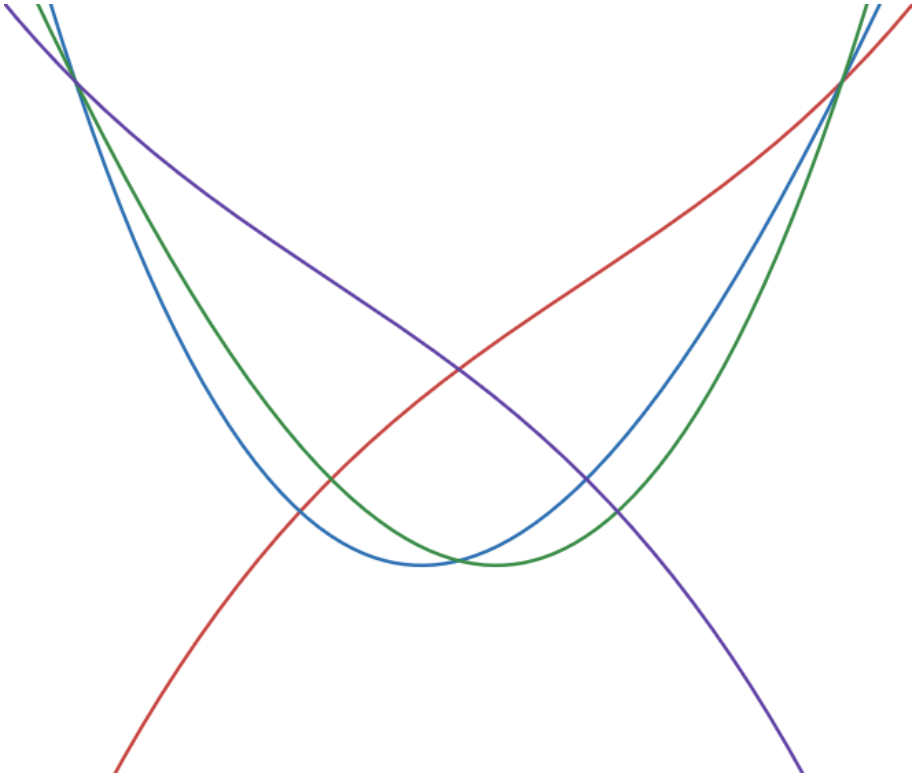

Exploring level-p-complexity for subclasses of Boolean functions

Master's thesis in Computer Science and Engineering

Arvid Rydberg
Selina Engberg

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Master's thesis 2025

Exploring level-p-complexity for subclasses of Boolean functions

Arvid Rydberg
Selina Engberg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Exploring level-p-complexity for subclasses of Boolean functions
Arvid Rydberg
Selina Engberg

Supervisor: Patrik Jansson, Department of Computer Science and Engineering
Examiner: Christian Sattler, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A set of polynomials representing the level-p complexity of the function represented by the Boolean expression:

$x_0\overline{x_1}x_2\overline{x_3} + \overline{x_0}x_1x_2 + \overline{x_0}\ \overline{x_1}\ \overline{x_2} + \overline{x_0}x_3 + x_1x_2x_3 + \overline{x_1}\ \overline{x_2}x_3$

This is an example of a 4-bit Boolean function with three maxima in $[0, 1]$.

Typeset in LaTeX
Gothenburg, Sweden 2025

Exploring level-p-complexity for subclasses of Boolean functions
Arvid Rydberg, Selina Engberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Boolean functions, which map n-bit inputs to a single output bit, can be used to describe digital systems, from logic gates to voting mechanisms. A key challenge lies in measuring the cost of evaluating such functions, specifically the number of input bits required to determine the output with certainty. Among various complexity measures, level-p-complexity, $D_p(f)$, evaluates the expected cost when input bits follow a Bernoulli(p) distribution. This paper aims to optimize an algorithm for computing level-p-complexities developed by Jansson and Jansson [1], as well as develop tools for further exploring level-p-complexity.

Key contributions include optimized algorithms for computing level-p-complexity through techniques such as hash-consing and minimization, which enhance memoization efficiency. Specialized representations for subclasses of Boolean functions, such as symmetric and threshold functions, enable faster computations for specific cases.

Furthermore, this work introduces tools to analyze piecewise polynomial representations of level-p-complexity, focusing on identifying critical points using algebraic numbers for exact calculations. A large number of properties have been implemented using QuickCheck to generate diverse test cases and increase confidence in the correctness of the algorithms.

Keywords: Boolean functions, level-p-complexity, sub-classes, majority function, Haskell, optimization.

Acknowledgements

We are deeply grateful to our supervisor, Patrik Jansson, for his never-ending support and invaluable guidance throughout this process. We also extend our heartfelt thanks to everyone who took the time to review our thesis and provide thoughtful feedback.

Arvid Rydberg, Selina Engberg, Gothenburg, 2025-04-24

Contents

List of Figures

List of Tables

List of Listings

List of definitions and theorems

1

Introduction

Boolean functions are fundamental for the understanding and development of various digital systems, from simple logic gates to more complex systems such as voting mechanisms [2].

Consider a simple Boolean function represented by an `AND` gate. The `AND` gate function takes two binary inputs, x_1 and x_2, and produces an output of **1** only if both inputs are **1**. In other words, the function can be described as $f(x_1, x_2) = x_1 \wedge x_2$.

An interesting question is "How many of the inputs do we need to evaluate in order to know what the result of f will be?". If we start by looking at x_1 and find that its value is **1**, then we also need to look at x_2 in order to know the result. However, if x_1 had been **0**, then we would have known that the result of $f(x_1, x_2)$ would be **0**, without even having to look at x_2. This concept of terminating the evaluation early when the result is called short-circuit logic and is used in a number of programming languages [3].

The concept of level-p-complexity is defined as the minimum expected evaluation cost of a Boolean function, given that its input bits are independent and identically distributed with Bernoulli(p) distribution [4]. This paper aims to optimize an algorithm for computing level-p-complexities [1], as well as develop tools for further exploring level-p-complexity.

1.1 Background

To start with, we give an explanation of the central topics needed to understand level-p-complexity and the computation of it.

Algorithms and decision trees

A possible representation of an algorithm for evaluating a Boolean function f is as a binary decision tree [1]. A definition of a binary decision tree for evaluating Boolean functions is given in listing 1.

The constructor `Res` represents a function with constant value, while the constructor `Pick` represents examining the value of an input variable i and choosing the first sub-algorithm when i is **0**, and the second if it is **1**.

As an example, a possible decision tree for the `AND` function is given in listing 2.

```haskell
data DecTree = Res Bool | Pick Index DecTree DecTree
```

Listing 1: A Haskell definition of a decision tree for evaluating boolean functions

```haskell
andAlg :: DecTree
andAlg = Pick 1 (Res False) (Pick 0 (Res False) (Res True))
```

Listing 2: A decision tree for the AND function in Haskell code

In the rest of the text, the terms algorithm and decision tree will be used interchangeably.

Expected cost of evaluating a Boolean function

The expected cost C_p of evaluating a Boolean function f using an algorithm A is given in [5]. This definition is shown in listing 3.

As seen in the type, the expected cost of evaluating a function is a polynomial of p with rational numbers as its coefficients.

Computing level-p-complexity

One obviously correct solution to the problem of finding the level-p complexity for a Boolean function f is to compute all possible decision trees for f, and choosing the ones with least expected cost. This results in a piecewise polynomial [1]. While this approach leads to a correct result, the implementation is very inefficient. To enhance efficiency, previous research [1] has introduced a library that employs the techniques thinning [6] and memoization. However, even with these optimizations, the method still has limitations, and although it can handle functions of arity up to 10 well, it starts to struggle when trying to compute the level-p-complexity of functions with higher arity.

1.2 Aim

Building on these foundations, this report has two main goals: optimization of complexity computations and developing tools for exploring properties of level-p-complexity.

In the case of optimization, limiting the scope to sub-classes of Boolean functions with certain properties can result in a faster computation of the level-p-complexity. An example of this is given in [7], where representing a 27-bit function as an

```haskell
expectedCost :: DecTree -> Poly Rational
expectedCost (Res _) = 0
expectedCost (Pick _ t1 t2) =
1 + (1 - p) * expectedCost t1 + p * expectedCost t2
```

Listing 3: The algorithm for `expectedCost`

iterated threshold function makes it possible to compute its level-p-complexity in a few minutes' time. This advancement opens the door for exploring other sub-classes of Boolean functions whose representations could yield more efficient complexity calculations than the representation from Jansson and Jansson (2023) [1].

In addition to optimizing computations for certain sub-classes of Boolean function, the report also aims to improve the method described in Jansson and Jansson (2023)[1] in order to improve the computation time of level-p-complexity for all Boolean functions.

Finally, the report aims to develop tools for further exploring level-p-complexity, specifically focusing on exploring extreme points of piecewise polynomials.

1.3 Contributions

The results presented in this report build on previous work by expanding the library presented in [1]. Among the contributions are the following:

Efficient representations of sub-classes of Boolean functions

Efficient representations have been created for two sub-classes of Boolean functions: symmetric functions and threshold functions. These representations result in much faster complexity calculations compared to [1]. However, only a small fraction of Boolean functions belong to these classes, limiting this optimization to a smaller scope. A description of these classes can be found in section 2.1.2, and the efficient representations can be found in chapter 4.

Optimization of the general algorithm for computing level-p-complexity

The algorithm for computing level-p-complexity presented in [1] has been optimized. This was achieved in two ways:

- **Hash-consing** [8] was used to achieve faster memoization-related lookups.
- **Minimization** was used to represent some sets of Boolean functions of identical complexity using only a single member of each set. This resulted in greater utilization of memoization, as only the representative of each set of functions had to be memoized. This concept is described in further detail in section 4.3.2.

A description of hash-consing can be found in section 2.1.6, and the implementation of the optimizations is presented in section 4.3.

Identifying extreme points of piecewise polynomials

As the level-p-complexity of a function is a piecewise polynomial, an interesting topic to explore is the extreme points of such a function. Developing this functionality proved a challenge, as the points where individual pieces intersect are not always rational, requiring the development of techniques for exactly determining the sign of a polynomial at an irrational coordinate.

The specific approach presented in this paper uses algebraic numbers, resulting in a way of exactly determining the extreme points of a piecewise polynomial. A definition of algebraic numbers is given in section 2.1.8 and the tools developed are described in chapter 5.

Multi-composed and iterated functions

A method for composing an n-bit Boolean function with a list of n other Boolean functions is introduced. This concept is then extended to repeated composition, resulting in what in this report is referred to as iterated functions.

Definitions of these two concepts are given in section 2.1.3, and the implementation details are found in section 5.3.1.

These definitions build upon the work in [7], but have better separation of concerns and are more general.

Testing to ensure correctness

A large number of properties regarding Boolean functions as well as complexities have been identified. These have been implemented and verified using the property based testing library QuickCheck [9]. Additionally, QuickCheck generators have been created in order to efficiently produce a large variety of test cases. Details about these properties and generators are found in section 3.2.

Code

All of the code referenced in this thesis is openly available at `https://github.com/Riddarvid/BoFunComplexitySubclasses`. The code referenced in this paper is tagged with the tag `report-revised` and has the hash `1c6c98410d29feda2c16680680d6de6636f38bf4`.

2

Theory

This chapter dives deeper into the background of the problem and the general algorithm for solving it. It also describes the advancements that have been done since the publication of the paper by Jansson & Jansson [1]. Finally, some core properties of Boolean functions are defined and proved.

2.1 Background

Here, we go over the topics that the work builds upon, focusing on Boolean functions and complexity measures.

2.1.1 Boolean functions

A Boolean function is a function from n bits to one bit, where n is a non-negative integer [10].

Majority functions

The majority function maj_3 is a function that takes three bits and returns the bit that is most frequent.

The function maj_3^2 is instead the iterated majority function. It takes nine bits, divides them into three groups of three bits each, computes the majority for each of those groups, and then returns the majority of the results.

$$maj_3^2(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) =$$
$$maj_3(maj_3(x_1, x_2, x_3), maj_3(x_4, x_5, x_6), maj_3(x_7, x_8, x_9))$$

This function has been studied in previous work [5] in the field of level-p-complexity, and it will be used throughout the report to illustrate examples.

Notation for Boolean functions

In order to better be able to reason about Boolean functions, we introduce a new notation for indexing the variables in the input to a Boolean function.

Definition 2.1.1 (The type of a Boolean function)**.**
We define the type of Boolean functions defined over a set of variables I:

$$BF(I) = (I \to B) \to B$$

In other words, a Boolean function is a function that takes a lookup vector over I as its input and returns a Boolean value. We say that the function's input is indexed by the set I.

As an example, we can look at the previously mentioned maj_3^2 function. Here, I could contain elements of any type, what is important is that it contains exactly nine elements.

$I = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

and

$I = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}$

are both valid sets for indexing into maj_3^2.

The `BoFun` typeclass

In the complexity computations one only needs two core operations on Boolean functions, which are described in listing 4. In this Haskell typeclass, `bf` is a type representing Boolean functions, and `Index` is the type for variable indices.

```
1 class BoFun bf where
2   isConst :: bf -> Maybe Bool
3   setBit :: Index -> Bool -> bf -> bf
4 type Index = N
```

Listing 4: The `BoFun` typeclass

To describe the `isConst` operation, we first need to define what it means for a function to be constant:

Definition 2.1.2 (Constant Boolean function)**.**
A Boolean function f of type $BF(I)$ is constant *if* there is a $b \in B$ such that $f(v) = b$ for all $v : I \to B$.

The `isConst` function checks if the Boolean function is constant, and in that case returns `Just b`, otherwise it returns `Nothing` [11].

The second method, `setBit`, restricts a Boolean function f by fixing the ith input bit to a Boolean value b. Using the formal notation, a `setBit` implementation should satisfy the following equality:

$$setBit(i, b, f) = \lambda v \to f(v')$$
$$\text{where}$$
$$v'(j) = \text{ if } i = j \text{ then } b \text{ else } v(j)$$

This operation produces a sub-function f_i^b with a lower arity than f. Although it is not enforced in the type, f_i^b can no longer use the variable i in its definition [1].

In the formal definition, if f has type $BF(I)$ and $i \in I$, then f_i^b has type $BF(I - i)$. Note that we use the notation $I - i$ to denote $I \setminus \{i\}$ throughout the paper.

2.1.2 Classes of Boolean functions

This section gives a short description of some common classes of Boolean functions [2] which will be referred to throughout the report. A more thorough description is given in chapter 4.

General - Any Boolean function is a general Boolean function.

Symmetric - A symmetric function is a function which yields the same value for all inputs containing the same number of **1**:s. Another way of phrasing it is that symmetric functions are invariant under permutations of input.

An example of a symmetric function is the 2-bit `XOR` function, which returns **1** *if* the number of **1**'s in the input is equal to 1.

Threshold - A threshold function with threshold $k \in \mathbb{N}$ is a function which yields the value **0** *if* the number of **1**:s in the input is greater than or equal to k. All threshold functions are symmetric, but not all symmetric functions are threshold functions.

Two threshold function examples are the 2-bit `AND` function (which has a threshold of two 1's) and the 2-bit `OR` function (which has threshold one). The previous example, `XOR`, is only symmetric, but not a threshold function.

Monotone - A function is monotone if, for any input where it outputs **1**, changing any input bit from **0** to **1** will still result in **1**.

An example of a monotone function is the 2-bit `const` function, returning its first argument. The fact that this function is monotone becomes obvious when one looks at the truth table:

x_1	x_2	`const`
0	0	0
0	1	0
1	0	1
1	1	1

Note that threshold functions are exactly the monotone symmetric functions.

Odd - An odd function is a function for which flipping every input bit results in negating the output.

As it happens, the `const` function from the monotone example is also odd.

Even - An even function is a function for which flipping every input bit results in no change of the output.

The 2-bit `XOR` function is an example.

Gates - The binary functions `AND`, `OR`, `XOR`, as well as the unary function `NOT`.

Balanced - A balanced function is a function which yields **1** for exactly half of its inputs [12].

Exactly one 1 - A function which yields **1** for exactly one of its inputs.

Read-once - A read-once function is any function that can be constructed from Boolean gates, where each variable appears at most once [13].

2.1.3 Composing Boolean functions

Here, we give two definitions of how to compose Boolean functions with each other.

Definition 2.1.3 (Multi-composition)**.**
Multi-composing a Boolean function f with a collection of Boolean functions gs is defined as follows:

$$multiCompose : BF(I) \to ((i : I) \to BF(J(i))) \to BF(DepPair(I, J))$$

where the typing rule for $DepPair$ is:

$$(i, j) : DepPair(I, J) \text{ when } i : I \text{ and } j : J(i)$$

$$multiCompose(f, gs) = \lambda v \to f(v')$$

where

$$v'(i) = gs(i)(v_i)$$

where

$$v_i(j) = v(i, j)$$

What this means is that if a Boolean function f is defined over an input of type I, then it can be composed over a collection of $|I|$ sub-functions.

A straightforward example of a multi-composed function is illustrated in fig. 2.1. In this example, the function h is applied to the outputs of the functions g_1, g_2, and g_3. These functions operate on 3-bit, 2-bit, and 4-bit inputs, respectively. When combined, they form a 9-bit function.

Note that the same function can be represented in multiple ways using iteration. For example, if AND_2 and AND_3 are the two- and three-bit AND functions respectively, then the five-bit AND function can be expressed as either:

$$\underbrace{\underbrace{\mathbf{0},\mathbf{0},\mathbf{1}}_{g_1}\quad\underbrace{\mathbf{0},\mathbf{1}}_{g_2}\quad\underbrace{\mathbf{1},\mathbf{1},\mathbf{0},\mathbf{1}}_{g_3}}_{h}$$

Figure 2.1: A simple example of a multi-composed function

$$multiCompose(AND_2, gs) \text{ where } gs(0) = AND_2, gs(1) = AND_3$$

or

$$multiCompose(AND_2, gs) \text{ where } gs(0) = AND_3, gs(1) = AND_2$$

Definition 2.1.4 (Iterated)**.**
Iteration is the concept of repeated multi-composition. A multi-composed function is either:

- the identity function, or
- the result of multi-composing a function over a collection of iterated functions.

This recursive definition results in a rose tree-like structure, where the value of each branch node is a function, and its children are the functions it is multi-composed over, and the leaves consist of either a 0-ary function or the identity function. A visual example of an iterated function can be seen in fig. 2.2 [14].

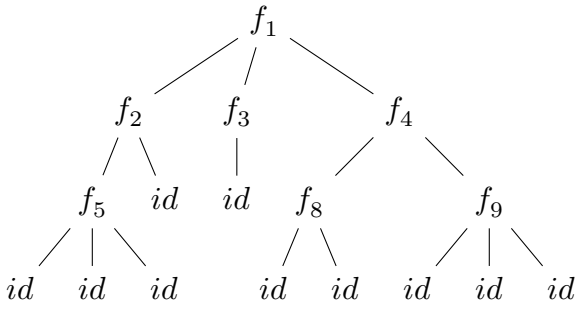

Figure 2.2: An example of an iterated function consisting of several layers of functions of the same function type.

An example of iteration is the read-once class of functions described in section 2.1.2. This class is exactly the class of functions that can be represented by iterating functions of the gate class.

Naturally, since iteration is simply the concept of repeated multi-composition, functions in this class can also be described in multiple ways.

2.1.4 Level-p-complexity

Assume that we have some Boolean function f and that we have a decision tree A representing the evaluation order of the input bits of f. An example of a decision

tree depicting one possible evaluation order for maj_3 can be seen in fig. 2.3. In the same figure we can see the polynomial $2 + 2x - 2x^2$ representing the level-p-complexity for maj_3. If the probability of a bit being 0 is high, then we only need to consider 2 bits on average. If the probability is closer to 50/50, the required number of bits increases slightly towards 2.5. If the a bit has a high probability of being 1, then we again only need to consider 2 bits.

If the input bits are independently distributed with a Bernoulli distribution, with $p \in [0, 1]$ being the probability that a bit is a 1, we can calculate the expected depth we will reach in the decision tree. This number represents the expected number of bits needed to evaluate. The level-p-complexity $D_p(f)$ of f is then defined as the minimum expected number of queries over all possible decision trees. The level-p-complexity of f can then be expressed as a piecewise polynomial in p [4].

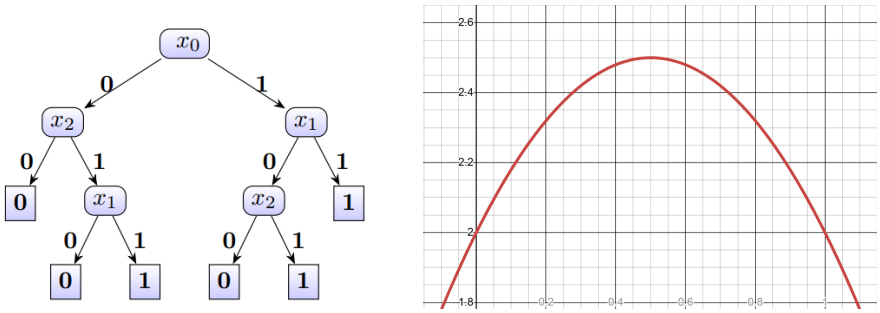

Figure 2.3: An illustration of a decision tree for computing $maj_3(x_0, x_1, x_2)$ [1] and the polynomial for the level-p-complexity plotted in a graph

2.1.5 Binary decision diagrams

Binary decision diagrams BDDs [15] is a data structure used to represent and manipulate Boolean functions efficiently. It is a type of directed acyclic graph (DAG) where each internal node represents a Boolean variable, and the edges correspond to the two possible values of that variable (usually **0** and **1**). The terminal nodes of the BDD are labeled with Boolean constants (**0** or **1**), representing the outcome of the function evaluation for different variable assignments. An example of a BDD can be seen in fig. 2.4.

To ensure uniqueness and compactness, BDDs are typically reduced. A **reduced ordered binary decision diagram (ROBDD)** is constructed by applying the following reduction rules to its graph:

- **Merge isomorphic subgraphs:** Any two subgraphs that represent the same function are merged into one.
- **Eliminate redundant nodes:** Any node whose two children are isomorphic is removed, and its parent node is connected directly to the shared child.

These reduction rules guarantee that, for a fixed variable ordering, every Boolean function has a unique representation as a BDD. This distinguishes BDDs from general binary decision trees, which may have multiple representations of the same

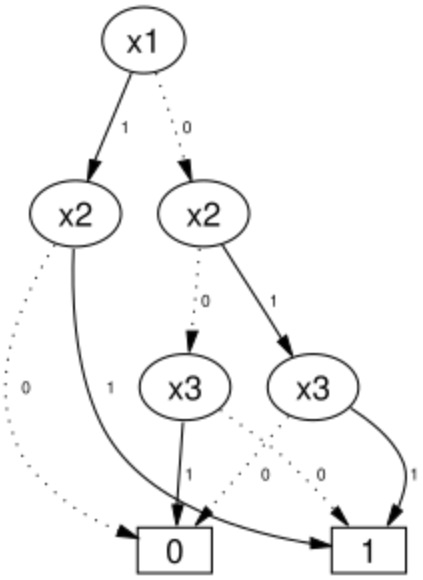

Figure 2.4: The BDD for the Boolean formula $\overline{x_1}\,\overline{x_2}\,\overline{x_3}+x_1x_2+x_2x_3$ using ascending variable ordering [16]

function even with the same variable ordering. The combination of ordering and reduction makes BDDs canonical, meaning they are unique up to isomorphism for a given function and variable order.

In this context, variable ordering refers to the sequence in which the Boolean variables are tested as one traverses from the root of the diagram to the terminal nodes. Specifically, there is a total ordering relation which must be respected throughout the structure to maintain the uniqueness property of the BDD.

The choice of variable ordering significantly impacts the size of the BDD; a poor variable ordering can result in an exponentially large BDD, while an optimal ordering can lead to a compact representation. This is of importance when comparing BDDs by traversing the structure of the underlying DAG.

The uniqueness property ensures consistency and efficiency when using BDDs to represent and manipulate Boolean functions, as we can be sure that if two boolean functions have different BDD representations, then they must be distinct functions. This structure makes BDDs a highly efficient way to represent Boolean functions for tasks like testing, verification and complexity analysis.

2.1.6 Hash-consing

Hash consing is a technique used for memory sharing and fast comparisons. In short, the technique is based on only ever creating one copy of a given value. This is usually done via a hash table, where values are looked up and if they already

exist, the existing value is used. Each value is also tagged with a unique integer identifier when it is first inserted into the hash table. This means that as long as the same hash table is used, values with the same tag are guaranteed to be equal, and, similarly, values with different tags are guaranteed to be different.

This technique is used by the library `decision-diagrams` and enables constant time comparisons of BDDs, given that they use the same variable ordering.

2.1.7 Memoization

Memoization is a technique used in dynamic programming, which enables reuse of earlier computations [17]. An example is the naive implementation of the Fibonacci function:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n+2) &= fib(n+1) + fib(n) \end{aligned}$$

The number of function calls for this function grows exponentially [1].

However, if we just compute $fib(0), fib(1), ..., fib(n)$ in that order, we can reuse the previous results instead of going through the recursive function, and in this way only spending linear time to calculate the result.

When evaluating the complexity of a Boolean function, you must in turn evaluate the complexity for the $2 \cdot n$ different sub-functions received as a result of setting each of the n bits to either **0** or **1**. In general, many of these sub-functions will be identical. It therefore makes sense to only have to evaluate the complexity of each distinct sub-function once.

This raises the question of how to determine whether two sub-functions are identical. In the algorithm for calculating level-p complexity, presented in [1], this is done using BDDs.

2.1.8 Algebraic numbers

An algebraic number [18] is a real number that can be expressed as a solution to a polynomial equation with integer coefficients. Algebraic numbers are used in section 5.1 in order to determine certain properties of level-p complexities exactly. An example of an algebraic number is $\sqrt{2}$. It is algebraic because it is a root of the polynomial $x^2 - 2$.

2.2 Recent advancements

This section gives background to what has been done to extend the library defined in [1] since its publication. This work was done Patrik Jansson and Christian Sattler. It has not yet been published but is publically available in a Github repository [7].

2.2.1 The new `BoFun` typeclass with the variable function

```
1
2 class BoFun f i | f -> i where
3   isConst   :: f -> Maybe Bool
4   variables :: f -> [i]
5   setBit    :: (i, Bool) -> f -> f
6
```

Listing 5: The new `BoFun` typeclass

The first change is the definition of the `BoFun` typeclass. A `BoFun` can now be indexed over any type i, rather than by the natural numbers. This gives the consumer of the library much greater freedom in how they choose to define a `BoFun` instance. It also allows for certain optimizations: a threshold function can for example be indexed over the type `()`. This is because of the fact that threshold functions are symmetric and do not depend on variable ordering, meaning that all variables are, in a sense, equivalent.

2.2.2 Piecewise polynomials

Rather than representing complexity indirectly as the minimum of a set of polynomial functions, a specialized data structure representing piecewise polynomials can be used to more efficiently compute the level-p complexity of a Boolean function [7].

In mathematics, a piecewise polynomial is a function that is defined by different polynomials over distinct intervals of its domain [19]. An example of a piecewise polynomial for the interval 0 to 1 is the function f below. This is also illustrated in fig. 2.5.

$$f(x) = \begin{cases} 2x^2 & \text{if } \mathrm{x} < 1/2 \\ x & \text{otherwise} \end{cases}$$

In the algorithm for computing level-p complexity, presented in [1], the thinning step was needed in order to keep the set of polynomials representing the complexity small. The use of a dedicated data structure representing piecewise polynomials simplifies this algorithm. As there is now only a single function to manage, there is no need to thin out redundant functions, as is required when working with a set. Additionally, it can increase efficiency, as sets of polynomial functions often contain many unnecessary elements, particularly at higher polynomial degrees. Results concerning these potential efficiency gains are presented in chapter 6. Lastly, using piecewise polynomial functions directly allows for immediate identification of the relevant intervals.

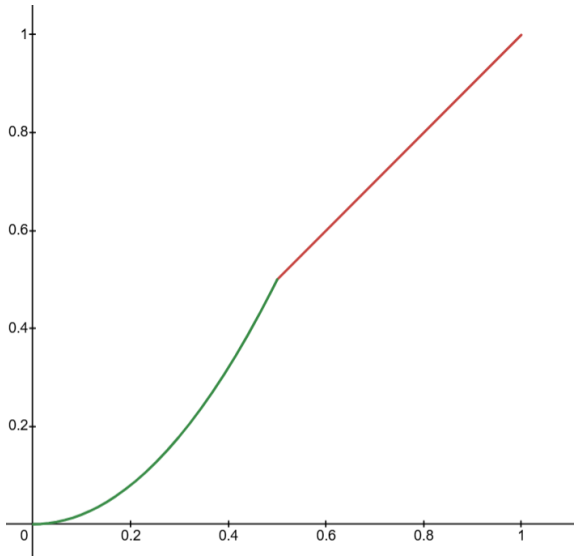

Figure 2.5: An illustration of a piecewise polynomial for function f(x)

2.2.3 Specialized classes of Boolean functions

Another area of improvement has been the implementation of representations of specialized function classes. A truth table for a general n-bit Boolean function needs 2^n bits of information, making the lookups needed for memoization exponential in these cases. Even with the use of BDDs, the number of nodes needed to be compared can still be large. A threshold function, however, only needs two integers of information: the number of bits needed to reach the threshold, and the arity of the function [7]. This makes the comparison used in memoization much faster. The drawback is of course that this representation only works for threshold functions.

2.3 Proving properties of level-p complexity

In this section, we prove two important properties that allow us to optimize the representations of Boolean functions. The first property lets us remove unnecessary bits, and the other lets us re-map the input variables. The two properties will later be used to mathematically justify the optimization described in section 4.3.2.

To start with, we give some definitions that will be used in stating and proving the properties. We also expand upon the definition of a decision tree given in [1].

Definition 2.3.1 (Decision tree)**.**
The set $DecTree(I)$ of decision trees using variables in I is inductively defined as:

- $Res(b)$ where $b \in B$
- $Pick(i, t_0, t_1)$ where $i \in I$ and $t_0, t_1 \in DecTree(I - i)$

Definition 2.3.2 (Semantics of a decision tree)**.**
The semantics of a decision tree is a Boolean function, defined as follows:

$$
\begin{aligned}
&[\_] : DecTree(I) \to BF(I) \\
&[Res(b)] = \lambda v \to b \\
&[Pick(i, t_0, t_1)] = \lambda v \to \text{if } v(i) \\
&\quad \text{then } [t_1](v|_{I-i}) \\
&\quad \text{else } [t_0](v|_{I-i})
\end{aligned}
$$

Definition 2.3.3 ($expCost$)**.**
The expected cost of evaluating a decision tree for a given probability p is defined as follows:

$$
\begin{aligned}
&expCost : DecTree(I) \to \mathbb{R} \to \mathbb{R} \\
&expCost(Res(b), p) = 0 \\
&expCost(Pick(i, t_0, t_1), p) = 1 + (1 - p) \cdot expCost(t_0, p) + p \cdot expCost(t_1, p)
\end{aligned}
$$

2.3.1 Redundant bits

This section aims to show that if a function does not depend on some of its inputs, then those inputs can be removed without affecting the complexity. We start by giving a formal definition of the concept of dependence.

Definition 2.3.4 (Dependence and independence)**.**
A function f of type $BF(I)$ is independent from a bit $i \in I$ if, for all input vectors v, $f(v_{i\to\mathbf{0}}) = f(v_{i\to\mathbf{1}})$, where $v_{i\to b}$ denotes the vector v but where $v(i) = b$. Similarly, f is dependent on a bit i if there exists an input vector v such that $f(v_{i\to\mathbf{0}}) \neq f(v_{i\to\mathbf{1}})$.

We denote the set of variables that a function f depends on as $dep(f)$.

Note that if f is of type $BF(I)$, then $dep(f) \subseteq I$. Note also that we must necessarily have $dep([t_0]) \subseteq dep([Pick(i, t_0, t_1)])$ for all i, and similar for t_1.

We also define a few helper functions that will allow us to state the property formally.

Definition 2.3.5 ($restrictDomain_{BF}$)**.**
We define the function shrinking the domain of a Boolean function as follows:

$$
\begin{aligned}
&restrictDomain_{BF} : (J : Set) \to BF(I) \to BF(J)\\
&restrictDomain_{BF}(J, f) = \lambda\ v \to f(v')\\
&\quad where\\
&\quad v'(i) = \text{if } i \in J \text{ then } v(i) \text{ else } \mathbf{0}
\end{aligned}
$$

where I and J are any two finite sets such that $J \subseteq I$.

Definition 2.3.6 ($restrictDomain_{DT}$)**.**
We define the function shrinking the domain of a decision tree. If a variable is not in the target set, then the $Pick$ containing it is replaced by its child with lowest expected cost, given some probability p.

$$
\begin{aligned}
&restrictDomain_{DT} : (J : Set) \to DecTree(I) \to \mathbb{R} \to DecTree(J)\\
&restrictDomain_{DT}(J, Res(b), p) = Res(b)\\
&restrictDomain_{DT}(J, Pick(i, t_0, t_1), p) = \text{if } i \in J\\
&\quad \text{then } Pick(i, t'_0, t'_1)\\
&\quad \text{else if } expCost(t'_0, p) < expCost(t'_1, p) \text{ then } t'_0 \text{ else } t'_1\\
&\quad where\\
&\quad t'_0 = restrictDomain_{DT}(J - i, t_0, p)\\
&\quad t'_1 = restrictDomain_{DT}(J - i, t_1, p)
\end{aligned}
$$

where I and J are any two finite sets such that $J \subseteq I$.

Definition 2.3.7 ($expandDomain_{DT}$)**.**
The function expanding the domain of a decision tree is simply the identity function, but expanding the type.

$$expandDomain_{DT} : DecTree(J) \to DecTree(I)$$
$$expandDomain_{DT}(t) = t$$

where I and J are any two finite sets such that $J \subseteq I$.

Stating the property

We are now ready to state the property formally:

$$\forall f : BF(I).\ D_p(f) = D_p(restrictDomain_{BF}(J, f))$$

where I and J are any two finite sets such that $J \subseteq I$ and $dep([t]) \subseteq J$.

In order to prove this property, we will first prove five lemmas.

Lemma 2.3.1 (Semantics of restricting a DecTree)**.**
$\forall t : DecTree(I),\ p \in [0,1],\ v : (I \to B)$ we have

$$[t](v) = [restrictDomain_{DT}(J, t, p)](v|_J)$$

where I and J are any two finite sets such that $J \subseteq I$ and $dep([t]) \subseteq J$.

Proof. Let I be a finite set and J any set such that $J \subseteq I$ and $dep(f) \subseteq J$. Let v be any input vector of type $I \to B$ and let $p \in [0,1]$ be any probability.

We will use a proof by structural induction on the structure of t.

Base case

In the base case, $t = Res(b)$ for some $b \in B$.

$$\begin{aligned} LHS &= [t](v) \\ &= [Res(b)](v) \\ &= (\lambda v' \to b)(v) \\ &= b \end{aligned}$$

$$\begin{aligned} RHS &= [restrictDomain_{DT}(J, t, p)](v|_J) \\ &= [restrictDomain_{DT}(J, Res(b), p)](v|_J) \\ &= [Res(b)](v|_J) \\ &= (\lambda v' \to b)(v|_J) \\ &= b \end{aligned}$$

LHS = RHS, so we have proved the base case.

Induction case

In the induction case, $t = Pick(i, t_0, t_1)$ for some $i \in I$ and $t_0, t_1 \in DecTree(I - i)$.

The induction hypothesis is that we have $[t_0](v_0) = [restrictDomain_{DT}(J_0, t_0, p')](v_0|_{J_0})$ for all v_0, p' and all J_0 such that $J_0 \subseteq I - i$ and $dep([t_0]) \subseteq J_0$. Similarly for t_1.

Let
$t_0' = restrictDomain_{DT}(J - i, t_0, p)$
$t_1' = restrictDomain_{DT}(J - i, t_1, p)$

By the induction hypothesis, $[t_0](v|_{I-i}) = [t_0'](v|_{I-i}|_{J-i}) = [t_0'](v|_{J-i})$, since the following prerequisites hold:

1. We trivially see that we fulfill the prerequisite $J - i \subseteq I - i$, since $J \subseteq I$.

2. The second prerequisite, $dep([t_0]) \subseteq J - i$, holds since $dep([t_0]) \subseteq dep([t]) \subseteq J$, and since $[t_0] : BF(I - i)$ it is clear that $i \notin dep([t_0])$ and thus $dep([t_0]) \subseteq J - i$.

The same logic holds for $[t_1](v|_{I-i}) = [t_1'](v|_{J-i})$.

$$
\begin{aligned}
LHS &= [t](v) \\
&= [Pick(i, t_0, t_1)](v) \\
&= \text{if } v(i) \text{ then } [t_1](v|_{I-i}) \text{ else } [t_0](v|_{I-i}) \\
&= \text{if } i \in J \\
&\quad \text{then if } v(i) \text{ then } [t_1](v|_{I-i}) \text{ else } [t_0](v|_{I-i}) \\
&\quad \text{else } [t_0](v|_{I-i}) \qquad \text{By Def. 2.3.4 (independence),} \\
&\qquad\qquad\qquad\quad \text{since } i \notin J \to i \notin dep([t]), \\
&\qquad\qquad\qquad\quad [t_0] = [t_1] \text{ so we choose } [t_0].
\end{aligned}
$$

$RHS = [restrictDomain_{DT}(J, t, p)](v|_J)$
$= [restrictDomain_{DT}(J, Pick(i, t_0, t_1), p)](v|_J)$
$=$ if $i \in J$
then $[Pick(i, t'_0, t'_1)](v|_J)$
else if $expCost(t'_0, p) < expCost(t'_1, p)$
  then $[t'_0](v|_{J-i})$
  else $[t'_1](v|_{J-i})$
$=$ if $i \in J$
then if $v|_J(i)$ then $[t'_1](v|_{J-i})$ else $[t'_0](v|_{J-i})$
else if $expCost(t'_0, p) < expCost(t'_1, p)$
  then $[t'_0](v|_{J-i})$
  else $[t'_1](v|_{J-i})$
$=$ if $i \in J$
then if $v|_J(i)$ then $[t'_1](v|_{J-i})$ else $[t'_0](v|_{J-i})$
else $[t'_0](v|_{J-i})$ — By Def. 2.3.4 (independence), $[t'_0] = [t'_1]$ so we choose $[t'_0]$.

$=$ if $i \in J$ — By the induction hypothesis,
  then if $v|_J(i)$ then $[t_1](v|_{I-i})$ else $[t_0](v|_{I-i})$ — as shown earlier.
  else $[t_0](v|_{I-i})$
$=$ if $i \in J$
  then if $v(i)$ then $[t_1](v|_{I-i})$ else $[t_0](v|_{I-i})$ — $(v(i) = v|_J(i))$
  else $[t_0](v|_{I-i})$

LHS = RHS, so we have proved the induction case and completed the induction. □

Lemma 2.3.2 (Semantics of restricting a BF)**.**
$\forall\ t : DecTree(I),\ p \in [0,1]$.

$$[restrictDomain_{DT}(J,t,p)] = restrictDomain_{BF}(J,[t])$$

where I and J are any two finite sets such that $J \subseteq I$ and $dep([t]) \subseteq J$.

Proof. Let I be any finite set and let J be any set such that $J \subseteq I$ and $dep([t]) \subseteq J$. Let $p \in [0,1]$ be any probability and t any $DecTree(I)$.

Let v be any vector and let $v'(i) =$ if $i \in J$ then $v|_J(i)$ else $\mathbf{0}$.

$$\begin{aligned} LHS &= [restrictDomain_{DT}(J,t,p)](v|_J) \\ &= [t](v) && \text{By lemma 2.3.1.} \\ &= [t](v') && \text{Since } [t] \text{ is independent from} \\ & && \text{the compliment of } J \text{ in } I. \end{aligned}$$

$$\begin{aligned} RHS &= restrictDomain_{BF}(J,[t])(v|_J) \\ &= [t](v') \end{aligned}$$

LHS = RHS, so we are done.

□

Lemma 2.3.3 (Semantics of expanding domain)**.**
$\forall\ t : DecTree(J), v : (I \to B)$.

$$[expandDomain_{DT}(t)](v) = [t](v|_J)$$

where I and J are any two finite sets such that $J \subseteq I$.

Proof. This is trivially true, since $expandDomain_{DT}(t)$ is identical in structure to t, and $v|_J$ by definition gives the same values as v for elements in J. □

Lemma 2.3.4 (Expected cost of restricting domain)**.**
$\forall\ t : DecTree(I), p \in [0,1]$.

$$expCost(t,p) \geq expCost(restrictDomain_{DT}(J,t,p),p)$$

where I and J are any two finite sets such that $J \subseteq I$ and $dep([t]) \subseteq J$.

Proof. Let I be any finite set and let J be any set such that $J \subseteq I$ and $dep([t]) \subseteq J$. Let $p \in [0,1]$ be any probability.

We will use a proof by induction on the structure of t to prove that $expCost(t,p) \geq expCost(restrictDomain_{DT}(J,t,p),p)$ for all t, p.

Base case

In the base case, $t = Res(b)$ for some $b \in B$.

$$\begin{aligned} LHS &= expCost(t,p) \\ &= expCost(Res(b),p) \\ &= 0 \end{aligned}$$

$$\begin{aligned} RHS &= expCost(restrictDomain_{DT}(J,t,p),p) \\ &= expCost(restrictDomain_{DT}(J,Res(b),p),p) \\ &= expCost(Res(b),p) \\ &= 0 \end{aligned}$$

$LHS \geq RHS$, so we have proved the base case.

Induction step

In the induction case, $t = Pick(i, t_0, t_1)$ for some i, t_0, t_1.

Let
$t'_0 = restrictDomain_{DT}(dep([t_0]), t_0, p)$
$t'_1 = restrictDomain_{DT}(dep([t_1]), t_1, p)$

For clarity, we let $c_0 = expCost(t_0, p)$, $c_1 = expCost(t_1, p)$, $c'_0 = expCost(t'_0, p)$, and $c'_1 = expCost(t'_1, p)$.

The induction hypothesis is that $expCost(t_0, p') \geq expCost(restrictDomain_{DT}(J_0, t_0, p'), p')$, where J_0 is any finite set such that $J_0 \subseteq I - i$ and $dep([t_0]) \subseteq J_0$ and p' is any probability. Similar for t_1.

Specifically, this means that $c_0 \geq c'_0$ and $c_1 \geq c'_1$.

$$
\begin{aligned}
LHS &= expCost(t, p) \\
&= expCost(Pick(i, t_0, t_1), p) \\
&= 1 + (1 - p) \cdot c_0 + p \cdot c_1 \\
RHS &= expCost(restrictDomain_{DT}(J, t, p), p) \\
&= expCost(restrictDomain_{DT}(J, Pick(i, t_0, t_1), p), p) \\
&= \text{if } i \in J \\
&\quad \text{then } expCost(Pick(i, t'_0, t'_1), p) \\
&\quad \text{else if } c'_0 < c'_1 \text{ then } c'_0 \text{ else } c'_1 \\
&= \text{if } i \in J \\
&\quad \text{then } expCost(Pick(i, t'_0, t'_1), p) \\
&\quad \text{else } min(c'_0, c'_1) \\
&= \text{if } i \in J \\
&\quad \text{then } 1 + (1 - p) \cdot c'_0 + p \cdot c'_1 \\
&\quad \text{else } min(c'_0, c'_1)
\end{aligned}
$$

We now need to show that $LHS \geq RHS$.

We start by looking at the first branch of the if-statement in RHS. By the induction hypothesis, we have $c_0 \geq c'_0$, and $c_1 \geq c'_1$, and from this it follows that $1 + (1 - p) \cdot c_0 + p \cdot c_1 \geq 1 + (1 - p) \cdot c'_0 + p \cdot c'_1$, since p and $1 - p$ are both greater than or equal to zero.

We now look at the second branch of the if-statement. We now have two cases, either $c'_0 < c'_1$ or $c'_0 \geq c'_1$. In the first case, $RHS = c'_0$. We rewrite this as $RHS = (1 - p) \cdot c'_0 + p \cdot c'_0$. By the induction hypothesis, we have $c_0 \geq c'_0$ and $c_1 \geq c'_1$. Since $c'_0 < c'_1$, we also have $c'_0 < c_1$. From this it follows that $RHS = (1 - p) \cdot c'_0 + p \cdot c'_0 \leq 1 + (1 - p) \cdot c_0 + p \cdot c_1 = LHS$.

A similar proof can be done for the second case, where $c'_0 \geq c'_1$.

We have now shown that both branches of the if-statement of RHS are less than or equal to LHS, and have thereby shown $LHS \geq RHS$, which is what we wanted to show.

□

Lemma 2.3.5 (Expected cost of expanding domain)**.**
Let I and J be finite sets such that $J \subseteq I$.

$\forall\ t : DecTree(J), p \in [0, 1]$ we have

$$expCost(t, p) = expCost(expandDomain_{DT}(t), p)$$

Proof. This is trivially true, since $expandDomain_{DT}(t)$ is structurally identical to t. □

We are now ready to prove the property.

Theorem 2.3.6 (Complexity of restricting domain)**.**
$\forall\ f : BF(I).\ D_p(f) = D_p(restrictDomain_{BF}(J, f))$

where I and J are any two finite sets such that $J \subseteq I$ and $dep([t]) \subseteq J$.

Proof. Let I be any finite set and let J be any set such that $J \subseteq I$ and $dep([t]) \subseteq J$.

Let $f : BF(I)$ be any Boolean function, and let $g = restrictDomain_{BF}(J, f)$. Note that the type of g is $BF(J)$.

Let $p \in [0, 1]$ be any probability.

Let t_f^* be any tree such that $[t_f^*] = f$ that minimizes $expCost(t, p)$. Similarly, let t_g^* be any tree such that $[t_g^*] = g$ that minimizes $expCost(t, p)$.

We will show that $expCost(t_f^*, p) = expCost(t_g^*, p)$. Since we will show that this holds for any probability, it will hold for all probabilities. Therefore, this implies that f and g by definition will have the same complexity.

Let $t_g = restrictDomain_{DT}(J, t_f^*, p)$.

By lemma 2.3.2 we have $[t_g] = g$ and by lemma 2.3.4 we have $expCost(t_g, p) \leq expCost(t_f^*, p)$. By definition, $expCost(t_g^*, p) \leq expCost(t_g, p)$. Thus, we have $expCost(t_g^*, p) \leq expCost(t_g, p) \leq expCost(t_f^*, p)$, which in turn gives us $expCost(t_g^*, p) \leq expCost(t_f^*, p)$.

Let $t_f = expandDomain_{DT}(t_g^*)$.

By lemma 2.3.3 we have $[t_f] = f$ and by lemma 2.3.5 we have $expCost(t_f, p) = expCost(t_g^*, p)$. By definition, $expCost(t_f^*, p) \leq expCost(t_f, p)$. Thus, we have $expCost(t_f^*, p) \leq expCost(t_f, p) = expCost(t_g^*, p)$, which in turn gives us $expCost(t_f^*, p) \leq expCost(t_g^*, p)$.

We now have $expCost(t_g^*, p) \leq expCost(t_f^*, p)$ and $expCost(t_f^*, p) \leq expCost(t_g^*, p)$, resulting in $expCost(t_f^*, p) = expCost(t_g^*, p)$, which is what we wanted to show.

Since this holds for all probabilities $p \in [0, 1]$, we have shown that f and g have the same complexity.

□

2.3.2 Re-mapping variables

This section aims to show that if we map the input variables of a Boolean function, using a bijective function, we will not affect complexity. Again, we give some definitions that will be used in stating and proving the property.

Definition 2.3.8 (map_{BF})**.**
We define the function mapping between $BF(I)$ and $BF(J)$ as follows:

$$map_{BF} : (I \to J) \to BF(I) \to BF(J)$$
$$map_{BF}(e, f) = \lambda v \to f(v \circ e)$$

where I and J are any two finite sets.

Definition 2.3.9 (map_{DT})**.**
We define the function mapping between $DecTree(I)$ and $DecTree(J)$ as follows:

$$map_{DT} : (I \to J) \to DecTree(I) \to DecTree(J)$$
$$map_{DT}(e, Res(b)) = Res(b)$$
$$map_{DT}(e, Pick(i, t_0, t_1)) = Pick(e(i), map_{DT}(e|_{I-i}, t_0), map_{DT}(e|_{I-i}, t_1))$$

where I and J are any two finite sets such that $|I| \leq |J|$ and e injective, in order to preserve the property that a decision tree can only inspect the same variable once for any given input vector.

Stating the property

We are now ready to state the property formally:

$$\forall\ f : BF(I),\ e : (I \to J), e \text{ bijective. } D_p(f) = D_p(map_{BF}(e, f))$$

where I and J are any finite sets such that $|I| = |J|$.

In order to prove this property, we will first prove three lemmas.

Lemma 2.3.7 (Inverse map_{BF})**.**
$\forall\ f : BF(I),\ e : (I \to J)$, e bijective, we have

$$f = map_{BF}(e^{-1}, map_{BF}(e, f))$$

where I and J are any two finite sets such that $|I| = |J|$.

Proof. Let I and J be any two sets such that $|I| = |J|$. Let f be any $BF(I)$ and let e be any bijective function from I to J.

$$
\begin{aligned}
RHS &= map_{BF}(e^{-1}, map_{BF}(e, f)) \\
&= \lambda v \to map_{BF}(e, f)(v \circ e^{-1}) \\
&= \lambda v \to (\lambda v' \to f(v' \circ e))(v \circ e^{-1}) \\
&= \lambda v \to f(v \circ e^{-1} \circ e) \\
&= \lambda v \to f(v) \\
&= f \\
&= LHS
\end{aligned}
$$

□

Lemma 2.3.8 (Semantics of mapping)**.**
$\forall\ e : (I \to J),\ e$ injective, $t : DecTree(I)$ we have

$$[map_{DT}(e, t)] = map_{BF}(e, [t])$$

where I and J are any two finite sets such that $|I| \leq |J|$.

Proof. Let I and J be any two finite sets such that $|I| \leq |J|$.

Let $t : DecTree(I)$ and let e be an injective function from I to J.

We will use a proof by induction on the structure of t to prove that $[map_{DT}(e, t)] = map_{BF}(e, [t])$ for all t.

Base case

In the base case, $t = Res(b)$ for some $b \in B$.

$$\begin{aligned} LHS &= [map_{DT}(e, t)] \\ &= [map_{DT}(e, Res(b))] \\ &= [Res(b)] \\ &= \lambda v \to b \end{aligned}$$

$$\begin{aligned} RHS &= map_{BF}(e, [t]) \\ &= \lambda v \to [t](v \circ e) \\ &= \lambda v \to [Res(b)](v \circ e) \\ &= \lambda v \to (\lambda v' \to b)(v \circ e) \\ &= \lambda v \to b \end{aligned}$$

$LHS = RHS$, so we have proved the base case.

Induction step

In the induction case, $t = Pick(i, t_0, t_1)$ for some i, t_0, t_1, and we assume that $[map_{DT}(e, t_0)] = map_{BF}(e, [t_0])$. Similarly for t_1.

We will now show that $[map_{DT}(e, t)] = map_{BF}(e, [t])$:

$$
\begin{aligned}
LHS &= [map_{DT}(e, t)] \\
&= [map_{DT}(e, Pick(i, t_0, t_1))] \\
&= [Pick(e(i), map_{DT}(e|_{I-i}, t_0), map_{DT}(e|_{I-i}, t_1))] \\
&= \lambda v_J \to \text{if } v_J(e(i)) \\
&\quad \text{then } [map_{DT}(e|_{I-i}, t_0)](v_J|_{J-e(i)}) \\
&\quad \text{else } [map_{DT}(e|_{I-i}, t_1)](v_J|_{J-e(i)}) \\
&= \lambda v_J \to \text{if } v_J(e(i)) && \text{(induction hypothesis)} \\
&\quad \text{then } map_{BF}(e|_{I-i}, [t_0])(v_J|_{J-e(i)}) \\
&\quad \text{else } map_{BF}(e|_{I-i}, [t_1])(v_J|_{J-e(i)}) \\
&= \lambda v_J \to \text{if } v_J(e(i)) \\
&\quad \text{then } (\lambda v_I \to [t_0](v_I \circ e|_{I-i}))(v_J|_{J-e(i)}) \\
&\quad \text{else } (\lambda v_I \to [t_1](v_I \circ e|_{I-i}))(v_J|_{J-e(i)}) \\
&= \lambda v_J \to \text{if } v_J(e(i)) \\
&\quad \text{then } [t_0]((v_J|_{J-e(i)}) \circ e|_{I-i}) \\
&\quad \text{else } [t_1]((v_J|_{J-e(i)}) \circ e|_{I-i}) \\
&= \lambda v_J \to \text{if } v_J(e(i)) \\
&\quad \text{then } [t_0]((v_J \circ e)|_{I-i}) \\
&\quad \text{else } [t_1]((v_J \circ e)|_{I-i})
\end{aligned}
$$

$$
\begin{aligned}
RHS &= map_{BF}(e, [t]) \\
&= \lambda v_J \to [t](v_J \circ e) \\
&= \lambda v_J \to [Pick(i, t_0, t_1)](v_J \circ e) \\
&= \lambda v_J \to (\lambda v_I \to \text{if } v_I(i) \\
&\quad \text{then } [t_0](v_I|_{I-i}) \\
&\quad \text{else } [t_1](v_I|_{I-i}))(v_J \circ e) \\
&= \lambda v_J \to \text{if } (v_J \circ e)(i) \\
&\quad \text{then } [t_0]((v_J \circ e)|_{I-i}) \\
&\quad \text{else } [t_1]((v_J \circ e)|_{I-i}) \\
&= \lambda v_J \to \text{if } v_J(e(i)) \\
&\quad \text{then } [t_0]((v_J \circ e)|_{I-i}) \\
&\quad \text{else } [t_1]((v_J \circ e)|_{I-i})
\end{aligned}
$$

$LHS = RHS$, so we have proved the inductive step and the induction is done.

□

Lemma 2.3.9 (Expected cost of mapping)**.**
$\forall\ p \in [0,1],\ t : DecTree(I),\ e : (I \to J), e$ injective, $\ v : (J \to B)$.

$$expCost(t, p) = expCost(map_{DT}(e, t), p)$$

where I and J are any two finite sets such that $|I| = |J|$.

Proof. Let I and J be any two finite sets such that $|I| = |J|$.

Let $p \in [0, 1]$.

Let $t : DecTree(I)$ and e be any injective function from I to J.

We will again use a proof by induction on the structure of t to prove that $expCost(t, p) = expCost(map_{DT}(e, t), p)$ for all p, t, e.

Base case

In the base case, $t = Res(b)$ for some $b \in B$.

$$\begin{aligned} LHS &= expCost(t, p) \\ &= expCost(Res(b), p) \\ &= 0 \\ RHS &= expCost(map_{DT}(e, t), p) \\ &= expCost(map_{DT}(e, Res(b)), p) \\ &= expCost(Res(b), p) \\ &= 0 \end{aligned}$$

$LHS = RHS$, so we have proved the base case.

Induction case

In the induction case, $t = Pick(i, t_0, t_1)$ for some i, t_0, t_1, and we assume that $expCost(t_0, p') = expCost(map_{DT}(e', t_0), p')$ for all probabilities $p' \in [0, 1]$ and all injective functions $e' : (I - i) \to (J - e(i))$. Similar for t_1.

$$\begin{aligned} LHS &= expCost(t, p) \\ &= expCost(Pick(i, t_0, t_1), p) \\ &= 1 + (1 - p) \cdot expCost(t_0, p) + p \cdot expCost(t_1, p) \end{aligned}$$

$$\begin{aligned} RHS &= expCost(map_{DT}(e, t), p) \\ &= expCost(map_{DT}(e, Pick(i, t_0, t_1)), p) \\ &= expCost(Pick(e(i), map_{DT}(e|_{I-i}, t_0), map_{DT}(e|_{I-i}, t_1)), p) \\ &= 1 + (1 - p) \cdot expCost(map_{DT}(e|_{I-i}, t_0), p) + p \cdot expCost(map_{DT}(e|_{I-i}, t_1), p) \\ &= \{\text{induction hypothesis}\} \\ &= 1 + (1 - p) \cdot expCost(t_0, p) + p \cdot expCost(t_1, p) \end{aligned}$$

$LHS = RHS$, so we have proved the inductive step and the induction is done.

□

We are now ready to prove the property.

Theorem 2.3.10 (Complexity of mapping variables)**.**
$\forall\ f : BF(I),\ e : (I \to J), e$ bijective. $D_p(f) = D_p(map_{BF}(f))$

where I and J are any two finite sets such that $|I| = |J|$.

Proof. Let I and J be any two finite sets.

Let $f : BF(I)$ be a Boolean function, let e be a bijective function from I to J, and let $g = map_{BF}(e, f)$. g is than of type $BF(J)$.

Let $p \in [0, 1]$ be any probability.

Let t_f^* be any tree such that $[t_f^*] = f$ that minimizes $expCost(t, p)$. Similarly, let t_g^* be any tree such that $[t_g^*] = g$ that minimizes $expCost(t, p)$.

We will show that $expCost(t_f^*, p) = expCost(t_g^*, p)$. Since we will show that this holds for any probability, it will hold for all probabilities. Therefore, this implies that f and g by definition will have the same complexity.

Let $t_g = map_{DT}(e, t_f^*)$.

By lemma 2.3.8 we have $[t_g] = g$ and by lemma 2.3.9 we have $expCost(t_f^*, p) = expCost(t_g, p)$. By definition, $expCost(t_g^*, p) \leq expCost(t_g, p)$. Thus, we have $expCost(t_g^*, p) \leq expCost(t_g, p) = expCost(t_f^*, p)$, which in turn gives us $expCost(t_g^*, p) \leq expCost(t_f^*, p)$.

Let $t_f = map_{DT}(e^{-1}, t_g^*)$. By lemma 2.3.7, we have $f = map_{BF}(e^{-1}, g)$.

Using the same lemmas and process as above, we can conclude that $expCost(t_f^*, p) \leq expCost(t_g^*, p)$.

We now have $expCost(t_g^*, p) \leq expCost(t_f^*, p)$ and $expCost(t_f^*, p) \leq expCost(t_g^*, p)$, resulting in $expCost(t_f^*, p) = expCost(t_g^*, p)$, which is what we wanted to show.

Since this holds for all probabilities $p \in [0, 1]$, we have shown that f and g have the same complexity. □

3

Method and evaluation

This chapter explains how we identified areas in need of improvement. We outline the approach taken to assess the different implementations, pinpoint inefficiencies, and propose optimizations. Furthermore, this chapter details the tools developed to verify the correctness of our and previous implementations.

3.1 What can be improved

When optimizing an algorithm, a key step is measuring both the total execution time and the time spent in specific parts of the code. For measuring total time, we use the `time` package [20].

To gain deeper insights into which functions consume the most time, we use profiling tools provided by the Glasgow Haskell Compiler (GHC), a compiler for Haskell programs [21]. By using the tool `ghc-prof-flamegraph` together with GHC profiling, we can generate flame graphs [22] that visually represent the proportion of time spent in the various functions of a Haskell program. We analyze three different implementations for calculating the level-p complexity.

The three different implementations:

- `genAlgMemoThin` - The original algorithm from Jansson & Jansson [1]. Represents complexities as sets of polynomials and uses thinning to limit the sizes of the sets
- `piecewiseComplexity` - The version using piecewise polynomials directly [7].
- `explicitpiecewiseComplexity` - A version of `piecewiseComplexity` using explicit memoization in order to enable fast memoization of BDDs.

3.1.1 Examining the general algorithm

A more thorough explanation of how flame graphs should be interpreted can be found in [22], but the important part is that each bar in the graph represents a Haskell function, and that the width of the bar represents the total amount of time the program spent evaluating a certain function.

As we can see in fig. 3.1, a large portion of time, around 80%, is spent evaluating the `memoize` function. Out of these 80%, 60 are spent on actual memoization, and around 20 are spent evaluating the `setBit` function. The remaining 20% is spent mostly on polynomial operations. The main reason for memoization taking so long is that each time we evaluate the complexity of a sub-function, we have to look up the corresponding BDD in the memoization table to see if its complexity has already been computed. This consists of traversing the structure of the BDD until a difference is found, resulting in a large number of comparisons having to be made each time the complexity of a sub-function is computed. The function `setBit` is slow since it involves reducing the resulting BDD, although it will make subsequent comparisons more efficient.

As is shown here, we could potentially gain a lot by improving memoization and the `setBit` function. Since memoization and `setBit` take up about 80% of the time, we could reach a theoretical maximum of a 5x speedup by improving this part of the algorithm.

3.1.2 Closed under setbit

An important property when representing Boolean functions is whether or not a class of Boolean functions is closed under the function `setBit`. What this means is that applying `setBit` on a function of class c should result in a function of class c [23]. This property is currently necessary for being able to create an instance of `BoFun` for a given class, as shown in the typeclass definition in section 2.2.1. In table 3.1, we present whether or not some common function classes are closed under `setBit`.

Type	Closed under `setBit`
General	Yes
Threshold	Yes
Symmetric	Yes
Monotone	Yes
Odd	No
Even	No
Gates	Yes
Balanced	No
Exactly one **1**	No
Read-once	Yes

Table 3.1: The function classes that are closed under the function `setBit`

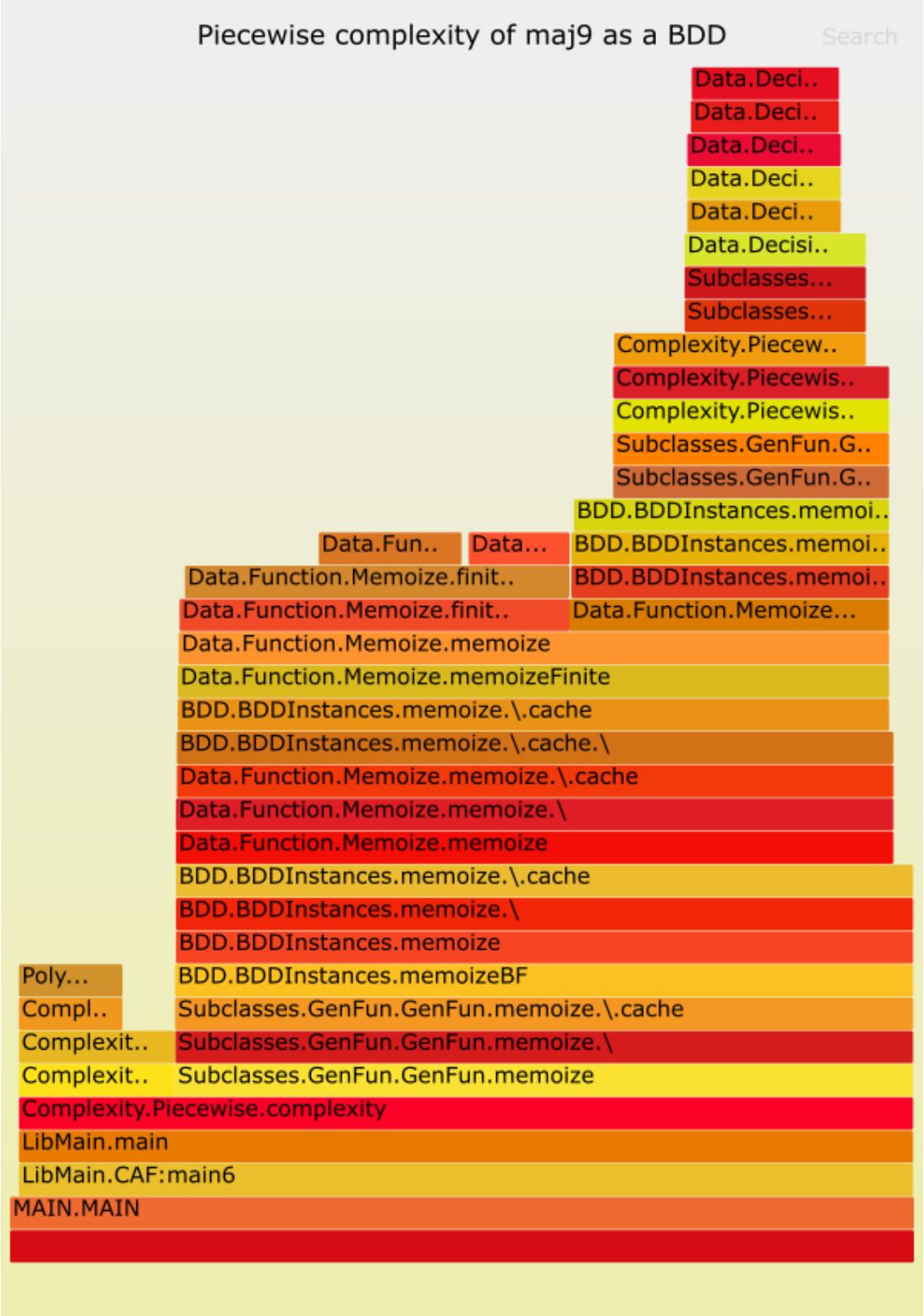

Figure 3.1: The flame graph for computing the level-p complexity of maj_9 represented as a BDD using the algorithm `piecewiseComplexity`.

3.1.3 Viable areas of improvement

The data presented in the earlier sections show us four main areas where improvements could be made:

1. Creating more efficient representations of subclasses of Boolean functions, resulting in less time spent on `memoize` and `setBit`.
2. Creating a more efficient representation of general Boolean functions, or in other ways making better use of BDDs here too, reducing the time spent in `memoize` and `setBit`.
3. If improvements are made with regards to `memoize` and `setBit`, improving the various operations on polynomials would be the next area of improvement.
4. Finally, we could find another algorithm to replace `piecewiseComplexity`, rendering some bottlenecks in `piecewiseComplexity` irrelevant.

As stated before, the main goal will be improving run time by finding more efficient representations of classes of Boolean functions, as well as improving the operations on general Boolean functions expressed as BDDs. This report will not focus on improving the operations on polynomials, and neither will it focus on changing the algorithm in `piecewiseComplexity`.

3.2 Generation and testing

To ensure the correctness of our implementation, we developed QuickCheck properties and generators for different components and the system as a whole [9]. Some of those properties include:

Correctness of complexity calculations

We test that `genAlgMemoThin`, `piecewiseComplexity`, and `explicitpiecewiseComplexity` all give the same complexity for the same function, pointing to the correctness of the different implementations.

- propPiecewiseComplexityCorrect: Ensures that `piecewiseComplexity` and `genAlgMemoThin` return the same piecewise polynomial when applied to the same boolean function. This step includes converting the piecewise polynomial returned by `piecewiseComplexity` into the corresponding representation as a set of polynomials used in [1].
- propExplicitpiecewiseComplexityCorrect: Checks that `piecewiseComplexity` and `explicitpiecewiseComplexity` yield the same complexity for the same function.

General properties of Boolean functions

Checking some general properties of Boolean functions.

- propFlipOutputCorrect: Ensures that inverting the output of a Boolean function yields a different result from the original function.

- propFlipOutputComplexity: Verifies that inverting the output of a function does not change its computed complexity.
- propFlipInputComplexity: Checks that the complexity is mirrored along the vertical line $x = \frac{1}{2}$ when the inputs are inverted.

Minimization

Properties that verify correct behavior after minimization.

- propMinimizedCorrectVars: Confirms that after minimization, a function has variables numbered from 1 to n, where n is the number of dependent bits in the original function.
- propMinimizedComplexity: Verifies that minimizing a given Boolean function does not affect its complexity. The concept of minimization is introduced in section 4.3.2 and follows from the theorems proved in section 2.3.

Other properties

- propConversionSymm: Ensures that converting from a symmetric Boolean function to the equivalent general Boolean function and back results in the original function.
- propRepsCorrect: Verifies that the flat majority function, expressed as a general, symmetric, or threshold function, yields the same result independent of the representation.
- propRationalSign: Checks that the sign of a polynomial at a rational point remains the same whether the point is represented as a rational number or an algebraic number.

To verify the correctness of our implementation it is important to generate a wide variety of examples to test it on. In order to do this we used QuickCheck generation to produce a large variety of test cases, with a wide range of complexities.

We can look at the generation of threshold functions as an example. The implementation defines a structure to generate a variety of threshold functions with adjustable complexity by setting an "arity" (or number of inputs). The `arbitraryArity` function creates instances of `ThresholdFun` by using the `generateThreshold` function, which randomly determines the count of "true" (nt) and "false" (nf) values in the output based on the arity. This randomization ensures a broad range of threshold functions, covering everything from simple cases (all true or all false) to complex ones.

4

Implementation of optimizations

This chapter presents the implementations of optimizations concerning level-p-complexity. The first two sections, 4.1 and 4.2, describe optimizations created by limiting the scope to sub-classes of Boolean functions, while the third section (section 4.3) details optimizations applicable for all Boolean functions.

4.1 Symmetric functions

We define a Boolean function to be *symmetric* if the result does not depend on the order of the input bits. In other words, the result for a given input depends only on the number of **1**'s in the input.

The representation of this class consists of a list of Boolean values `xs` of length $n+1$, where n is the arity of the function. If the number of **1**'s in the input is k, then the function yields the value `xs !! k`.

4.2 Threshold functions

We define a Boolean function to be a *threshold* function if it has n bits which yields the value **1** if and only if the number of **1**'s in the input is greater than or equal to some threshold k, where $0 \leq k \leq n+1$.

If $k = 0$, we know that the function is constant and must yield **1**. If $k = n+1$, we know that the function is constant and must yield **0**, as we can never have $n+1$ **1**'s in an input consisting of n values.

The main benefit of this representation is the fact that memoization is very efficient (we only have to lookup the two values in the tuple (n, k), and the `setBit` operation only requires decrementing either just n, or n and k.

In practice, the datatype representing a threshold function is represented as a tuple of two integers (k_t, k_f), representing the number of **1**'s and **0**'s needed for the function to yield **1** and **0** respectively. Note that for an n-bit function we have the invariant $k_t + k_f = n + 1$.

The intuition for this representation is as follows: suppose the input contains n_t **1**'s and n_f **0**'s, so $n_t + n_f = n$. Then, the condition $(n_t - k_t) + (n_f - k_f) = -1$ holds.

Since these are integers, exactly one of $n_t - k_t$ and $n_f - k_f$ must be non-negative. In other words, exactly one of $n_t \geq k_t$ and $n_f \geq k_f$ is true, which is exactly the desired behavior.

4.3 General Boolean functions

Finally, we describe the implementation of general Boolean functions. We identify two areas of improvement and present solutions to make them more efficient.

As previously stated, the representation of general Boolean functions presented in Jansson and Jansson [1] used binary decision diagrams, and specifically the Hackage library `decision-diagrams`. This data structure efficiently represents general Boolean functions as DAGs, but, as seen in section 3.1.1, a significant amount of time is needed to compare BDDs in the memoization step, as we need to traverse the entire DAG each time.

Another inefficiency is that we currently do not use the all the information we have available to reduce the number of times we have to calculate complexities. As an example, we look at the 3-bit functions $g(x_1, x_2, x_3) = x_1 \wedge x_2$ and $h(x_1, x_2, x_3) = x_2 \wedge x_3$. Since they are different functions, we currently have to calculate their complexities separately. However, by the properties proved in section 2.3, we can show that these functions have the same complexity. Therefore, it would be advantageous to us if we could use this fact in the complexity calculation, in order to only have to calculate this complexity once.

4.3.1 Hash consing for fast comparison

The problem of slow comparisons by comparing DAGs can be easily solved using functionality from the `decision-diagrams` library. As stated previously in section 2.1.6, `decision-diagrams` uses hash consing for fast comparison of BDDs. Employing this type of fast hashing and comparison lets us speed up the memoization step greatly, as we go from having to compare the DAGs up until the first point of difference, to only having to compare a single pair of values with each other.

A problem with this solution is that we can no longer use the `memoization` library, as `decision-diagrams` does not expose the internal node IDs. We have circumvented this issue by rewriting `piecewiseComplexity` to use a `Data.HashMap` in the `State` monad in order to perform explicit memoization.

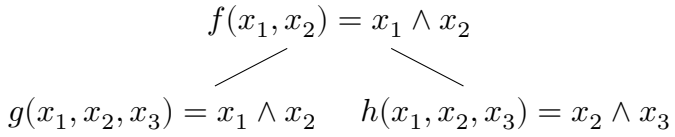

Figure 4.1: Two 3-bit functions g and h being minimized to the same function f

4.3.2 Minimized general functions

The problem of functions like $g(x_1, x_2, x_3) = x_1 \wedge x_2$ and $h(x_1, x_2, x_3) = x_2 \wedge x_3$ having the same complexity but not utilizing memoization can be solved through a process of minimization.

Definition 4.3.1 (Minimization)**.**
We say that an n-bit Boolean function is minimized if it is dependent on all of its bits, and its input domain is precisely the variables $\{x_1, x_2, ..., x_n\}$.

In order to minimize a given function, we make use of the properties proved in section 2.3. This process consists of two steps:

Step 1

For the first step, theorem 2.3.6 tells us that for any function f that is only dependent on some of its input bits, we can create another function f' with the same complexity as f and having only the bits that f depends on in its input domain. Using fig. 4.1 as an example, the function $g(x_1, x_2, x_3) = x_1 \wedge x_2$, is only dependent on x_1 and x_2. By theorem 2.3.6, we know that the function $h'(x_1, x_2) = x_1 \wedge x_2$ has the same complexity as h. Similarly, we know that $g'(x_2, x_3) = x_2 \wedge x_3$ has the same complexity as g.

Step 2

For the second step of the minimization process, we make use of theorem 2.3.10. This theorem states that we can remap the variable indices of a function using a bijective function without affecting complexity. Continuing our example, the function $h'(x_2, x_3) = x_2 \wedge x_3$ can be remapped to $h''(x_1, x_2) = x_1 \wedge x_2$. g' already fulfills the conditions for being minimized, and therefore we simply say $g'' = g'$.

As we can see, $f = g'' = h''$. This means that we can now use the same memoization record for g and h, namely the minimized version of them, as we have proved that they have the same complexity.

As a more complex example, the function $f(x_1, x_2, x_3, x_4, x_5, x_6) = x_2 \wedge (x_5 \vee x_3)$ would be minimized to $f'(x_1, x_2, x_3) = x_1 \wedge (x_3 \vee x_2)$.

Minimization and BDDs In the special case of minimizing Boolean functions represented as *BDDs*, special care is taken in the remapping step. According to theorem 2.3.10, we could choose any bijective function with $\{x_1, x_2, ..., x_n\}$ as its codomain in the second step of the minimization step. However, this would likely require a complete rebuild of the BDD, if the variable ordering is not preserved. This problem is solved by ensuring that the mapping function is chosen in such a

way that the mapped variables follow the same ordering.

In other words, if a function f is dependent on the variables $x_{i_1}, x_{i_2}, ..., x_{i_n}$, where $i_1 < i_2 < ... < i_n$, we map x_{i_1} to x_1, x_{i_2} to x_2, ..., x_{i_n} to x_n. This ensures that ordering is preserved, and allows us to simply rename the variables in the underlying BDD, without needing to change the structure of it.

The concrete implementation of this consists of creating a newtype `MinimizedGenFun` wrapping the BDD and adding the minimization step to `setBit`. Thus, an invariant for `MinimizedGenFun` is that it is always Minimized. A smart constructor ensures that a non-Minimized instance of `MinimizedGenFun` cannot be created.

4.3.3 Canonical BDDs

One property of level-p-complexity for BDDs is that flipping or inverting the Boolean function does not change its complexity. Building on this, we developed a concept we call canonical form.

We define a BDD as canonical if the leftmost path reaches the terminal node **0**. Under this definition, a canonical BDD evaluates to **0** when all input variables are set to **0**. This approach is chosen to keep representations consistent, simplifying comparisons of BDDs.

The function toCanonicForm is used to convert a BDD to canonical form. This function checks whether the BDD meets the canonical criteria by using a helper function, inCanonicForm. If the BDD already satisfies the canonical condition, toCanonicForm returns it unchanged. When the BDD does not meet the criteria, the function inverts it, ensuring that the leftmost path reaches **0**.

The goal of this approach was to provide a straightforward method to standardize the BDD structure, simplifying comparison.

5

Tools for exploring level-p-complexity

We have implemented several tools and functionalities to better examine the level-p-complexity of Boolean functions.

5.1 Comparing polynomials

Calculating the critical points of a piecewise polynomial can give us valuable insight into the complexity of a Boolean function. It would require us to compare the polynomials the piecewise polynomial consists of. The question then becomes: how do we compare polynomials?

First, we must note that not all polynomials are easily comparable. An example is $p = x$ and $q = 1 - x$, where $q \leq p$ when $0 \leq x \leq 1/2$, but $p \leq q$ when $1/2 \leq x \leq 1$. Neither is strictly less than the other one; $p(x) < q(x)$ when $0 \leq x < 1/2$, but $q(x) < p(x)$ when $1/2 < x \leq 1$.

One way to compare two polynomials such as p and q, is to check whether one of the polynomials is greater than the other within some interval. Since this thesis is concerned with probabilities, that interval is $[0, 1]$. We want to know if $p \leq q$ in this interval, which is equivalent to $0 \leq q - p$. By examining the roots of the polynomial $q - p$, we can easily find out whether p and q are comparable, and if so which one of them is greater.

5.2 Critical points of piecewise polynomials

However, when calculating the critical points of our piecewise polynomials, there were some issues with the accuracy. Although the coefficients of the polynomials were rational, the roots themselves were real but not necessarily rational. This discrepancy led to minor inaccuracies in pinpointing the exact root values. To resolve this, we implemented a method using algebraic numbers (see section 2.1.8) for precisely analyzing the roots of polynomials at irrational points.

To identify critical points in piecewise polynomials, we divided the task into three primary parts: finding critical points within individual polynomial pieces, locating

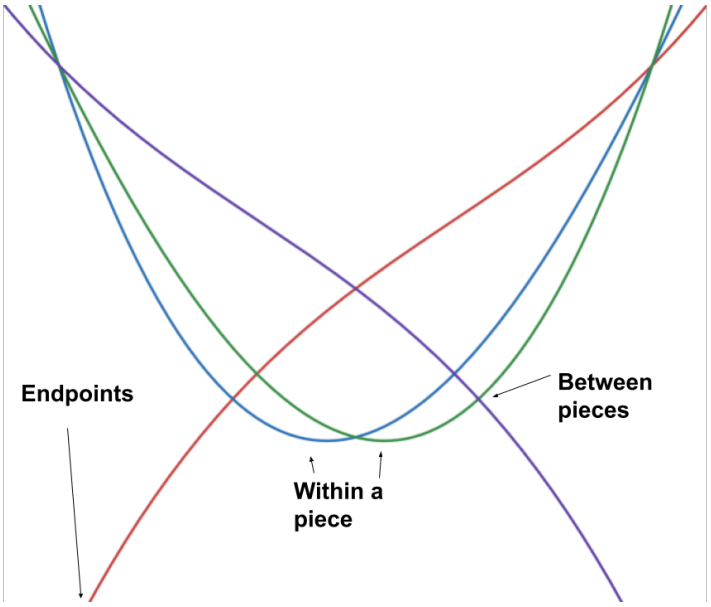

Figure 5.1: A set of polynomials representing the level-p complexity for a 4-bit Boolean function containing the three different types of critical point.

critical points at boundaries between polynomial pieces, and addressing the behavior of endpoint on the interval [0,1]. Examples of these types of critical points appear in fig. 5.1.

5.2.1 Critical points within a piece

The function `criticalPointsInPiece` finds critical points (maxima or minima) within a single piece of a piecewise polynomial. Given two boundary points and a polynomial, the function checks if the polynomial is constant (using the `constP` function). If the polynomial is constant, it returns a single uncertain critical range (`URange`), representing the possibility that the entire interval is a flat segment. If the polynomial is not constant, the function calculates its derivative (using `derP`) and looks for places where the derivative changes sign, which signals a critical point. These sign changes indicate critical points: a change from positive to negative signifies a maximum, while a change from negative to positive signifies a minimum.

To ensure the boundaries of the critical points are accurate, the function calls `findRationalEndpoints` to determine rational approximations for the algebraic boundaries. The function `shrinkIntervalStep` then helps narrow the interval around the critical point. This step progressively reduces the size of the interval, making it smaller and more precise until it gets the critical point's location. More details on how `shrinkIntervalStep` works can be found in section 5.2.4.

The function `criticalPointsInPiece'` handles the core logic of finding critical points by recursively examining the interval. It uses the derivative of the polynomial and `numRootsInInterval` to check how many roots (i.e., places where the derivative is zero) exist within the current interval. Depending on the number of roots found:

1. If there are no roots, there are no critical points in the interval.
2. If there is one root, the function checks the type of critical point using `criticalType`. If a valid type is found, it records the critical point.
3. If there are multiple roots, the function splits the interval in half and continues the search in both halves until the intervals are small enough to identify individual critical points.

It is important to note that the function `numRootsInInterval` does not return the exact number of roots in the specified interval, but instead uses Descarte's rule of signs [24] in order to find a number k such that the number of roots n is less than or equal to k, and $k - n$ is even. However, this means that if the result of `numRootsInInterval` is 0 or 1, then we know that the number of roots must be exactly 0 or exactly 1 respectively, and this is the only information we need.

This shrinking process continues until the interval is refined enough that no further splitting is needed, telling us that we have accurately identified a critical point. The `criticalPointsInPiece` function can be found in listing 9 in the appendix.

5.2.2 Critical points between pieces

Critical points can also appear where two polynomial pieces meet. The function `criticalPointBetweenPieces` checks the shared boundary (a point represented by s) between two adjacent pieces to see if the derivatives of the pieces change sign at that boundary. To do this, it first calculates the derivatives of both polynomials using `derP` and evaluates the signs of these derivatives at the shared boundary s using `signAtAlgebraic`.

If the signs of the derivatives differ (for example, one is positive and the other is negative), this indicates a change in the function's direction, meaning there is a critical point (either a maximum or a minimum) at that boundary. The function returns this critical point as an uncertain critical point (`UPoint`), which will be resolved later to confirm whether it is indeed a maximum or minimum.

If the signs are the same, no critical point exists at the boundary because the function does not change direction. The result is Nothing, indicating no critical point between these pieces. The `criticalPointBetweenPieces` function can be found in listing 10 in the appendix.

5.2.3 Endpoints of [0,1]

The endpoints 0 and 1 define the boundaries of the interval where the piecewise polynomial is evaluated. The function `addEndPoints` makes sure to check these points for possible maxima or minima. It evaluates the polynomial at both endpoints and compares the values (`zeroVal` and `oneVal`). If the values are different, the function assigns a minimum to the lower value and a maximum to the higher value. If the values are the same, the entire range might be a flat segment.

When uncertain critical points are already present, `addEndPoints` uses two helper functions, `addFirst` and `addLast`, to manage the boundaries. The `addFirst` function ensures that the starting point 0 is classified correctly by examining the first uncertain critical point. If the first point suggests a maximum, it adds a minimum at 0. Similarly, if it suggests a minimum, it adds a maximum at 0 as well.

The `addLast` function handles the endpoint 1 in a similar way. It checks the last uncertain critical point and, if it indicates a maximum, adds a minimum at 1. If the last point indicates a minimum, it adds a maximum at 1. These functions ensure that the endpoints are not overlooked and are classified correctly. The `addEndPoints` function, as well as its helper functions can be found in listing 11 in the appendix.

5.2.4 `shrinkInterval` and `signAtAlgebraic`

The `shrinkIntervalStep` function shrinks the interval around potential critical points to improve precision. If the input is a rational number, it returns the value using Haskell's arbitrary precision representation. For algebraic numbers, it repeatedly shrinks the interval by calculating the midpoint and checking where roots of the polynomial's derivative lie, in a similar way to a binary search.

In `shrinkIntervalStep'`, the interval $[low, high]$ is divided at the midpoint. If the midpoint is a root, it returns the midpoint. Otherwise, the function checks which half of the interval contains a root using `numRootsInInterval` and continues shrinking the appropriate half. This process ensures that the interval becomes small enough to accurately capture the critical point.

An example of this process can be found in fig. 5.2. For the polynomial $x^3 + 2x^2 + 1$, we begin with an initial upper and lower bound. The upper bound is first shrunk to the midpoint between the original bounds. This process is repeated, shrinking the upper bound to the new midpoint. Subsequently, the lower bound is shrunk to the current midpoint. By iteratively narrowing the interval through this method, we can "zoom in" on an interval containing a single root to an arbitrary degree.

The `signAtAlgebraic` function checks the sign of a polynomial q at an algebraic number a. If the algebraic number is rational, the function directly proceeds to `signAtAlgebraic'`. If the algebraic number is an algebraic number, the function first checks if the algebraic number is a root with the polynomial q. If they share a root, the sign is zero. If not, the function normalizes the limits of the algebraic number and proceeds to check whether the polynomial q has the same sign in the entire interval describing the algebraic number. If it does, the value of q is calculated at the midpoint of the interval. Otherwise, the interval is shrunk, ensuring that it still contains the correct root. This process is repeated until the polynomial q no longer changes sign within the interval. The implementation of `signAtAlgebraic` can be seen in listing 6.

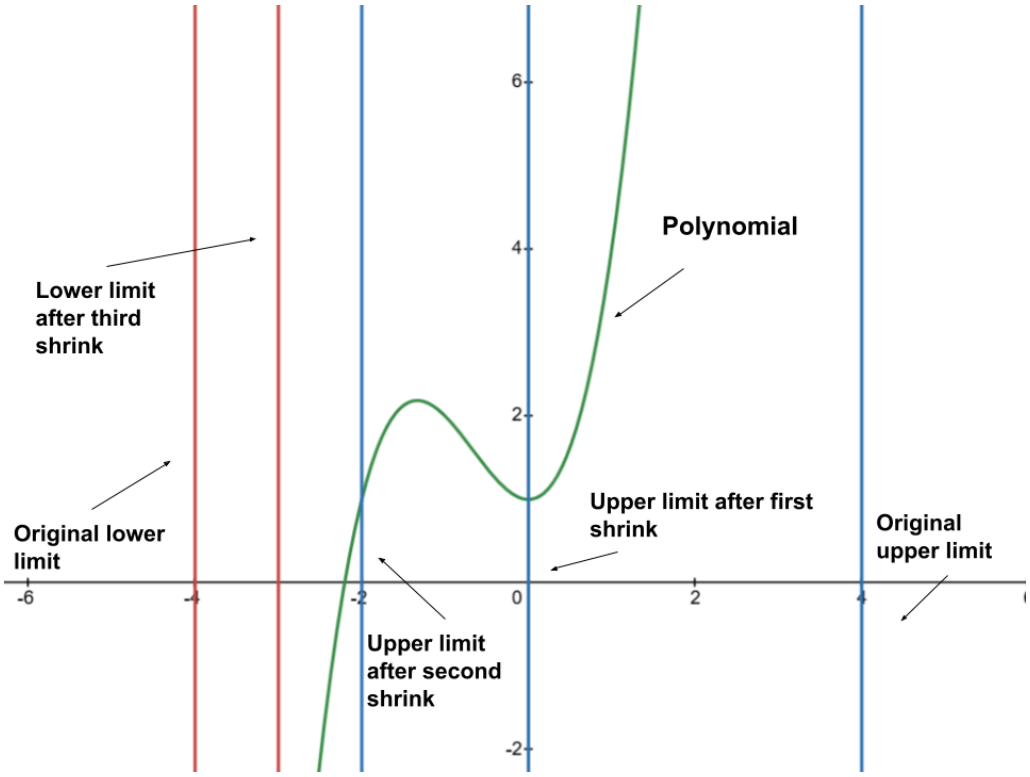

Figure 5.2: An illustration of the shrinking process for a polynomial with x^3+2x^2+1 as its derivative, in order to find the derivative's root.

5.2.5 Finalizing critical points

The function `critcalPointsPW` brings everything together to create a complete list of critical points for a piecewise polynomial. It does this by performing three main steps:

1. Finding midpoints: The function `findMidPoints` locates potential critical points within each polynomial piece and at the boundaries between pieces.
2. Adding endpoints: The function `addEndPoints` ensures that the critical points at the boundaries of the interval [0, 1] are considered.
3. Determining certainty: Finally, `determineUncertain` resolves any uncertain points by classifying them definitively as maxima or minima.

This process ensures that all potential maxima and minima are accounted for, whether they occur within a piece, at the boundary between pieces, or at the endpoints of the interval. The method of shrinking intervals and checking for sign changes in the derivative guarantees precision, even when dealing with tricky boundary cases or subtle changes in the function's behavior. The `criticalPointsPW` function can be found in listing 12 in the appendix.

```
1
2 signAtAlgebraic :: Algebraic -> Poly Rational -> Sign
3 signAtAlgebraic a@(Rational _) p = signAtAlgebraic' a p
4 signAtAlgebraic (Algebraic a) q
5   | shareRoot a q = Zero
6   | otherwise = signAtAlgebraic' (Algebraic $ normalizeLimits a) q
7
```

Listing 6: The `signAtAlgebraic` Function

5.3 Composing Boolean functions

Here, we give implementations for multi-composed and iterated functions, described in section 2.1.3.

5.3.1 MultiComposed

Building on definition 2.1.3, we implement a multi-composed function h as a data type containing an n-ary Boolean function f and a list of n Boolean functions gs. f is limited to being indexed over the set of `Int`s, as an `Int` is needed to index into gs. The definition, as well as the `BoFun` instance for `MultiComposed` is given in listing 7. In practice, a smart constructor is used in order to insure that no sub-functions are constant.

5.3.2 Iterated

The implementation of iterated functions builds upon definition 2.1.4. The implementation is given in listing 8. Note that a `Const` constructor has been added in order to handle the case where `setBit` is called on the identity function. The `BoFun` instance is also given in listing 8.

Looking at the `BoFun` instance for `Iterated`, and again looking at iterated functions as rose trees [14], we see that an iterated function is constant only if the function in the root node is constant, or if it consists of the `Const` constructor. The variables are of the type `[i]`, where each `i` determines which sub-function should be chosen at each node in the function tree. For example, the variable `[3, 2, 3]` means "At the root node, choose the third sub-function. Then, for that function, choose its second sub-function. Finally, for that function, choose its third sub-function.". Looking at the definition of `setBit`, we see that this path specification must end at a leaf node, i.e. a node containing `Const v` or `Pure ()` rather than another sub-function.

Note that in this implementation, all of the multi-composed functions must be of the same type. As an example, we could create an iterated threshold function by only composing threshold functions over iterated threshold functions, but we don't allow multi-composing a gate function over a threshold function.

```haskell
data MultiComposed f g = MultiComposed {
  mcFun     :: f,
  mcSubFuns :: [g]
}

instance (BoFun f Int, BoFun g j)
=> BoFun (MultiComposed f g) (Int, j) where
  isConst :: MultiComposed f g -> Maybe Bool
  isConst = isConst . mcFun
  variables :: MultiComposed f g -> [(Int, j)]
  variables lf = do
    (subFun, i) <- zip (mcSubFuns lf) naturals
    j <- variables subFun
    return (i, j)
  setBit :: ((Int, j), Bool) -> MultiComposed f g -> MultiComposed f g
  setBit ((i, j), v) lf = case isConst subFun' of
    Nothing -> lf{mcSubFuns = start ++ (subFun' : end)}
    Just v' -> lf{mcFun = setBit (i, v') $ mcFun lf,
        mcSubFuns = start ++ end}
    where
      (start, subFun, end) = fromJust $ splitList i $ mcSubFuns lf
      subFun' = setBit (j, v) subFun

splitList :: Int -> [a] -> Maybe ([a], a, [a])
splitList n xs = case end of
  []       -> Nothing
  (x: end') -> Just (start, x, end')
  where
    (start, end) = splitAt n xs
```

Listing 7: The complete definition of the `BoFun` instance for `MultiComposed`

```haskell

type SubFun f = MultiComposed f (Iterated f)
data Iterated f = Const Bool | Id | Iterated (SubFun f)

instance (BoFun (SubFun f) (i, [i])) => BoFun (Iterated f) [i] where
  isConst :: Iterated f -> Maybe Bool
  isConst (Const v)    = Just v
  isConst Id           = Nothing
  isConst (Iterated f) = isConst f

  variables :: Iterated f -> [[i]]
  variables (Const _)    = []
  variables Id           = [[]]
  variables (Iterated f) = map (uncurry (:)) $ variables f

  setBit :: ([i], Bool) -> Iterated f -> Iterated f
  setBit _              (Const _)    = error "setBit on const"
  setBit ([], v)        Id           = Const v
  setBit _              Id           = error "Too many levels in path"
  setBit (i : is, val) (Iterated v) = Iterated $ setBit ((i, is), val) v
  setBit ([], _)        (Iterated _) = error "Too few levels in path"
```

Listing 8: The `Iterated` data type and its `BoFun` instance

6

Results

In this chapter we present and evaluated the performance of three different implementations:

- `genAlgMemoThin` - The original algorithm from [1]. Represents complexities as sets of polynomials and uses thinning to limit the sizes of the sets.
- `piecewiseComplexity` - The improved version using piecewise polynomials directly [7].
- `explicitpiecewiseComplexity` - A version of `piecewiseComplexity` using explicit memoization in order to enable fast memoization of BDDs.

We calculated the complexity of three example functions: flat 11-bit majority, 2-level 3-bit majority, and 3-level 3-bit majority. For each of these example functions, we analyzed the impact of using canonical forms, minimization, both canonical and minimization (referred to as Both or BothGenFun), and specialized representations for threshold, symmetric, and iterated function classes.

The example functions have the following complexities:

maj_{11}

$6 + 6x + 6x^2 + 6x^3 + 6x^4 + 6x^5 - 786x^6 + 2184x^7 - 2436x^8 + 1260x^9 - 252x^{10}$

maj_2^3

$4 + 4x + 6x^2 + 9x^3 - 61x^4 + 23x^5 + 67x^6 - 64x^7 + 16x^8$

maj_3^3 The complexity for this function consists of a very long and complex piecewise polynomial and can be found in appendix A.1.

6.1 Data

The measurements for each of the algorithms, example functions, and configurations can be seen in table 6.1, table 6.2 and table 6.3.

Function	General	Canon.	Mini	Both	Thresh.	Iter.Thresh.	Symm.	Iter.Symm.
maj_{11}	9.545	9.412	0.079	0.081	0.0004	0.0002	0.0009	0.0002
maj_2^3	0.653	0.652	0.061	0.061	N/A	0.0215	N/A	0.0201
maj_3^3	Out of memory	Out of memory	Out of memory	Out of memory	N/A	$>$ 15 min	N/A	$>$ 15 min

Table 6.1: `genAlgMemoThin`'s total computing time for example functions (in seconds)

Function	General	Canon.	Mini	Both	Thresh.	Iter.Thresh.	Symm.	Iter.Symm.
maj_{11}	9.556	11.296	0.078	0.087	0.0006	0.0002	0.0014	0.0002
maj_2^3	0.967	0.961	0.097	0.094	N/A	0.050	N/A	0.053
maj_3^3	Out of memory	Out of memory	Out of memory	Out of memory	N/A	$\sim$15 min	N/A	$\sim$15 min

Table 6.2: `piecewiseComplexity`'s total computing time for example functions (in seconds)

Function	General	Canon.	Mini	Both
maj_{11}	1.354	1.362	0.015	0.016
maj_2^3	0.577	0.579	0.063	0.063
maj_3^3	$>$ 30 min	$>$ 30 min	35 min	35 min

Table 6.3: `explicitpiecewiseComplexity`'s total computing time for example functions (in seconds)

6.2 Computing time of the 11-bit majority function

The result for maj_{11} is illustrated in fig. 6.1 and fig. 6.2. Fig. 6.1 shows how long it takes, in seconds, to compute the maj_{11} function with each of the three implementations: `genAlgMemoThin`, `piecewiseComplexity`, and `explicitpiecewiseComplexity`. Each bar represents the different configurations.

Similarly, fig. 6.2, shows how much each optimization technique improves computing time, compared to the original value. The factor decreases in total computing time. Higher bars mean a bigger time-saving. Equivalent graphs for the maj_2^3 can be found in the appendix.

The use of specialized representations such as threshold, iterated threshold, symmetric, and iterated symmetric drastically reduced computation times compared to the general method. For the maj_{11} function, times decreased from several seconds to milliseconds, resulting in about a 10000x speedup in certain cases.

Minimization also consistently improved computation efficiency across all implementations. As we can see in fig. 6.2, minimization resulted in a 100x speedup. We

also see in fig. 6.1 that `genAlgMemoThin` and `piecewiseComplexity` perform very similarly for the function maj_{11}, tellings us that while there is no significant gain from using piecewise polynomials instead of a list of polynomials, there is also no significant loss.

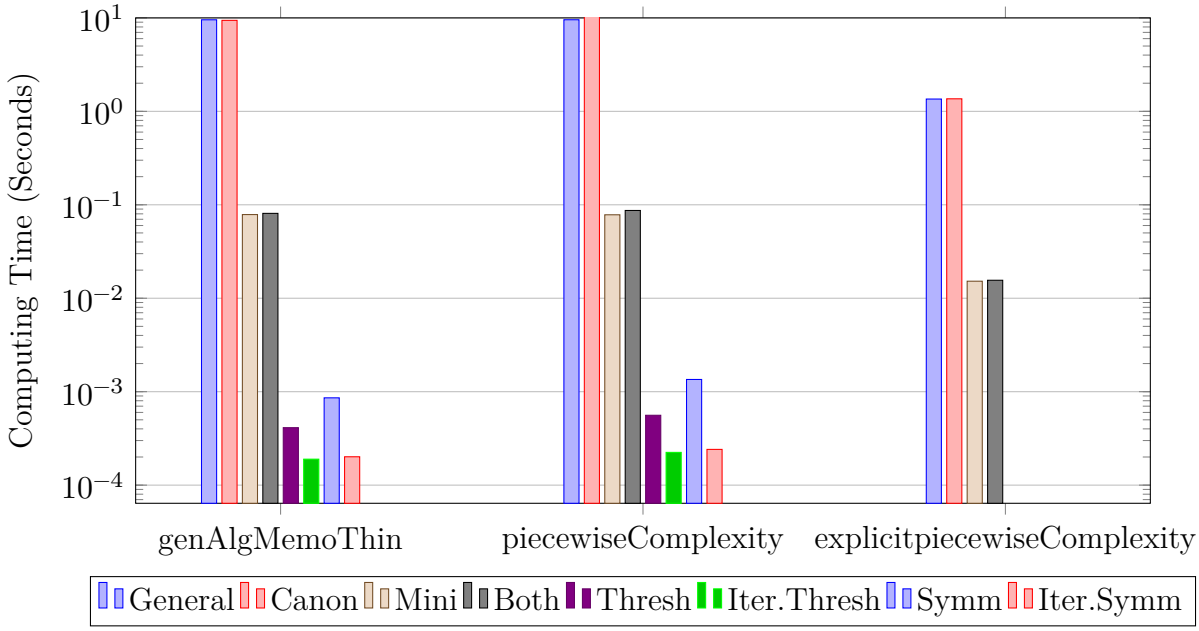

Figure 6.1: Log-scale bar chart for maj_{11} showing the total computing time (s)

6.3 Computing time of n-bit majority

We also explored how the computing time changes depending on the number of bits for the different configurations. This can be seen in fig. 6.3. For all implementations, General and CanonicalGenFun show rapid exponential growth in computing time as the number of bits increases beyond 7 and both configurations only reach around 11 bits in 10 seconds. MinimizedGenFun and BothGenFun grow less steeply compared to General and CanonicalGenFun. Computing time is manageable up to 17 bits before it reaches around 10 seconds. We also see the benefits of minimization become more and more apparent the higher the number of bits the function has as we have a higher number of comparisons.

Thresh and Symm show slower growth rates, staying steady even for larger numbers of bits. For example, at 301 bits, computing time is just over 6 seconds. IterThresh and IterSymm exhibit sharp increases in computing time beyond 17 bits. They both manage to get to 51 bits in 4 seconds.

Explicit memoization (Hash Consing) improved performance for all the configurations tested. Similarly to minimization, the benefits of the explicit memoization become more and more apparent the higher the number of bits the function has as we have a higher number of comparisons. For the maj_{11} function, the improvement was around 5x to 7x in speedup, but increasing to 17 bits resulted in an improvement of around 60x to 70x speedup.

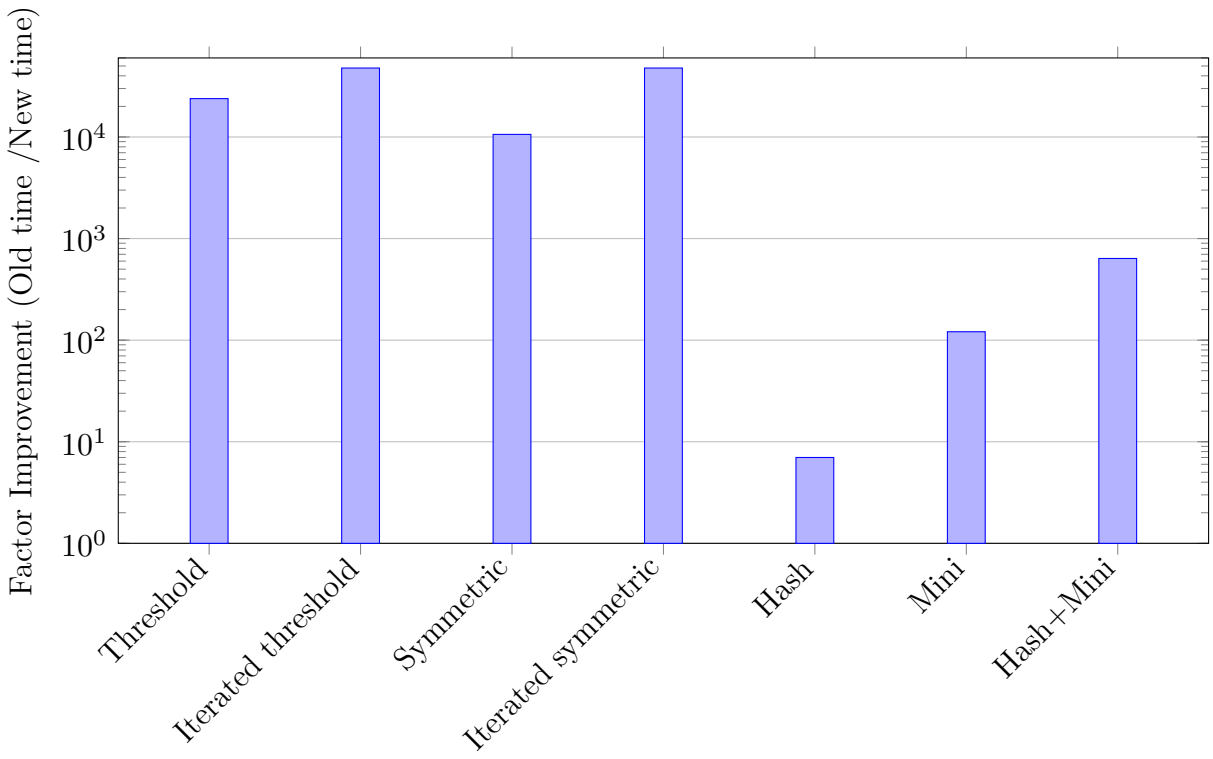

Figure 6.2: Log-scale bar chart for maj_{11} showing the factor improvement compared to the original (old time /new time)

6.4 Computing time for randomly generated functions

We also examined how computing time differed across various implementations when applied to randomly generated functions. Box plots were used to summarize the distribution of computing times for different approaches.

In fig. 6.4 for randomly generated GenFuns, we see that adding minimization resulted in a more consistent performance, with a narrower inter quartile range (IQR) and fewer outliers. In addition, through observing the dashed lines we can see that using the `explicitpiecewiseComplexity` method generally resulted in faster computing times, with reduced variability compared to `genAlgMemoThin`.

As seen in fig. 6.5 for randomly generated function representations, the threshold functions were consistently faster than for example symmetric. However, symmetric demonstrated relatively consistent performance across all complexity levels. Both of these two representations could handle a much larger number of bits for than the general method. Iterated had higher variance. Most likely cause an iterated function with 10 bits can have a different number of layers.

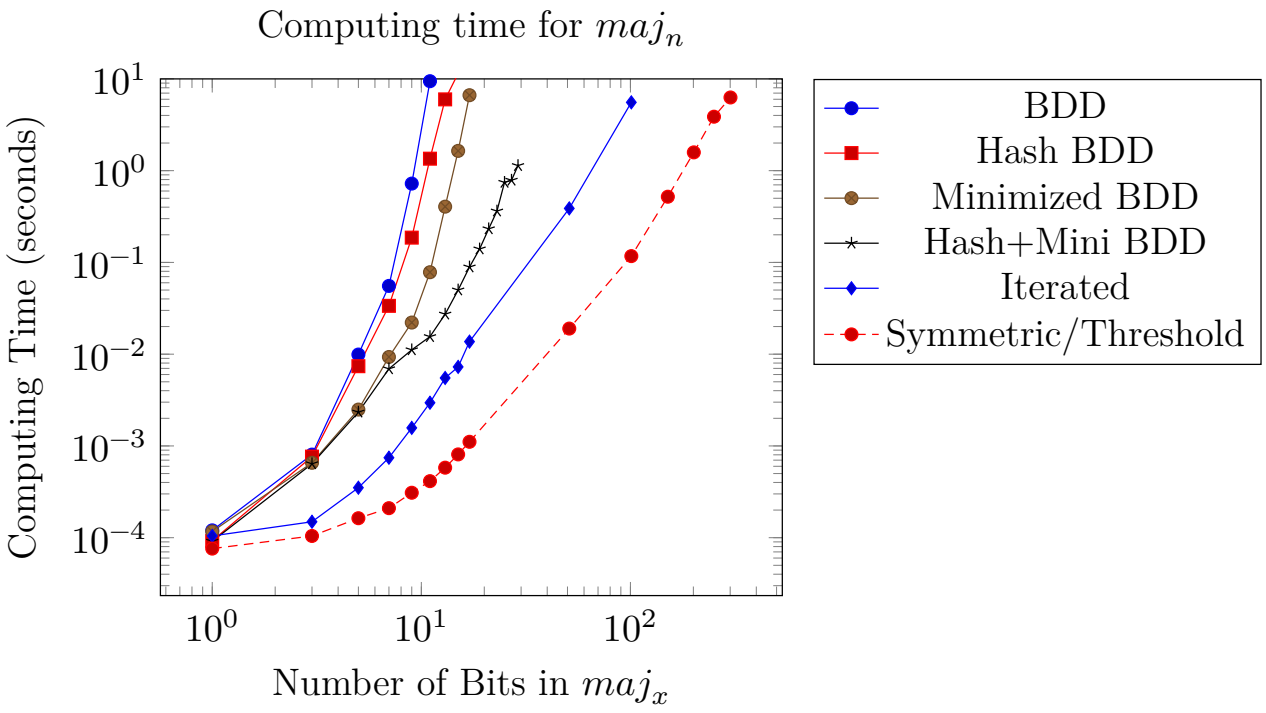

Figure 6.3: Log scale line chart for Computing Time for the Majority Function (maj_x) where x is the number of bits. Calculated for each of the different configurations

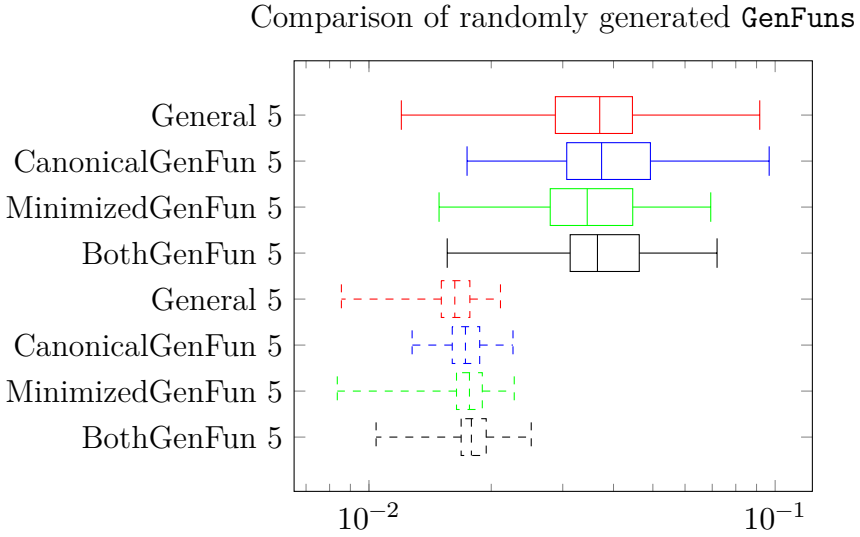

Figure 6.4: Computing time for the various implementations of the general function class, with 100 random samples of each function class. Solid lines represent complexities computed using `genAlgMemoThin`, dashed using `explicitpiecewiseComplexity`.

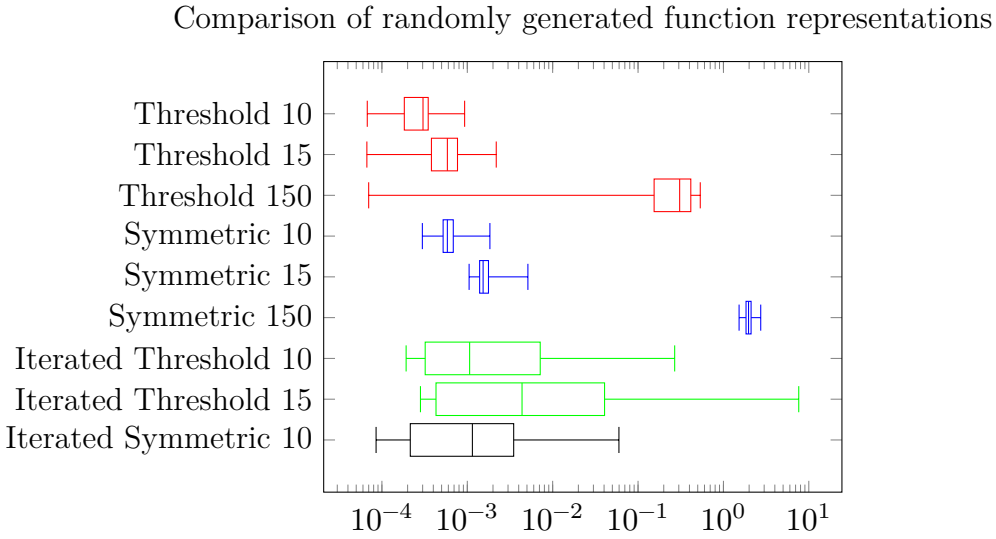

Figure 6.5: Computing time for selected function classes, using `genAlg` with 100 random samples of each function class.

6.5 Efficiency gains

If we instead look at fig. 6.6, which represents the evaluation of the 301-bit majority function represented as a threshold function using the implementation described in section 2.2.3, we see that the vast majority of the time, around 95%, is spent solely on polynomial calculations, while most of the remaining 5% is spent on memoization. The total time spent evaluating the function was 7.45 seconds, so the time spent is not insignificant. This is especially interesting when we consider the fact that the number of polynomial calculations will generally grow as the number of input bits grows. Comparing this division of compute time to the one in fig. 3.1, we see that the memoization part of the program has shrunk from 60% to 5%, with no increase in the time spent on polynomial operations.

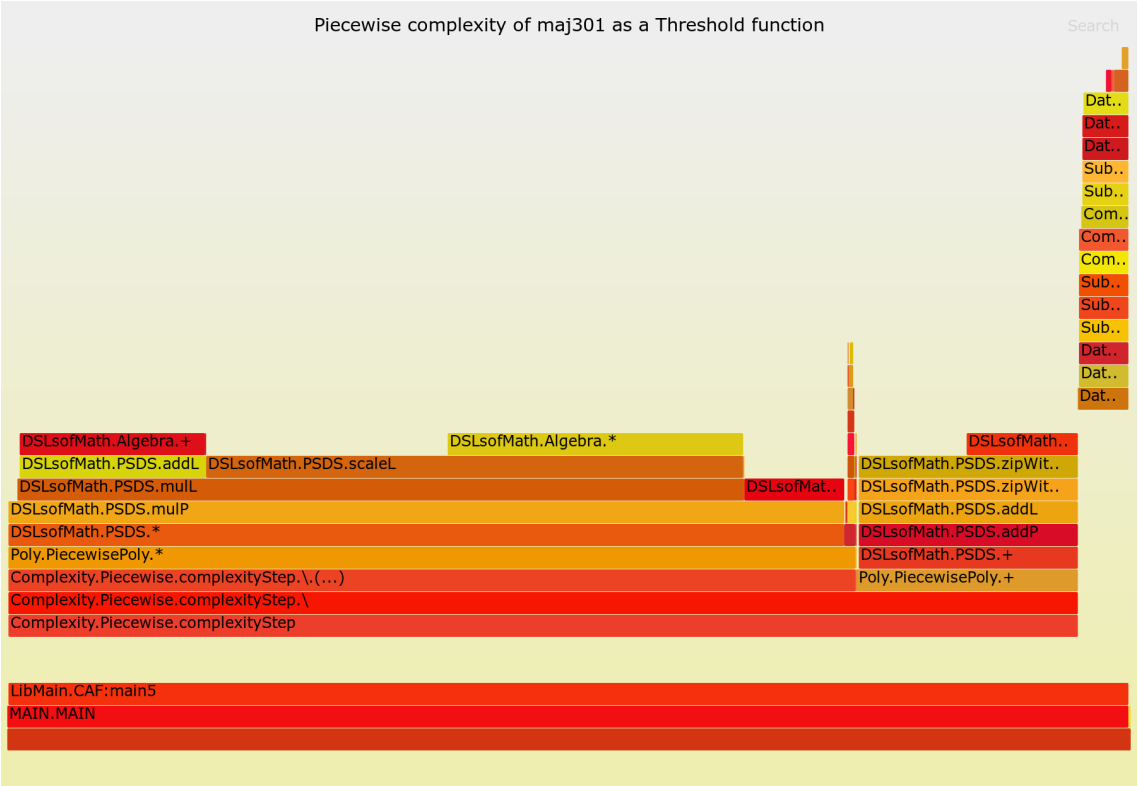

Figure 6.6: The flame graph for computing the level-p complexity of maj_{301} expressed as threshold function using `piecewiseComplexity`.

7

Conclusion

In this thesis, we explored the level-p-complexity of subclasses of Boolean functions. Our main goal was to improve the efficiency of evaluating these functions by extending existing methods and developing specialized representations. The results presented show significant progress in achieving this goal.

One of the key contributions was extending the library to handle specialized classes of Boolean functions more effectively. By introducing specialized representations like threshold and symmetric, we were able to reduce the time spent on computations significantly. For example, our optimized implementations for these classes outperformed the original general method by a large margin. For the maj_{11} function, times decreased from several seconds to milliseconds, resulting in about a 10000x speedup. In addition, they can handle a much larger number of bits, going from 11 bits in 10 seconds to over 300 bits in 10 seconds for Symmetric and Threshold functions. The specialized methods performed far better than the general method for these specific function classes and developing specialized methods and/or representations definitely has potential for further exploration. However, these classes are still relatively small compared to the total number of boolean functions.

Looking at general optimizations, canonization did not significantly reduce runtime, whereas minimization, however, had a substantial impact. By using minimized representations, we reduced the number of redundant calculations, as functions like $g(x_1, x_2, x_3) = x_1 \wedge x_2$ and $h(x_1, x_2, x_3) = x_2 \wedge x_3$ were treated as equivalent, allowing memoization to be more effective. This resulted in efficiency improvements around 100 times faster.

Hash-consing led to significant improvements in performance. For both hash-consing and minimization, the advantages become increasingly evident as the number of bits in the function grows, due to the corresponding increase in comparisons. While both methods did not achieve speedups comparable to those seen with specialized representations, they offer the added benefit of being applicable to a much broader range of Boolean functions.

The implementation of multi-composed and iterated boolean functions, while not giving a lot of gain time-wise, has contributed to making the library more user-friendly and broadened its functionality. Additionally, various tools have been developed to analyze and validate our results. These include examining the critical points of our level-p complexities, generating diverse examples, and implementing properties to ensure correctness.

7.1 Future work

While our improvements have led to substantial gains in efficiency, there are still areas that could be further explored. For instance, the handling of polynomial calculations remains an area where more work could be done, particularly for higher-degree polynomials. Additionally, exploring and developing more specialized methods for other complex subclasses of Boolean functions that were not covered in this thesis.

Future work could also involve developing adaptive algorithms that dynamically select the most efficient representation or method for a given function based on its characteristics. This would further enhance the versatility and performance of our approach.

Another question for assessing the impact of this improvement is: 'How many functions belong to the given class?'. The answer to this is not always so straightforward. How one calculates this and which representations one chooses to include could highlight the more relevant classes of Boolean functions.

Bibliography

[1] J. Jansson and P. Jansson, "Level-p-complexity of Boolean functions using thinning, memoization, and polynomials," *Journal of Functional Programming*, vol. 33, e13, 2023. DOI: 10.1017/S0956796823000102.

[2] R. O'Donnell, *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[3] J. A. Bergstra, A. Ponse, and D. J. Staudt, "Short-circuit logic," *arXiv preprint arXiv:1010.3674*, 2010.

[4] C. Garban and J. E. Steif, *Noise Sensitivity of Boolean Functions and Percolation*. Cambridge University Press, 2014, vol. 5.

[5] J. Jansson, "Level-p-complexity for Boolean functions," Available at https://odr.chalmers.se/server/api/core/bitstreams/6ee6ecde-3020-4563-8fd0-257da0672cab/content, MSc thesis, Chalmers University of Technology, 2022.

[6] R. Bird and J. Gibbons, *Algorithm Design with Haskell*. Cambridge University Press, 2020.

[7] J. Jansson, P. Jansson, and C. Sattler, *Bofuncomplexity*, https://github.com/juliajansson/BoFunComplexity, Accessed: 2024-11-04.

[8] E. Goto, "Monocopy and associative algorithms in an extended lisp," Technical Report TR 74-03, University of Tokyo, Tech. Rep., 1974.

[9] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000, pp. 268–279.

[10] Encyclopedia of Mathematics, *Boolean functions*, https://encyclopediaofmath.org/wiki/Boolean_function, Accessed: 2024-07-29.

[11] J. Spanier and K. B. Oldham, "The constant function c," in *An Atlas of Functions*, Washington, DC: Hemisphere, 1987, ch. 1, pp. 11–14.

[12] I. Benjamini, O. Schramm, and D. B. Wilson, "Balanced boolean functions that can be evaluated so that every input bit is unlikely to be read," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, H. N. Gabow and R. Fagin, Eds., Baltimore, MD, USA: Association for Computing Machinery, 2005, pp. 244–250, ISBN: 1-58113-960-8. DOI: 10.1145/1060590.1060627. arXiv: math.PR/0410282.

[13] M. C. Golumbic and V. Gurvich, "Read-once functions," in *Boolean Functions*, ser. Encyclopedia of Mathematics and its Applications, Y. Crama and P. L. Hammer, Eds., vol. 142, Cambridge: Cambridge University Press, 2011, pp. 519–560, ISBN: 978-0-521-84751-3. DOI: 10.1017/CBO9780511852008.

[14] R. Bird, *Introduction to functional programming using Haskell*. Pearson Educación, 1998.

[15] Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986. DOI: 10.1109/TC.1986.1676819.

[16] D. Beyer, *Bdd simple*, https://commons.wikimedia.org/wiki/File:BDD_simple.svg, Accessed: 2024-11-10, 2024.

[17] Hackage. "Memoize - hackage." Accessed: 2025-01-07. (2025), [Online]. Available: https://hackage.haskell.org/package/memoize.

[18] Editors of Encyclopaedia Britannica, *Algebraic number*, Accessed: 2024-06-12, Jun. 2006. [Online]. Available: https://www.britannica.com/science/algebraic-number.

[19] E. W. Weisstein, *Piecewise function*, https://mathworld.wolfram.com/PiecewiseFunction.html, Accessed: 2024-11-01. From MathWorld–A Wolfram Web Resource.

[20] A. Yakeley, *Time: A time library*, https://hackage.haskell.org/package/time, Accessed: 2024-11-01.

[21] GHC Team, *8. profiling - glasgow haskell compiler 9.10.1 user's guide*, https://downloads.haskell.org/ghc/latest/docs/users_guide/profiling.html, Accessed: 2024-11-01.

[22] B. Gregg, *Flame graphs*, https://www.brendangregg.com/flamegraphs.html, Accessed: 2024-10-30.

[23] Wikipedia contributors, *Closure (mathematics)*, https://en.wikipedia.org/wiki/Closure_(mathematics), Accessed: 2024-12-16, 2024.

[24] R. Descartes, *La géométrie*. BoD-Books on Demand, 2022.

A

Appendix

This appendix contains complexity expressions, code listings, and plots that were too large to fit in the main part of the report.

A.1 Complexity of maj_3^3

The list of polynomials (a,b,c, etc) that make up the complexity for maj_3^3

a(x) $= 6 + 6x + 9x^2 + 12x^3 + 76x^4 - 22x^5 - 877x^6 + 946x^7 - 40x^8 - 612x^9 + 9158x^{10} - 20297x^{11} - 166x^{12} + 47146x^{13} - 54720x^{14} + 3534x^{15} + 43536x^{16} - 43921x^{17} + 20780x^{18} - 5060x^{19} + 512x^{20}$

b(x) $= 6 + 6x + 9x^2 + 12x^3 + 76x^4 - 16x^5 - 950x^6 + 1079x^7 + 108x^8 - 1178x^9 + 9366x^{10} - 19534x^{11} - 1105x^{12} + 47140x^{13} - 53792x^{14} + 2476x^{15} + 44197x^{16} - 44179x^{17} + 20839x^{18} - 5066x^{19} + 512x^{20}$

c(x) $= 6 + 6x + 9x^2 + 12x^3 + 76x^4 - 4x^5 - 1042x^6 + 1122x^7 + 554x^8 - 1815x^9 + 8855x^{10} - 17525x^{11} - 2854x^{12} + 46642x^{13} - 51024x^{14} - 786x^{15} + 46350x^{16} - 45032x^{17} + 21028x^{18} - 5084x^{19} + 512x^{20}$

d(x) $= 6 + 6x + 9x^2 + 12x^3 + 76x^4 + 2x^5 - 1097x^6 + 1244x^7 + 607x^8 - 2366x^9 + 9519x^{10} - 17282x^{11} - 4204x^{12} + 47890x^{13} - 51023x^{14} - 1796x^{15} + 47367x^{16} - 45539x^{17} + 21162x^{18} - 5099x^{19} + 512x^{20}$

e(x) $= 6 + 6x + 9x^2 + 12x^3 + 76x^4 + 14x^5 - 1179x^6 + 1358x^7 + 851x^8 - 3220x^9 + 10108x^{10} - 16188x^{11} - 6731x^{12} + 49501x^{13} - 50024x^{14} - 4416x^{15} + 49556x^{16} - 46520x^{17} + 21398x^{18} - 5123x^{19} + 512x^{20}$

f(x) $= 6 + 6x + 11x^2 + 7x^3 + 74x^4 - 11x^5 - 1019x^6 + 1130x^7 + 774x^8 - 2608x^9 + 9244x^{10} - 15646x^{11} - 6566x^{12} + 48810x^{13} - 49382x^{14} - 4672x^{15} + 49544x^{16} - 46464x^{17} + 21376x^{18} - 5120x^{19} + 512x^{20}$

g(x) $= 6 + 6x + 11x^2 + 7x^3 + 79x^4 - 9x^5 - 1097x^6 + 1250x^7 + 879x^8 - 3013x^9 + 9558x^{10} - 15516x^{11} - 7134x^{12} + 49687x^{13} - 50388x^{14} - 3828x^{15} + 49063x^{16} - 46291x^{17} + 21341x^{18} - 5117x^{19} + 512x^{20}$

h(x) $= 6 + 6x + 11x^2 + 17x^3 + 38x^4 + 32x^5 - 1044x^6 + 1056x^7 + 1277x^8 - 3686x^9 + 9992x^{10} - 14560x^{11} - 9703x^{12} + 52124x^{13} - 50775x^{14} - 5288x^{15} + 50699x^{16} -$

$47128x^{17} + 21561x^{18} - 5141x^{19} + 512x^{20}$

$\mathrm{i}(x) = 6 + 6x + 11x^2 + 17x^3 + 48x^4 - 14x^5 - 970x^6 + 1026x^7 + 1167x^8 - 3240x^9 + 8954x^{10} - 13149x^{11} - 10498x^{12} + 51328x^{13} - 48526x^{14} - 7774x^{15} + 52334x^{16} - 47788x^{17} + 21712x^{18} - 5156x^{19} + 512x^{20}$

$\mathrm{j}(x) = 6 + 6x + 21x^2 - 9x^3 + 40x^4 + 53x^5 - 1073x^6 + 1313x^7 + 845x^8 - 3850x^9 + 10708x^{10} - 14214x^{11} - 11661x^{12} + 53588x^{13} - 49720x^{14} - 8140x^{15} + 53207x^{16} - 48317x^{17} + 21865x^{18} - 5174x^{19} + 512x^{20}$

$\mathrm{k}(x) = 6 + 6x + 21x^2 + x^3 + 9x^4 + 68x^5 - 1043x^6 + 1230x^7 + 1189x^8 - 4503x^9 + 10885x^{10} - 13201x^{11} - 13166x^{12} + 54324x^{13} - 49430x^{14} - 8822x^{15} + 53719x^{16} - 48539x^{17} + 21920x^{18} - 5180x^{19} + 512x^{20}$

A.2 Plots

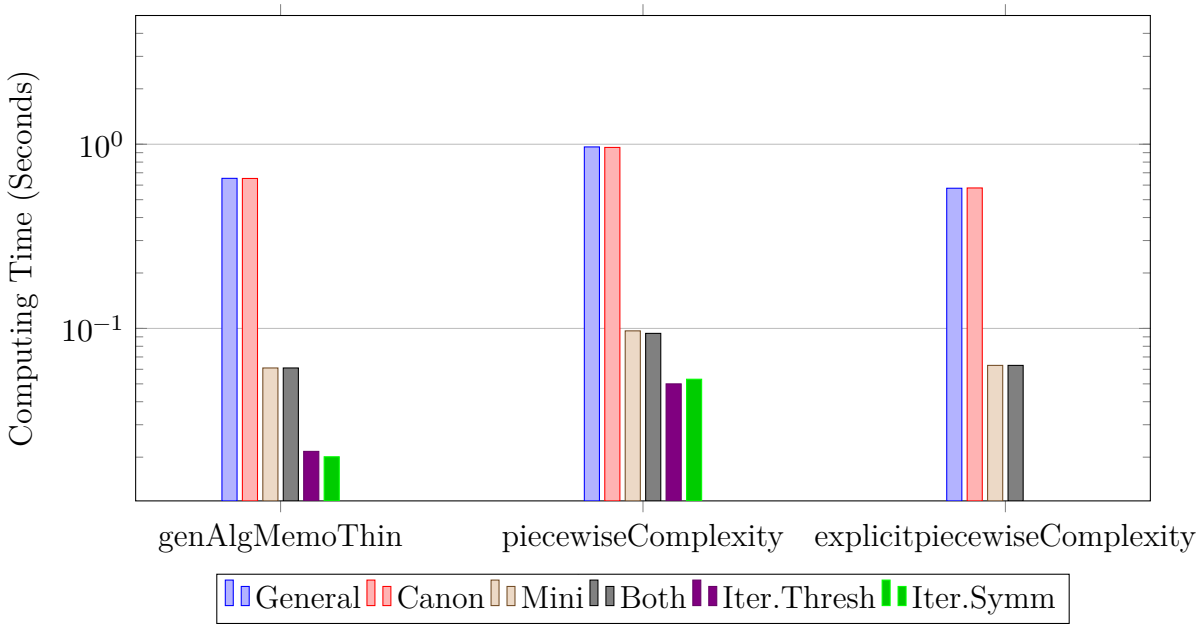

Figure A.1: Log-scale bar chart for maj_3^2 showing the total computing time (s)

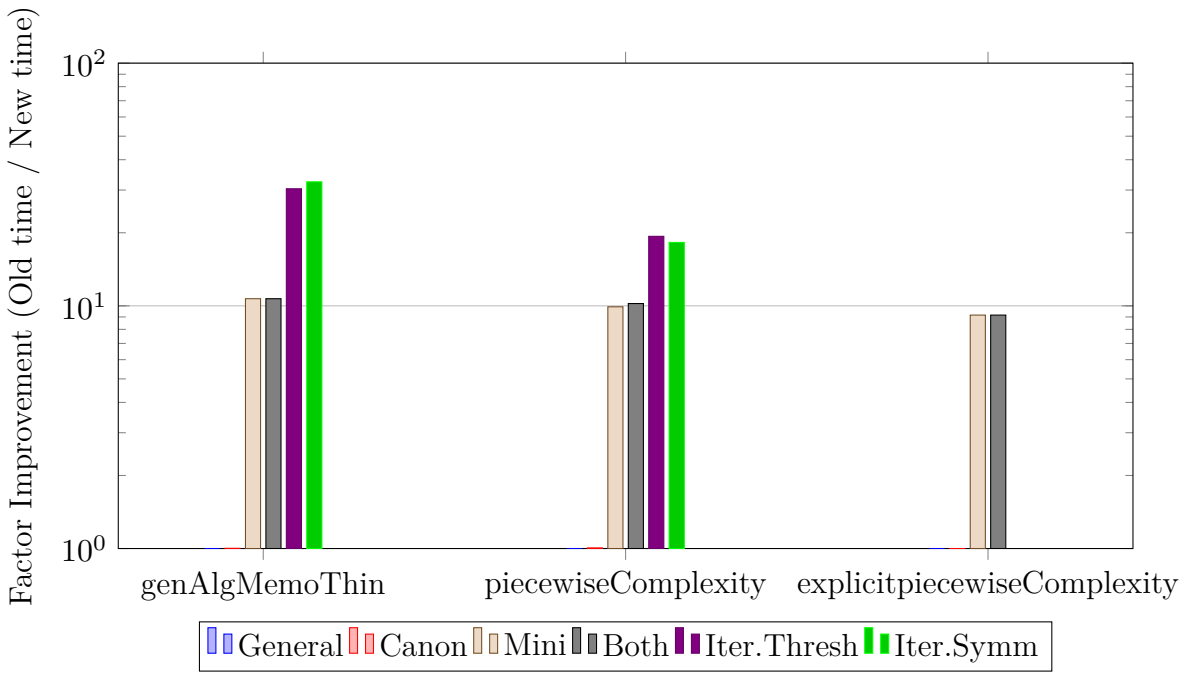

Figure A.2: Log-scale bar chart for maj_3^2 showing the factor improvement compared to General (old time / new time)

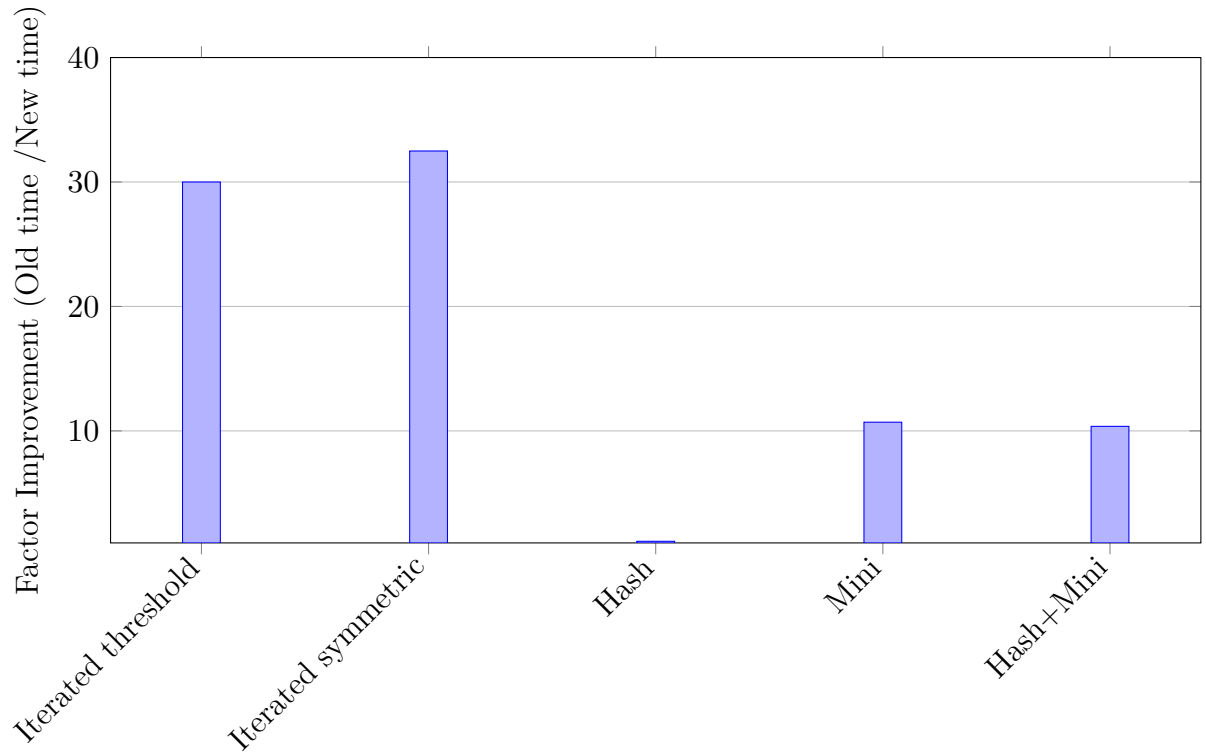

Figure A.3: Log-scale bar chart for maj_2^3 showing the factor improvement compared to the original (old time /new time)

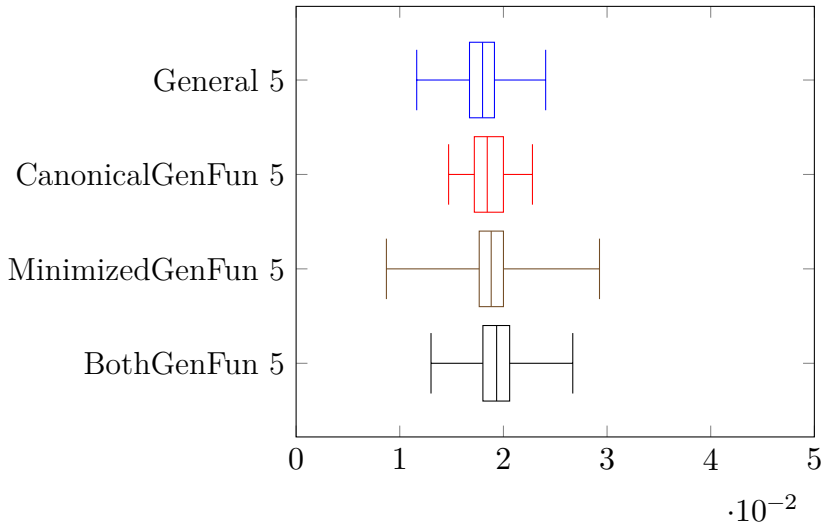

Figure A.4: Computing time for the various implementations of the general function class, using `piecewiseComplexity` with 100 random samples of each function class.

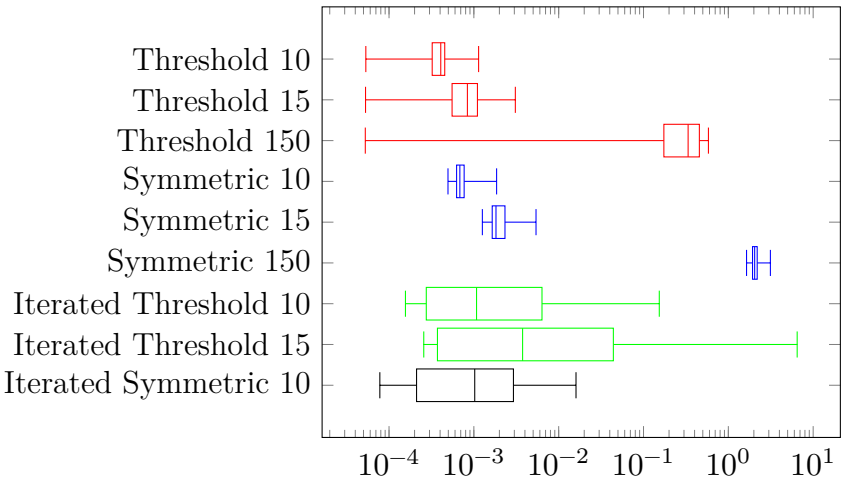

Figure A.5: Computing time for selected function classes,
using `piecewiseComplexity` with 100 random samples of each function class.

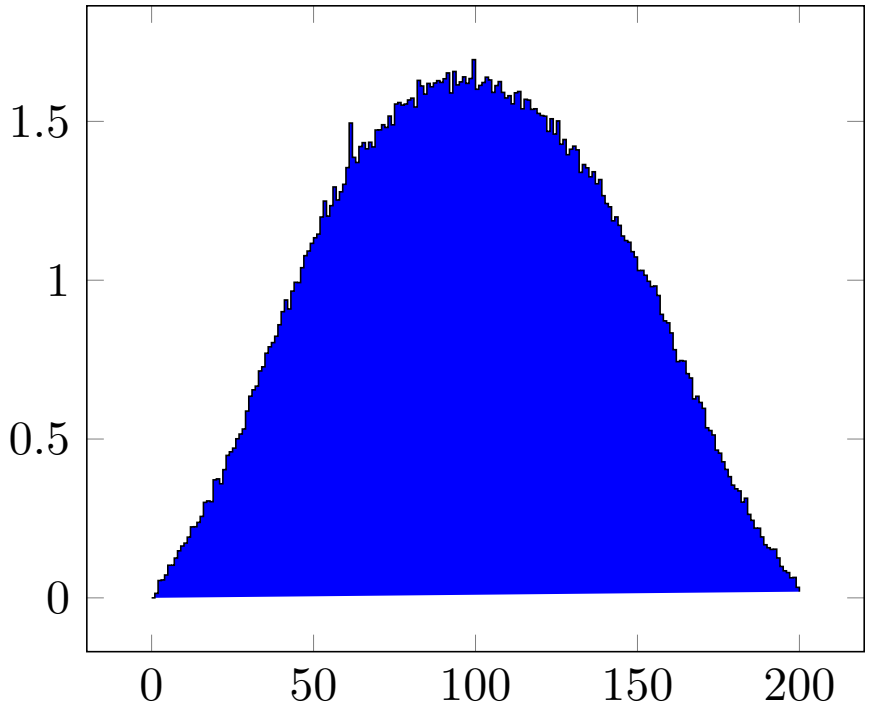

Figure A.6: Diagram showing the computing time of the complexity of all 200-bit threshold functions, using `piecewiseComplexity`.

A.3 Code listings

```haskell
criticalPointsInPiece :: Algebraic -> Poly Rational ->
Algebraic -> [UncertainCriticalPoint]
criticalPointsInPiece a1 _ a2
  | a1 >= a2 = error "a1 must be < a2"
criticalPointsInPiece a1 p a2 = case constP p of
  Just v  -> [URange a1 a2 v]
  Nothing -> criticalPointsInPiece' r1 p r2
  where
    (r1, r2) = findRationalEndpoints a1 p a2

criticalPointsInPiece' :: Rational -> Poly Rational
-> Rational -> [UncertainCriticalPoint]
criticalPointsInPiece' startLow p = go startLow
  where
    p' = derP p
    noDoubleRootsP' = removeDoubleRoots p'
    go low high = case numRootsInInterval noDoubleRootsP' (low, high) of
      0 -> []
      1 -> case criticalType low p' high of
        Nothing -> []
        Just t  -> [UPoint (Algebraic $ AlgRep p' (low, high)) t]
      _ -> go low mid ++ go mid high
      where
        mid = (low + high) / 2
```

Listing 9: The `criticalPointsInPiece` function

```haskell
criticalPointBetweenPieces :: Poly Rational -> Algebraic ->
Poly Rational -> Maybe UncertainCriticalPoint
criticalPointBetweenPieces p1 s p2 = do
  c <- criticalType' s1 s2
  return (UPoint s c)
  where
    p1' = derP p1
    p2' = derP p2
    (s1, s2) = (signAtAlgebraic s p1', signAtAlgebraic s p2')
```

Listing 10: The `criticalPointBetweenPieces` function

```haskell
addEndPoints :: [Poly Rational] -> [UncertainCriticalPoint] ->
[UncertainCriticalPoint]
addEndPoints ps [] = case compare zeroVal oneVal of
  LT -> [UPoint (Rational 0) Minimum, UPoint (Rational 1) Maximum]
  GT -> [UPoint (Rational 0) Maximum, UPoint (Rational 1) Minimum]
  EQ -> [URange (Rational 0) (Rational 1) zeroVal]
  where
    zeroVal = evalP (head ps) 0
    oneVal = evalP (last ps) 1
addEndPoints _ xs = addFirst $ addLast xs

addFirst :: [UncertainCriticalPoint] -> [UncertainCriticalPoint]
addFirst [] = error "Should not happen"
addFirst rest@(x : _) = case x of
  URange {}        -> rest
  UPoint _ Maximum -> UPoint endPoint Minimum : rest
  UPoint _ Minimum -> UPoint endPoint Minimum : rest
  where
    endPoint = Rational 0

addLast :: [UncertainCriticalPoint] -> [UncertainCriticalPoint]
addLast [] = error "Should not happen"
addLast xs = case last xs of
  URange {}        -> xs
  UPoint _ Maximum -> xs ++ [UPoint endPoint Minimum]
  UPoint _ Minimum -> xs ++ [UPoint endPoint Maximum]
  where
    endPoint = Rational 1
```

Listing 11: The `addEndPoints` function

```haskell
critcalPointsPW :: PiecewisePoly Rational -> [CriticalPoint]
critcalPointsPW pw = determineUncertain points
  where
    points = addEndPoints (pieces pw) midPoints
    midPoints = findMidPoints pwList
    pwList = map (fmap fromPWSeparation) $ linearizePW pw
```

Listing 12: The `criticalPointsPW` function