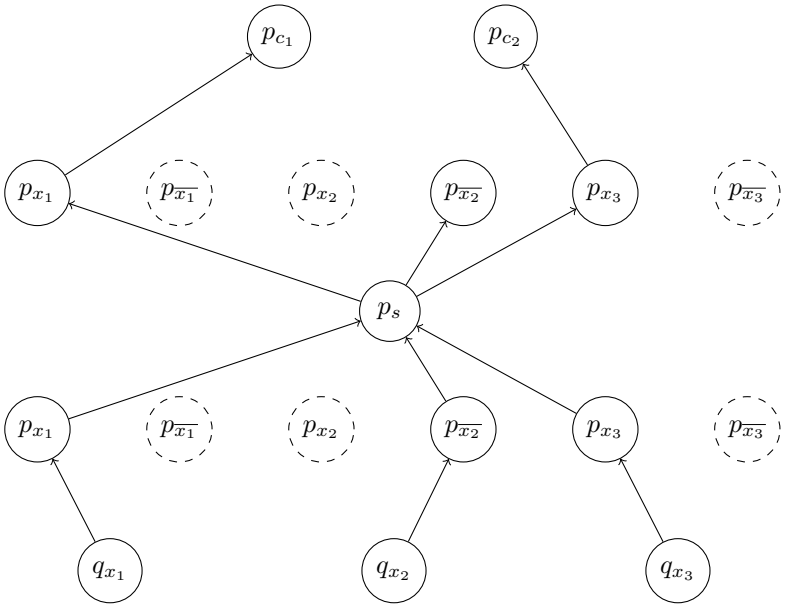


Figure 3.4: Constructed graph based on the example. Each node represents an indexed root expression and the label of each node is the associated polynomial. Relations between indexed root expressions are indicated by directed edges between the nodes while the nodes whose polynomials are in E are indicated by dashed outlines.

Due to the construction we know that for each variable x_i we must have either $p_{x_i} \in E$ or $p_{\overline{x_i}} \in E$ while the literal that is not in E is protected by the ordering, and we therefore know that we can also cover the indexed root expressions ξ_{x_i} with cost 0.

If the interpretation α satisfies the SAT instance then we also know that indexed root expressions representing the clauses are also protected by the ordering. The constructed solution is therefore valid and has a cost of at most n because we need to add the cost of exactly three discriminants.

"$\Leftarrow$": Let $(\preceq, E)$ be a solution to the Optimal-ordering-EC instance. We will now construct an interpretation $\alpha : AP \mapsto \{0, 1\}$ for the SAT instance based on the solution:

$$\alpha(x_i) = \begin{cases} 1 & \text{if } I.u \preceq \xi^u_{x_i} \\ 0 & \text{otherwise} \end{cases}$$

Since the cost of the polynomial of the section defining indexed root expression and the polynomial for each clause c_j is $+\infty$, we know that $p_{c_j} \notin E$. Due to this we know that for each indexed root expression ξ_{c_j} there must exist an indexed root expression $\xi^l_{x_i}$ such that $\xi^l_{x_i} \preceq xi_{c_j}$. We also know that $\xi_s \preceq \xi^l_{x_i}$ since otherwise the ordering would not be valid and that the literal x_i must be in the clause c_j due to the definition of the cost function. This means that the interpretation α satisfies all clauses in the SAT instance.

Each indexed root expressions ξ_{x_i} must also be covered by the ordering and therefore there must exist an indexed root expression $\xi^l_{x_i}$ such that $\xi_{x_i} \preceq \xi^l_{x_i}$.

This means that for each variable x_i we must have either $p_{x_i} \in E$ or $p_{\overline{x_i}}$ since otherwise the cost of the discriminants would exceed the cost limit. The assignment is therefore consistent and if the Optimal-ordering-EC instance has a solution with a cost of at most n the SAT instance must also be satisfiable.

Since the reduction is polynomial and the SAT problem is NP-hard the Optimal-ordering-EC problem is also NP-hard. □

Since the Optimal-ordering-EC problem is NP-hard we will probably not be able to find an optimal solution in polynomial time, and we will instead rely on a heuristic to find a good set of equational constraints. Once we have found an optimal indexed root ordering we will construct the set of equational constraints based on the ordering:

$$\begin{aligned} E = &\{\xi.p \mid \xi \in \mathrm{dom}(\preceq), (\xi_s, \xi) \in \preceq \text{ and } \nexists \xi' \in \mathrm{dom}(\preceq) : (\xi, \xi') \in \preceq\} \cup \\ &\{\xi.p \mid \xi \in \mathrm{dom}(\preceq), (\xi, \xi_s) \in \preceq \text{ and } \nexists \xi' \in \mathrm{dom}(\preceq) : (\xi', \xi) \in \preceq\}. \end{aligned}$$

This set of equational constraints will ensure that the ordering is valid and optimized for the cost of resultants while still reducing the number of discriminants.

3.5 Resultant cost metrics

In this chapter we will discuss the different cost metrics that can be used to determine the cost of a resultant. The cost of a resultant is an important factor in the optimization problem since it determines the weight of the edges in the graph. The weight of the edges is used to determine the optimal edge cover, which in turn determines the optimal set of resultants to compute. The cost of a resultant can be determined by different factors, such as the total degree of the resultant, the number of variables, the number of monomials, the number of resultants, and the feature based cost. In this chapter we will discuss these different cost metrics and how they can be used to determine the cost of a resultant.

Total degree upper bound (Tdub)

As explained in Chapter 3, the total degree of a resultant is an important factor in the running time of the CAD algorithm and as such it is a good metric to use for the cost of a resultant. Since we do not know the exact total degree of the resultant we can use an upper bound for the total degree as an approximation for the total degree of the resultant. The total degree of a resultant is bounded by the product of the total degrees of the input polynomials. This upper bound can be derived from Bézout's theorem.

Sum of total degree (Sotd)

As described in [DSS04], we define the cost of a resultant as follows: For both input polynomials we calculate the total degree of each monomial and sum these total degrees. The cost of the resultant is then the sum of these two numbers.

In the short term, this cost metric may not produce resultants with the lowest total degree. However, since this cost metric looks at all monomials of the input polynomials we hope that in the long term, it should hopefully produce resultants with a lower average total degree.

Number of variables (Nv)

Setting the cost of the resultant to the number of variables in the resultant means that we can potentially reduce the number of elimination steps in the CAD algorithm. This could be beneficial since the runtime of the CAD algorithm is dominated by the number of elimination steps.

The number of variables in the resultant can be determined by taking the size of the union of the variables of the two input polynomials and subtracting one since we are eliminating one variable.

Number of monomials (Nm)

The best upper bound for the number of monomials we found is presented in [Kal93] and depends on the degrees of the input polynomials. Unfortunately, the number of monomials in the resultant can quickly explode and as such the upper bound can return vastly different results even for polynomials of similar degrees.

For this reason we instead decided to simply sum the number of monomials in the input polynomials. This approach is simpler to compute and makes the cost of a resultant more predictable and comparable.

Number of resultants (Nr)

By setting the cost of a resultant to 1 we can use the number of resultants as the cost of a resultant. This could be beneficial since we can potentially reduce the number of resultants that need to be computed.

Feature based (Fb)

In order to reduce the running time of the CAD algorithm, we can choose the variable ordering for the projection and lifting phases [DSS04] and finding better variable orderings is an active area of research. Some statistical analysis of this problem has been done in [PdEC24] and presents some metrics that can be used to construct efficient variable orderings. The metrics presented in [PdEC24] all use simple operations such as the sum, average, or maximum of some set of polynomials.

In our case we will adapt one of these metrics to define the cost of a resultant. Specifically, the set of polynomials S will be our input polynomials, and we will use $sum(max(v_i(S)))$ as our cost, which is the sum of the maximum exponent of the variable x_i in the polynomials in S. There are many more possible combinations of operations as shown in [PdEC24], but we will only use the above-mentioned metrics.

Variable depth (Vd)

Similar to minimizing the number of variables, we can also minimize the level of the resultant of the input polynomials. Here the level of a polynomial is defined as the maximum level of any variable in the polynomial.

Total degree exact (Tde)

For experimental purposes we will also use the exact total degree of the resultant by simply calculating the resultant and taking the total degree of the resultant. This is not a practical cost metric since it requires computing the resultant, but we will use

this metric to see what effect the exact total degree has on the number of cells and size of the cells.

Chapter 4

Evaluation

As part of this thesis, we implemented the discussed algorithm variants and tested them to evaluate their performance. Specifically, we implemented the algorithm to compute the optimal indexed root ordering along with the different cost metrics as discussed in Section 3.5. For this purpose, we implemented the algorithms in C++ as part of the SMT-RAT solver [CKJ+15]. SMT-RAT is a state-of-the-art SMT solver that focuses on real arithmetic. Most importantly, SMT-RAT already contains an implementation of the levelwise algorithm for computing CADs. The solver is also highly optimized and allows for easy integration of new algorithms. Each of the cost metrics is represented by a separate strategy in the solver, which allows for easy switching between the different cost metrics. Additionally, we will also evaluate the performance of the Biggest Cell (BC) and Lowest Degree Barrier (LDB) heuristics (see Definition 2.3.1 and Definition 2.3.2 respectively).

The benchmarks were run on the QF_NRA benchmark set from the SMT-LIB benchmark library [BFN+24]. This benchmark set was created using the SMT-LIB language for the purpose of having a standardized set of benchmarks to evaluate the accuracy and performance of SMT solvers. As of the time of writing, the QF_NRA benchmark set contains 12134 benchmarks that are taken from real-world problems in other research areas. We therefore consider this benchmark set to be a good representation of the problems that are typically solved using SMT solvers and have decided to use it for our evaluation.

To evaluate the performance of the algorithms on the benchmarks, we ran the SMT-RAT solver using the benchmax tool[1] with a time limit of 1 min and a memory limit of 4 GB on the RWTH High Performance Computing Cluster. Specifically, we ran the solver on the CLAIX-2018 HPC segment, which consists of 1243 nodes with 2x Intel Xeon Platinum 8160 processors each[2]. Besides the total running time of a benchmark, we measured different parts of the algorithm in microseconds since milliseconds are not precise enough to measure some parts of the implementation.

We used the statically linked version of the SMT-RAT solver for the evaluation with the build type set to `RELEASE` to ensure optimal performance. In the following sections, we present the results of the evaluation and we will also discuss the bottlenecks of the algorithm more broadly.

[1] https://ths-rwth.github.io/smtrat/dd/d0f/benchmax.html
[2] https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/

	sat	unsat	wrong	timeout	memout	segfault	solved
BC	5123	5006	0	1711	103	191	10129
LDB	5120	5005	0	1700	114	195	10125
FB	5121	5009	0	1684	103	217	10130
NM	5122	5006	0	1703	99	204	10128
NR	5119	5009	0	1693	103	210	10128
NV	5123	5009	0	1691	92	219	**10132**
SOTD	5119	5007	0	1724	91	193	10126
TDE	5123	5008	0	1708	108	187	10131
TDUB	5120	5009	0	1702	91	212	10129
VD	5119	5007	0	1703	112	193	10126

Table 4.1: Number of answers returned by each strategy. The number wrong denotes instances where the strategy returned either sat or unsat while the correct answer was the opposite. The number timouts, memouts and segfaults are instances where the solver could not solve the benchmark due to a lack of time, memory or an error during execution. The number solved shows the number of benchmarks where the solver returned either sat or unsat.

4.1 Result overview

We first want to look at the results of the benchmarks and check if the algorithm is able to solve the benchmarks correctly: From the results in Table 4.1, we can see that the algorithm was able to solve the majority of the benchmarks within the time limit. Out of those that were solved, the algorithm was able to solve all of them correctly.

We can see that all strategies have a high number of segfaults, timeouts and memouts. When looking at the output of the solver, we can see that the segfaults are caused by the solver running out of memory and are misreported as segfaults. The number of timeouts and memouts is expected since some benchmarks are very hard to solve and require more resources to solve. Similar tests (like the ones in [NÁS+24]) have shown that these numbers are not unusual when testing on the QF_NRA benchmark set.

Most importantly, the results show that the new strategies are generally not able to solve more benchmarks than the original strategies. In some cases, the new strategies solve a few more benchmarks than the original strategies while in other cases they solve fewer benchmarks.

4.2 Performance profiling

Next, we want to look at the runtime of the benchmarks and see how the new strategies compare to the originals: The results in Table 4.2 show that the new strategies are generally slightly faster than the original strategies by some milliseconds and the standard deviation is also slightly lower, which means that the new strategies are more consistent in their runtime.

When looking at the scatter plot in Figure 4.1, we can see how the LDB and TDUB strategies compare to each other in each instance. Most notably, we can see some instances where the runtime was reported to be above 60 s. These are the instances

	runtime mean (s)	runtime stddev (s)	runtime median (s)
BC	0.7335	3.0750	0.0610
LDB	0.7257	3.0511	0.0600
FB	0.7035	2.8759	0.0600
NM	0.7041	2.8952	0.0610
NR	**0.7040**	2.8893	0.0610
NV	0.7059	2.9402	**0.0590**
SOTD	0.7090	2.9047	0.0600
TDE	0.7204	3.0104	0.0600
TDUB	0.7086	2.8886	0.0600
VD	0.7106	2.9229	0.0600

Table 4.2: Runtime statistics for each strategy. Only the instances that were solved by each solver are considered.

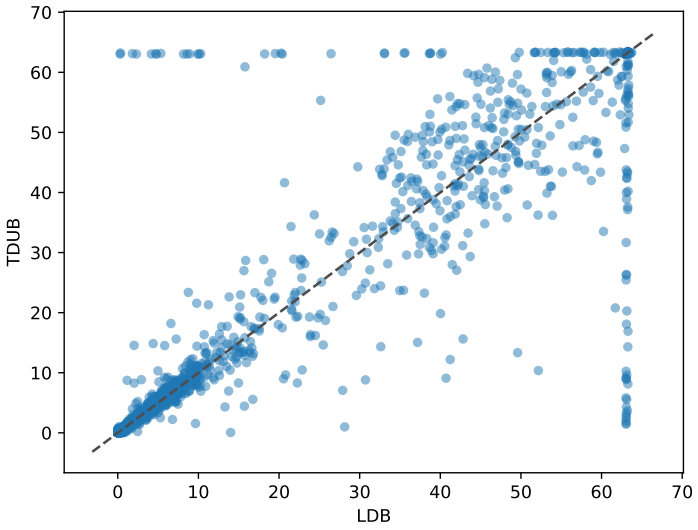


Figure 4.1: Scatter plot comparing the total running time for the LDB and TDUB strategies. All instances are considered in the plot.

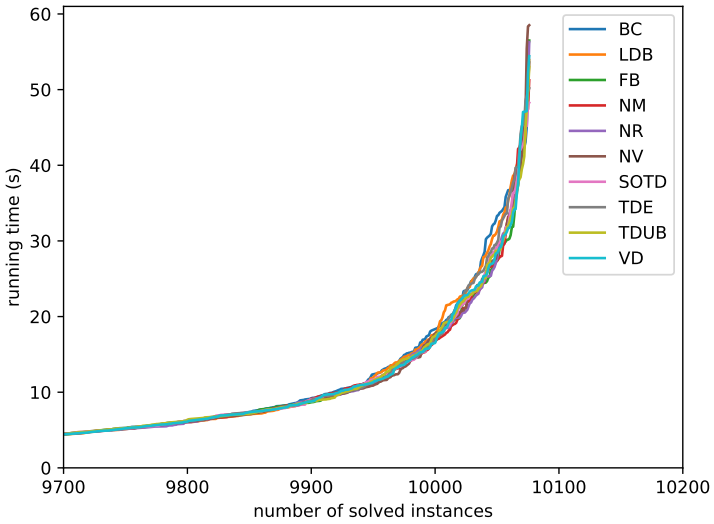


Figure 4.2: Performance profile showing the number of solved instances in relation to their running time for each strategy.

where the respective strategy was not able to solve the benchmark within the time limit and instead the solver timed out. This shows us that there are some instances where the new strategies are able to solve an instance while the original strategies are not able to solve the same instance and vice versa. When looking at these instances it is difficult to find a reason why one strategy is able to solve the instance while the other is not. This is because small differences in the explanation calls can lead to large differences in the execution path later on. Since the optimization algorithm can not effectively predict what will happen after the indexed root ordering is computed, it is difficult to conclude why one strategy is better than the other.

If we look at the performance profile in Figure 4.2, we can again see that the new strategies do not perform significantly better than the original strategies. The number of solved instances compared to the running time is similar for all strategies.

We also want to look at the performance of the optimization algorithm itself and see how much time is spent on finding the optimal indexed root ordering. Table 4.3 shows that the runtime of the optimization algorithm is about the same for most strategies at around 1.5 ms. The outlier is the TDE strategy which can be explained by the fact that the TDE strategy spends more time on computing the cost of each resultant. When looking at the box plot in Figure 4.3, we can also see a more detailed view of the runtime of the optimization algorithm for the TDUB strategy. It shows that there are some outliers that require significantly more time to compute the optimal indexed root ordering. There was an additional outlier in the TDUB strategy that took around 6 s which is not shown in the box plot for better readability. These outliers are caused by intersecting indexed root expressions as explained in Section 3.1, and it shows that the optimization algorithm is not perfect. However, since these outliers are very few and far between, we can assume that the optimization algorithm is generally working as intended.

	ordering runtime mean (ms)	ordering runtime median (ms)
FB	1.57	**0.01**
NM	1.52	**0.01**
NR	**0.97**	**0.01**
NV	1.52	**0.01**
SOTD	1.52	**0.01**
TDE	245.14	**0.01**
TDUB	1.43	**0.01**
VD	1.71	**0.01**

Table 4.3: Runtime statistics for the optimization algorithm.

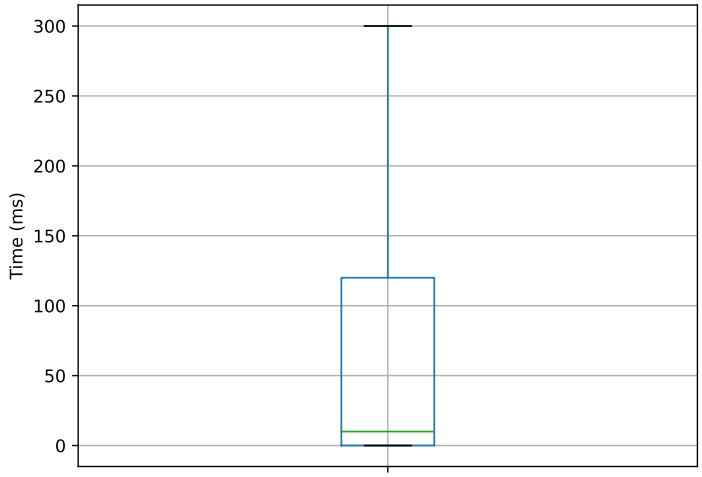


Figure 4.3: Box plot showing the distribution of the runtime of the optimization algorithm.

	total degree mean	total degree stddev
BC	2.67	5.43
LDB	2.68	5.54
FB	2.57	4.89
NM	2.56	4.79
NR	2.60	4.95
NV	2.49	4.07
SOTD	2.56	4.82
TDE	**2.45**	4.00
TDUB	2.56	4.88
VD	2.49	4.07

Table 4.4: Average total degree and standard deviation of all projections computed. Only the instances that were solved by each solver are considered.

4.3 Total degree

Next, we want to further analyze the effects of the cost metrics. The results in Table 4.4 show that the optimized strategies are generally able to slightly reduce the total degree of the resultants. When comparing the average total degree of the resultants, we can look at the TDE strategy to see how far we can expect to reduce the total degree in an optimal case. The closest strategies to the TDE strategy are the NV and VD strategies which both have a slightly higher average total degree. We could possibly explain this by the fact that these strategies reduce the total number of elimination steps in the projection phase and this in turn leads to fewer resultants that are computed and thus a lower total degree.

4.4 Number of cells

Similar to [NÁS+24], we also want to look at the number cells that are computed because the number of cells is a good indicator of the complexity of the problem. To measure the number of cells, we will look at the number of explanation calls that are made. Since the number of explanation calls is proportional to the number of cells, we can use this as a proxy for the number of cells. Table 4.5 shows that the number of cells that are computed is lower overall. When looking at the scatter plots for the LDB strategy compared to the NV and VD strategies in Figure 4.5, we can see that the number of cells stays mostly the same for the new strategies compared to the original strategies except for a few outliers where the number of cells is lower for the new strategies. One possible explanation for the lower number of cells is the lower total degree of the polynomials. Since the number of roots of any polynomial depends on the degree of the polynomial, the number of roots should be lowered for the newer strategies. This also means that the cells that are computed are bigger because the number of roots is lower and thus the number of cells is also lower.

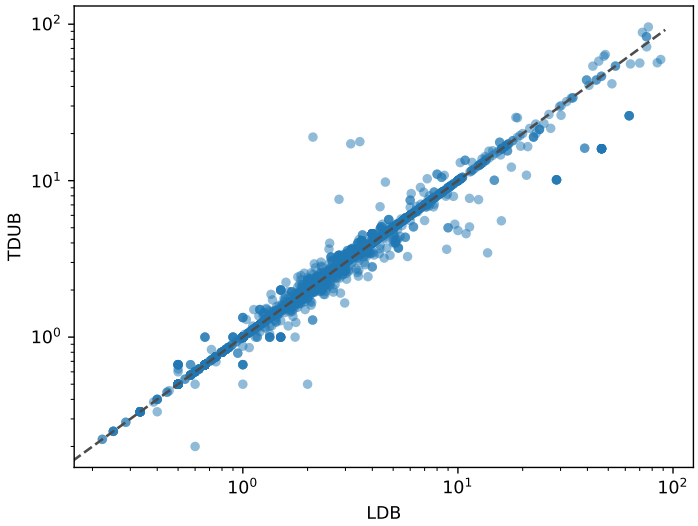
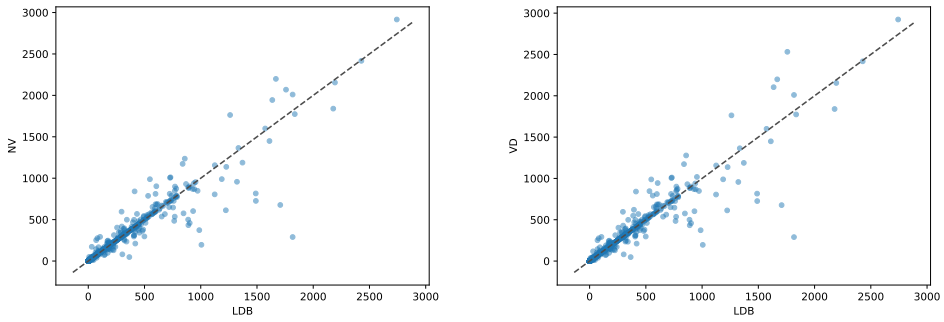


Figure 4.4: Scatter plot of the average total degree for the LDB and TDUB strategies.

	mean #cells per instance	stddev #cells per instance
BC	21.05	112.25
LDB	21.05	112.25
FB	20.64	109.72
NM	20.59	109.85
NR	20.70	110.12
NV	20.58	109.22
SOTD	**20.56**	109.11
TDE	20.65	109.72
TDUB	20.58	109.21
VD	20.74	111.10

Table 4.5: Number of cells computed for each strategy. Only the instances that were solved by each solver are considered.



(a) Comparison of number of computed cells between the strategies LDB and NV

(b) Comparison of number of computed cells between the strategies LDB and VD

Figure 4.5: Scatter plots showing the number of cells computed for the LDB strategy compared to the NV and VD strategies respectively.

	resultant mean runtime (ms)	discriminant mean runtime (ms)	is zero mean runtime (ms)	real roots mean runtime (ms)	factorization runtime (ms)
BC	9.37	60.77	4.49	17.58	33.16
LDB	9.24	61.02	4.16	17.65	33.30
FB	7.11	60.70	4.51	17.04	**30.29**
NM	6.68	**56.10**	4.33	**17.01**	31.72
NR	6.96	60.68	4.48	17.38	30.35
NV	**6.38**	62.51	**4.23**	17.14	31.12
SOTD	7.08	58.43	4.79	17.84	30.89
TDE	7.13	60.87	4.48	20.91	31.19
TDUB	7.22	61.75	4.49	17.37	32.12
VD	7.12	61.81	4.32	17.31	30.85

Table 4.6: Amount of time spent on computing different parts of the projection. Only the instances that were solved by each solver are considered.

4.5 Projection runtime profiling

As discussed earlier, we want to also investigate how much time it takes to compute projections and how the time is distributed over all the calculated projections. Specifically, we want to know what causes the computation times of the projections to be particularly high and if there are any patterns that we can identify. Alongside the resultants and discriminants, we also want to consider

- factorization of polynomials,
- computation of the real roots of polynomials and
- the function which checks whether a polynomial is zero at a given point.

Table 4.6 shows that the mean computation times of the resultants for the new strategies are generally lowered by approximately 2 ms compared to the original strategies. However, the average computation times of the resultants are already

	mean #resultant	mean #discriminant	mean #is zero	mean #real roots	mean #factorizations
BC	47.39	99.87	261.54	209.93	167.05
LDB	47.37	99.87	261.53	209.93	167.04
FB	46.62	98.42	257.92	206.95	164.41
NM	46.36	98.04	256.72	206.07	163.85
NR	46.67	98.77	258.33	207.51	164.79
NV	46.43	98.03	256.46	205.90	163.73
SOTD	**46.28**	**97.83**	**255.87**	**205.49**	**163.47**
TDE	46.60	98.45	257.28	206.64	164.38
TDUB	46.39	98.00	256.30	205.87	163.71
VD	46.81	98.98	258.80	207.98	165.19

Table 4.7: Number of times each part of the projection was computed. Only the instances that were solved by each solver are considered.

	resultant #timeouts	discriminant #timeouts	is_zero #timeouts	real_roots #timeouts	factorization #timeouts	ordering #timeouts
BC	29	472	3	18	107	**0**
LDB	28	474	3	18	110	**0**
FB	49	**427**	1	**15**	123	**0**
NM	47	441	3	24	104	**0**
NR	52	450	2	22	110	**0**
NV	52	449	**0**	23	**86**	**0**
SOTD	51	446	1	17	101	**0**
TDE	**18**	438	2	19	108	48
TDUB	45	448	4	21	108	**0**
VD	55	446	4	18	101	**0**

Table 4.8: Number of times the timer was interrupted, i.e. still running at the end of execution.

rather low with and the difference between the original and new strategies is also very small.

The results in Table 4.7 show that each part of the projection is computed fewer times for the new strategies compared to the original strategies. This is probably caused by the reduction in the number of cells that are computed as we saw in Table 4.5. Furthermore, the results could be a possible explanation for the reduction in computation times of the resultants that we saw in Table 4.6.

The table in Table 4.8 shows how often the timers were interrupted in total for each strategy. While we can not conclude that any timeout was specifically caused by the part of the projection that was being computed, we will still look at the results to see if there are any patterns since more expensive parts of the projection should be more likely to cause a timeout.

It shows that the number of interrupted timers actually increased during the computation of the resultants for the new strategies. This is surprising because we saw that the new strategies were able to reduce the computation times of the resultants.

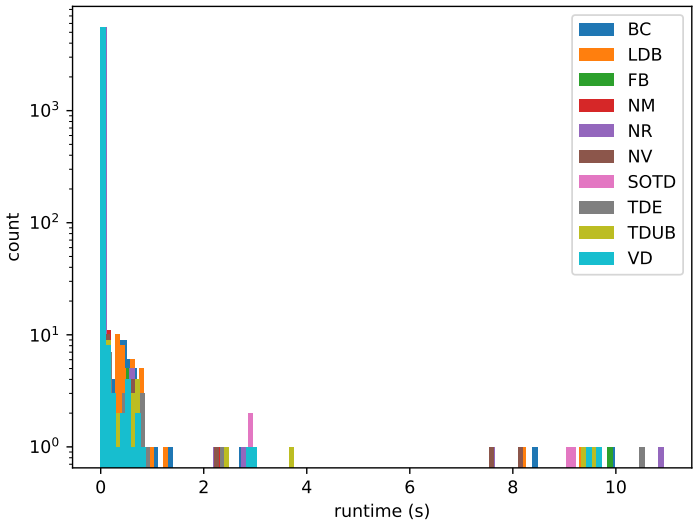


Figure 4.6: Number of resultants distributed over amount of time required to compute the resultant.

The only exception is the TDE strategy which has significantly fewer interrupted timers for the resultants. However, this can be explained by the fact that the TDE strategy has many timeouts during the cost calculation instead.

Only for the discriminants, we see a clear pattern where the new strategies have fewer interrupted timers. For the other parts of the projection, the number of interrupted timers do not show a clear pattern. Most importantly, the results show that the major bottleneck in the computation times of the projections are the discriminants and factorization.

4.5.1 Resultants

Looking further into the computation times of the resultants and discriminants, we want to know if long computation times are caused by a few resultants that take a long time to compute or if the time is evenly distributed over many resultants. Looking at the results in Figure 4.6, we can see that the majority of the resultants take only a fraction of a second to compute. In a few instances, the solvers spent around 10 seconds on computing the resultants. While these instances are problematic because they take up a large portion of the one-minute timeout limit, they are not the majority of the instances. Together with the previous results, we can see that resultants are not as problematic as previously thought.

4.5.2 Discriminants

For the discriminants, we can see a similar pattern in Figure 4.7 compared to the resultants. However, discriminants generally take a lot longer to compute than resultants as we already saw in Table 4.6, but the extreme values are a lot more

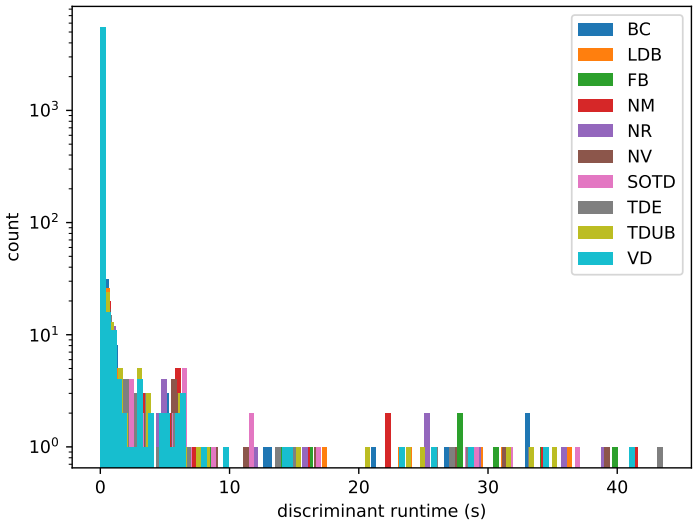


Figure 4.7: Number of discriminants distributed over amount of time required to compute the discriminant.

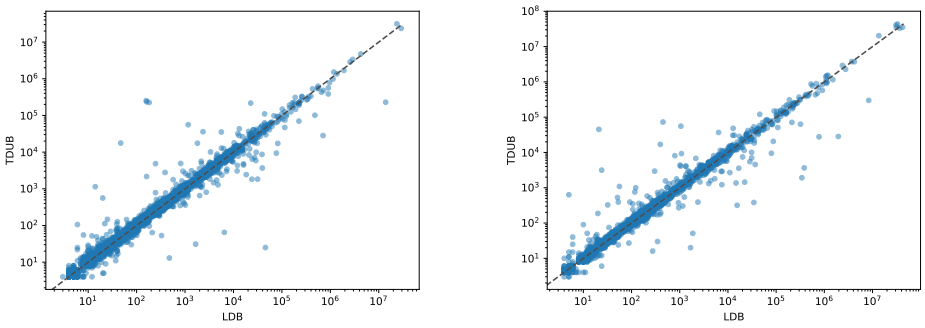
pronounced for the discriminants. Here we can see for some instances the solver spent up to around 40 seconds on computing the discriminants, but these instances are also very few and far between.

For the purpose of finding the cascading effect of the resultants and discriminants, we also want to further investigate how much impact the resultants and discriminants have on the computation times of other discriminants. Specifically, we want to know how much time is spent on computing the discriminants of the resultants and the discriminants of the discriminants. This is important because the complexity of the CAD algorithm is exponential due to the fact that we have to compute resultants and discriminants of polynomials that are themselves resultants and discriminants of other polynomials, which causes the total degree of the resultants and discriminants to grow exponentially [Col76]. When comparing the results from Table 4.9 to the results from Table 4.6, we can see that a significant amount of time is spent on computing the discriminants of the resultants and the discriminants of the discriminants. When comparing the different strategies, we can see the optimized strategies are able to slightly reduce the computation times of the discriminants of the resultants. For the discriminants of discriminants, we do not see a clear pattern that shows that the new strategies are better than the original strategies. The reduction in computation time here could be a possible explanation for the reduction in timeouts that we saw in Table 4.8.

The scatter plots in Figure 4.8 show a finer grained view for each instance in the LDB and TDUB strategies. Unfortunately, we do not see a clear pattern that shows that the new strategies are better than the original strategies in any of the scatter plots.

	disc of res timer mean (ms)	disc of disc timer mean (ms)
BC	19.33	23.95
LDB	18.07	23.83
FB	15.73	23.55
NM	**13.90**	23.15
NR	16.11	25.26
NV	17.39	25.14
SOTD	16.26	**22.92**
TDE	15.68	23.80
TDUB	15.91	23.56
VD	16.37	25.60

Table 4.9: Amount of time spent on computing discriminants of resultants and discriminants of discriminants. Only the instances that were solved by each solver are considered.



(a) Comparison of total time spent on computing discriminants of resultants for the LDB and TDUB strategies

(b) Comparison of total time spent on computing discriminants of discriminants for the LDB and TDUB strategies

Figure 4.8: Scatter plots showing the total time spent on computing discriminants of either resultants of other discriminants for the LDB strategy compared to the SOTD strategy.

Chapter 5

Conclusion

At the beginning of this thesis, we introduced the problem of finding the optimal indexed root ordering for the computation of cells in the levelwise single cell construction algorithm. Now that we have presented the optimization problem and evaluated the implementation, we can conclude on the results and discuss possible future work.

5.1 Future work

While the implementation of the optimization problem is already quite efficient, there are still some areas where the model could be improved.

Model modifications

First, the model cannot yet handle the case where multiple indexed root expressions intersect. We have shown that this case is more complex and have explored ways to solve it, but we have not yet found a suitable solution. It would be interesting to investigate if there is a way to integrate this case into the model or if this case makes the problem NP-hard. Alternatively, we could investigate if there is a way to solve this case in a heuristic way that is efficient and still returns good results.

In Section 3.3 we have shown the difficulties of addressing the connectedness of the cell in the model. We presented a possible solution, but we have not yet been able to find a solution that is efficient and guaranteed to yield optimal results. Future work could investigate if there is a way to solve this problem in a more efficient way.

Cost functions

Another area that could be investigated further is the cost functions. We have shown wide range of cost functions that could be used, but there are still many more that could possibly be used. For example, we could try to accurately calculate the total degree of the resultants and use that as a cost function by calculating only the monomial with the highest total degree in the resultant. This could potentially allow us to reduce the computation time of the TDE strategy while still maintaining the same quality of the resultants.

The ideas from the feature based cost function could also be extended to include more features as in [PdEC24]. We imagine that the features could be combined as a

sort of cost vector. This would require some work to adapt the optimization algorithm to handle vectors, but it could potentially lead to better results.

Discriminants

In Chapter 4 we have shown that a major bottleneck in the computation of projections is the computation of the discriminants. We already saw that optimizing for the cost of resultants can reduce the time spent on computing discriminants since we have to also compute discriminants for the resultants later in the projection, but discriminants still take up a large portion of the computation time. As such it would be interesting to investigate if there is a way to optimize the computation of discriminants in the same way as we have optimized the computation of resultants. In the context of this optimization problem, we can only optimize the computation of discriminants for the section case by using equational constraints, and it would be interesting to investigate if there are ways to reduce the impact of discriminants more broadly.

5.2 Summary

In this thesis we have presented an optimization problem for the computation of indexed root orderings. We have shown that the problem can be solved using graph-based optimization techniques and have presented a model for the optimization problem. We have also investigated different cost functions for resultants and have shown that the cost function can have a significant impact on the quality of the resultants. The implementation of the optimization problem has been evaluated on a set of benchmarks, and we have shown that the implementation is efficient and can be used to optimize the computation of indexed root orderings. However, we also looked at current bottlenecks in the levelwise single cell construction algorithm and have shown that the computation of discriminants is a major bottleneck. During our evaluation we have shown that the optimization algorithm is able to slightly reduce the computation times of the resultants, the average total degree and the average number of computed cells. However, we also found that the optimization algorithm does not help the levelwise single cell construction algorithm to answer more queries within the timeout limit and that the major bottlenecks in the computation times of the projections are the discriminants and factorization. Lastly, we have presented some ideas for future work that could be done to improve the model and the implementation of the optimization problem.

Acknowledgements

A special thank you to my advisor, Jasper Nalbach, for their guidance and assistance throughout the entire process. Your input has greatly contributed to the development of my thesis. I want to express my sincere appreciation to Prof. Dr. Erika Ábrahám for giving me the opportunity to work on my thesis at THS and Prof. Dr. Jürgen Giesl for being the second examiner of my thesis.

Simulations were performed with computing resources granted by RWTH Aachen University under project thes1647.

Bibliography

[BDE+16] Russell Bradford, James H. Davenport, Matthew England, Scott McCallum, and David Wilson. Truth table invariant cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 76:1–35, September 2016. https://www.sciencedirect.com/science/article/pii/S0747717115001005.

[BFN+24] Clark Barrett, Pascal Fontaine, Aina Niemetz, Mathias Preiner, Hans-Jörg Schurr, and Cesare Tinelli. SMT-LIB release 2023 (non-incremental benchmarks). https://zenodo.org/records/10607722, February 2024.

[BK15] Christopher W. Brown and Marek Košta. Constructing a single cell in cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 70:14–48, September 2015. https://www.sciencedirect.com/science/article/pii/S0747717114000923.

[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 Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 360–368, Cham, 2015. Springer International Publishing.

[Col76] George E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition: A synopsis. *SIGSAM Bull.*, 10(1):10–12, February 1976. https://dl.acm.org/doi/10.1145/1093390.1093393.

[CR99] William Cook and André Rohe. Computing Minimum-Weight Perfect Matchings. *INFORMS Journal on Computing*, 11(2):138–148, May 1999. https://pubsonline.informs.org/doi/10.1287/ijoc.11.2.138.

[dJ13] Leonardo de Moura and Dejan Jovanović. A Model-Constructing Satisfiability Calculus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 1–12, Berlin, Heidelberg, 2013. Springer.

[DPS18] Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling Algorithms for Weighted Matching in General Graphs. *ACM Trans. Algorithms*, 14(1):8:1–8:35, January 2018. https://dl.acm.org/doi/10.1145/3155301.

[DSS04] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. Efficient projection orders for CAD. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 111–118, New York, NY, USA, July 2004. Association for Computing Machinery. https://doi.org/10.1145/1005285.1005303.

[Duc00] Lionel Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145(2):149–163, January 2000. https://www.sciencedirect.com/science/article/pii/S0022404998000814.

[EBD20] Matthew England, Russell Bradford, and James H. Davenport. Cylindrical algebraic decomposition with equational constraints. *Journal of Symbolic Computation*, 100:38–71, September 2020. https://www.sciencedirect.com/science/article/pii/S0747717119300859.

[FOSV17] Pascal Fontaine, Mizuhito Ogawa, Thomas Sturm, and Xuan Tung Vu. Subtropical Satisfiability. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems*, pages 189–206, Cham, 2017. Springer International Publishing.

[Kal93] Michael Kalkbrener. An upper bound on the number of monomials in the Sylvester resultant. In *Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation*, ISSAC '93, pages 161–163, New York, NY, USA, August 1993. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/164081.164116.

[McC98] Scott McCallum. An Improved Projection Operation for Cylindrical Algebraic Decomposition. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 242–268, Vienna, 1998. Springer.

[McC99] Scott McCallum. On projection in CAD-based quantifier elimination with equational constraint. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*, ISSAC '99, pages 145–149, New York, NY, USA, July 1999. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/309831.309892.

[NÁS+24] Jasper Nalbach, Erika Ábrahám, Philippe Specht, Christopher W. Brown, James H. Davenport, and Matthew England. Levelwise construction of a single cylindrical algebraic cell. *Journal of Symbolic Computation*, 123:102288, July 2024. https://www.sciencedirect.com/science/article/pii/S0747717123001025.

[PdEC24] Lynn Pickering, Tereso del Río Almajano, Matthew England, and Kelly Cohen. Explainable AI Insights for Symbolic Computation: A case study on selecting the variable ordering for cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 123:102276, July 2024. https://www.sciencedirect.com/science/article/pii/S0747717123000901.

[Sch02] Alexander Schrijver. *Combinatorial Optimization*. Springer Berlin, Heidelberg, December 2002. https://link.springer.com/book/9783540443896.

[TVKO17] Vu Xuan Tung, To Van Khanh, and Mizuhito Ogawa. raSAT: An SMT solver for polynomial constraints. *Form Methods Syst Des*, 51(3):462–499, December 2017. https://doi.org/10.1007/s10703-017-0284-9.

[Wei97] V. Weispfenning. Quantifier Elimination for Real Algebra — the Quadratic Case and Beyond. *AAECC*, 8(2):85–101, January 1997. https://doi.org/10.1007/s002000050055.

Appendix A

Evaluation details

A.1 Version details

We used the following software libraries for the evaluation:

Library	Version
Boost	1.83.0
CArL	24.02
gmp	6.3.0
Eigen3	3.3.9

Table A.1: Versions of the libraries used for the evaluation.

All code was compiled using GCC version 13.0.0. Additionally, we used code from the following repository for the implementation of the minimum cost perfect matching algorithm[1].

[1] `https://github.com/dilsonpereira/Minimum-Cost-Perfect-Matching` (commit c916cc1)