Towards Spectral Synthesis: Field Expansions for Partial Functions and Logic Modules for FPGAs

by

Zeljko Zilic

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Zeljko Zilic, 1997
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-28104-3
Towards Spectral Synthesis: Field Expansions for Partial Functions and Logic Modules for FPGAs

Doctor of Philosophy, 1996

Zeljko Zilic

Department of Electrical and Computer Engineering
University of Toronto

Abstract

Spectral methods have long been contemplated for use in the analysis and synthesis of logic functions, but their applicability in CAD tools became possible only with the introduction of modern graph-based representations. There remain several problems in using the spectral methods for practical applications. In this dissertation, we give new results on one problem, that of handling the partial, or incompletely specified functions. We concentrate on the field expansions, which generalize Reed-Muller transform to multiple-valued logic domain, and on the related forms. The case of the incompletely specified functions is studied here using the methods of multivariate polynomial interpolation. Several new methods are developed, based on the sparsity of the function, complexity of the algorithm and the representation of the forms used. We present a polynomial time deterministic interpolation algorithm, which includes the decomposition of the problem into possibly many smaller problems. Some of the immediate applications of the interpolation algorithm are presented as well. The second part of the thesis examines the design of Universal Logic Modules, intended for use in Field Programmable Gate Arrays (FPGAs). We give an outline of a method for achieving optimal architectures with respect to the encoding length of the functions that are to be realized with such logic blocks. The proposed class of blocks is of special interest in reconfigurable logic applications of FPGAs.
Acknowledgments

First and foremost I wish to thank to my supervisor, Professor Zvonko Vranesic, for his guidance, understanding and unequivocal support. While he left me the freedom to investigate the topics that I appreciate the most, his constant advice kept me solidly on the ground.

I wish to thank to the members of my dissertation committee. Professors Jon Muzio, David Lewis, Jonathan Rose, Frank Kschischang and Rudi Mathon provided many useful comments.

Conversations with Charles Rackoff, Allan Borodin, Shuhong Gao, Victor Shoup, Rudi Mathon and Daniel Panario helped me greatly to understand computational issues of dealing with finite fields. Special thanks go to Professor Charles Rackoff who helped me enormously.

For their help with posers, I am grateful to Derek Corneil, Gara Pruesse, Hans-Jürgen Hoehnke, Miroslav Haviar, and Bernhard Ganter.

I am also grateful to many members of the Department. Professors Jonathan Rose and Steve Brown provided lots of help, understanding and advice throughout the last six years. Guy Lempieux, Alireza Kaviani and Mircea Gusat gave many useful comments while preparing this dissertation. Kelvin Loveless and the whole NUMAchine group provided good company and understanding while working on the NUMAchine project, which itself generated a lot of inspiration.

On personal level, my fiancee, Kasia Radecka, provided a lot of love and support.
## Contents

1 Introduction .......................... 1
   1.1 Motivation and Overview ........... 1
   1.2 Representation of Functions ........ 3
   1.3 Realization of Functions .......... 5
   1.4 Thesis Outline .................... 6

2 Background ........................... 8
   2.1 Reed–Muller Transform ............ 8
      2.1.1 Related Forms ................. 9
   2.2 MVL Generalization of RM Transform .... 10
      2.2.1 Field Expansions - Polynomials over Finite Field ... 10
   2.3 Fast RM Transform ................. 11
      2.3.1 Hypercube-based Realization .... 12
      2.3.2 Recursive Implementation of RM Transform .... 14
   2.4 BDDs and Graph-Based Representations ... 15
   2.5 Using Reduced Representations to Obtain RM Transform .... 17
      2.5.1 Using BDDs .................. 17
      2.5.2 Transform to FDDs ............ 19
   2.6 Incompletely Specified Functions in Logic Synthesis .... 21
      2.6.1 Sources of "don't cares" ....... 22
      2.6.2 Using "don't cares" ........... 23

3 RM Transforms For Incompletely Specified Functions - Dense and BDD-based Interpolations ........ 25
   3.1 Introduction ....................... 25
3.2 RM Transform for Incompletely Specified Functions ................................................. 26
  3.2.1 Problem Definition and Known Results .............................................................. 26
  3.2.2 Exhaustive Search for Optimal Assignment of Don't Cares ................................ 27
  3.2.3 Heuristics and Approximations to Optimal Assignment ..................................... 28
  3.2.4 Sparse Case - Searching for Optimal Terms .................................................... 30
3.3 Interpolation Approach ............................................................................................... 31
  3.3.1 Univariate Interpolation ....................................................................................... 31
  3.3.2 Multivariate Interpolation .................................................................................... 37
3.4 Dense Interpolation Algorithm .................................................................................. 40
  3.4.1 Detailed Description of Dense Interpolation ....................................................... 41
  3.4.2 Empirical Comparison ......................................................................................... 43
3.5 Sparse Interpolation over GF2 ................................................................................... 45
  3.5.1 Recursive Sparse Interpolation Algorithm ........................................................... 45
  3.5.2 Straightforward Implementation of the Interpolation Algorithm ......................... 46
  3.5.3 BDD Implementation of the Sparse Interpolation Algorithm .............................. 47
  3.5.4 Empirical Comparison of the Algorithms ............................................................. 51
3.6 Dense BDD-based Algorithm and Related Forms ..................................................... 52
  3.6.1 Extensions to Pseudo Reed-Muller and Pseudo Kronecker Forms ......................... 53
  3.6.2 Empirical Evaluation of all Methods Considered ................................................ 54
3.7 Sparse Interpolation and Higher Radices .................................................................... 55
3.8 Concluding Remarks .................................................................................................. 57

4 New Finite Field Polynomial Interpolation Algorithms ................................................... 58
  4.1 Reformulation of the Problem - Additional Definitions ............................................ 59
  4.2 Interpolation over GF2 - Simple Case ....................................................................... 60
  4.3 Interpolation over Larger Fields - Problem Decomposition ...................................... 65
    4.3.1 Partial Ordering of Interpolation Spaces ........................................................... 65
    4.3.2 Structure of the System Matrix ........................................................................ 66
    4.3.3 Structure of the Solution Space .......................................................................... 68
  4.4 Binary Case Revisited ............................................................................................... 74
    4.4.1 Using Other Reduced Representations .............................................................. 75
  4.5 Conditions for Singularity and Selection of Terms .................................................. 76
1.5.1 Selection of Terms ............................................. 78
1.5.2 Comparison of Term Selection Schemes ...................... 81
1.5.3 Retry Strategies and Randomization ....................... 82
1.6 Deterministic Interpolation Algorithm ......................... 84
1.6.1 Nullspaces and Increasing the Rank ....................... 85
1.6.2 Searching for Replacement Terms Efficiently ................. 88
1.6.3 Algorithm Implementation and Performance ................... 90
1.7 Extensions .......................................................... 92
1.7.1 Interpolation and Fixed Polarity RM Forms .................. 92
1.7.2 Incremental Interpolation and Learning ....................... 93
1.7.3 Incremental Interpolation and Optimal Forms ................. 95
1.7.4 Parallel Interpolation ........................................... 96
1.7.5 Bandwidth Requirements ....................................... 97
1.8 Concluding Remarks ................................................ 98

5 Universal Logic Modules and Programmable Logic ............. 100
5.1 Introduction ......................................................... 100
5.2 Realization of ULMs .............................................. 103
  5.2.1 Equivalence Classes of Switching Functions ................. 103
  5.2.2 Realization of ULMs using BDDs ................................ 104
5.3 Realization of ULM.3 ............................................ 106
  5.3.1 Super BDD as a ULM ............................................. 106
  5.3.2 Encoding of Programming Bits ................................ 108
  5.3.3 Implementation Issues ......................................... 111
5.4 Larger Blocks - ULM.4 ........................................... 113
  5.4.1 Unifying Representative BDDS ................................ 114
  5.4.2 Generating Super BDD (SBDD.4) ............................ 116
  5.4.3 Input Encoding for Programming Bits ...................... 117
  5.4.4 Larger Blocks .................................................... 119
5.5 Other Issues ........................................................ 120
  5.5.1 Technology Mapping Using ULMs .............................. 120
  5.5.2 Functionally Incomplete Blocks ............................... 121
5.3.3 Using Other Graph-based Representations - FDD Case .................. 121
5.6 Concluding Remarks ......................................................... 123

6 Conclusions ........................................................................ 124
6.1 Field Expansions and Incompletely Specified Functions .............. 124
  6.1.1 Future Work ................................................................. 126
6.2 Universal Logic Modules ..................................................... 127
  6.2.1 Future Work ................................................................. 127

A Summary of Useful Facts ..................................................... 128

B Proofs on Term Selection ....................................................... 133

C Enumeration of 4-Variable Functions by BDDs .......................... 137
List of Figures

2.1 GF4 operations ................................. 11
2.2 Hypercube execution of fast RM transform - functions of three variables 14
2.3 Application of the basic RM transform to a BDD representation 18
2.4 Application of the basic RM transform to a BDD representation 19
2.5 Skipping levels in BDD in construction of FDD 20
2.6 Obtaining an FDD from a BDD 21

3.1 Circuit for small field interpolation over GF4. 36
3.2 Circuit for interpolation using Menger theorem. 36
3.3 Comparison - Functions of two variables 43
3.4 Comparison - Functions of three variables 44
3.5 Transform from a BDD_DC to an FDD via an interpolated BDD 48
3.6 Performance of sparse interpolation. 51
3.7 Reduction to BDDs: a) BDD_DC, b) assignment of values to DC's, c) RM transform 52
3.8 Comparison of all methods considered. 54

4.1 All z-subspaces for functions of three variables 67
4.2 Solving z-subspaces in order - geometric presentation for three variables 69
4.3 Example of binary RM transform using posets 75
4.4 More points than terms in projection - example 79
4.5 Example of greedy term selection 81
4.6 Failure rates for three selections of terms - GF3 81
4.7 Initial failure rates for three selections of terms - GF11 83
4.8 Distributions of number of retries in the algorithm - GF3 83
List of Tables

2.1 An example of RM transform and related forms .................................. 10

3.1 Differences for $F = [1.0. d. 3]$ .................................................. 33
3.2 Formal Differences for $F = [1.0. d. 3]$ ........................................ 35
3.3 Algorithm Comparisons - Nonreduced Representation of RM Transform .. 47

5.1 Number of bits needed to encode all functions of $n$ variables. ............... 104
5.2 Equivalence classes for functions of three variables. .......................... 106
5.3 Programming bits for three-variable representative functions. ............... 109
5.4 Programming bits for permuted representative functions. ..................... 109
5.5 Encoding for the first four switches. ............................................. 111
5.6 Optimized programming bits. ...................................................... 111
5.7 Comparison between LUT.3 and ULM.3. ........................................ 114
5.8 Encoding groups. ........................................................................ 117
5.9 Encoding for first four switches. .................................................... 118
Chapter 1

Introduction

1.1 Motivation and Overview

Manipulation of logic functions is of great importance in many areas, of which the design, or synthesis, of digital circuits is the most important. Designers of circuits today rely almost exclusively on Computer Aided Design (CAD) tools. Most of the existing CAD tools are based on classical synthesis methods in the function domain. The work presented in this dissertation is aimed at promoting the use of spectral methods, which provide more insight about the logic functions by using the spectral domain information, not unlike using the Fourier transform elsewhere. It has been shown that spectral methods can be used in logic circuit analysis, for classification of functions [55], detection of symmetries [45], decomposition [50], and certain synthesis tasks [50], [96], [95], [94]. Additionally, spectral methods are useful for testing and fault detection purposes [51], [29], [30].

The main difficulty in the application of spectral methods is the inherent complexity of operations involved. The number of operations required to produce a spectral representation is always exponential in the number of variables, if standard matrix operations are used. This has hindered the application of spectral methods, despite many interesting theoretical results showing their usefulness. Renewed interest in spectral methods has been sparked with the application of graph-based representation, known as Binary Decision Diagrams (BDDs) [25], which reduces the size of the spectral representations and the algorithm execution times. Additional impetus for spectral methods came from the renewed interest in XOR-based synthesis, for which the old methods are not effective.

Reed-Muller (RM) transform, and numerous related forms, are linked closely to the synthesis using XOR circuits. The RM transform [65], [71] has been the subject of numerous
investigations. The transform is either used as a representation of logic functions useful for analysis purposes, or as a starting point for realization, or synthesis through one of its many related forms. Unlike many other transforms used for Boolean function representations, the RM transform can be extended easily to multiple-valued logic (MVL) domain. One such extension, field expansion, is of our primary interest.

This dissertation contains two main contributions to the utilization of RM transform. BDDs and related forms. First, we develop algorithms for efficient RM representations of incompletely specified functions. Second, using BDDs and the RM transform, we design programmable logic building blocks that allow optimal length encoding of functions that they realize.

Spectral transformations are defined only for total, or completely specified functions. Handling of partial, or incompletely specified functions, which are ubiquitous in logic synthesis, has not been resolved. We investigate the problem of producing concise RM representations for incompletely specified functions. This is a hard problem, which becomes even worse for extension forms of the RM transform, which are especially useful in the synthesis of digital circuits using XOR primitives. Recently, some algorithms have been proposed for simplifying the RM transform using don't cares. All of these algorithms use some version of combinatorial search, or an approximation to it. Since the problem is XP-hard [75], it is believed that no such approach can yield an efficient algorithm that can produce optimal results.

We adopt a completely different approach which, unlike others, can lead to provably effective algorithms, both in terms of the computational resources needed and the bounds on the obtained forms. We concentrate on algebraic properties of the RM transform to solve the problem. More precisely, we exploit the fact that the RM transform can be defined as a polynomial interpolation over finite fields to derive new algorithms. We present several algorithms for computing RM forms using multivariate polynomial interpolation, including an efficient finite field interpolation algorithm.

The second part of the dissertation concentrates on the classification of logic functions and the derivation of Universal Logic Modules (ULMs). We introduce a new type of ULMs suited for use in modern programmable logic devices and derive a design procedure for such modules using BDDs and RM transform.

Since the first part of the thesis deals with the representation, and the second part with
the realization of logic functions, we position next our work relative to these two areas.

1.2 Representation of Functions

Representation of Boolean and MVL functions is of great importance. As functions become large, it is the representation method alone, i.e., the data structure, that sets the limit on the size of the function that can be handled. Algorithms are important as well, but they are often restricted only by the size of the data structures encountered.

Representations of logic functions are usually defined using tabular representations, logic formulas, and the BDD-related representations. Since the tabular representations are based on explicit enumeration of function outputs for each valuation of inputs, their size is always exponential in the number of inputs. The simplest such representations are truth tables, which are only useful for small functions, and are usually replaced by the cube representations. Cubes are compact descriptions of the sets of input assignments. For example, in cube \( x_1x_3x_4x_7 \), the variables \( x_2 \), \( x_5 \) and \( x_6 \) can take any value. To define an arbitrary Boolean function, it is sufficient to enumerate all cubes for which the function is 1 (or 0).

Logic formulas are the expressions that use Boolean and MVL operators. A given function is evaluated for a specific input valuation by computing the expression. The most common representations are the sum-of-products and the dual product-of-sums expressions, both of them known as two-level expressions. Multiple-level forms are the most general extension of these concepts, as they involve a hierarchical nesting of simpler forms and Boolean expressions. Less familiar forms are those arising from the spectral methods of representation. The basic idea is that the function is represented using orthogonal operators, much like the Fourier expansions used elsewhere. Many such expansions are known: Reed-Muller, Walsh, Haar, Hadamard, several wavelet and Fourier expressions are among the most popular. These forms possess many useful properties and provide the means to analyze the functions.

This dissertation concentrates on two representations of logic functions. One is the RM expansion and its related forms, while the other is the graph-based BDD representation. A combination of the RM form and the BDD approach has become popular, as the size of the RM form becomes much more manageable if the BDD concept is applied to it.
Handling incompletely specified functions which possibly have many don't cares is extremely difficult for both RM transforms and BDD-based forms. Little is known about exploiting the don't cares in the best way in the analysis and synthesis process. In this thesis we concentrate on the simplification of the resulting forms. Having incompletely specified binary and MVL functions, we want to obtain simple, or manageable resulting forms, within a limited computation time.

In addition to the fact that the minimization of RM transform using don't cares is NP-hard even for binary functions, it is shown that no good approximation to the optimal solution can be found in polynomial time when the graph representations of the RM form is considered [13]. Therefore, we concentrate on the approaches that can provide efficient algorithms for producing representations of bounded size. As long as the representation size is kept limited by the bound depending on the size of the input set, it can be used in subsequent steps in synthesis and analysis.

We use an approach that exploits the algebraic properties inherent in these forms. This approach has several advantages. First, we obtain powerful algebraic tools that eventually lead to provably effective algorithms. We will show that with our approach the resulting forms bounded by the size of the input set can be obtained in polynomial time. Second, we can produce not only the RM expansion, but also the related forms. Finally, in a broader context, by adopting the interpolation approach in dealing with incompletely specified functions, we are solving the problem that is interesting on its own.

The algorithms are presented in increasing order of sophistication and performance. We start with a simple modification to the fast RM transform that can be used for dense functions, which have few don't cares. Since such a transform always has an exponential running time in the number of variables, we refine the algorithm to use a sparse data structure, such as BDDs. Although the obtained algorithm still has exponential running time in the worst case, we obtain linear bounds for the complexity of the resulting forms. Additionally, several related forms can also be synthesized using the extensions of the algorithm. We further improve the running time by the polynomial interpolation algorithms based on the partial ordering between the points at which the function is specified.
Multiple-Valued Logic (MVL)

We treat MVL as a representation, rather than an implementation issue. We note that MVL circuits and signalling are used commercially, from memories (Intel EEPROMs), to communication circuits (HP 100MB/s Ethernet) and even in some FPGAs. MVL representations appear naturally even when dealing with binary circuits. Many problems can best be stated and solved when defined using multiple-valued representations. Among the two-level expression optimization heuristics, the multiple-valued Espresso-MV \([76]\) is the most commonly used method. Other examples include the \textit{input and output encoding} \([99]\), and the \textit{state assignment} for finite state machines \([5]\). In these problems, a set of multiple-valued symbols has to be embedded into the binary system such that the final implementation is the simplest. The best algorithms for these problems take into account the MVL nature of symbols that are used.

Multiple valued logic is much harder to deal with, and it will become apparent that the problems that we consider are substantially harder for the MVL domain than for the binary case. Many challenging open problems exist in this area, from the underlying mathematical logic and algebraic theories to the practical representation and implementation problems. In this dissertation, the results related to binary logic may be of more practical value, but the MVL-related results are much more intellectually rewarding.

1.3 Realization of Functions

The main task of logic synthesis is the realization of given logic functions by circuits. A large amount of research has been devoted to algorithms that synthesize circuits which minimize some complexity measure, such as area, delay or power consumption, or have some other useful properties, such as easy testability. Most of such optimization problems, when they can be exactly defined, are known to be NP-hard.

Several \textit{design styles} are used in practice, with the traditional \textit{two-level} AND-OR circuits and networks of such circuits, called \textit{multiple-level} circuits being used the most. Recently, some specialized three-level design styles involving XOR gates have been investigated, especially AND-OR-XOR gates, which can be very useful with some programmable logic technologies. They were shown to be superior to the two-level technologies, for both practical circuits \([81],[78]\) and with respect to the theoretical bounds on the complexity of realized
circuits [34]. The results from the first part of this dissertation can be applied to the synthesis using AND and XOR primitives. Generalized to the MVL domain, this corresponds to the circuits that use finite-field operations, which are shown to be powerful primitives [17].

Rapid emergence and acceptance of field-programmable gate arrays (FPGAs) has created many possibilities. FPGAs are general purpose logic devices that contain programmable logic blocks and flexible interconnect. Lookup Tables (LUTs) are the most common logic blocks; they are capable of implementing any function with a fixed number of inputs. Since these devices can be programmed without being removed from a printed circuit board, they can be used as reconfigurable machines. While many useful algorithms (such as sorting and processing images) can in the best case be executed by circuits in the time logarithmic in the number of inputs, the reconfigurability allows us to run some algorithms in a constant number of cycles [62]. The attractiveness of this approach is clearly dependent on the time to reconfigure the device. With existing FPGAs this might take several orders of magnitude longer than the time to compute a function.

This dissertation brings some new results on logic blocks which are especially suited for the reconfigurable computing. Chapter 5 of the thesis uses the classification of logic functions using BDDs and RM transform to derive a new class of ULMs [108],[109], which act as LUT replacement and are more practical than ULMs proposed earlier. Their main advantage is in allowing shorter reconfiguration time: the theoretical minima have been reached for the length of the description of functions implemented by such blocks. While this chapter is not directly related to the preceding chapters, the underlying theme of using the algebraic tools and the relation to spectral methods plays a dominant role.

1.4 Thesis Outline

Chapter 2 contains the necessary definitions and examples of algorithms for RM transform and BDDs. It also introduces the necessary notation - we rely heavily on recursive function definitions which are useful for BDDs. Chapter 3 defines the problem of concise RM forms for incompletely specified functions and provides algorithms which use the polynomial interpolation approach. The difficulties with obtaining interpolation algorithms are explained. Chapter 4 contains the new results on solving the interpolation problem. The cases of binary and MVL RM transforms are considered separately, for both the presenta-
tion purposes and applications in mind. Chapter 5 contains a detailed description of the design of ULMs which can be used in FPGAs, especially those intended for reconfigurable computing applications.
Chapter 2

Background

This chapter provides most of the background required in this dissertation. Definitions of the RM transform and related forms are followed by the description of BDDs, fast transform algorithms and the origin and use of don't cares in logic synthesis. The basic algebraic background information is contained for reference in Appendix A.

2.1 Reed–Muller Transform

Reed–Muller transform was introduced independently by Reed [71] and Muller [65] in two different applications: representation of Boolean functions and error correcting codes. The same forms were contemplated long before, in an extensive work by Zhegalkin [101], [102] which, by appearing too early, did not attract much attention.

The basic RM transform, or expansion, is a polynomial

\[ f(x_1, x_2, \ldots, x_n) = \sum_{i_1=0}^{1} \sum_{i_2=0}^{1} \ldots \sum_{i_n=0}^{1} c_{i_1i_2\ldots i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} \]  

(2.1)

in which the summation sign represents the XOR operation. Since the exponents and coefficients take binary values, the expression is an exclusive-sum of product terms. For example, in 3-variable case the RM transform is given by 8 spectral coefficients \( c \). in the form:

\[ c_{000} = c_{100}x_1 = c_{010}x_2 = c_{001}x_3 = c_{110}x_1x_2 = c_{101}x_1x_3 = c_{011}x_2x_3 = c_{111}x_1x_2x_3. \]

Each Boolean function can be uniquely represented using the RM transform. For each \( n\)-
variable Boolean function given by $2^n$ function values, there is a unique set of $2^n$ coefficients of the polynomial from Equation 2.1.

The RM transform has been used in many applications, in analysis, synthesis and testing of logic functions [95]. It is an early example of spectral methods for representing Boolean functions [50], [43].

2.1.1 Related Forms

While the RM transform is canonical, it may require exponentially many terms for a given function. Various related forms relax the canonicity requirement, and consequently can be much simpler. The canonical forms are more useful in the analysis and have good testability properties, while the other forms are considered in the synthesis using XOR gates.

The simplest such form is the fixed polarity RM form, in which each variable may be inverted consistently throughout the expression. The RM transform is often called the positive polarity RM form because each variable is present in positive polarity. For $n$ variable functions, there is a total of $2^n$ fixed polarity forms. For selected polarity, these forms are still canonical.

If the variables in an expression are inverted inconsistently, then we have mixed polarity forms. There are several such forms, depending on the way in which the polarities are selected. They should not have more terms of the same degree than a corresponding polynomial. A broad extension of the basic form is the Kronecker form, which is a mixed polarity expression with the added possibility of using Shannon expansion $f = x f_0 - x f_1$, in addition to RM expansion. Considerable work has been done on optimizing these forms [77].

Table 2.1 shows an example of a RM transform and three related forms for the same function. While the RM expansion requires all but one nonzero coefficient of the polynomial, by inverting variable $x_1$ consistently, the number of terms required is reduced to 4. The inconsistent forms reduce the number of terms further, and the Kronecker form, which can have more terms of the same degree than the polynomial forms, requires only two terms for the same function.
Table 2.1: An example of RM transform and related forms

<table>
<thead>
<tr>
<th>RM Transform</th>
<th>Fixed Polarity</th>
<th>Mixed Polarity</th>
<th>Kronecker</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \div x_1 \div x_2 \div x_3 \div x_1 x_2 \div x_1 x_3 \div x_2 x_3$</td>
<td>$x_1 \div x_1 x_2 \div x_1 x_3 \div x_2 x_3$</td>
<td>$x_1 x_2 x_3 \div x_1 x_2 x_3$</td>
<td>$x_1 x_2 x_3 \div x_1 x_2 x_3$</td>
</tr>
</tbody>
</table>

2.2 MVL Generalization of RM Transform

There are many possible generalizations of RM forms to MVL domains. Many known MVL spectral representations can be defined as a suitable extension of the binary RM transform. In this dissertation, we concentrate on one such generalization which is the best for the purpose we have in mind.

2.2.1 Field Expansions - Polynomials over Finite Field

The basic RM transform can be generalized to nonbinary domains in the form of polynomials known as ring sum expansions, where the ring operations are modulo addition and multiplication. If the domain possesses the structure of a finite field, then we have field expansions. We concentrate on the field expansions ("RM expansions" henceforth), which are defined as polynomial representations of switching functions over finite fields $GF(q)$ with $q = p^m$, for some prime $p$ and integer $m$:

$$f(x_1, x_2, \ldots, x_n) = \sum_{i_1=0}^{q-1} \sum_{i_2=0}^{q-1} \cdots \sum_{i_n=0}^{q-1} c_{i_1 i_2 \ldots i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$$

It is apparent that the binary RM expansion, as defined in Equation 2.1, is a special case for which $q = 2$. Figure 2.1 shows the example of $GF(4)$ addition and multiplication operations, as they will be used in the text. We illustrate next the simplest 4-valued transform, that of a single-variable function.

Example 1 Let $q = 4$ and let the function $f(x)$ be given by the following value vector: $F = [f(0), f(1), f(2), f(3)] = [1, 0, 3, 2]$. The transform corresponds to finding the coefficients of the polynomial in one variable that implements the function $f$. A theorem by Menger [47] can be used for that purpose for all finite fields $GF(q)$. The theorem gives the polynomial coefficients from the function values at points $f(0), f(1), \ldots, f(q - 1)$.
Figure 2.1: GF4 operations

\[
\begin{array}{c|cccc}
+ & 0 & 1 & 2 & 3 \\
0 & 0 & 1 & 2 & 3 \\
1 & 1 & 0 & 3 & 2 \\
2 & 2 & 3 & 0 & 1 \\
3 & 3 & 2 & 1 & 0 \\
\end{array}
\quad
\begin{array}{c|cccc}
\times & 0 & 1 & 2 & 3 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 2 & 3 \\
2 & 0 & 2 & 3 & 1 \\
3 & 0 & 3 & 1 & 2 \\
\end{array}
\]

\[
c_0 = f(0) \\
c_1 = \sum_{j \geq 0} [f(0) - f(g^j)] g^{-j}
\]  \hspace{1cm} (2.2)

The application of this theorem produces a vector of coefficients \(C = [c_0, c_1, c_2, c_3] = [1, 1, 0, 0]\), which corresponds to the polynomial \(f(x) = 1 + x\).

**Extensions of MVL RM transform – related forms**

A basic RM expansion can have exponentially many coefficients, and more compact representations might exist, as in the binary case. By adding a constant to a variable consistently throughout the polynomial expression, namely by using the polynomial \(f(x_1 - a_1, x_2 - a_2, \ldots, x_n - a_n)\) instead of \(f(x_1, x_2, \ldots, x_n)\), we have fixed polarity RM forms, similar to the binary case. Various inconsistent mixed polarity forms also exist.

### 2.3 Fast RM Transform

In computing the RM transform we can, as with many other spectral methods, exploit the existence of fast transform. For completely specified functions, we can use the fast RM transform [110] which is shown in [58] to be equivalent to the inverse FFT over finite fields. For binary functions, it can be easily shown that the RM transform and its inverse are the same. There are several ways in which the fast transform can be described and executed. The idea behind all these is to exploit the repeated application of expansion in one-variable.

For binary case, the single-variable transform, which produces the column vector of coefficients \(c = [c_0, c_1]\) from the function values \(f = [f(0), f(1)]\) can be given in matrix form
as \( c = Tf \), where:

\[
T = \begin{bmatrix}
1 & 0 \\
1 & 1
\end{bmatrix}
\]

Using the \( GF2 \) arithmetic, multiplication corresponds to the logical AND and addition corresponds to the XOR operation, resulting in expressions \( c_0 = f(0) \) and \( c_1 = f(0) - f(1) \). The symbol "+" will denote XOR operation in the rest of the chapter.

**Kronecker product extension**

While the Kronecker matrix product extension to the multivariate case is conceptually the simplest, it does not lead to fast transforms. The \( n \)-variable extension is defined using the single-variable transform as:

\[
T_n = \begin{bmatrix}
1 & 0 \\
1 & 1
\end{bmatrix} \otimes T_{n-1}
\]

which produces, after applying the Kronecker product, the recursive matrix definition

\[
T_n = \begin{bmatrix}
T_{n-1} & 0 \\
T_{n-1} & T_{n-1}
\end{bmatrix}
\]

This matrix has \( 2^{2^n} \) entries and the transform obtained by multiplying \( T_n \) with the vector of values requires \( O(2^{2^n}) \) operations. The fast RM transform is faster than that; it can be done in \( O(n2^{n-1}) \) time and \( O(2^n) \) space; it is fast compared to the other approaches that lead to at least \( O(2^{2^n}) \) time algorithms.

The most usual way to represent the execution of fast transforms is by means of butterfly diagrams, which are the data flow graphs of the computation steps performed. These diagrams are widely used among the VLSI implementers of various fast transforms. For our purposes, another implementation is more useful.

**2.3.1 Hypercube-based Realization**

In this dissertation, we rely on the hypercube-based description and execution of the fast transforms. In [110] we derived it for the general MVL case and proved that the transform is indeed a fast transform. With this method, the input, the intermediate results and the output of the algorithm are kept in nodes of a \( q \)-valued hypercube. Separate transform
steps are done over the hypercube, starting with the original array of values $F$, and ending with the array of transform, or polynomial coefficients $C$:

$$ [F] = [B]_n \leftarrow [B]_{n-1} \leftarrow \ldots \leftarrow [B]^1 \leftarrow [B]^0 = [C] $$

Each transformation step, denoted as "$\leftarrow$", is the transform with respect to a variable. The transform with respect to $x_i$ is a set of $k^{n-1}$ one-dimensional transforms for which all variables but $x_i$ are assigned (all possible) constant values. Geometrically, it can be interpreted as a transform over a face of an $n$-dimensional cube. Figure 2.2 illustrates such an approach for $n = 3$, by associating with each transform with respect to a variable an arrow pointing to a face of a hypercube. The transforms with respect to a variable can be performed in any order. A 2-variable, 4-valued case is given in more detail in Example 2, which includes the process of creating intermediate matrices $[B]^i$.

**Example 2** Let $k = 4$ and let the function $f(x_1, x_2)$ be given by the following value matrix, in which $x_1$ selects the row and $x_2$ selects the column:

$$ F = \begin{bmatrix} 1 & 0 & 3 & 2 \\ 1 & 2 & 0 & 1 \\ 2 & 0 & 3 & 2 \\ 0 & 1 & 3 & 2 \end{bmatrix} $$

The first intermediate matrix $[B]^2$ is equal to the matrix of function values $F$. To create the next matrix, $[B]^1$, the variable $x_1$ is set to 0, 1, 2 and 3 and the one-dimensional algorithm is applied to the rows $F_{0x} = [1.0.3.2], F_{1x} = [1.2.0.1], F_{2x} = [2.0.3.2], \text{ and } F_{3x} = [0.1.3.2]$. By using, for example, the theorem by Menger, this leads to the following matrix:

$$ [B]^1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \end{bmatrix} $$

The second step of the algorithm consists of the application of the single-variable algorithm to the columns of the matrix: $B^1_{x0} = [1.1.2.0], B^1_{x1} = [1.0.1.0], B^1_{x2} = [0.1.0.1]$ and
Figure 2.2: Hypercube execution of fast RM transform - functions of three variables

$B_{x3}^1 = [0,2,3,0]$, which produces the solution:

$$[B]^0 = [C] = \begin{bmatrix}
1 & 1 & 0 & 0 \\
0 & 3 & 3 & 0 \\
2 & 2 & 2 & 3 \\
2 & 0 & 0 & 1
\end{bmatrix}.$$  

The resulting RM transform coefficients are contained in this matrix. Its entries are the coefficients of the polynomial in two variables over $GF(4)$ that implements the given function.

### 2.3.2 Recursive Implementation of RM Transform

The fast RM transform can be concisely described by a recursive function. We will deal explicitly with the binary case only: extension to other radices follows directly. First, some additional notation is necessary.

A function is represented by a *list* of its values, in both the functional and transform domains. The list and *vector* representations are equivalent and we use these two terms interchangeably. A *construction* of an $n+1$ variable function is obtained as a concatenation of lists that represent $n$-variable functions. We adopt the convention that the first half of the list corresponds to the case when the variable $x$ is zero. For example, vector $[1,1]$ corresponds to the single-variable function $1+x$ in the transform domain, and to the constant 1 in the functional domain. An inverse operation to the construction is the restriction $f |_{x=r}$, or assignment of a value $r$ to the variable $x$. Each restriction decreases the dimensionality
by one, and hence corresponds to selecting one half of the list. Single-variable transform can be obtained by considering the following identity for functions over $GF2$:

$$f = f \mid_{x=0} + (f \mid_{x=0} + f \mid_{x=1}) \quad \text{(2.3)}$$

where "+" is the binary XOR operation.

Since the RM transform is defined as a polynomial representation, a direct transcription of Equation 2.3 results in a recursive algorithm for computing the RM transform. Let the polynomial be represented as a list that can be concatenated by operation $\mid$. Then the RM transform produces the list of transform coefficients as follows:

**Algorithm 1 Recursive RM Transform**

$$RM(n, f) = RM(n - 1, f \mid_0) \parallel RM(n - 1, f \mid_0 + f \mid_1)$$
$$RM(0, z) = z, \quad z \in 0.1$$

This produces recursively the RM transform (interpolation polynomial) for a completely specified binary function $f$ in $n$ variables. The function restrictions are denoted by $f \mid_0$ and $f \mid_1$ when it is clear which variable is assigned a value of 0 or 1, respectively.

Since this algorithm is defined recursively, we can use a full decision tree description of the function in the following way. A restriction of the function when a variable is 0 (1) is equivalent to selecting the subtree obtained by selecting the 0 (1) edge. This corresponds to the following redefinition of Algorithm 1 using the list notation:

**Algorithm 2 RM Transform on Lists**

$$RM(f) = RM(f.0) \parallel RM(f.0 + f.1)$$
$$RM(z) = z, \quad z \in 0.1$$

### 2.4 BDDs and Graph-Based Representations

Many advances in manipulation of Boolean functions have been achieved recently by employing the BDD representation, as introduced in [2] and [19]. The origins of this representation can be traced back to the work of Shannon [85] in 1930's and Lupanov [59] and Lee [56] in
1950's. The same or similar representations have for long time been successfully applied in studies of complexity theory, where the name branching programs was used. These representations describe a Boolean function by a series of binary decisions that depend on the input values; the overall decision takes one of the two possible values, corresponding to the function value.

A BDD can be constructed from the tabular description as follows. First, the complete binary decision tree is formed by assigning the function values to the leaves of a tree. The BDD is then obtained by extracting the isomorphic subtrees in the decision tree and creating edges to only one representative of each isomorphic class [19]. In this way, a graph is obtained that represents the same function as the original decision tree.

From this construction, it follows that BDDs are directed acyclic graphs, with a unique root node and 2 terminal nodes that have no successors. Associated with these two nodes are binary values 0 and 1. All other nodes have a constant outdegree (number of outgoing edges) of 2. The value of the function is calculated by traversing the BDD from the root to one of its terminal nodes. The nodes are grouped according to levels, and an edge can point only to a node that is at a higher level. At each level i node, an edge is taken according to the value of the variable $x_i$, and the function value corresponds to the terminal node reached. A successor node reached by an edge 0 (1) is called 0-successor (1-successor). In diagrams, we show an edge to 0-successor (0-edge) as a dotted line, and 1-edge as a solid line.

**Performing operations on BDDs**

The significance of BDDs is that they are canonical for a fixed order of inputs and that many useful operations can be done in time polynomial in their size. It is shown in [19] that any binary operation (including XOR) over two subgraphs $F$ and $G$ can be computed in $O(|F||G|)$ time. To achieve such running time, the algorithm maintains a table of all intermediate results. If the result of an operation for a pair of visited nodes is already in the table, there is no need to recompute the operation.

In all the manipulations on BDDs to follow, we use the recursive function descriptions, as introduced in Section 2.3.2. ¹

¹Interestingly, a large majority of publications on BDDs do not exploit this elegant description.
2.5 Using Reduced Representations to Obtain RM Transform

We can use BDDs to represent RM expressions in a more compact way. This representation, known as Functional Decision Diagrams (FDDs), has received a lot of attention recently \[13\]. The idea of combining both BDDs and FDDs in Kronecker Decision Diagrams (KDDs) has also been tried \[52\]. Although the obtained representations can still be exponentially long in the worst case, substantial reductions are possible for many classes of functions.

2.5.1 Using BDDs

The RM transform can be obtained using BDDs. For this, we use a recursive version of the fast RM transform. Since the ordinary recursive execution of the RM transform involves traversing the decision tree, when the transform is executed using the BDD representation, a corresponding traversal of the graph occurs. In the context of this traversal, a subtree of a decision tree maps to a lower subgraph of a BDD. We say that the transform is performed at a node \(v\) when it is executed over a function represented by a subgraph whose root is the node \(v\).

To obtain the RM transform, we must perform an XOR operation between two BDDs. Starting from the decision tree description of a function, this operation can be obtained as:

\[
f \oplus g = (f \cdot 0 \cdot g \cdot 0) \bigoplus (f \cdot 1 \oplus g \cdot 1)
\]

Note that in a BDD it is possible to "skip" levels, i.e. an edge at level \(i\) can point to a node that is not at level \(i + 1\). This situation can be handled by keeping a record of the level of a node and explicitly checking if the two nodes over which an XOR operation is performed are at the same level. A detailed description of such an algorithm is given in \[52\].

A simple example of skipping a level while performing an XOR operation over two BDDs is given in Figure 2.3. Implicitly, an edge leading to the root of the first BDD skips one level. The transform is obtained by XOR-ing recursively the \(0(1)\)-successors of nodes in the first BDD with the \(0(1)\)-successors of nodes in the second BDD starting with a root node. The root of the right BDD is at level \(n - 1\) and to compensate for an absence of the level \(n - 1\) node in the left BDD, both 0- and 1-successors of the root node are set to be equal.
to node 1. This leads to XOR-ing node 1 with node 2, which produces terminal node 0. Similarly the 1-successors of the root node are the nodes 1 and 3, and the XOR between these two nodes results in terminal node 1.

Level skipping in an algorithm that computes the RM transform can be handled by "padding" the transform that is produced at a node that skipped levels. Skipping a level is equivalent to the statement $f.0 = f.1$, which means that the second half of the transform vector must contain all 0s. by Algorithm 2. This leads to the following algorithm:

**Algorithm 3** *Postorder RM Transform using BDD*

\[
RM (f) = (a_0 = Pad(RM(f.0))) \| a_0 + Pad(RM(f.1))
\]

\[
RM (z) = z, \quad z \in \{0, 1\}
\]

where the function $Pad$ calculates the difference between the current level of a node in the BDD and the level of its immediate successor and expands the transform vector accordingly.

Note that this scheme differs from Algorithm 2 in the fact that it reuses the result of the execution of the algorithm over the 0-successor, and XORs it with the transform over the 1-successor. These versions of the transform algorithm amount to the difference in the order in which the XOR operations are performed. According to Algorithm 3, the XOR operations are performed after the subtrees (subgraphs) have been visited, while in Algorithm 2, the 1-successor subtree (subgraph) is changed before it is processed. We call these two strategies *postorder* and *preorder*, respectively. For the basic RM transform, we use the postorder scheme, because its execution corresponds to a straightforward traversal of the graph.
An example of an application of such transform is given in Figure 2.4, which shows a completely specified function represented by a BDD. For each node in the BDD, the value of the transform at that node is given by a value vector. The transform at node 1 is obtained as $1 \lor 1 + 0 = [1 \ 1]$, because the 0-successor is 1 and the 1-successor is 0. To obtain the transform at node 2, the transform of the 0-successor has to be padded. Since edge 0 of node 2 points to the terminal 0 node, the transform would result in the vector [0 0]; since one level is skipped, the transform is padded to become [0 0]. Then, the transform at the node 2 is $[0 \ 0] \lor [0 \ 0] + [1 \ 1] = [0011]$. Similarly, the transform at node 3 is obtained after padding vector [1] twice and concatenating the transform obtained at node 2 with the sum of these two: $[0011] \lor ([0011] + [000]) = [0011] \lor [1011] = [0011011]$.

### 2.5.2 Transform to FDDs

A more efficient way to represent an RM transform is possible using FDDs, which are BDD representations of the RM transform vectors. We now adapt the recursive transform to produce an FDD representing the RM transform.

As above, a recursive procedure can be devised to build the resulting data structure bottom up. For each node in the original BDD, a corresponding transform is constructed. Associated with each node in the BDD is a root node in the FDD which represents the transform of the function represented by the subgraph with the root at that node. The edge 0 of the node points to a node in the FDD corresponding to the transform of the 0-successor, while the 1-successor of a node in the FDD is obtained by XOR-ing two successor
transform nodes.

Again, the transforms have to be padded, because skipping the levels in the original BDD amounts to assigning 0s in the RM transform to those coordinates where the skipped variables are 1. Applied to FDDs, this means that for each skipped level, a node will be created whose 1-successor is the terminal node 0. This is illustrated in Figure 2.5, where for s skipped levels there are s nodes created in the FDD. Note that if the function to be transformed is \( f = 0 \), then there is no need to pad the transform. Since padding amounts to augmenting zeros, in the case of 0 transform, this padding does not change the result.

This translates into the following scheme:

**Algorithm 4 Transform to FDDs**

\[
FDD(f) = \text{JoinBDD}(\{a_0 = \text{Pad}(FDD(f,0)), a_0 + \text{Pad}(FDD(f,1))\})
\]

\[
FDD(z) = z, \quad z \in 0.1
\]

where the function \( \text{JoinBDD}(f_0, f_1) \) creates a reduced BDD with the two restrictions given by BDDs \( f_0 \) and \( f_1 \).

We can estimate the number of operations needed for the RM transform as follows. Each XOR operation over two subgraphs \( F, G \) can be obtained in \( O(|F||G|) \) time as shown in [19]. We use the postorder model of evaluation by which the transform is performed over individual nodes in the input BDD. If we denote the size of the input BDD as \( |G_1| \) and the size of the largest transform generated by the algorithm (not only the final result, but also
the intermediate steps as $G_2$, then the computation time of the algorithm is $O(|G_1||G_2|^2)$ because for each node in the BDD there is an XOR operation performed over the obtained transforms.

Figure 2.6 shows, step by step, how such a transform can be obtained for the BDD in Figure 2.1. Node 1 in the BDD is transformed to node 4. Node 2 is transformed into node 5 whose 0-successor is the terminal node 0, while the 1-successor is the XOR of 0 and node 1. Since node 4 has both edges leading to the terminal node 1, it can be omitted; instead an edge can be formed directly to the terminal node 1. Node 3 is transformed into node 8, using node 5 as the 0-successor and the XOR of node 5 and the transform of the 1-successor of node 3. Since node 3 points to the terminal node 1, which is not at the level adjacent to the terminal level, the transformed subgraph is padded with nodes 1 and 6. Finally, the 1-successor of node 8 is obtained by XORing BDDs whose top nodes are nodes 5 and 6. This XOR operation produces the BDD with the root node 7.

2.6 Incompletely Specified Functions in Logic Synthesis

Incompletely specified functions are ubiquitous in logic synthesis. In this section we first briefly discuss the possible sources of don't cares, based on the detailed investigation by De Micheli [33] and several other recent works. Then, we describe their use in analysis and synthesis of logic functions.
2.6.1 Sources of "don't cares"

Generally, don't cares occur when interfacing the systems, that is, in embedding a circuit into a specific environment. For a given subsystem, the external don't care conditions come from outside, and consist of controllability and observability components. These conditions are called don't care sets or DC sets. The input controllability don't care sets consist of input patterns that are never present, while the output observability don't care sets are those that are never observed by the environment. When considering larger systems composed of several subsystems, we define the local don't cares as the combinations of internal variables that are never present. In [33], algorithms are presented that calculate the possible don't care sets for a system from the description of its subsystems and the external don't care sets for the overall system.

Several don't care conditions occur often in sequential systems. There, not only the environment and the interface produce the don't cares, but the network itself generates many such conditions. In a finite state machine with \( S \) states, encoded using \( n \) bits, there will be \( 2^n - S \) don't care conditions, i.e., unreachable states. The state encoding is often sparse. For example, a commonly used encoding technique is one hot encoding, in which \( n \) bits are used to encode \( n \) states; in this case, the remaining \( 2^n - n \) combinations of state bits are don't cares. Additionally, the state transitions may not be given for some combinations of inputs and state bits, or some states cannot occur after a specific state sequence. The latter type of DC set is called a sequential don't care set.

We are not aware of any attempt to characterize such sets in circuits that are of interest in practice. Many authors indeed operate with the abstract don't care conditions which are assumed to be random, with some unknown distribution. This interpretation is close to another area in which many don't cares are present, namely, the machine learning of concepts. Here, the goal is to construct programs (machines) which will learn some concepts (e.g., cats vs. dogs) or tasks (e.g., robot motion in environment) based on external inputs. The concepts learned are usually represented as binary or MVL functions. In these applications, too, the external don't care conditions are impossible to characterize. In these applications the functions are assumed to be highly unspecified, i.e., with many don't cares.
2.6.2 Using "don't cares"

In practice, don't care (DC) sets are used mostly to minimize some measure of circuit (implementation) or representation complexity. We discuss first the issue of reducing the complexity, and then describe some other possible uses of don't cares.

Minimizing the cost

In two-level AND-OR circuits (sum-of-product expressions), the question of dealing with DC sets is relatively easy. Using modern CAD programs, such as ESPRESSO-SIGNATURE [18], the circuits can be minimized exactly for very large functions, such as all but three large MCNC benchmark tests.

In multilevel logic synthesis, several logic synthesis methods are centered on exploring the existence of DC sets in simplifying the circuit. The examples are the methods in [22] and [66].

In sequential synthesis, the problem is largely open but several synthesis methodologies have been tried. We have used successfully don't cares in the state machine designs [105], [104] that achieved high clock rates using the relatively slow FPGA and CPLD technologies in [106].

The problem of finding optimal spectral representations of incompletely specified functions is hard. Since they act globally, over the whole domain of the definition, the assignment of a specific value to a single don't care can change many spectral coefficients. While it was proven in [50] that the the number of spectral coefficients can be bounded by the number of specified points, finding efficient algorithms that realize these bounds is not straightforward, as will be evident in the chapters that follow. The case of the Haar transform is relatively easy. This transform, by being "localized", allows the existence of an efficient algorithm for don't care simplification, as shown in [50].

Other uses of don't cares

There are many other ways in which the presence of don't cares can be exploited. Functional decomposition, detection of symmetries, classifications under permutations and negations (called also "isomorphism" in "matching" in the literature) and several other problems are of interest. It is interesting to know if the problem is easier or harder in the presence of don't
cares. We know of several results which are general enough: many heuristic approaches to use don’t cares for these problems have been considered. While some of them are practical, they generally provide no solid conclusions about the question if the presence of don’t cares makes the decision easier or harder. It was shown [45] that the presence of don’t cares can be \textit{tolerated} to detect the symmetries in Boolean functions using spectral methods. In other words, the decision is neither easier nor harder.

Results on decomposition indicate that the task is harder in the presence of don’t cares. It was shown in [16] that for some simple decompositions, the polynomial algorithms exist when the functions are completely specified, while the presence of don’t cares makes the same problems NP-complete.
Chapter 3

RM Transforms For Incompletely Specified Functions - Dense and BDD-based Interpolations

This chapter considers the Reed-Muller (RM) transform for incompletely specified functions, and outlines the polynomial interpolation approach. A background on multivariate polynomial interpolation is provided and the difficulties in solving the problem are explained. Two simple interpolation algorithms are given. First, an algorithm which incorporates the polynomial interpolation in one variable into the fast RM transform framework is presented. This algorithm is as fast as the fast RM transform, and is suited for the MVL case. A recursive algorithm for obtaining the RM transform for incompletely specified functions with a provable bound on the number of terms is presented next. Implementation using Decision Diagrams is given. Extensions of these algorithms to several other forms are presented as well.

3.1 Introduction

We define the problem of RM expansion and related forms for incompletely specified functions and give an outline of solving the problem by polynomial interpolation over finite fields. While we illustrate the difficulties with the multivariate interpolation, especially in the finite field case, the algorithms that follow are relatively simple and do not tackle
the interpolation problem in its full complexity. However, these algorithms are attractive, related to the problem considered. They also offer a way to deal with the related forms, unlike the known approaches.

### 3.2 RM Transform for Incompletely Specified Functions

There are several problems involving the incompletely specified functions and their spectral representations, depending on the target application area. One can consider using don't cares to alleviate problems like the detection of symmetries, isomorphisms between the functions, decompositions and so on. We deal here with possibly the simplest but still not resolved question, that of using don't cares to produce concise representation. We treat the binary and MVL case in as uniform manner as possible. Again, no known approach has such properties.

#### 3.2.1 Problem Definition and Known Results

An incompletely specified function, which is unspecified in $D$ points, represents in a $q$-valued case a set of $q^D$ functions. Therefore, there are $q^D$ possible transform domain representations of that function. A natural goal is to find the minimum cost form, where the cost can be specified in various ways. The most common metric used for binary RM forms is the number of nonzero coefficients, which corresponds to the number of terms used. A more sophisticated scheme given in [97] takes into account the cost of each product term in the polynomial. In the MVL domain, the cost of a term is proportional to the sum of the powers in the product term, i.e.

$$\text{cost}(x_1^{i_1}x_2^{i_2}\ldots x_i^{i_n}) = \text{const} \times \sum_{i=1}^{n} i_i$$  \hspace{1cm} (3.1)

We want to pursue another objective: while the transform of functions given in terms of truth tables is conceptually simple, we really want to employ our algorithms using the reduced representations, such as the cubes and, especially, BDD-based representations. In these cases, the cost function is defined simply as the size of the representation obtained.

In the remainder of this section, we describe the known approaches to the problem of concise RM representations of incompletely specified functions. These approaches can be categorized into the following:
• Search for optimal assignment of don't cares
• Approximation to optimal don't care assignment and heuristics
• Search for optimal selection of terms
• Interpolation approach

We will show that neither the exact optimization nor the approximation to the minimization problem is feasible, and we will retreat to a relaxed set of goals: we will be satisfied if the algorithms produce the resulting forms that are bounded with the input size, and that the running time is polynomial in the size of the input. In realizations of logic functions, the simplest circuit is the must even at the expense of exponential-time algorithm. On the other hand, as long as the representation is kept bounded relative to the input size, it is manageable, and can be used in subsequent analysis and synthesis steps. We will achieve provably these goals only in the next chapter, but the main exposition of the problem is confined to this section.

3.2.2 Exhaustive Search for Optimal Assignment of Don't Cares

The conceptually simplest synthesis approach is to generate the RM forms for all the assignments of unspecified points and find the one that has the minimum cost. Green [40] describes how this search can be organized for the binary case. By using the linearity property of the RM transform, the functions at specified points and the contributions of each unspecified point can be transformed separately:

\[ RM(f_z + d_1 + d_2 + \ldots + d_D) = RM(f_z) + RM(d_1) + RM(d_2) + \ldots + RM(d_D) \]

where \( f_z \) is the function that has assigned zeros to all unspecified points and each function \( d_i \) is 1 at the corresponding don't care \( i \) and 0 otherwise. All possible assignments of values to don't cares can be tried by summing these transforms. The complexity of such search in the binary case is

\[ (D + 1)n2^{n-1} + 2^D \times 2^n \]

i.e., in addition to \( (D + 1) \) fast transforms it is necessary to try all possible assignments of
the transformed unspecified points to the transformed specified part. The MVL version of this approach can also be done. The points at which the function is not specified have to be transformed only once, although they can take \( q \) values. The linearity property of the RM transform guarantees that this is sufficient.

An algorithm for assigning values to unspecified points is proposed in [60] in terms of a tabular technique [3], which is applicable to the binary domain only. The branch and bound search is accomplished by considering the influence of unspecified points one at a time, and pruning the branches that are guaranteed not to lead to the optimal solution. The complexity of the search reaches the complexity of the exhaustive search in the worst case.

It has been shown that the problem of representing incompletely specified binary functions by polynomials with a minimal number of coefficients is NP-hard [75]. This forces us to search for good approximations or heuristics.

### 3.2.3 Heuristics and Approximations to Optimal Assignment

A heuristic for the binary case was proposed by Varma and Trachtenberg [97], which decreases the number of operations needed while providing a suboptimal assignment. Instead of trying all possible combinations, the influence of individual points on the RM form is considered. The unspecified points are ordered by the number of spectral coefficients that they influence. The unspecified point is set to one if the coefficients that can be changed solely by that point become cheaper. One pass through the unspecified points is sufficient and the algorithm runs in time proportional to

\[
(D + 1)n2^{n-1} + D \times 2^n
\]

This heuristic has been improved to handle fixed-polarity multiple output functions in [21]. While the extension to multiple-output functions is trivial for the problem considered, the generation of related forms is not. The approach proposed in [21] is to assign the values to DC's first, and then to search for the best polarity. However, there is no guarantee that this approach is the best. In Chapter 4, we will propose alternative ways for related forms for which we can prove optimality in some, relaxed, way.
Using reduced representations - transforming cubes

A more efficient way to transform individual points and Boolean cubes is also given in [97], which can decrease the execution time of the transform algorithm. A cube is transformed componentwise by replacing each $x_i$ with $1 + x_i$, where "+" denotes the XOR operation: for example, the cube $x_1 x_2 x_3$ is transformed to $x_1 x_2 (1 + x_3)$ in the RM domain. This approach can be generalized to the MVL case. If a cube to be transformed is given by $x_1 = l_1, x_2 = l_2, ..., x_s = l_s$, then we can rewrite the binary RM transform as:

$$[1 - (x_1 - l_1)] [1 - (x_2 - l_2)] ... [1 - (x_s - l_s)]$$  \hspace{1cm} (3.2)

Indeed, if the coordinate $l_i$ is equal to one, then the term $x_i$ is present in the transform. Otherwise, the term $(1 + x_i)$ is present.

The MVL extension can be defined using the arithmetic in $GFq$ by using the fact that the expression $a^{q-1}$ is equal to 1, unless $a = 0$. Then, to transform the cube $x_1 = l_1, x_2 = l_2, ..., x_s = l_s$, we can employ the expression:

$$[1 - (x_1 - l_1)^{q-1}] [1 - (x_2 - l_2)^{q-1}] ... [1 - (x_s - l_s)^{q-1}]$$ \hspace{1cm} (3.3)

Such transform has already been used in [68]. These expressions can be used as characteristic functions of a cube.

By comparing Equations 3.3 and 3.2, we note that the fast point- and cube-wise transform cannot be used effectively in the MVL domain to speed up the computation. In the binary case, each coordinate equal to zero multiplies the total number of coefficients in the RM transform by two, while a coordinate that is equal to one does not. If $Z$ is the total number of coordinates where $l_i = 0$, then the RM expression requires $2^Z$ multiplications and terms in the expression. In MVL case, every coordinate for which $l_i \neq 0$ contributes $k - 1$ terms that multiply the rest of the expression. If $l_i = 0$, then there are two such terms. Thus, the total number of multiplications is $(q-1)^s - Z 2^Z$. This is much worse than in the binary domain - the number of terms grows exponentially in either case.

Using BDDs and the non-approximability results

It is natural to consider the BDD-based representations in both the functional and spectral domains. The question of simplifying both BDDs and FDDs using don't cares arises in
many applications [86], most notably in verification or model checking [27], [61].

It has been proven recently that using don’t cares for both the BDD and FDD optimizations is NP-hard [13]. A natural goal is to consider algorithms for approximations to the optimal forms, which are worse than the optimal result by only a constant factor. A strong negative result is contained in [13]: it is proven that producing an approximation to the optimal solution is as hard as obtaining the optimal solution. In other words, unless \( NP = P \) there can exist no good approximation algorithm for don’t care FDD minimization.

This non-approximability result justifies the approach that we pursue in this section: since both the exact minimization and an approximation to it are impossible, we will concentrate on the approaches that provide results within guaranteed bounds, and can be executed in polynomial time. Before explaining our approach, we describe one more published algorithm that deals with the important case of highly unspecified, or sparse, functions.

3.2.4 Sparse Case - Searching for Optimal Terms

All previous approaches lose attractiveness as the number of unspecified points increases. It is impossible to search the whole search space, and the heuristics perform poorly [21], [110].

A recent paper by Zakrevskij [100] describes an approach that is more suited for the case of highly unspecified functions, i.e. those that have many don’t cares. He proposes an exhaustive search for the cheapest linear combination of polynomial terms. Although the problem is NP-hard, this approach can be applied for Boolean functions with a very large number of DC’s and a small number of specified points, such that the number of terms over which the search is done is small.

The approach by Zakrevskij is important to us for one reason: it relies on the principle “don’t care about don’t cares”. This means that instead of assigning the values to don’t cares, one can approach the problem by selecting the terms of the polynomial. The selection must be such that the function coincides only at the specified points. This is the principle behind most of the algorithms that we propose.
3.3 Interpolation Approach

The problem of deriving RM transforms for incompletely specified functions can be considered as a polynomial interpolation problem over finite fields. We are given an $n$-variable partial function $f : S \rightarrow GF(q)$, where $S \subseteq GF(q)^n$ is called the "rare" set. The complement $GF(q)^n \setminus S$ is called the "don't care" set, DC. The interpolation polynomial sought (which is a total function) must coincide with the function only at its care set. This approach is based on fitting a curve through a set of points; the only difference compared to fitting a real-valued function is that we operate with finite fields.

We base our algorithms on interpolation for several reasons. First, the underlying algebraic structure will give us more tools to reason about the problem than with any combinatorial search approach. Second, since both the optimization and approximation problems are hard, the interpolation provides us with an opportunity to devise algorithms that are provably effective in a different way, i.e., we can obtain the resulting forms within the time polynomial in the size of input that have provable linear bounds on the size. Third, we will show that this approach allows us to deal not only with the RM transform and FDDs, but also with several related forms which can be useful in analysis and synthesis. Fourth, although the algorithms for binary and MVL cases might differ, this approach provides a uniform way to deal with both problems. Finally, the interpolation algorithms are interesting on their own and can be applied to other problems.

Next, we briefly describe what is known about interpolations and provide a glimpse of the results that can be expected for our application.

3.3.1 Univariate Interpolation

The simplest interpolation problem is for single-variable functions $f : F \rightarrow F$ over any field $F$. Any interpolation method, such as Lagrange or Newton interpolation, can be used in this case to obtain a bounded-degree interpolation polynomial:

$$f(x) = \sum_{i=0}^{t-1} c_i x^i$$

where $t$, the number of specified points, cannot be larger than the size of the field. Note that these approaches, while yielding the interpolation polynomial of degree bounded by $t$, do not guarantee the minimal number (or cost) of terms. However, they are considered to
be good, because the highest order coefficients are the most costly to implement, and, on average, the biggest gain is obtained if these coefficients are zero. This kind of algorithms, which provide the bound on the result will be sought in the multivariate case.

**Newton interpolation - decreasing the degree by don't cares**

To illustrate the application to incompletely specified functions, we describe briefly the use of Newton interpolation. With this approach, a set of differences is formed, which is used to calculate the coefficients of the interpolation polynomial. The differences of the function, for which we know the \( t \) values \( f_{x_1}, f_{x_2}, \ldots, f_{x_t} \) given at points \( x_1, x_2, \ldots, x_t \), are defined recursively as:

\[
\begin{align*}
[x_i, x_i] &= 0 \\
[x_i, x_j] &= \frac{f_{x_i} - f_{x_j}}{x_i - x_j} \\
[x_i x_{i+1} \ldots x_{i+j}] &= \frac{[x_i x_{i+1} \ldots x_{i+j-1}] - [x_{i+1} x_{i+2} \ldots x_{i+j}]}{x_i - x_{i+j}}
\end{align*}
\]

The polynomial representation of the function is then determined as:

\[
F(x) = f(x_1) + \sum_{i=1}^{t} (x - x_1)(x - x_2)\ldots(x - x_i)[x_1 x_2 \ldots x_{i+1}]
\]  

(3.1)

In order to obtain the coefficients \( c_i \) of the polynomial, we have to expand this expression.

The computation can be presented in form of a difference table, with each column containing differences \( \Delta^i \) of the same order \( i \). The order of a difference is a number that equals to the number of points involved minus 1. The recursive definition of the differences is used to derive the differences of the order \( i + 1 \) from the differences of order \( i \).

**Example 3**: Consider a partial function given over \( GF(4) \) by a vector of values: \( F = [f_0, f_1, f_2, f_3] = [1, 0, d, 3] \), where \( d \) denotes a don't care. For partial functions, the unspecified points are simply omitted from the equations, and the points are re-indexed. The highest order differences become zero for each don't care.

Table 3.1 shows the differences obtained using the \( GF(4) \) operations, as defined in Figure 2.1. The next step is to calculate the coefficients of the polynomial using the differences.
Table 3.1: Differences for $F = [1, 0, d, 3]$ which are placed in boxes in Table 3.1. According to Equation 3.4, they can be determined as:

$$F(x) = \boxed{1} + \boxed{1} \cdot x + \boxed{1} \cdot x(x - 1) = 1 + x^2$$

Thus, the coefficients are $C = [c_0, c_1, c_2, c_3] = [1, 0, 1, 0]$. Notice that each don't care contributes with one 0 in the highest order coefficient.

**Solving a system of equations - Lagrange interpolation**

The problem can be solved in a more direct way by setting up a system of equations which describe the condition that the polynomial fits the function values $f_x$ at each point $x$:

$$c_0 + c_1 x_0 + c_2 x_0^2 + \ldots + c_{t-1} x_0^{t-1} = f_{x_0}$$

$$c_0 + c_1 x_1 + c_2 x_1^2 + \ldots + c_{t-1} x_1^{t-1} = f_{x_1}$$

$$\ldots$$

$$c_0 + c_1 x_{t-1} + c_2 x_{t-1}^2 + \ldots + c_{t-1} x_{t-1}^{t-1} = f_{x_{t-1}}$$

This requires solving a system $Tc = f$, where the matrix $T$ is the Vandermonde matrix:

$$T = \begin{pmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^{t-1} \\
1 & x_1 & x_1^2 & \ldots & x_1^{t-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{t-1} & x_{t-1}^2 & \ldots & x_{t-1}^{t-1}
\end{pmatrix}$$

(3.5)
for which the determinant is equal to:

\[ \text{det}(T) = \prod_{0 \leq i < j \leq t-1} (x_j - x_i) \]

As long as the points \( x_i \) of the definition differ, the determinant is nonzero, and the system has a unique solution. The solution of this system, it can be shown [114], leads directly to the Lagrange interpolation method.

The solution to the interpolation problem by any of these methods is computable in \( O(t^2) \) time. We mention here for completeness the algorithm by Fiduccia [37], which achieves \( O(t \cdot \log^2(t)) \) running time for Lagrange interpolation by using a clever trick in obtaining the intermediate results. This algorithm, however, cannot speed up the computation in our case because \( t \) is always smaller than the field size.

**Small finite field interpolation**

There exist several univariate interpolation methods tuned specifically for finite fields. For completely specified functions, one such method can be derived from Menger's theorem [47], given in Chapter 2. A scheme that deals with incompletely specified functions for the important case of small finite fields (of size up to four) was reported in [103] and [110].

Starting with the linear system, as with the previous scheme, and using the properties of small finite fields, the scheme can be performed in a a way similar to the Newton method. A set of formal differences is formed, in order to keep track of the right hand sides of the equations, by the following definition:

\[
[x_1 x_j] = \frac{F_{x_j} - F_{x_1}}{S_{1j}}
\]

\[
[x_1 x_{i+1} \ldots x_{i+j-1}] = \frac{[x_1 x_{i+1} \ldots x_{i+j-1}] - [x_1 x_2 x_3 \ldots x_j]}{S_{1(i+1) \ldots (i+j-1)}}
\]

where the set of denominators \( S \) is determined as follows:

\[
S_{1j} = j - 1
\]

\[
S_{1(i+1) \ldots (i+j-1)} = S_{1(i+1) \ldots (i+j-1)} - S_{1 \ldots j}
\]

Obtaining the coefficients of the polynomial is simpler here than with Newton interpolation.
Table 3.2: Formal Differences for $F = \{1, 0, d, 3\}$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$F$</th>
<th>$[x_1 x_2]$</th>
<th>$[x_1 x_2 x_3]$</th>
<th>$[C]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$-1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$\frac{2-1}{1-0} = 1$</td>
<td>1 - 1 = 0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$d$</td>
<td>$\frac{3-1}{1-0} = 1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>$\frac{3-1}{1-0} = 3$</td>
<td></td>
</tr>
</tbody>
</table>

The denominators are used when calculating the coefficients $c_m$ from the already computed coefficients $c_{m+1}, c_{m+2}, \ldots, c_{r-1}$. The equations:

$$C_m = [x_1 x_2 \ldots x_{m+1}] - S_{m+1} c_{m+1} - S_{m+2}^2 c_{m+2} - \ldots - S_{m-1}^{r-2} c_{r-1}$$  \hspace{1cm} (3.6)

where $S_r = S_{12, \ldots, r-1}$ and

$$c_0 = f_0$$  \hspace{1cm} (3.7)

describe this process. Additionally, if the function is completely specified, the coefficient $c_{r-1}$ is obtained as

$$c_{r-1} = -(f_0 + f_1 + \ldots + f_{r-1})$$  \hspace{1cm} (3.8)

**Example 4**: Consider again the function in Example 3. Then a third-degree polynomial can be obtained as shown in Table 3.2. The highest-order coefficient is zero, because there is one don’t care, the coefficient $c_1$ is equal to $[x_1 x_2 x_3]$, according to Equation 3.6. While $c_1$ is obtained using the same equation, the coefficient $c_0$ is equal to $f_0$, yielding $C = [1, 0, 1, 0]$. The resultant polynomial is: $f(x) = 1 + x^2$.

We have shown in [110] that this interpolation performs better than the other algorithms for both serial and parallel (circuit) model of computation. Figure 3.1 contains a circuit implementing our interpolation over $GF4$, while Figure 3.2 uses Menger algorithm, which requires fewer operations than Newton or Lagrange interpolations.

For larger finite fields, it seems that the Lagrange or Newton method must be used, except if $t$ can be large enough to justify using the Fiduccia method. If the evaluation points can be chosen freely, then the $O(t \cdot \log(t))$ Fast Fourier Transform algorithm can be used, by selecting the roots of unity (so called "Fourier points"). This algorithm reaches
Figure 3.1: Circuit for small field interpolation over $GF4$.

Figure 3.2: Circuit for interpolation using Menger theorem.
the lower bound on the complexity of univariate interpolation, proven by Strassen [89] (for an overview in English, see [90]).

There exist several related problems in univariate interpolations, which are of interest in applications such as decoding of error-correcting codes. A question of univariate interpolation with a bounded degree polynomial was settled by the Berlekamp-Welch algorithm [10]. Related is the randomized algorithm by Sudan [91] that produces all polynomials of bounded degree which interpolate the function for a given set of points. This algorithm can be used for maximum-likelihood decoding of Reed Solomon codes. The question of interpolating with an arbitrary set of terms is still open for finite fields [87], while it is trivial over real numbers because the generalized Vandermonde matrix, in which the exponents are arbitrary, can be solved in that case. A list of open problems related to univariate interpolation over integers and rational numbers is listed in [15].

3.3.2 Multivariate Interpolation

The problem is much more difficult for functions of several variables. There are no good algorithms for that case. The reason is that while the selection of terms is unique in the univariate case (1, x, x^2, ..., x^n), in the multivariate case it is not and can lead to systems that cannot be solved, depending on the position of interpolation points [38]. There is no known algorithm to decide in advance if this will be the case. The solution exists only for very restricted instances of the problem. So far, it is known how to characterize only up to 6 points for which the interpolation always exists among the two-variable polynomials of degree 1 [12].

One can describe the problems that are encountered with the multivariate interpolation in several ways. Using the language of vector spaces, we say that it is not known in advance if the selected set of terms spans the desired space. Geometric interpretation is that it is not known if there exists a curve of a selected degree that passes through the interpolation points. Using the matrix notation, the multivariate interpolation is described in terms of the multivariate generalization of the Vandermonde matrix. This matrix, unlike in the univariate case, cannot be simply characterized and inverted. An equivalent statement is in terms of zero avoidance problem [26]. Namely, if there exists a polynomial which is equal to zero for all points given and terms selected, then there is no solution to the interpolation problem. We will return to these characterizations in the next chapter in more detail.
Because of the difficulty of the problem, the majority of results on multivariate interpolation are related to somewhat relaxed problems. We give in the next subsection an overview of these results, for completeness and to illustrate the algorithms for some applications like coding, learning and symbolic computation, for which these algorithms suffice.

**Overview of results for related problems**

Much work on multivariate interpolation has been done in the area of numerical analysis [12], [31]. These methods are not directly applicable to our problem, because the main concern in these applications is the numerical stability, or approximations, both of them not present in our case. Interpolations are used in statistics, but for them too, the main goal is smoothing, rather than interpolating functions exactly. Also, these approaches rely on the properties not found in finite fields. The algorithm by de Boor and Ron [31] uses properties of exponential functions over real numbers to construct the term set. The interpolation scheme developed by Kergin [53] requires the function to have sufficiently many derivatives. The algorithm by Sauer [83] is an $n$-dimensional version of Newton interpolation: a set of basis multivariate polynomials is formed, each of them being nonzero only at one point of the function definition. The resulting interpolation polynomial is then expressed by a linear combination of these basis polynomials. While the algorithm attempts to limit the degree of these basis polynomials, there is no guarantee that their degree will be bounded. The authors did not give the worst case complexity of the algorithm. Instead, the algorithm is very useful for dealing with real and rational numbers, since the numerical error can be controlled.

Interpolation over finite fields [114] has been used mostly in symbolic computation and in complexity and learning theories. In these applications it is assumed that the points of evaluation can be selected freely and/or that the degree of the polynomial is known in advance, which is not true in our case. Even for these restricted interpolation problems, three cases need to be considered, depending if the fields are $GF2$ [35], small finite fields [41], or large finite (and infinite) fields [8]. For $GF2$, a related problem has been studied, in which the maximum number of coefficients, $t$, is known in advance (so called $t$-sparse interpolation). An effective procedure exists for selecting the interpolation points [75], [35] and solving the problem using the theory of error-correcting codes. For the case of large finite and infinite fields, an algorithm for $t$-sparse interpolation has been devised by Ben-Or
and Tiwari [8], which relies heavily on the large size of the field. The practicality of this algorithm is diminished by the computations with very large numbers that can occur in the process. Two more practical algorithms are derived from the Ben-Or-Tiwari algorithm. A randomized algorithm is presented in [49] for interpolation over rational numbers, while the randomized (parallel) algorithm for large, but finite, fields was presented recently in [44]. The worst situation is with the small finite fields that are different than $GF2$. Here, it has been shown that the solution of $t$-sparse interpolation exists only in the extension fields that are large enough to allow the computation required by the algorithm [41], [26].

**Algorithms for finite field interpolation**

Much less is known about finding a solution when the interpolation points cannot be selected freely, and the degree bounds are not given, as in our problem. One known algorithm [113] is probabilistic, i.e., it is not guaranteed to complete successfully. The algorithm uses univariate interpolations to fit the one-dimensional projections of the interpolation points. It first finds a polynomial in one variable, e.g., $x_1$, at some random assignment of other variables $x_2, x_3, \ldots, x_n$. When this univariate polynomial $\sum_0^{t_1} c_i x_i$ is found ($t_1$ depends on the selected points), then, by repeating the procedure, the coefficients $c_i$ are expressed as a function of variables $x_2, x_3, \ldots, x_n$ in subsequent steps. This algorithm can be best described as a sparse version of the hypercube-based transform, described in Section 2.3.1 in Chapter 2. The difference is that this algorithm might fail if the coefficients obtained in transform with respect to a variable are not sufficient for the interpolation in the subsequent steps. Additionally, the algorithm uses the inversion of the matrix which might be singular, which is the second source of failures.

There are the other problems with using this algorithm for our problem. First, this algorithm is suited for larger fields, as the probability of failure is quadratic in the number of points and decreases with the field size. Second, the algorithm does not provide a bound, or even a probabilistic estimate, on the number of terms needed. Hence, there can be exponentially many terms in the polynomial obtained.

**Dense vs. sparse interpolation**

The size of the care set is important in our considerations. If this set is relatively small compared to $q^n$, then we have *sparse interpolation*. otherwise it is *dense*. We use different
techniques for these two cases. In the former case, we exploit the fact that there are relatively few points to be processed and solve the system of equations, while in the latter case we use the fast RM transform as much as possible. In the extreme, where there are only a few unspecified points, even the exhaustive methods can be used.

The data structures and algorithms for the two problems are adjusted to the expected complexity. For the dense case, it is assumed that we are willing to pay the price for the nonreduced representations, e.g., tabular representation. Consequently, the running times will be exponential in the number of variables. For the sparse cases, the functions of many more variables will be treated by having some sparse representation. In that case, we prefer to have algorithms polynomial in the input representation size, or in the number of points given.

### 3.4 Dense Interpolation Algorithm

It is shown in [110] how a simple interpolation scheme can be applied for the dense case such that only one fast RM transform is used over any finite field. The method uses the hypercube-based execution of the computation of the fast RM transform, in which separate steps are done for each of the $n$ input variables:

$$[F] = [B]^n \leftarrow [B]^{n-1} \leftarrow \ldots \leftarrow [B]^1 \leftarrow [B]^0 = [C]$$

Each transformation step "\leftarrow" is the transform with respect to a variable, as explained in Section 2.3.1.

In the dense interpolation scheme the univariate sparse interpolation is used in the first step of the computation, and all the subsequent steps are done as in the ordinary fast transform.

$$[F] = [B]^n \leftarrow [B]^{n-1} \leftarrow \ldots \leftarrow [B]^1 \leftarrow [B]^0 = [C]$$

where the first step, "\leftarrow", consists of $q^{n-1}$ univariate transforms for incompletely specified functions, such as those presented in Section 3.3.1.
3.4.1 Detailed Description of Dense Interpolation

The dense interpolation incorporates the one-dimensional transform for incompletely specified functions into a fast multidimensional transform. When we perform the one-dimensional transform for incompletely specified functions, each unspecified point leads to a zero in the place of one highest-order coefficient in the transform domain, as a result of the univariate interpolation. A single application of the one-dimensional transform in execution of the fast multidimensional transform corresponds to an application over one value-vector. An n-dimensional transform with respect to \( x_i \) involves \( q^{n-1} \) one-dimensional transforms over all value-vectors in \( x_i \). If there exists at least one unspecified point, \( d \), in each value-vector in \( x_i \), then it is guaranteed that the hyperplane \( x_i^{k-1} \) will contain all zeroes. Hyperplane, as usual, is a subspace of dimension \( n - 1 \). If all the coefficients in some hyperplane become zero, then any further transform step will leave these zeroes unaffected. In addition to zeroes created in the hyperplane \( x_i^{k-1} \) by transforming value-vectors with unspecified points, we note that sometimes zeroes are obtained from completely specified value-vectors, if such vectors comprise values whose sum is zero (as shown by Equation 3.8). The above observations may affect the lower-order hyperplanes, \( x_i^{k-2}, x_i^{k-3}, \ldots \).

**Example 5** : Consider the following 2-variable 4-valued function with 4 unspecified points:

\[
\begin{pmatrix}
  d & 3 & d & 0 \\
  1 & 0 & d & 3 \\
  d & 2 & 1 & 1 \\
  0 & 1 & 2 & 2
\end{pmatrix}
\]

\[ f(x_1, x_2) = \]

When this function is transformed by choosing columns first, the matrix \([B]^{1}\) is:

\[
[B]^{1} = \begin{pmatrix}
  2 & 3 & 0 & 0 \\
  3 & 3 & 3 & 3 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0
\end{pmatrix}
\]

and the RM coefficient matrix is:

41
\[
[C] = \begin{bmatrix}
2 & 3 & 3 & 1 \\
3 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

From the definition in Equation 3.1 (\textit{const} = 1), the cost of this solution is 7. Two rows (hyperplanes) have been eliminated due to the assignment of unspecified points.

If we select rows first, we obtain the solution:

\[
[C] = \begin{bmatrix}
1 & 2 & 0 & 0 \\
0 & 3 & 2 & 2 \\
2 & 3 & 2 & 3 \\
2 & 2 & 1 & 1
\end{bmatrix}
\]

which has the cost of 42. Finally, we note that if the function is transformed by assigning zeroes to all unspecified points, an even more expensive solution, with a cost of 48, is obtained:

\[
[C] = \begin{bmatrix}
0 & 3 & 3 & 3 \\
1 & 2 & 1 & 1 \\
1 & 2 & 1 & 2 \\
1 & 2 & 1 & 2
\end{bmatrix}
\]

It is clear that the effectiveness of this transform depends on the order in which the variables are considered in the fast RM transform. However, only the selection of the first variable has to be considered, because the unspecified points are assigned concrete values in this step.

This algorithm is slightly slower than the RM transform for completely specified functions. The bound on the search for the solution is \(O(n)\). This is very small compared to either the optimal solution search with a bound of \(O(k^n)\), or the Varma-Trachtenberg (VT) method with the search complexity of \(O(k^n)\). The search complexity should be multiplied by the number of operations executed in one search pass, and added to the complexity of obtaining the initial transforms, before the search procedure starts. When all the components of the algorithms are taken into account, the ratio of worst case complexities between
3.4.2 Empirical Comparison

We have tested the efficacy of our method on 10000 randomly generated incompletely specified 4-valued functions having a wide range of unspecified points. We generated the RM transforms using our method as well as the MVL extension of the VT method. We also tried a simple solution where all don't cares were replaced with zeroes. The results for the 2-variable case are shown in Figure 3.3. The curves correspond to the average relative cost differences with respect to our method. It is clear that the VT method produces excellent results when the proportion of don't cares is low. Our method gains superiority as this proportion increases. In view of this fact, we also show a curve (called "Combination") that is obtained if one chooses the better solution of either VT or our method for each function. While the VT method is superior on average for functions with a low percentage of unspecified points, there are some cases where better solutions are obtained with our method even for functions of this type. The overall effectiveness of these methods can be assessed by comparing with the optimal solutions produced by the exhaustive search, which is also indicated in the figure.

The running times of the program (run on the SPARCStation II) were 53 seconds for our algorithm, 2.7 minutes for the VT algorithm and 18.5 hours for finding the optimal solution. Similar results are shown for the functions of three variables in Figure 3.4. The results for the optimal solution could not be obtained in this case, due to the excessive computational
requirements, as the solution space can grow as quickly as \( O(k^n) \). It is impossible to produce similar statistically significant results for higher dimensionality functions, due to the computational requirements of the VT method. While the crossover point between our and the VT method moves to the right when the dimensionality increases, we have observed (in tests with functions of up to 5 variables) that even to the left of the crossover point our method produces results that are within 13\% (on average) of those obtained with the VT method.

The dense interpolation was the first feasible MVL method for dealing with dense functions that still have a substantial DC set such that the exhaustive methods or heuristics cannot be used. The complexity of this algorithm is equal to the complexity of \( n \)-times repeated fast RM transforms: it is applicable in all those cases in which the fast RM transform is feasible.

There are some limitations of this method: it is not directly suited for binary functions and its performance worsens when the number of variables increases. While the computing resources required are on the order of the fast RM transform, there is no bound on the resulting forms. Finally, the algorithm does not use the reduced representations of functions.

In the remainder of this chapter, we will derive approaches that work well with binary functions and whose performance does not deteriorate with an increase in the number of variables: instead, the number of nonzero coefficients will be bounded by the number of specified points. Additionally, we will use some form of sparse representation for both the input definition and the output polynomial. Finally, the extensions to some related RM
forms will be presented.

3.5 Sparse Interpolation over GF2

In this section, we present new methods for dealing with incompletely specified functions. The main idea that the RM transform is treated as a polynomial interpolation is augmented with the notion of "sparseness", familiar in numerical and symbolic computing. In many applications, given logic functions have large DC sets, hence they are sparse.

We first derive a recursive algorithm for interpolating incompletely specified functions. Since the basic, naive implementation of the algorithm still requires exponential time, we next derive a more efficient algorithm which uses the data structure of BDDs.

3.5.1 Recursive Sparse Interpolation Algorithm

The reasoning in this section is based on the fact that the number of terms in a multivariate polynomial is bounded. For the case of GF2, there is a theorem by Rackoff [70]:

**Theorem 1.** A multivariate partial function over GF2 given by t points can be interpolated with a polynomial that has at most t terms.

**Proof:** By induction on the number of variables.

**Induction Basis:** For \( n = 1 \), univariate interpolation algorithms exist which satisfy the statement of the theorem.

**Induction Hypothesis:** Every partial \( n \)-variable function \( \tilde{f} \) specified at \( t \) points can be interpolated by a polynomial with at most \( t \) terms.

**Induction Step:** Suppose the statement is true for the \( n \)-variable case. Then the interpolation \( f \) of an \((n + 1)\)-variable partial function specified at \( t \) points can be written as:

\[
f = f_0 + x_{n+1}f_1
\]

By setting \( x_{n+1} \) to 0, a partial function in \( n \) variables is obtained. Suppose that it is specified in \( t_0 \) points. Then, the function can be interpolated by \( f_0 \) alone. By induction hypothesis, \( f_0 \) has at most \( t_0 \) coefficients.
Now, restrict the function by setting $x_{n+1}$ to 1. Since $f_0$ is total, we can subtract it from the original function:

$$\tilde{f} - f_0 = \tilde{f}_1$$

The expression $\tilde{f} - f_0$ is specified only where $\tilde{f}$ is specified for $x_{n+1} = 1$, which is in $t_1 = t - t_0$ points. By induction hypothesis, it can be interpolated by an $n$-variable polynomial $f_1$ with at most $t_1$ terms. From these observations, we conclude that the interpolating polynomial has at most $t = t_0 + t_1$ nonzero coefficients. □

3.5.2 Straightforward Implementation of the Interpolation Algorithm

The above theorem is constructive and its straightforward implementation is as follows. If we denote the unspecified points in the function definition as $d$, and the partial operation $+_d$ to be $d$ whenever the second operator is $d$, then

Algorithm 5 Sparse Interpolation

$$RM_d(n, f) = (f_0 = RM_d(n - 1, f|_0) \| RM_d\left(n - 1, RM^{-1}(n - 1, f_0) +_d f|_1\right)$$

$$RM_d(1, f) = 0 \| 0, \text{ if both points are } d$$

$$RM_d(1, f) = z \| 0, \text{ if one point is } d, \text{ and the other is } z$$

$$RM_d(1, f) = RM(1, f), \text{ if neither point is } d$$

gives the construction given in the above proof. We can also use the fact that for a binary function $RM^{-1} = RM$.

**Lemma 1** Algorithm 5 produces the RM transform for partial functions in $O(n^2 2^n)$ time.

**Proof:** The running time of the algorithm can be determined as follows. At each stage $n > 1$ there are two invocations of the algorithm with $n - 1$ variables, plus one invocation of the ordinary RM transform algorithm. This leads to the recurrence for the number of operations:

$$C(n) = 2C(n - 1) + 2^{n-1} + (n - 1)2^{n-2}$$

for which we can establish the complexity bound:

$$C(n) < n^2 2^{n-2}$$

46
\[
C(n+1) < 2n^2 \cdot 2^{n-2} + 2^n + n2^{n-1} < (n+1)2^{n-1}
\]

Similar to the dense interpolation in Section 3.4, we repeat the transform \(n\) times, for all possible selections of the first variable; the complexity is then \(O(n^32^n)\). Note that in this case there are more than \(n\) different input variable permutations with respect to the final result. We recapitulate the complexity bounds for the four algorithms discussed above for the binary case in Table 3.3.

The bound on this algorithm is tight when the recursive tree-based execution and matrix representation of the RM transform are assumed to be comparable with other algorithms discussed. We will show how this algorithm can be improved by using the BDD and FDD data structures.

### 3.5.3 BDD Implementation of the Sparse Interpolation Algorithm

We now describe an implementation of Algorithm 5 that uses sparse representations of both the function and its transform. The input sparse representation is in form of BDDs adapted for the presence of the third value, \textit{don't care}, which we call BDD\_DC.

This extension allows representation of incompletely specified functions by only one BDD. Figure 3.5a shows one such BDD\_DC where the third value is \(d\). for don't care. This kind of BDDs can be constructed with only a slight performance penalty compared to the usual BDDs and we use it here to represent incompletely specified functions. This representation was used successfully elsewhere [63].

The recursive definition of the sparse interpolation transform allows us to use the BDD data structure - each node in a BDD can be interpreted as a recursively applied Shannon decomposition of a given function. The sparse interpolation scheme can be directly

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive</td>
<td>(O(2^n))</td>
</tr>
<tr>
<td>Varma-Trachtenberg</td>
<td>(O(n2^n))</td>
</tr>
<tr>
<td>Dense Interpolation</td>
<td>(O(n^22^n))</td>
</tr>
<tr>
<td>Sparse Interpolation</td>
<td>(O(n^32^n))</td>
</tr>
</tbody>
</table>

Table 3.3: Algorithm Comparisons - Nonreduced Representation of RM Transform

Indeed, the induction step in the inductive proof of this bound shows that for \(n > 2\):

Indeed, the induction step in the inductive proof of this bound shows that for \(n > 2\):

Indeed, the induction step in the inductive proof of this bound shows that for \(n > 2\):
Figure 3.5: Transform from a BDD-DC to an FDD via an interpolated BDD

implemented using BDD-DCs and applying Algorithm 5 in preorder fashion.

The algorithm can be organized as follows. At each node of the BDD-DC being processed, a pointer to the current FDD and a BDD that represents the interpolated function (the one which has values for DCs assigned) is maintained. To produce the interpolated BDD, we have to calculate the inverse RM transform, if the edge points to the terminal node. Otherwise, it is sufficient to merge the two interpolated BDDs from the successor nodes of the current node. To produce the resulting FDD, the 0-successor of the current node in the FDD corresponds to the FDD of the 0-successor of the current node, while the 1-successor is obtained by XOR-ing the two successor FDDs. This is equivalent to the transform for completely specified functions. The complexity of the overall algorithm is dominated by performing XOR operations and the inverse transform.

Figure 3.5 gives an example of this transform. Shown are the original BDD-DC, interpolated BDD (BDD obtained when this algorithm assigns values to DCs) and the resulting FDD. Nodes 2 and 3 are transformed in a straightforward way, because they represent completely specified functions. When node 4 is transformed, specific values are assigned to DCs for the first time. Since its 1-edge points to the 0 node, any addition of the type \( +_1 \) must produce 0 in Algorithm 5, and the transform for incompletely specified functions will produce a node 0. If 0 is a transform of the 1-successor, then we know that both edges will point to the same successor. Hence, node 4 disappears in the interpolated BDD. The FDD obtained from node 4 has node 7 as its root. Since node 4 is the 0-successor of the root node
of the BDD DC, none of these results will change in subsequent steps of the algorithm. The transform of the 1-successor of the root (node 3) is obtained by XOR-ing the left subgraph of the interpolated BDD and the right subgraph of the original BDD DC. The result of this operation is shown in the earlier example in Figure 2.3. A transform for completely specified functions can be applied to the result of this operation. The final result is shown in Figure 3.3c.

We now give several lemmas that give some insight into the operation of the algorithm and the sizes of BDDs and FDDs used. We use the (generic) terms BDD DC, BDD and FDD to denote the input to the algorithm, the interpolated BDD and the result, respectively.

**Lemma 2** For an n-variable function given at t points, the FDD produced using Algorithm 3 is linear in t:

$$|FDD| = O(nt)$$

**Proof:** By Theorem 1, each polynomial representing such a function has at most t terms. Then, an FDD representing such a polynomial will have at most t paths leading to the terminal node 1. All nodes in the FDD must be on these paths. If we suppose by way of contradiction, that there is a node which is not on one of these paths, then both of its outgoing edges must be on a path to terminal node 0. Since the BDD is reduced, we know that such nodes do not exist.

An upper bound on the number of nodes is then $n \cdot t$, because in the worst case all the paths (except for the root and terminal nodes) will be disjoint. $\square$

As a corollary, the intermediate FDDs will be limited in size by $nt$, unlike in the general case, in which intermediate FDDs can be much larger than the final result.

**Lemma 3** Each assignment of values to DCs in Algorithm 5 will produce an interpolation BDD of the size equal to or smaller than the size of the original BDD with DCs being processed

$$|BDD| < |BDD_{DC}|$$

**Proof:** The values are assigned to DCs only when a DC node (node d) is encountered the first time in a traversal. We show that in such cases the size of the interpolated BDD is either smaller or it stays the same. There are two possible cases: the terminal d node is either the 0- or the 1-successor - it cannot be both.
If that node is the 0-successor of a node \( r \), then the transform would be

\[
a_0 = R.M_d(r,0) = 0
\]

and the transform for the 1-successor is obtained as

\[
R.M_d \left( R.M^{-1}(a_0) + i \right) = R.M_d(r,1)
\]

Here, the fact that \( R.M^{-1}(a_0) = 0 \) causes the edge 0 of the interpolation BDD to point to the terminal node 0. Hence, in this case, the size of the \( \text{BDD}_\text{DC} \) and BDD will be the same.

If the edge 1 points to the terminal \( d \) node, then the 0-successor is transformed to \( a_0 = R.M_d(r,0) \), and the interpolated value is

\[
\hat{f} = R.M^{-1}(a_0).
\]

The 1-successor of the \( RM \) transform is then obtained as

\[
R.M_d \left( \hat{f} - i \right) = R.M_d(d) = 0
\]

by definition of the function \( \hat{f} \). Since the transform is \( a_0 \| 0 \), we conclude that the node \( r \) will be eliminated from the interpolated BDD. \( \square \)

This lemma indirectly shows that the transform algorithm will have at least as many 0s as the number of DCs in the given function.

**Lemma 4** For a function given by \( t \) points, the size of the input \( \text{BDD}_\text{DC} \) is limited by \( \| \text{BDD}_\text{DC} \| < nt \).

**Proof:** Consider the number of paths leading to either terminal node 1 or 0. This number is \( t \), and by the same reasoning as in Lemma 2, we know that all nodes of a \( \text{BDD}_\text{DC} \) must lie on these paths. Again, the number of nodes is upper bounded by \( nt \). \( \square \)

These three lemmas are not sufficient to prove that the algorithm will run in polynomial time. Although the input and output representations, as well as the intermediate transforms are linear in \( t \), the algorithm is executed in preorder fashion, which means that the algorithm does not simply traverse the original BDD and assigns the transform to each node. Instead,
a new subgraph is formed for each 1-successor in a BDD-DC, and it is not possible to bound the size of graphs obtained this way. However, the lemmas show that it is beneficial to use BDDs and FDDs. First, the size of the input and output is linear in \( nt \) and second, the dynamically updated BDD is smaller than the BDD maintained by the RM transform for completely specified functions.

### 3.5.4 Empirical Comparison of the Algorithms

To assess the performance of the proposed algorithms, we performed experiments over large sets (over 10000) of randomly generated incompletely specified functions. Figure 3.6 shows the relative cost difference of other schemes versus the new sparse interpolation algorithm for incompletely specified functions, for which the difference is 0.

The worst behavior is exhibited by algorithm Simple, which assigns all the values to zero: it departs fast from the interpolation method as the percentage of don't cares increases. Sparse interpolation method performs on average better than the Varma-Trachtenberg method (VT in Figure 3.6) and its performance gain increases with the increase in per-
Figure 3.7: Reduction to BDDs: a) BDD. DC, b) assignment of values to DC's. c) RM transform

centage of DC's, as expected. The performance of the combination of the new interpolation and VT method is also shown, which indicates that a gain can be obtained when the VT and improved interpolation schemes are both used. Graphs with the same trends were obtained for varying numbers of inputs.

3.6 Dense BDD-based Algorithm and Related Forms

We have shown two transforms, dense and sparse, for incompletely specified functions. We want to extend these algorithms to related forms. To achieve this, we now show another fast method that uses the BDD representation, like the sparse algorithm, but it is based on the dense interpolation scheme. We use the same reasoning: the first time when a d is encountered, it can be assigned the value such that the resulting local RM transform is the simplest. This is achieved if both outgoing edges point to the same node. Then, it is guaranteed that the upper half of the coefficients is zero, according to Equation 2.3.

Figure 3.7 shows an example of this assignment. It produces the completely specified BDD in Figure 3.7c, from which we can easily calculate the RM transform for completely specified functions. Associated with each node in Figure 3.7c are the resulting local RM transforms - the one associated with the root node is the transform of the whole function. Note that using BDDs each assignment can eliminate more coefficients than the dense interpolation method. However, in this case up to n! orderings of variables should be tried to obtain the best possible result. We do not consider here the problem of ordering the
variables.

### 3.6.1 Extensions to Pseudo Reed-Muller and Pseudo Kronecker Forms

Various extension forms can be produced in the same step of obtaining the transform from BDDs. Fixed polarity RM expressions (FPRME) can be obtained if all nodes at the same level are inverted consistently. If the polarity change is done independently at each node, then we obtain a pseudo Reed-Muller form (PRM). At each node, the expansion can be either as in Equation 2.3, or of the type:

$$ f = f|_{x=1} + \bar{x} (f|_{x=0} + f|_{x=1}) $$

This expansion is known as the negative Davio expansion, or the RM transform with respect to the inverted variable.

A simple way to reduce the complexity of the resulting form is to compare the number of 0s in the transforms of the two subgraphs: whichever one is simpler becomes the 0-successor of the resulting list, while the 1-successor is always equal to the sum of the successor transforms. In this way, we obtain the function PRM:

$$ PRM(v) = Pad(PRM(v,0)) || Pad(PRM(v,0)) + Pad(PRM(v,1)) \text{ or } Pad(PRM(v,1)) || Pad(PRM(v,0)) + Pad(PRM(v,1)), $$

whichever is simpler

This operation is equivalent to interchanging the outgoing edges from a node in a BDD. The polarities are assigned in a greedy bottom-up manner.

To obtain the pseudo Kronecker expressions, we use the Shannon expansion at each node. This expansion can be written using the $GF2$ operations as:

$$ f = (1 + x)f|_{x=0} + xf|_{x=1} = \bar{x}f|_{x=0} + xf|_{x=1} $$

A greedy bottom-up construction of pseudo Kronecker expressions can be described as:

$$ KR M(v) = Pad(KRM(v,0)) || Pad(KRM(v,0)) + Pad(KRM(v,1)) \text{ or } $$
As before, the values can be assigned to DCs the first time the \( GF_2 \) polynomial expansion has to be produced.

Although optimality is not claimed for these procedures, we note that the KRM algorithm that selects the expansion type in the bottom-up fashion will not increase the number of nonzero terms in any step, which is not the case with the greedy PRM algorithm.

### 3.6.2 Empirical Evaluation of all Methods Considered

We have implemented and compared all of the discussed methods using randomly generated sets of functions. The percentage of incompletely specified points varied from 0 to slightly over 50 percent. Each set comprised over 10000 functions. The measurements were done on functions of up to 10 inputs: no such graphs could have been obtained for larger functions.
Our goal was to understand the relative performance of all methods, especially relative to the percentage of unspecified points. Figure 3.8 shows that comparison.

The method that uses BDDs to assign the DC values is not much better than the simple assignment. However, we have tried only one ordering of variables, even though more than \( n \) orderings should be considered. For comparison we also recorded the results for the sparse interpolation method when only one ordering of variables is considered (Interp-1).

The lowest two curves contain the comparison with the costs of inconsistent forms - pseudo RM and pseudo Kronecker forms. These two were produced by BDDs, using just one ordering of variables. The best results produced by pseudo Kronecker expansions can be up to 40 percent better than those obtained by interpolation alone. These results are consistent with those published by Sasao [77], [79], [80].

We can gain much more with assignment of DCs when their percentage is high, which makes interpolation schemes more favorable. Since the improvement by choosing alternative RM forms can be more significant, it is important to have algorithms that perform the assignment of unspecified points fast. Even with the poor DC assignment, like in the BDD-based scheme with only one ordering, the overall savings with reduced forms dominate. Finally, while our approach can be used easily with various extensions to RM forms, this is not the case with the VT or any other previous method.

### 3.7 Sparse Interpolation and Higher Radices

The proposed method can be extended to finite fields \( GF_q \), for \( q > 2 \) [111]. However, compared to the RACKOFF theorem, the number of terms in the interpolation polynomial cannot be bounded by the number of specified points. Instead, we can derive bounds on the number of terms which are polynomial in the number of points. First, we illustrate how the RM transform can be obtained for \( q \)-valued incompletely specified functions.

For example, for \( q = 3 \), the recursive definition of the RM transform (GF3 polynomial) has the form:

\[
f_n = (a_0 + a_1 x_n + a_2 x_n^2) f_{n-1}
\]  

(3.11)

where \( f_n \) is an \( n \)-variable RM transform, while the expression in brackets is a polynomial in the last variable.

The sparse interpolation approach cannot guarantee the linear bounds as in the binary
case. Even if we assume that the polarity of a variable can be selected in each step of the RM transform (as in pseudo-RM transform), these bounds are superlinear. For the case of $GF3$, the bound can be obtained as follows.

In $n$-th recursive step of the RM transform over $GF3$, the polynomial is expanded around the variable $x_n$. By assigning this variable to zero, we obtain the coefficient $a_0$ from Equation 3.11. By assigning the values $x_n = 1$ and $x_n = 2$, other coefficients are obtained after solving the system of equations obtained.

We now count the number of terms in the worst case. Knowing that we can select the polarities freely, the worst case is when the number of specified points is distributed equally among the subsets of points given by $x_n = 0$, $x_n = 1$ and $x_n = 2$. By assigning $x_n = 0$, the number of terms in the expression is equal to the number of terms for the problem that has one third of specified points, i.e.

$$T(t, x_n = 0) = T(t/3).$$

When the other two values are assigned, we know that the polynomial will be a linear combination of the two expressions in $n - 1$ variables. i.e.

$$T(t, x_n = 1) = T(t, x_n = 2) \leq T(t/3) + T(t/3).$$

By summing these expressions, we obtain:

$$T(t) = T(t, x_n = 0) + T(t, x_n = 1) + T(t, x_n = 2) \leq 3T(t/3)$$

The given recurrence equation has the solution:

$$T(t) \leq 5^{\log_2(t)} = t^{\log_2(5)}$$

which amounts to $T(t) \leq t^{1.46}$.

For any radix $q > 2$, there will be $(q - 1)$ terms that are obtained as a linear combination of $(q - 1)$ expressions for a polynomial in $n - 1$ variables. Then, the bound can be obtained from the expression:

$$T(t) \leq T(t/q) + (q - 1)^2T(t/q)$$
which has the solution

\[ T(t) \leq t^{\log_4(q^2 - 2q + 2)}. \]

When \( k \) becomes large, this bound can be approximated as

\[ T(t) \leq t^{\log_4(q^2)} = t^2. \]

Hence, using this approach, the bound on the number of terms is quadratic in the number of specified points in the limit, when \( q \) goes to infinity.

### 3.8 Concluding Remarks

We considered polynomial interpolation for obtaining concise RM transforms for incompletely specified functions. While the minimal RM representation and the close approximations to it are not feasible to obtain, we have shown that the interpolation approach produces forms with bounded maximum cost of realization. The interpolation problem over finite fields is reviewed and two cases, that of dense and sparse functions, are established. Using dense interpolation, a simple algorithm which is as costly as the fast RM transform can be used for MVL functions. The bounds on the number of terms needed in the RM transform using the sparse interpolation approach are linear in the case of binary functions and quadratic for the worst MVL case. The actual average performance over sets of randomly generated functions compares favorably with the best known heuristic methods, while the complexity of the proposed approaches is significantly lower. These approaches have been used in conjunction with the BDD representations and extension of RM forms.
Chapter 4

New Finite Field Polynomial Interpolation Algorithms

This chapter presents polynomial time algorithms for the sparse multivariate polynomial interpolation over finite fields, which can be used for optimizing Reed-Muller forms for binary and MVL functions. To derive such algorithms, we use new results on interpolation over finite fields. In the previous chapter, we explained the difficulty with multivariate interpolation. One formulation of the problem can be given in terms of a multivariate Vandermonde matrix, for which there were no known characterizations. In this chapter, we concentrate on this formulation, and provide some characterization of the structure of that matrix. First, based on the partial order between the interpolation points, we show how the interpolation problem can be decomposed into interpolations over suitable subspaces of the function definition domain. Second, we derive an algorithm that solves the interpolation problem over each subspace; the approach presented can be applied to interpolation over any field and can have many possible applications.

This chapter is organized as follows. In Section 4.2, we give an algorithm for sparse interpolation over $GF2$ which is a quadratic running time improvement of the algorithm presented in the previous chapter. The algorithm relies on ordering between the interpolation points. In Section 4.2, the algorithm is given in its simplest form, by using the lexicographic ordering between points. The extension of this approach to larger fields, together with the more general formulation, is given in Section 4.3. It is shown that the algorithm decomposes the original problem into possibly many smaller problems. While
this decomposition is sufficient to achieve the quadratic time GF2 interpolation, the resulting algorithms for finite fields other than GF2 are not as straightforward. A series of steps are required, each of which increases the rank of the matrix of the system by one. In deriving the algorithm, we first use heuristic reasoning in Section 4.5.3 and show that a practical probabilistic algorithm is possible. We extend this approach to fixed polarity RM forms and show that not only the resulting forms become simpler, but that the algorithms also run faster. Finally, we discuss the incremental and parallel versions of the algorithm and its applications.

4.1 Reformulation of the Problem - Additional Definitions

To describe the algorithms for multivariate finite field interpolation, we introduce some additional notation. We are given an \( n \)-variable partial function \( f : S \rightarrow GF_q \), where \( S \subseteq GF_q^n \) is called the “care” set. Let \( S \) consist of \( t \) points, \( p_1, p_2, \ldots, p_t \). To reduce the number of subscripts we use small letters, like \( p \), for individual points, and indexed letters, as in \( p_i \), when several points are considered. When the coordinates of points are discussed, we use double-indexed small letters. For example, \( p_{ij} \) denotes coordinate \( j \) of point \( p_i \). A set of points is denoted by the capital letter \( P \).

We want to find the multivariate polynomial \( f = \sum_{j=1}^{t^*} c_j \cdot m_j \), which is a linear combination of at most \( t^* \), terms (monomials)

\[
m_j = \prod_{k=1}^{n} x_k^{m_{jk}}
\]

that must interpolate the function exactly at the care set \( S \). Since the terms are specified by their \( n \) exponents, we treat the points and terms uniformly, as points in \( n \)-dimensional space. The set of all terms is denoted by \( M \).

We want to provide a bound on the set of all terms that can appear in the interpolation polynomial. It was proven in the previous chapter that for the binary case \( t_s \leq t \) always holds. Assuming that the structure of the term set \( M \) is known and the bound on its size is \( t \), a solution to the interpolation problem can (in principle) be obtained by solving a system of linear equations

\[
f = Te
\]

(4.1)
which corresponds to the condition that the function values \( f \) and the sought polynomial coincide at the points given. Vector \( c \) then contains the coefficients of a chosen set of terms.

Provided that a solution can be found, the complexity of solving this system is \( O(t^3) \). Note that if we were able to use the fast RM transform, the algorithm would require only \( O(t \cdot \log(t)) \) operations. While we will avoid the explicit construction of the transform matrix \( T \), the solution will be based on investigating the structure of that matrix. We will characterize the conditions under which the solution to the problem can be found.

The size of the finite field also plays a role in the derivation of the interpolation algorithm. The problem is the simplest for \( GF2 \). For large fields the methods will be similar to those used for real and rational numbers, while for small fields we have to search for alternatives. Since we are primarily interested in interpolation over small finite fields, we deal first with a relatively easy case of \( GF2 \) and then extend these results to deal with other finite fields.

### 4.2 Interpolation over GF2 - Simple Case

By considering the linear system of equations that has to be solved in the binary case, we give a characterization of the structure of the resulting terms. This allows us to derive a quadratic time algorithm.

Let a function be given by a list of points \( p_1, p_2, \ldots, p_t \in GF2^t \) and the function values at these points \( f_1, f_2, \ldots, f_t \). The interpolation polynomial may have up to \( t \) terms, \( m_1, m_2, \ldots, m_t \), in a linear combination

\[
\sum_{j=1}^t c_j \cdot m_j
\]  

where \( c_1, c_2, \ldots, c_t \) are the coefficients associated with each term. To find the interpolation polynomial, we have to:

- find the terms \( m_j \), and
- determine the values of coefficients \( c_j \).

Consider the structure of a system of linear equations used to find the interpolation polynomial. The system is given by the condition that the polynomial equals the function
value at the interpolation points

\[ \sum_{j=1}^{t} c_j \cdot m_j(p_i) = f_i, \quad i = 1, 2, \ldots, t \quad (4.3) \]

which can be represented in matrix form as \( Tc = f \), where the vectors \( c \) and \( f \) are the coefficients and the function values, respectively. The multivariate Vandermonde matrix \( T \) consists of all applications of terms to points.

\[ T_{ij} = m_j(p_i). \]

The application of a term to a point is the value that the term takes at that point. We will also use the symbol \( p_{i}^{m_j} \) to denote the application, because for an \( n \)-variable case the application value is obtained as coordinate-wise exponentiation in \( GF(2) \):

\[ m_j(p_i) = p_{i1}^{m_{j1}} p_{i2}^{m_{j2}} \cdots p_{in}^{m_{jn}}. \]

We will use extensively the lexicographic order \( <_L \) among the points in \((GF(2))^n\). The main results in this section rely on the relation between the order and the monomial application. The following three facts are useful:

**Lemma 5** Consider two \( n \)-tuples, \( p_1 = p_{11}p_{12}\cdots p_{1n} \), and \( p_2 = p_{21}p_{22}\cdots p_{2n} \) of \( GF(2) \) elements. If \( p_1 <_L p_2 \), then there exists some \( p_{1i} \) that is zero, while \( p_{2i} \) is not.

**Proof:** Suppose that there is no such \( i \). Then, for each \( p_{2j} \), points \( p_{1j} \) are greater than or equal, which contradicts the fact that \( p_1 <_L p_2 \). \( \square \)

**Lemma 6** When a term \( p \) is applied to itself, the result is

\[ p(p) = 1. \]

**Proof:** For each \( p_i \) in an \( n \)-tuple, the application

\[ p_{1}^{p_{1}^{p_{1}}} p_{2}^{p_{2}^{p_{2}}} \cdots p_{n}^{p_{n}^{p_{n}}} \]

yields either \( 0^0 = 1 \) or \( 1^1 = 1 \). \( \square \)
Lemma 7 If two terms $p_1, p_2$ are in relation $p_1 \leq p_2$, then

$$p_2(p_1) = 0$$

Proof: The application has the form:

$$p_{11}^{p_{21}} p_{12}^{p_{22}} \cdots p_{1n}^{p_{2n}}$$

By Lemma 5, there exists an index $i$ for which $p_{1i} = 0$ and $p_{2i} \neq 0$. For this index, the application $p_2(p_1)$ results in $0^i = 0$.

To be able to solve the system defined by Equation 4.3, the matrix $T$ has to be nonsingular. To achieve this, the set of terms (monomials) has to be carefully selected. We are able, using the lemmata just proven, to characterize a choice of terms that always leads to the solution by the following theorem.

Theorem 2 If the point and term sets coincide, the system in Equation 4.3 is nonsingular.

Proof: Elements of the matrix are of the form $m_j(p_i) = p_j(p_i)$. For an $n$-variable function, each point $p_i$ is expressed by an $n$-tuple $p_{i1}p_{i2}\cdots p_{in}$. Application of a term $p_j$ to a point $p_i$ then results in

$$p_{i1}^{p_{j1}} p_{i2}^{p_{j2}} \cdots p_{in}^{p_{jn}}$$

which is always: $a_j$ nonzero for diagonal elements $p_i(p_i)$ and $b_j$ zero above the diagonal $p_j(p_i)$, with $i \geq j$. This gives a lower-triangular system, which is nonsingular.

This characterization allows us to define an $O(n \cdot t^2)$ algorithm for interpolation over $GF(2)$, which solves the system from Equation 4.3 by elimination.

Theorem 3 Interpolation of an $n$-variable function over $GF(2)$ given by $t$ points can be done with $O(n \cdot t^2)$ operations.

Proof: By selecting the points using Theorem 2 one obtains a lower-triangular system. The solution for such a system is:

$$c_1 = f_1$$

$$c_i = f_i - \sum_{j=1}^{i-1} T_{ij} \cdot c_j = f_i - \sum_{j=1}^{i-1} p_j(p_i) \cdot c_j$$

(4.4)
This result can be computed with \((3/4)n \cdot t^2 - (5/4)nt\) bitwise operations. This time dominates the sorting step that may precede it if the points are not in order. \(\Box\)

The following example will illustrate the interpolation algorithm. We will first apply the full matrix description, which involves solving a system of linear equations. Then, we show the benefits of using the sparse representation. We use simple lists to store both input and output of the algorithm: no additional storage is required.

**Example 6** Consider a function given by the following sparse representation:

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Matrix Solution:** Using Theorem 2, we select the polynomial terms that correspond to the points. After ordering the points lexicographically, we compose the system of equations

\[
c_{000} + c_{001}(000)^{001} + c_{101}(000)^{101} + c_{110}(000)^{110} = 1 \quad (= f_{000})
\]

\[
c_{000} + c_{010}(001)^{001} + c_{101}(001)^{101} + c_{110}(001)^{110} = 1 \quad (= f_{001})
\]

\[
c_{000} + c_{001}(101)^{001} + c_{101}(101)^{101} + c_{110}(101)^{110} = 1 \quad (= f_{101})
\]

\[
c_{000} + c_{010}(110)^{001} + c_{101}(110)^{101} + c_{110}(110)^{110} = 0 \quad (= f_{110})
\]

Note that we use the shorthand notation \(p^n\), as in \((001)^{101}\), for coordinate-wise exponentiation. The matrix \(T\) is constructed from this system by performing all possible applications of the set of points \(p_i\) \((p_k)\). The row \(i\) of the matrix is obtained by applying the set of terms to a point \(p\), and the exponents in the column \(j\) correspond to the term \(p_j\):

\[
T = \begin{bmatrix}
0^00^00^0 & 0^00^00^1 & 0^10^00^0 & 0^10^00^1 \\
0^00^10^0 & 0^00^10^1 & 0^10^10^0 & 0^10^10^1 \\
1^00^01^0 & 1^00^01^1 & 1^10^01^0 & 1^10^01^1 \\
1^01^00^0 & 1^01^00^1 & 1^11^00^0 & 1^11^00^1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1
\end{bmatrix}
\]
The inverse of this matrix is:

\[
T^{-1} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1
\end{bmatrix}
\]

and the vector of coefficients is obtained by multiplying \( T^{-1} \) by the vector of values:

\[
\mathbf{c} = T^{-1} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.
\]

**Sparse Solution:** The input representation used in the interpolation algorithm is a list consisting of couples \((p_i, f_i)\).\(^1\) The lexicographically sorted list of points, together with the corresponding values, is

\[
P = ((0, 0, 0), 1), ((0, 0, 1), 1), ((1, 0, 1), 1), ((1, 1, 0), 0).
\]

Since the matrix \( T \) is triangular, as anticipated by Theorem 3, the first coefficient obtained is equal to the function value at the first sorted point \( c_{000} = f_{000} = 1 \), while other coefficients can be obtained by repeated application of Equation 4.4: \( c_{001} = f_{001} - c_{000} = 0, c_{101} = f_{101} - c_{001} - c_{000} = 0 \) and \( c_{110} = f_{110} - c_{000} = 1 \). By using this equation, we avoid the explicit construction of \( T \) and \( T^{-1} \). The computation is performed instead by traversing the input list and adding the obtained terms to the output list. It is sufficient that the algorithm outputs only the list of terms \( m_i \) that are multiplied with nonzero coefficients \( c_i \). The two nonzero terms of the RM polynomial obtained in this way are

\[
\mathbf{c} = ((0, 0, 0), (1, 1, 0))
\]

Hence, the interpolating polynomial is: \( f = 1 + x_1x_2 \).

\(^1\)We will describe later how to use other reduced representations, such as cubes and BDDs, that are more common in manipulation of logic functions.
4.3 Interpolation over Larger Fields - Problem Decomposition

We now consider the interpolation over larger fields. We show that there is a generalization of the approach from the previous section that can be used for interpolation over fields other than $GF(2)$. By investigating the structure of the system $Te = f$, we show that the original problem can be decomposed into several smaller problems which can be then solved independently. While the decomposition can be used for any field, it is most effective for small finite fields, which interest us the most.

4.3.1 Partial Ordering of Interpolation Spaces

For dealing with the structure of the Vandermonde matrix $T$, we treat both points $p$ and terms $m$ as points of interpolation spaces. For $n$-variable functions, these are the sets of $n$-tuples over $GF(q)$. Each number in an $n$-tuple is a coordinate. We alternatively write $p^n$ instead of $m(p)$ to indicate that an application of a term to a point involves coordinate-wise exponentiation.

We say that two points $u$ and $v$ are equivalent, $u \approx v$, if they have 0s in the same coordinates. This relation is an equivalence relation over the interpolation space, and the equivalence classes are called $z$-subspaces. Since $z$-subspaces are distinguished only by coordinates which are 0, we denote them with expressions like $S_{p_0q_0}$ to indicate which coordinates are zero and which have only nonzero values.

A relation of partial order $\preceq$ is defined between $z$-subspaces. A $z$-subspace $S_1$ precedes or equals $S_2$ if the set of coordinates that are 0 in $S_2$ is a subset of those in $S_1$. The relation $\preceq$ can be computed coordinate-wise as:

$$S_1 \preceq S_2 \iff \forall i\ S_{1i} = 0 \lor S_{2i} \neq 0.$$ 

**Example 7** Points 10201, 20301 and 30102 belong to $z$-subspace $S_{102020z}$, and the point 10111 belongs to $S_{1011110z}$. Then:

$$S_{102020z} \preceq S_{1011110z}.$$ 

The order is partial because there exist incomparable $z$-subspaces, those whose zero coordinates form mutually non-inclusive sets. We define the incomparable $z$-subspaces
The z-subspaces are equal if the coordinates that are 0 are the same. Relations < and \( \preceq \) are defined using \( \preceq \) and equality in an intuitive way.

In Example 7, it is also true that \( S_{x0y0z} < S_{x0y0z} \) since these two z-subspaces are not equal.

Example 8 Points 1020 and 2010 belong to the z-subspace \( S_1 = S_{x0y0z} \). Points 1210 and 1012 belong to the z-subspaces \( S_2 = S_{x0y0z} \) and \( S_3 = S_{x0y0z} \), respectively. Since the sets of zero coordinates in both \( S_2 \) and \( S_3 \) are the subsets of zero coordinates in \( S_1 \), it follows that \( S_1 < S_2 \) and \( S_1 < S_3 \). Also, the latter two z-subspaces are not comparable, i.e. \( S_2 \sim S_3 \), because the coordinate \( S_{22} \neq 0 \) while \( S_{32} = 0 \) and \( S_{24} = 0 \) but \( S_{34} \neq 0 \).

These order relations can also be applied to individual points, because of the equivalence relation \( \approx \). The subspaces defined this way form a structure of a partially ordered set, or poset.

Example 9 Let \( n = 3 \), for any finite field \( GF(q) \). The z-subspaces are ordered as in the Hasse diagram shown in Figure 4.1. This diagram represents the Boolean algebra \( B_3 \). \( B_n \) is usually defined as a poset consisting of all subsets of an \( n \)-element set ordered by set inclusion; it is easy to see that the order between z-subspaces is related to this order.

Generally, for sparse functions, the poset representing the existing z-subspaces and the order between them can be any sub-poset of \( B_n \). The order between the z-subspaces can be obtained by checking the reachability in the diagram. The maximal number of incomparable subspaces corresponds to the width of the poset; subspaces belonging to the same level (sets of z-subspaces with the same number of 0 coordinates) are always incomparable. The length of the poset, defined as the length of the longest path, is clearly bounded by \( n \).

4.3.2 Structure of the System Matrix

Using z-subspaces, we can study the structure of the system matrix \( T \), and eventually characterize the selection of terms that can help in solving the system. The following statement holds:
Figure 4.1: All z-subspaces for functions of three variables

**Theorem 4** An element \( m(p) \) of a matrix \( T \) is:

1. zero if \( p < m \)
2. zero if \( m \) and \( p \) are not comparable
3. nonzero if \( p \geq m \)

**Proof:** The proof follows from the definition of relation \( < \). Case 1) Some coordinate of \( p \) is zero, while it is nonzero in \( m \), for which \( p^m \) results in a zero application \( p^m \). Case 2) \( m \) and \( p \) have zero coefficients such that in both \( m(p) \) and \( p(m) \) there is a term \( \epsilon(0) = 0^r \cdot \epsilon \neq 0 \).

Case 3) whenever a coordinate in \( p \) is 0, it is 0 in \( m \) as well, and \( 0^0 = 1 \).

The relations between the partial order and the application of polynomial terms to points can be summarized as:

\[
\begin{align*}
\text{\( u < v \Leftrightarrow u(v) \neq 0 \land v(u) = 0 \)}
\end{align*}
\]

\[
\begin{align*}
\text{\( u \asymp v \Leftrightarrow u(v) = 0 \land v(u) = 0 \)}
\end{align*}
\]

\[
\begin{align*}
\text{\( u \approx v \Leftrightarrow u(v) \neq 0 \land v(u) \neq 0 \)}
\end{align*}
\]

This characterization allows us to grasp the structure of the system matrix. Assuming for a moment that the points and terms coincide, the system matrix is block-triangular as in the following example.
Example 10 Let a sparse function be specified at $z$-subspaces $S_{00r}, S_{0rx}, S_{x0r}$ and $S_{rxx}$. Then, the matrix $T$ consists of block matrices, each of which contains applications of terms from one $z$-subspace to points in another $z$-subspace. The rows correspond to the points and the columns to the terms. These block matrices consist either of all zeros, or all nonzero elements, depending on the relative order between the point and term subspaces. The matrix in this example takes the form:

$$
T = \begin{bmatrix}
[0^00^0r] & 0 & 0 & 0 \\
[0^0r^0x] & [0^0x^0r] & 0 & 0 \\
[x^00^0r] & 0 & [r^00^0x] & 0 \\
[x^0x^0r] & [x^0x^0r] & [r^0x^0r] & [r^0x^0r]
\end{bmatrix}
$$

where each nonzero block matrix is represented by values that the point and term coordinates can take. Note that for each block, if there exist a coordinate in which 0 is raised to nonzero coefficient, a block matrix filled with zeros is obtained: informally, this corresponds to the rule $0^r = 0$.

While the matrix of the overall system is sparse, that is, it can have many 0s, the nonzero matrices for each subspace will have all nonzero entries. This decomposition of sparse matrices is perfect because no zero element is assigned to the residual dense matrices. Additionally, no new nonzero elements (fill-in) are created in the process, which is rarely the case in sparse matrix computations.

4.3.3 Structure of the Solution Space

We can now describe the solution space, i.e., the set of terms $m$ which is a generalization of the solution in the binary system. Instead of assigning the terms that coincide with points, the condition that has to be satisfied is that the $z$-subspaces of terms and points coincide.

**Theorem 5** Let the sparse interpolation point set consist of points in $z$-subspaces $S_1, S_2, \ldots, S_z$. Then, the selections of terms that lead to solutions in these subspaces considered in isolation constitute a solution to the complete problem.

**Proof:** By using the partial relation $<$ to order the $z$-subspaces, we obtain the bottom $z$-subspaces. We know that points $p_i$ in these subspaces have no nonzero matrix entries $m_j(p_i)$ with terms in any other subspaces, hence these subspaces can be treated independently.
Assuming that they can be solved, the number of terms in each z-subspace is equal to the number of points in it.

By solving these systems, the corresponding z-subspaces are eliminated from the system, and new bottom z-subspaces can be solved in the same way. The solution is found when all unknowns have been eliminated. □

Equivalent to this process is the inversion of block-triangular matrices. Another view is based on the geometric meaning of z-subspaces, which are the points, lines, planes, etc. in $n$-dimensional space. Figure 4.2 shows the execution order of the algorithm for the 3-dimensional case. The solution is first obtained for the z-subspace $S_{000}$ (the point at origin), if present. Next, the interpolations along lines $S_{x00}$, $S_{0x0}$ and $S_{00x}$ can be obtained, followed by planes $S_{0x0}$, $S_{x0x}$ and $S_{x00x}$. The last z-subspace to be solved in $S_{xxx}$. Since this order of execution is given by the poset of z-subspaces, we use that poset as a primary way to describe the interpolation by decomposition into z-subspaces.

From Theorem 5, it follows that it is sufficient to know how to solve the system over a z-subspace to solve the interpolation problem. We will later search for the assurance that the inversion of matrices in z-subspaces can be done, but assuming that this is possible, we can define the interpolation algorithm. We illustrate it first by inverting the matrix $T$ from...
Example 10, which we rewrite in the form:

\[
T = \begin{bmatrix}
A & 0 & 0 & 0 \\
E & B & 0 & 0 \\
F & 0 & C & 0 \\
G & H & I & D \\
\end{bmatrix}
\]

To invert this matrix, it is sufficient to invert the diagonal block matrices, \(A, B, C\) and \(D\). It can be verified that the inverse of \(T\) is:

\[
T^{-1} = \begin{bmatrix}
A^{-1} & 0 & 0 & 0 \\
-B^{-1}EA^{-1} & B^{-1} & 0 & 0 \\
-C^{-1}FA^{-1} & 0 & C^{-1} & 0 \\
-D^{-1}(G - HB^{-1}E - IC^{-1}F)A^{-1} & -D^{-1}HB^{-1} & -D^{-1}IC^{-1} & D^{-1} \\
\end{bmatrix}
\]

by multiplying \(T\) and \(T^{-1}\) in any order.

This inversion is used for solving the system \(Tc = f\) for an unknown vector \(c\). Although a closed-form solution can be obtained in the matrix form, we prefer an alternative which is both computationally efficient and has a simple description. For this, we present vectors \(c\) and \(f\) in the block forms, with elements of the block belonging to the same \(z\)-subspace. In our example, this results in four blocks. By multiplying

\[
T^{-1} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}
\]

we obtain blocks \(c_1, c_2, c_3\) and \(c_4\). The multiplication results in expressions that can be written as:

\[
c_1 = A^{-1}f_1
\]

\[
c_2 = B^{-1}f_2 - B^{-1}EA^{-1}f_1 = B^{-1}(f_2 - EA^{-1}f_1)
\]

\[
c_3 = C^{-1}f_3 - C^{-1}FA^{-1}f_1 = C^{-1}(f_3 - FA^{-1}f_1)
\]

\[
c_4 = D^{-1}\left(f_4 - (G - HB^{-1}E - IC^{-1}F)A^{-1}f_1 - HB^{-1}f_2 - IC^{-1}f_3\right)
\]

70
The expression $c_1 = A^{-1} f_1$ from the first equation can be substituted in other equations. Note that in $T$ each block matrix under the diagonal evaluates a polynomial from a subspace corresponding to its column at a subspace corresponding to its row. For $z$-subspace $S_k$, we denote with $f_{<k}$ the values obtained by evaluating polynomials from all $z$-subspaces lower than $S_k$ at points in $S_k$. By reusing the coefficients obtained in subspaces that are solved, and by adding all the evaluations, the last three equations reduce to:

$\begin{align*}
    c_2 &= B^{-1}(f_2 - Ec_1) = B^{-1}(f_2 - f_{<2}) \\
    c_3 &= C^{-1}(f_3 - Fc_1) = C^{-1}(f_3 - f_{<3}) \\
    c_4 &= D^{-1}(f_4 - Gc_1 - HB^{-1}(f_2 - Ec_1) - IC^{-1}(f_3 - Fc_1)) = D^{-1}(f_4 - Gc_1 - Hc_2 - IC^{-1}(f_3 - Fc_1)) = D^{-1}(f_4 - f_{<4})
\end{align*}$

Hence, for each $z$-subspace $S_k$, the values should be first updated by subtracting the values of the polynomial obtained in subspaces that are lower. Then, the inversion of each diagonal block matrix directly recovers coefficients for a given $z$-subspace.

An interpolation algorithm can be defined, based on the traversal of the poset of $z$-subspaces. After performing interpolation over each subspace, the algorithm evaluates the polynomial just obtained at all higher subspaces.

**Algorithm 6** Interpolation using $z$-subspaces

- Sort $z$-subspaces according to $<$
- For each $z$-subspace $S_i$ with points $P_i$ and values $f_i$ in increasing order
  - Select all polynomial terms $M_i$ and create $T_i = [P_i^{M_i}]$
  - Interpolate in $S_i$ to obtain a vector of coefficients
    
    \[ c_i = T_i^{-1} f_i \]

  - For all $S_j \succ S_i$, update vectors of values
    
    \[ f_j = f_j - [P_j^{M_i}] c_i \]

So far, we have not shown that we can always select the terms $M_i$, such that the inverse of $T_i$ exists: we deal with this issue later. The strength of this algorithm is that it allows
selecting the terms independently for each subspace and even changing them during the execution because the overall matrix $T$ is never constructed explicitly. All non-diagonal block matrices used in the derivation above need not be known in advance. The following three-valued example illustrates the proposed algorithm, including the decomposition into smaller problems in $z$-subspaces.

**Example 11** Consider the three-variable partial function given by the points in $(GF3)^3$:  

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The first two points belong to the $z$-subspaces $S_{z00}$ and $S_{z01}$, respectively, while the remaining three points belong to $S_{z22}$. The ordering relation $<$ between these subspaces allows us to solve the first two subsystems independently. This gives the following two terms: $c_{101} = 1$ and $c_{120} = 1$. With these two coefficients, the system can be updated for the remaining points. We adjust the function values for points in $S_{z22}$ by subtracting the value of the calculated term at these points. This results in the value vector $[1, 1, 2]$ after the first coefficient is calculated and $[0, 2, 0]$ after both coefficients have been taken into account. The final step consists of setting up and solving a system of equations for points in $S_{z22}$. The matrix $T_{z22}$ is constructed by calculating all possible values $p_j(p_i)$:

$$T_{z22} = \begin{bmatrix}
1 & 2 & 1 \\
2 & 1 & 1 \\
2 & 2 & 1 
\end{bmatrix}$$

whose inverse is equal to

$$T^{-1} = \begin{bmatrix}
2 & 0 & 1 \\
0 & 2 & 1 \\
2 & 2 & 0 
\end{bmatrix}$$
After multiplying this matrix with the vector of values $[0, 2, 0]$, the last three coefficients are $c_{121} = 0$, $c_{211} = 1$, and $c_{221} = 1$. We can check the validity of this result by evaluating the resulting polynomial $f = x_1 x_3 + x_1 x_2^2 + x_2^2 x_3 + x_1^2 x_3^2 x_1$ at all possible inputs. For example, the value at point $[1, 2, 1]$ is $f(1, 2, 1) = 1 \cdot 1 + 1 \cdot 2^2 + 1 \cdot 2 \cdot 1 + 1 \cdot 2^2 \cdot 1 = 1 + 4 + 2 + 4 = 11$.

Of practical importance is the construction of the relation $\prec$ and the construction of $z$-subspaces (classification of points). The ordering relation between any two points can be calculated by comparing all $n$ coordinates of these points in $O(n^t)$ time by computing the relation between each pair of points. A refinement is possible by separating the points into subsets with $i$ zeroes, for $i = 0, 1, 2, \ldots, n$, which correspond to the levels in the poset diagram. Since no $z$-subspace can span two such subsets, all subspaces will be classified by performing comparisons inside the subsets only. Similarly, the comparison using $\prec$ has to be performed only between the subspaces at different levels. Therefore, the practical data structure for this algorithm consists of the lists of subspaces at level $i$, $i \in \{0, 1, \ldots, n\}$; each individual subspace can be represented by a simple list of points. The overall computation in the algorithm will not be dominated by the operations over these lists.

The complexity of the algorithm is dominated by solving the system in each subspace, which requires $O(t^3)$ operations. The decomposition lowers the number of points that have to be considered at a time. In the worst case, all points will be in one $z$-subspace, but for any fixed distribution of points, the complexity can be lowered considerably. For example, the uniform distribution will produce the largest $z$-subspace of

$$\frac{(q - 1)^n}{q^n} \cdot t = (1 - \frac{1}{q})^n \cdot t$$

points, reducing the largest matrix inversion exponentially in $n$. This decomposition is obviously more beneficial for interpolations over smaller fields. We will show that the number of points in the largest subspaces can be lowered for any distribution of input points by using the fixed-polarity forms.

Before investigating in more detail the issue of solving the problem in a single $z$-subspace, we return briefly to the important case of binary functions.

73
4.4 Binary Case Revisited

The decomposition of the interpolation just given can be applied to the binary case, to characterize further the solutions for that problem. In this case, each z-subspace consists of a single point - consequently, the solution for each of them trivially exists.

In Section 4.2, a solution using the lexicographic order was presented. The solution obtained using posets and the partial order is slightly more general: lexicographic order is one possible linear order obtained from a poset. While previously we knew only that the matrix entries above the diagonal are 0, now we know which elements below the diagonal will be zero.

The algorithm from Theorem 3 can be restated in a more general way.

**Algorithm 7 GF2 Interpolation by Posets**

- Sort points according to $\prec$
- For all bottom points $p_\perp$
  
  \[ e_\perp = f_\perp \]

- For all other points $p_i$ in increasing $\prec$ order

  \[ e_i = f_i - \sum_{j=1}^{i-1} T_{ij} \cdot e_j = f_i - \sum_{j < i} p_j \cdot e_j = f_i - \sum_{j < i \prec p_i} e_j \tag{4.5} \]

From Algorithm 7, we obtain additional information about the structure of the matrix $T$. Only the points that are lower than the point $p_i$ according to $\prec$ contribute to the coefficient value at that point. This can be formalized as follows:

\[
\begin{align*}
  u \prec v & \Rightarrow u(v) = 1 \land v(u) = 0 \\
  u \preceq v & \Rightarrow u(v) = 0 \land v(u) = 0 \\
  u = v & \Rightarrow u(v) = 1 \land v(u) = 1
\end{align*}
\]

where $u$ and $v$ are the points in an interpolation space.

The actual computation, hence, takes form of the traversal of the poset diagram. It is sufficient to know the function values at each point in the poset, to calculate the coefficients.
of the polynomial at the corresponding points.

Example 12 Consider the function given by the poset diagram in Figure 4.3. The function values $f_p$ associated with each point $p$ are shown next to the node representing that point. Traversal of the poset produces the coefficients $c_p$ for each term (point) $p$ using Algorithm 7; the results are shown in the figure.

4.4.1 Using Other Reduced Representations

This algorithm can be adjusted to other reduced representations of logic functions. Assuming that the function is given by the cubes, there is no need to expand them to individual points. Efficient algorithms exist to compute the ordering relation $<$ between the cube $C$ and a point $p$, which return the set of points $c_p$ in the cube, such that $c_p < p$. The ordering relation computation dominates the overall computation. We note, however, that the output result will not be polynomial in the input representation, for the simple reason that each cube can represent exponentially many points.

A similar situation occurs when we use BDD representations, for both the input and the output of the algorithm. Using the BDD_DC representation from the previous chapter, the need for preprocessing, which would extract the information and form the list or poset, disappears. By traversing the input in lexicographic order, we can use the version of the algorithm presented in Section 4.2. Consequently, there is no additional memory requirement, because the BDD_DC can be used as a direct input to the algorithm. We know that the algorithm presented will output the FDD in size that is at most linear in the number of points, by Lemma 2 (Chapter 3).
Hence, the algorithm presented runs in time that is polynomial in the number of specified points and produces the output presentation (FDD, cubes, list of coefficients) bounded by the number of points given.

4.5 Conditions for Singularity and Selection of Terms

Section 4.3 showed how the interpolation problem can be decomposed into smaller problems and solved in principle. We now take a look at the yet unsolved part of the interpolation problem. Our main concern is that the systems of equations obtained in this way have a solution. We express the criterion for solvability of the system as a requirement that the matrix $T$ is not singular. To discuss the conditions for singularity of a system of equations, we use here the matrix notation, rather than the more abstract vector space notation. The matrix of a system $T$ can be singular as shown in the following simple example.

Example 13 Consider a two-variable $GF3$ function specified at points $p_1 = 12$ and $p_2 = 21$. If the terms coincide with the points, the matrix of the system is

$$
\begin{bmatrix}
1^{12} & 1^{21} \\
2^{12} & 2^{21}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 2 \\
2 & 1
\end{bmatrix}
$$

which is singular over $GF3$. This example illustrates the difficulties encountered when interpolating over small finite fields, as opposed to large finite and infinite fields. This matrix is nonsingular over real numbers, and some other finite fields, e.g. $GF5$.

Systems over two points in finite fields

We explain the difficulties with interpolation over finite fields by analyzing the simplest case, that of two points in a system. We assume that the terms correspond to the points. It is sufficient to concentrate on matrices with nonzero elements because they belong to the same $z$-subspace $S$, as shown in Theorem 5. For example, with two points, $p_1$ and $p_2$, we

\footnote{Although systems with singular matrices can be solvable, we dismiss this in our case because the algorithms should be independent of the values of the function.}
have the following system

\[
\begin{bmatrix}
   p_1(p_1) & p_2(p_1) \\
   p_1(p_2) & p_2(p_2)
\end{bmatrix}
\begin{bmatrix}
   p_1^{p_1} & p_2^{p_1} \\
   p_1^{p_2} & p_2^{p_2}
\end{bmatrix}
\]

with the determinant

\[p_1^{p_1}p_2^{p_2} - p_1^{p_2}p_2^{p_1}\]

This equation has many solutions for \(n\)-tuples over \(GF(q)\). We gave an example over \(GF(3)\), where \(p_1 = 12\) and \(p_2 = 21\). A class of such solutions can be constructed by noticing that the formal exponentiation operation \(p_1^{p_2} = p_1^{p_1^{p_2}} \cdots p_1^{p_n^{p_2}}\) can be used in the same way as ordinary exponentiation. Coefficients can be subtracted, leading to the following condition for singularity: \(p_1^{(p_1-p_2)} = p_2^{(p_1-p_2)}\), or

\[
\left(\frac{p_1}{p_2}\right)^{(p_1-p_2)} = 1.
\tag{4.6}
\]

There can be many solutions for this system over finite fields. Even in the extreme case when each point has one coordinate, i.e., in univariate case, such solutions exist over finite fields, while over real numbers there is no solution for \(a^{x-b} = b^{x-b}\), such that \(a \neq b\).

The case of \(GF(3)\) can be characterized further because it possesses an interesting combinatorial structure. In the case of \(2 \times 2\) matrices, we notice that:

- the diagonal is always 1 (since \(1^1 = 2^1 = 1\)).
- all symmetrical \(2 \times 2\) matrices with nonzero entries are singular over \(GF(3)\).

Consequently, it is sufficient that the entries \(p_1^{p_2}\) and \(p_2^{p_1}\), \(p_1 \neq p_2\), are equal to have a singular system when straightforward selection of terms is used. These cases can be constructed easily - the maximal construction involves the parity function among the coordinates, as in Example 14, in which 8 out of 16 points in \(z\)-subspace \(S_{xxxx}\) are included.

**Example 14** Consider a function given at points with an even number of coordinates that are 1: \(p_0 = 1111\), \(p_1 = 1122\), \(p_2 = 1212\), \(p_3 = 1221\), \(p_4 = 2112\), \(p_5 = 2121\), \(p_6 = 2211\), \(p_7 = 2222\). For any two points, if the terms are equal to the points, \(M = P\), we obtain a singular system. Furthermore, the system is singular when all the points are considered. Its rank is 4 because the rows \(p_j^M\) and \(p_j^{M'}\) obtained by applying all the terms to points \(p_j\) and \(p_{7-j}\)
respectively, are equal. Extended to the \( n \)-variable case, this provides a construction of a system with as many as \( t = 2^n / 2 \) points in which the rank of the matrix is as low as \( t/2 \).

**Decomposition to \( z \)-subspaces and singularity condition**

In the previous section, we showed that the problem can be decomposed if the \( z \)-subspaces of points and terms coincide. We mentioned that it can be solved "in principle" because no guarantee was provided that the linear system can be solved. We now describe the condition when such an approach fails.

**Theorem 6** A system \( Te = f \) is singular if and only if there are \( z \)-subspaces \( S_1, S_2, \ldots, S_z \), \( z > 0 \) where each \( z \)-subspace considered in isolation contains a singular system.

**Proof:** \(( \Rightarrow \) Suppose that the matrix is singular. This requires that some of the rows sum to zero (null). It is impossible for this to happen when we apply terms to points from incomparable subspaces. If such points are in \( z \)-subspaces \( S_1 \) and \( S_2 \), then the matrix \( T \) will be block diagonal and singular if at least one of the blocks is singular.

It is impossible that the \( z \)-subspaces that are comparable but not equal, \( S_1 \neq S_2 \), will null, while those over \( S_2 \) and \( S_1 \) alone will not. We could then solve the system for \( S_1 \) and obtain the system defined only over \( S_2 \) by subtracting the computed values.

This means that only points in the same \( z \)-subspace can create a singular system.

\(( \Leftarrow \) Conversely, if at least one subspace contains a singular system, then solvability of this subspace will not be improved if additional subspaces are taken into account. \( \Box \)

We are interested in always obtaining the solution, and this observation gives us a condition for singularity. It is sufficient to investigate the \( z \)-subspaces in isolation to eventually remove the singularity. To achieve this, we have the freedom in selecting the terms of the resulting polynomial.

**4.5.1 Selection of Terms**

So far, we have considered selecting the terms equal to points with no guarantee of finding a solution over finite fields. This selection might also lead to polynomials that have large exponents, which are expensive to implement. We now seek a better approach for the selection of terms. We would like to have as small exponents as possible while preserving or even improving the prospects of finding a solution.
Selection of terms will be based on the following observation. If we consider any projections of points, then the number of terms in these projections should correspond to the maximum number of points. Otherwise, if there are more points than terms in some projection, there may exist no solution. Figure 4.4 shows an example, in which a one-dimensional projection contains three points. Selecting the polynomial with two terms in $x_1$ might not be sufficient. Hence, the number of points in any projection defines the minimal degree of the polynomial in the variables determined by that projection.

This is still not a sufficient condition for the system to be solvable. Example 13 shows one case in which the points and terms have the same projections and the system is singular. We now propose algorithms for selection of terms, which decrease the cost of the polynomial and preserve the topological relation between the points. While straightforward selection can lead to a singular system, we show that under some conditions the new selection methods lead to a result.

**Bijective selection of terms**

If we select each coordinate of the term set by some bijective mapping from the coordinates in the point set, then the number of points and terms in each projection will be the same. The cost can be minimized by selecting the minimum degree terms, by assigning the lowest degrees to the most common point coordinates. The algorithm can be stated as follows:

**Algorithm 8 Bijective Term Selection**

- For each coordinate $i = 1 \ldots n$
  - $CS_i = \text{set of } i\text{-th coordinates of all points}$
  - Order the elements of $CS_i$ according to how often they appear
For each point \( p \) replace the coordinate \( p_i \) with its position in the ordered \( C S_i \).

While this produces an optimal bijective selection with respect to the total cost of the terms that might be used, the optimality cannot be guaranteed for the actual forms since it is not known which of these terms will be multiplied by nonzero coefficients. Such selection works well for large fields and point sets, as will be shown later, but for small fields, it can still often fail to produce a nonsingular system. Example 13 presents one such case.

**Greedy term selection**

For small fields and few points, another approach for selection of terms can lead to a nonsingular system, while producing even less costly terms. It does not preserve the projections: on the contrary, it tends to increase many of them.

Starting from the initial point \( 11\ldots1 \) in a \( z \)-subspace (0s are ignored since they will not change), a coefficient in only one direction that has changed is increased. The coordinate to increase can be chosen, for example, by lexicographically ordering the points. Intuitively, this corresponds to "greedy compaction" of the points, which lowers the coordinates of the chosen points each time there is an empty place where the point can be moved. The highest coordinate changed will increase by 1, while all lower ones will be set to one. This mapping can be done in linear time if the points are ordered:

**Algorithm 9 Greedy Term Selection**

- Order the points lexicographically
- For each point

  - the highest coordinate that has changed is increased by 1.

  - all of the lower order coefficients are set to 1 (zeros ignored).

Figure 4.5 shows an example of such term selection. The terms are "compacted", which is good for reducing the cost of the resulting form.

To show that this algorithm can actually improve the system (eliminate singularity) in the case of small finite fields, we note that the greedy selection of terms leads to nonsingular systems for examples of previous singular systems.
Example 15 Consider the system from Example 13. Greedy selection of points produces terms $m_1 = 11$ and $m_2 = 21$. This results in the nonsingular system

$$
\begin{bmatrix}
1^{12} & 1^{32} \\
2^{11} & 2^{12}
\end{bmatrix} =
\begin{bmatrix}
2 & 2 \\
2 & 1
\end{bmatrix}
$$

Greedy selection and nonsingular systems

The greedy selection of terms produces nonsingular system for any three points in any field, and for some configurations of $n$ points in any field. The proofs of these claims are given in Appendix B.

4.5.2 Comparison of Term Selection Schemes
Since it is impossible to analyze even the case with two points in full detail, we bring some empirical evidence on the usefulness of the term selection schemes. We compare their performance on randomly generated sparse functions over several smallest prime finite fields. Figures 4.6 and 4.7 give the percentage of systems for which these term selections fail to produce a nonsingular system, for large (> 10000) randomly generated sets of functions over $GF3$ and $GF7$. The x-coordinate on the graphs does not show the actual number of points. Since the scale is logarithmic, the highest power of the field size smaller than the actual number of points is shown. Since there is no apparent difference between the average performance of the identical and bijective term selection schemes, we conclude that the bijective term selection is the preferred method, because it produces a minimal-cost term selection.

The greedy selection works better for fewer points, while the bijective selection becomes better as the size of the field increases. The results for $GF5$ and $GF7$, not shown here, show the gradual transition between the two cases plotted. It is not surprising that for random functions the ratio of failures approaches $1/q$. Then, the assumption that the matrix is singular with probability of $1/q$ holds because the points are selected at random and the determinant will be zero with probability of $1/q$ [114]. We observed also that all three term selections produce the matrices of ranks close to the full rank. There is a result in [114], which for a similar class of random matrices shows a relatively high lower bound on their rank. It is possible that a similar guarantee can be provided in our case.

A more interesting question is how these selections behave in the worst case, and if there exist "pathological" configurations of points for which they will perform poorly. Example 14 shows a construction for which the rank is $t/2$ for identical term selection. Since randomization offers an excellent tool for avoiding the worst cases, we point to the use of randomization next.

### 4.5.3 Retry Strategies and Randomization

Since the proposed schemes for selection of terms can lead to a singular system, additional action must be taken to solve the system. The simplest scheme is use a randomized approach: if the selection fails to produce a nonsingular system, then we can either try another term selection scheme, another permutation of inputs for the greedy term selection, or a slight perturbation of selected terms. Perturbation of a (randomly chosen) term is : replace-
Figure 4.7: Initial failure rates for three selections of terms - GF11

Figure 4.8: Distributions of number of retries in the algorithm - GF3

ment of a coordinate in a term by a randomly chosen coordinate such that the resulting term is different from all existing terms.

We have conducted several experiments to compare the random retry strategies, and to verify the validity of our assumptions. For all cases in which the initial selection of terms led to a singular system, the number of retries was recorded using either a perturbation of terms or a random permutation of inputs in the greedy scheme. We used only the greedy and bijective term selections, because the identical mapping produces costlier terms than the bijective selection with the same failure rate.

Figure 4.8 shows the distribution of the number of retries that were needed for GF3 functions. Three methods were tried: greedy original term selection with random perturbation of terms, greedy term selection with random permutations, and bijective selection with random perturbations. For most functions that are singular after the first attempt, the number of permutations that have to be tried is small. The permutation approach.
4.6 Deterministic Interpolation Algorithm

The matrix of the system is a multivariate generalized Vandermonde matrix over finite fields which cannot be characterized, unlike the case of univariate interpolation over real numbers. It is not known how to select the terms to make this matrix nonsingular in advance. Instead, we start with a matrix that can be singular and in subsequent steps increase its rank until the matrix becomes nonsingular. Then, the inverse exists and the interpolation problem for a z-subspace can be solved by inverting this matrix. The higher the rank of the starting matrix is, the fewer such replacements are needed; in the worst case, $O(t)$ such replacements are sufficient.
4.6.1 Nullspaces and Increasing the Rank

We now present an algorithm for replacing a term in the interpolation polynomial, which deterministically increases the rank of the matrix $T$ by one.

For this, we study the nullspace of the matrix. When the rank is not full, then there exists a linear combination of columns (or rows) which is zero. Concentrating on rows, for each row $i$, the following condition must hold

$$c_1p_i^{m_i} + c_2p_i^{m_2} + \ldots + c_mp_i^{m_r} = 0$$

We say that the vector $c = c_1c_2\ldots c_r$ belongs to the column nullspace $C_{null}$ of matrix $T$.\(^3\)

All such vectors are linearly independent, as they form a basis for the nullspace, whose dimension is called nullity. For each $m \times k$ matrix, the relation between the rank $\rho$ and the nullity $\nu$ is $\rho - \nu = k$.

Hence, each column nullspace vector $c$ satisfies the following equation

$$\sum_{i=1}^{t} c_i p_i^{m_i} = 0$$

(4.7)

for all rows $i, i = 1, \ldots, t$. Alternatively, since the row and column ranks are the same, we can consider the nullspace of the transpose of the matrix $T$. The vectors in this nullspace satisfy the condition

$$\sum_{i=1}^{t} r_i p_i^{m_i} = 0$$

(4.8)

for $j = 1, 2, \ldots, t$. Again, all such vectors $r$ form the basis of the row nullspace, $R_{null}$.

For our purposes, we can freely choose only the terms of a polynomial, and considering the row nullspace will help us select the terms. For a fixed row nullspace vector $r$, it is sufficient to find a term $m_r$ for which \(\sum_{i=1}^{t} r_i p_i^{m_r} \neq 0\) to remove that vector from the nullspace. Such a term must exist as proven next.

**Theorem 7** Consider a matrix $T$ given over a $z$-subspace $Z$ with $t$ points in an $n$-variate interpolation problem. For any vector $r$ in row nullspace, there exists a term $m_r$ for which

\(^{3}\)Equation 4.7 also implies that there exists a polynomial which equals 0 at given points, hence vector $c$ consists of the coefficients of such polynomial.
r does not annul:

\[ \sum_{i=1}^{t} r_i p_i^{m_r} \neq 0 \]  \hspace{1cm} (4.9)

**Proof:** Consider a rectangular matrix that has \( t \) rows and \((q - 1)^r\) columns, consisting of applying all possible terms in a given \( z \)-subspace \( Z \) to the points given. We show that such matrix has rank \( t \), and consequently that there exists a term \( m_r \) with the property 4.9. To show this, we first consider a matrix with elements \( p_i^{m_r} \) obtained by applying all \( q^r \) terms, not only of those in \( Z \), to the given \( t \) points. Starting with the total function specified on all \( q^r \) points, we know that the columns of the matrix are linearly independent. Restricting rows to the considered points then reduces its rank to \( t \). Removing all the terms from outside \( Z \) will not decrease its rank. First, the terms from all \( z \)-subspaces which are incomparable or \( > \) relative to \( Z \) will result in a zero column. Second, the terms which are \( < \) than \( Z \) have 0 coordinates where points in \( Z \) do not. Each such coordinate \( a^0 \) is equal to \( a^{t-1} \) in \( Z \). and consequently, each such term can be replaced by a term from \( Z \). Hence, there must exist a term \( m_r \) from \( Z \) which does not annul \( r \). \( \Box \)

The algorithm for interpolation relies on the following theorem that describes a replacement step.

**Theorem 8** The rank of a system increases by 1 if a column that does not annul the row vector \( r \) replaces a column corresponding to a nonzero coefficient \( c_r \) in a column nullspace vector \( c \).

**Proof:** By adding a term \( m_{t+1} \) for which \( \sum_{i=1}^{t} r_i p_i^{m_{t+1}} \neq 0 \), we eliminate the row nullspace vector \( r \). Then

\[ \sum_{i=1}^{t} r_i p_i^{m_i} = 0, j \neq t + 1 \]  \hspace{1cm} (4.10)

and

\[ \sum_{i=1}^{t} r_i p_i^{m_{t+1}} \neq 0. \]  \hspace{1cm} (4.11)

Now we show that we can remove the column corresponding to a nonzero coordinate \( c_v \) in the column nullspace vector \( c \). Then, we obtain the nonzero linear combination:

\[ \sum_{j=1, j \neq v}^{t} c_j p_j^{m_j} = -c_v p_v^{m_v} \]  \hspace{1cm} (4.12)

We now claim that replacing the term \( m_v \) with \( m_{t+1} \) cannot possibly create a new nullspace
vector \( c' \). For this to be true, \( c'_{t+1} \) must be nonzero; otherwise, the sum from Equation 4.12 is nonzero. Then, \( c' \) is a nullspace vector if

\[
\sum_{j=1, j \neq v}^{t+1} c'_j p_i^m = 0
\]

for each \( i \). Multiplying each sum with \( r_i \) and adding them together also results in 0. This sum can be written as:

\[
\sum_{i=1}^{t} r_i \sum_{j=1, j \neq v}^{t+1} c'_j p_i^m = \sum_{j=1, j \neq v}^{t+1} c'_j \sum_{i=1}^{t} r_i p_i^m = \sum_{j=1, j \neq v}^{t+1} c'_j \sum_{i=1}^{t} r_i p_i^m + c'_{t+1} \sum_{i=1}^{t} r_i p_i^{m_{t+1}}.
\]

Using Equations 4.10 and 4.11 this expression reduces to

\[
c'_{t+1} \sum_{i=1}^{t} r_i p_i^{m_{t+1}} \neq 0
\]

because \( c'_{t+1} \neq 0 \); this contradicts the assumption that \( c' \) is a null vector. Hence, replacing the column \( v \) by the column \( t+1 \) eliminates one nullspace vector. This replacement increases the rank only by 1, because we can always choose a basis of \( C_{null} \) which has only one nonzero coordinate \( c_v \), among all vectors \( c \) in the basis. Then, all the other base vectors will remain in \( C_{null} \) after this replacement step. \( \square \)

We demonstrate the replacement step in Example 16.

**Example 16** Let the list of points and values in \( GF(3) \) be

\[
(((1.0.0.1.0.0.2.1).2).((2.0.0.1.0.0.2.2).0)).
\]

Using the identical term selection, the matrix \( T \) becomes of

\[
T = \begin{bmatrix}
1 & 1 \\
1 & 1
\end{bmatrix}
\]

Both row and column nullspaces consist of a single vector \([2 \ 1]\). The next step consists of selecting a new term which is not orthogonal for the row nullspace. By selecting:

\[
m_3 = (1.0.0.1.0.0.2.2)
\]
and replacing the first term with this term, we obtain the matrix

\[ T = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \]

which can be inverted since its nullspace is empty.

This replacement step can be used in a deterministic polynomial interpolation algorithm. There can be at most \( t \) replacement steps. In each step the nullspace vectors can be obtained in \( O(t^3) \) time by, for example, Knuth's or Berlekamp's algorithm [54]. However, it is not apparent that searching for the replacement term can be done in polynomial time, because there are \((q - 1)^n\) possible terms to search from.

### 4.6.2 Searching for Replacement Terms Efficiently

We now show that the replacement term can be found in \( O(nt^2) \) time. We first deal with the case in which there is only one vector in the nullspace.

**Theorem 9** Let a row nullspace of matrix \( T \) be spanned by one vector \( r \). There exists a term \( m \) for which changing a coordinate \( m_d \) by 1 results in a column that does not anull the vector \( r \). It is sufficient that \( p_{i \cdot d}, i = 1, 2, \ldots, t \) are not constant for nonzero coordinates of \( r \).

**Proof:** For the selected term \( m = m_1 m_2 \ldots m_n \), we know that

\[ \sum_{i=1}^{t} r_i p_i^m = 0 \]

There must exist a coordinate \( d \) for which \( p_{i \cdot d} \) is not constant for all the points, because otherwise all points would be the same. By increasing the coordinate \( m_d \) by 1, the following expression must hold

\[ \sum_{i=1}^{t} r_i p_{i \cdot d} p_i^m = 0 \]

Since coordinates \( p_{i \cdot d} \) are not constant for all \( i \)'s, this expression will be true for all the terms only if a vector with coordinates \( r'_t = r_t p_{i \cdot d} \) is a null vector. However, the dimension of the nullspace would then be 2, which contradicts our assumption. Thus, there must exist a term for which changing a coordinate \( m_d \) produces a column which does not anull \( r \). \( \square \)
In the case when nullity $\nu$ is greater than 1, it is again sufficient to consider only one vector in the nullspace. For that, we use a simple fact about the basis vectors in the nullspace. The nonzero coordinates in such vectors correspond to the rows which annul in a linear combination. For any such basis vector $r$, we always ensure that none of its nonzero coordinates $c_r$ can be removed such that the vector is still in the nullspace.

**Theorem 10** Let nullity of row nullspace of matrix $T$ be $\nu > 1$. Changing a coordinate in an existing term, as in Theorem 9, results in a column that does not annull a vector $r$ in the nullspace.

**Proof:** We consider a row nullspace vector $r$ whose nonzero coordinates cannot be removed. We select a coordinate $d$ as before, and consider increasing the corresponding coordinate in all terms. In this way, in addition to the vector $r$ with coordinates $r_d$, the vector $r'$ with coordinates $r'_d = r_d + \epsilon$ must be in the nullspace if the theorem is not true. Since any linear combination of $r$ and $r'$ must exist in the nullspace, and since these two are not collinear, we can always create from these two vectors a vector $r''$ whose nonzero coordinates are a subset of those in $r$. However, $r''$ cannot be a nullvector since the vector $r$ is chosen such that none of its zero coordinates can be removed.

This proof guarantees that the replacement term will be found among $O(t)$ alternatives, which is small relative to searching through all possible $(q - 1)^2$ terms.

**Theorem 11** For a vector $r$ in the nullspace, the replacement term can be found in $O(n^2)$ time.

**Proof:** We have to find a coordinate $d$ for which the points $p_d$ are not constant whenever $r_d$ is nonzero. There are at most $n$ coordinates, and $t$ checks to be made. The next $O(t)$ checks are needed to find an existing term to increase. In each check, we perform $tn$ operations.

To obtain a replacement term, we increased the coordinate $m_{i,d}$ of an existing term $m_i$ by one. Technically, to keep the new term in the same nullspace, this increase must be calculated as $m_{i,d} = (m_{i,d}) mod(q-1) + 1$. The interpolation algorithm in a $z$-subspace can be stated as follows.

**Algorithm 10** Interpolation in $z$-subspace

- Find initial terms $M$ (using e.g. bijective term selection)
• Create initial matrix $T$ by calculating

$$P^M = p_i^{m}, i = 1, \ldots, l, j = 1, \ldots, t$$

• If $\text{det}(T) \neq 0$ return terms $M$ and coefficients $T^{-1} \ast f$, else $\nu = \text{nullity}(T)$

• While $\nu > 0$

  - Obtain nullspace vectors $R = r_1, r_2, \ldots, r_\nu, C = c_1, c_2, \ldots, c_\nu$ for which

    $$c_i^T T = T c_i = 0, i = 1, 2, \ldots, \nu$$

  - Find $d$ such that $p_{i,4}$ is not constant for nonzero coordinates $r_1$;

  - Find term $m_j$ for which

    $$\sum_{i=1}^{t} r_i p_{i,4} * p_i^{m_j} \neq 0$$

  - Create $m_{j+1}$ from $m_j$ by adding 1 to coordinate $m_{j,4}$

  - Replace column $\nu$ corresponding to nonzero coordinate $c_1$ with column $P_{m_{j+1}}$

• Return terms $M$ and coefficients $T^{-1} \ast f$

Assuming that $t > n$, we can calculate the number of operations needed for the interpolation algorithm.

**Theorem 12** The multivariate interpolation algorithm for $t$ points in a $z$-subspace produces an interpolation polynomial with $t$ terms in $O(t^4)$ time.

**Proof:** Since the While loop can be executed at most $\nu = O(t)$ times, and the operations inside the loop are dominated by computing the nullspace, there will be at most $O(t^4)$ field operations executed. $\square$

### 4.6.3 Algorithm Implementation and Performance

The interpolation can be executed in several ways. It is easy to show that the algorithm would work correctly if the decomposition to $z$-subspaces is not performed at the beginning. Furthermore, instead of selecting a set of terms in advance and replacing some of them afterwards, the algorithm can instead start with no terms, and in each step add one term.
Figure 4.10: Number of replacement steps compared to random retries - GF3

until there are \( t \) terms. Applied to the univariate case, this algorithm would then select the \( t \) lowest-degree terms. Comparing this algorithm to the term selection schemes from Section 4.5.1, it is apparent that the greedy term selection attempts to approximate this process of obtaining the term set. Namely, by increasing the term coordinate corresponding to the point coordinate that has changed, the greedy term selection ensures that the terms that are selected consecutively are linearly independent. It does not guarantee that each new term will be independent relative to all previously selected terms, because such a guarantee must involve checking if there is a linear combination of columns that becomes zero. In the interpolation algorithm, this is performed by computing the nullspace vectors.

This algorithm has been implemented and successfully executed for more than 10000 randomly generated functions. We used the bijective term selection as a primary term selection. Ranks of matrices obtained in this way were high. For example, for over 80% of GF3 cases that initially failed, one term replacement was sufficient, while for the next 13% two replacements provided a nonsingular system. Figure 4.10 compares the number of term replacements (Exact) with the random retries discussed previously for GF3 functions. The conclusion from this graph is that the probabilistic scheme can also be attractive.

Since there are no other methods to compare against, we compared the cost of the resulting forms against the maximal cost of the terms, assuming that the terms correspond to the points. On average, the cost was about 50% of the maximum. For random functions, there is \( 1/q = 1/3 \) coefficients that are zero, which reduce the cost by \( 1/3 \). Further, bijective mapping reduces further the cost of the form. The fact that there were few term replacements ensures that the maximal cost does not deviate much from the cost of the
bijectionally selected terms.

4.7 Extensions

We present three immediate extensions of the proposed algorithm that can be used in various applications. The concept of fixed polarity RM forms can be applied as a way to use the polynomials in variables that are shifted for a constant. An incremental version of the algorithm can be useful in logic synthesis and in learning algorithms. A parallel version of the algorithm is interesting because the derivation of parallel algorithms is a challenging problem on its own, independent of the underlying computational model. This interpolation can be very practical for many parallel machine models.

4.7.1 Interpolation and Fixed Polarity RM Forms

It is possible to decrease the complexity of the interpolation polynomial by using fixed polarity RM forms, or *shifted polynomials*. These are the polynomials in variables \( x_1 + a_1 \cdot x_2 + a_2 \ldots x_n + a_n \). The cost of the resulting form, which takes into account the cost of each term, can be reduced by minimizing the number of coordinates that are nonzero in the terms set. By selecting additive constants \( a_i \) such that there is a maximal number of points whose \( i \)-th coordinate is \( a_i \), we maximize the number of zeros in that coordinate. This forces the solution terms \( m \), that belong to the same \( z \)-subspaces as the set \( p \), to have the maximal number of zeros in each coordinate. A small amount of preprocessing to determine the most common point in each coordinate and shift the polynomial can be performed in time linear in the number of points and quadratic in the number of dimensions:

**Algorithm 11 Polarity Selection**

- For each variable \( i = 1 \ldots n \)
  - Find the most common coordinate \( p_{shift} \) among \( p_j \), for all points \( p_j \), \( j = 1 \ldots t \)
  - Polarity in variable \( i = p_{shift} \)

This approach has an advantage of doing both don't care and polarity simplification of RM forms. For binary case, these shifts are optimal, since this algorithm finds the minimal “total height” of a poset [112], defined as a sum of all nonzero coordinates, or heights of all
points. An example of obtaining these forms is shown in Figure 4.11. Starting with a poset representing the points at which the function is specified, the optimal shift is performed. The resulting poset, which represents all the terms used is of minimal height.

The optimality of such shift cannot be guaranteed for the case of finite fields other than $GF(2)$. Although by maximizing the number of coefficients that are zero, the size of the largest $z$-subspace can be reduced, it is possible to construct the case in which the largest subspace is of size $(q - 1)/qt$ after such shift, while the optimal shift would produce the subspace with $1/qt$ points.

4.7.2 Incremental Interpolation and Learning

Decomposition of the interpolation problem can be used to derive an efficient *incremental* (on-line) algorithm. This can be useful in many situations in logic synthesis. For example while assigning values to some don't cares because of the environment in which the circuit is embedded [22]. Other applications, like machine or computational learning can be derived from the algorithm [112].

A simple learning model which the algorithm readily supports is the one in which the polynomial is computed for a set of points observed so far (either in the training phase, or at any later moment). For each new point encountered, the interpolation polynomial has to be recomputed (update phase) only when a mismatch is detected between the polynomial and the actual function value. Knowledge of the partial order among the $z$-subspaces allows us to update only the necessary part of the representation. Figure 4.12 illustrates the execution.
Figure 4.12: Incremental interpolation step

of the incremental step in the poset. For a new point \( p \), the set of subspaces that are lower is used to compute the polynomial coefficients. After the new coefficients of the polynomial have been computed, the solution in \( z \)-subspaces higher than \( p \) has to be updated.

This learning model resembles the learning by membership and equivalence queries [84], except for the essential difference that with that model the program can choose the interpolation points, which is the opposite to the assumptions we made throughout this dissertation. While the algorithm [84] works only for \( GF(2) \), several other algorithms for learning polynomials have been been presented recently [20], [9]. The algorithm from [20] exploits the existence of interpolation over large fields, such as the algorithm by Zippel.

**Adversaries and random shifts**

The efficacy of learning, as presented here, depends on the order in which the points are given, and we now demonstrate the power of randomization in dealing with this problem.

There are several metrics for assessing the efficiency of learning algorithms. It is important to know in advance how well the algorithm generalizes, that is, for a given output hypothesis, how good is the match with the samples that will appear in future. Another measure is the number of expected operations for each update step expected in the future. It is apparent that if the points in the \( z \)-subspaces that are low in the partial order are introduced more often at the end, then the amount of updating is larger.

In our case, with the underlying assumption that the points are given in the order dictated by the environment, the algorithm just presented may perform very poorly. If we assume that the environment is adversarial, then it can present the examples in an order which suits us the least, even though we might not know what order that is. It is a common
practice to use this adversarial model to describe such situations, and assuming that the adversary is all-powerful is not uncommon.

Randomization offers us an elegant solution to the problem just outlined. Using the same incremental interpolation, we can make the partial order between interpolation points random. A simple solution is to shift all the points randomly; that is, interpolate in variables $x_1 + r_1, x_2 + r_2, \ldots, x_n + r_n$ where $r_1, r_2, \ldots, r_n$ are randomly selected. This is sufficient to reduce the worst case input to the average case. A possible shortfall of this scheme is in dealing with adversaries that can use the information about the time needed for the update phase. Such adversary model is not unrealistic because some recent code-breaking algorithms use exactly the same knowledge. For dealing with such an adversary, it is feasible to change randomly the shifts at random instances of time. Recomputing the relation takes approximately constant time for each new shift, so no information is disclosed while performing these.

The learning model in [84] addresses these problems in another way. With that model, the interpolation points are selected by the program during the update phase. It was shown that it is sufficient to select these points such that the set of all points forms a meet-semilattice. Meet-semilattice is a poset in which for each couple of points, their meet is present. In our case, meet operation is computed by bitwise logical AND operation on binary representation of the poset elements.

### 4.7.3 Incremental Interpolation and Optimal Forms

Incremental interpolation can be used in heuristics for the minimal RM representations, primarily for Boolean functions. Producing the sparse transform from $t$ points requires $O(t^2)$ operations, with the resulting form having at most $t$ nonzero terms. If the goal is to reduce the cost of the form even more, then this incremental algorithm provides an inexpensive way to achieve this goal.

Since the incremental step of the transform can be done in $O(t)$ time, the heuristics, such as that of Varma and Trachtenberg, can be used to augment the interpolation approach. Starting with the don't cares that are lowest according to $<$, the function value at each point can be obtained by traversing the poset up to that point and adding the point to the poset, as in the incremental algorithm. Once the function value is computed, it can be inverted and for all higher points according to $<$, the polynomial and its cost can be
recomputed in linear time. The lower cost form should be kept, and the procedure can continue until the cost is low enough. Notice that by assigning the value to a don't care we can increase the number of terms required by one; hence, when this assignment does not improve the cost, it should not be used.

It is apparent that by considering the points in $\prec$ order, we are first considering the points that influence the form the most. Furthermore, a bound is provided on the influence of each point, which allows us to estimate the possible improvement by changing the value of each don't care. The point at level $i$ in the poset can change at most $2^{(n-i)}$ coefficients in general. Having the poset already built, we can provide a more accurate estimate. The same estimate can be used to provide more information about when to stop trying to find the improvement. Clearly, a small set of don't cares that are low in $\prec$ order offers the most savings.

These insights allow us to define heuristics that are better than the VT heuristic, requiring less computational resources. For example, while VT heuristic considers each point only once, with the poset built we can afford to perform a more detailed search among the don't cares at the same level in the poset, at least for the most influential don't cares.

### 4.7.4 Parallel Interpolation

The decomposition of the problem leads to a parallel interpolation algorithm, as discussed in [107]. Since the dependencies are defined with the partial order $\prec$ on $z$-subspaces, all incomparable subspaces can be solved in parallel. After the interpolation problem is solved in a $z$-subspace, the system can be updated for all the remaining $z$-subspaces in parallel.

The decomposition into $z$-subspaces does not automatically lead to a $NC^1$ class algorithm for Parallel RAM (PRAM) machines. Although the major steps undertaken can be performed by a $NC^1$ class algorithm, the length of the poset is $O(n)$, which implies that $n$ sequential steps have to be taken. In each step the incomparable $z$-subspaces are solved in parallel and values are updated for the successor subspaces.

In practice, it is important that all available processors can be used and that the communication and synchronization overheads are not dominant at execution time. To achieve this, the scheduling and processor allocation algorithms must be provided with information about the algorithm executed. The proposed algorithm, unlike most such "irregular" algorithms in practice, provides much of this information.
Let tasks be parts of the algorithm performed over a z-subspace: each task can be performed either by a parallel or sequential algorithm, whichever is faster for a particular z-subspace. With the proposed algorithm, the dependencies among the tasks are given by the same relation \( \prec \) used in obtaining the z-subspaces. The parallelism among the tasks, which is the maximal number of independent tasks, is equal to the width of the poset. Also, the processor scheduler may be provided with the exact execution times of each subtask. The only tasks that exhibit complex interdependence patterns are the function updates in the substitution phase, which are deterministic and of known duration. Figure 4.13 shows how such an algorithm is executed. The poset of dependencies among the tasks is given by the same poset, plus the dependence on the inversions, which can be done in parallel for all z-subspaces. The radii of circles and balls in the diagram symbolize the number of points in each subspace. Circles denote quadratic time in the size of inputs and the balls denote the time that is probabilistic cubic or worse. Randomized interpolation can be useful for the parallel model. Instead of doing deterministic changes of the matrix in serial, the existence of several processing units can be used to spawn independent retries in parallel.

### 4.7.5 Bandwidth Requirements

One additional feature of the algorithm is that the communication bandwidth required for the program execution is small. This is an important feature for any machine model that
takes into account the communication limits. Any of the subproblems can be communicated in time linear in \( nt \). For each matrix in a \( Z \)-subspace \( S_i \), it is sufficient to communicate points and terms to recreate the matrix in its full size (\( O(nt^2) \)). These matrices allow efficient encoding, like other structured matrices [73]. This is used in both inversion and substitution phases of the algorithm. In a situation when inversion fails due to the singularity of the matrix, a new task can be spawned to one or more processors by sending \( O(nt) \) field elements.

If broadcast operation is available, then the amount of information transmitted during the substitution phase is \( O(nt^2) \). For each subspace \( S_i \), the terms of the interpolated polynomial from the subspaces \( S_k \leq S_i \) are broadcast to the subspaces \( S_j > S_i \).

If broadcast is not available, then the worst case occurs when both the \( Z \)-subspaces and the number of \( Z \)-subspaces are \( O(t) \) and when the largest size \( Z \)-subspace is the lowest in \( < \) order. Assuming that the lowest \( Z \)-subspace is \( O(t) \), while all other \( Z \)-subspaces are in a chain and of size \( O(1) \), a total of \( O(t) \) terms of size \( n \) will be sent \( O(t) \) times to \( O(t) \) successors each. To reduce the actual number of messages sent, the terms in the interpolation polynomial can be merged for each \( Z \)-subspace from the terms obtained in the lower order \( Z \)-subspaces. It is then sufficient to transmit the update messages only to the immediate successors and to wait for updates only from the immediate predecessors. In the case described above, the amount of information transferred using this approach is \( O(nt^2) \) because the messages of size \( O(nt) \) will be sent \( O(t) \) times.

The knowledge of the pattern and amount of information to be transferred between the subtasks is perfect; this information can be used for the allocation of processors for a fixed communication network topology.

### 4.8 Concluding Remarks

In this chapter we presented new results for sparse interpolation over finite fields, which can be used in Reed-Muller transform for binary and MVL functions and related forms. Algorithms for interpolation exploit the decomposition of the problem into subspaces. The heuristic methods for reducing the failure rate of the algorithm are presented and empirically evaluated for random functions over several small prime fields. A deterministic algorithm is given, which increases the rank of the matrix by a series of substitutions.
An extension for fixed polarity RM forms is defined, which minimizes the number of nonzero coordinates in the selected terms of the polynomial, while also reducing the size of the largest subproblems and speeding up the algorithm.

An incremental algorithm is considered, which can be used for machine learning, logic synthesis or to improve the known heuristics for don't care assignment. Parallel algorithms are also derived from our characterization of the solution set.
Chapter 5

Universal Logic Modules and Programmable Logic

5.1 Introduction

The first commercially available Field-Programmable Gate Arrays (FPGAs) had an array of 3-input logic blocks, where each block could realize any function of three variables using an 8-bit RAM. Such a block is a lookup-table with 3 inputs (LUT:3). A decade or two before that, there was a significant amount of theoretical research on Universal Logic Modules (ULMs), which are logic blocks capable of realizing all functions of a fixed number of variables assuming that permutations and negations of variables are provided outside these blocks. Some of the established FPGA families from Actel, Xilinx and Pilkington, use blocks derived from such a concept of ULMs. However, more systematic research on the use of ULM circuits as logic blocks in FPGAs appeared only recently [37], [93]. In this chapter, we propose a new type of ULMs for use in SRAM-based FPGAs. Practical designs for 3- and 4-input LUT (LUT:3 and LUT:4) replacements are presented together with the methodology to systematically derive such blocks.

ULMs are traditionally defined as blocks with \( m \) general purpose inputs that can realize any function of up to \( n \) inputs, \( n < m \), under the assumption that permutations and negations of signals are generated cost-free outside the logic block [88]. While the inversions are not free, as will be shown in Section 5.5, the permutations of inputs are free for some routing architectures: Altera 8k and 10k series [4] allow that all the possible input signals.
can be connected to all the input pins of a logic block. Such architectures with fully connected inputs are considered in [11] exclusively for routing in hierarchical FPGAs [1]. The ULM blocks achieve their functionality by bridging some inputs and/or assigning them to a constant; these are assumed to be costless operations. This concept is illustrated in Figure 5.1a. Classical ULM research was based on this definition of ULMs. Lower and upper bounds are known for \( m \) as a function of \( n \), and they asymptotically approach each other. To realize all \( n \)-input functions, the total number of inputs \( m \) needed is on the order of \( 2^n / \log(n) \). Several methods have been proposed for constructing such ULMs [23],[69],[88].

Recent research on ULMs has been focused on investigating the tradeoff between the functionality of logic blocks and their usefulness in real applications. The research presented in [57] and [93] attempted to find a subset of functions that a ULM can realize so that it behaves as close as possible to the LUT. These papers deal with blocks that have functionality comparable to LUT.3 [57] and LUT.4 [93], but they are not functionally complete. In this chapter, we propose a more practical type of ULMs. It is known that adding pins to logic blocks in realistic FPGAs is very costly [74]. Since standard ULMs require a total of \( m = O(2^n / \log(n)) \) inputs for realization of an \( n \) input function, an unreasonable amount of routing resources may be needed if such blocks are used. In addition to providing the access to all \( m \) input pins, the routing network must provide resources for bridging the input

\[ B < 2^n \]

Figure 5.1: ULM alternatives.

The block in [57] is named "semi-ULM" to express the fact that it is not functionally complete.

It was noticed early [88] that the standard ULMs are not very practical because of the large number of input pins needed.
pins. There are $O(m^2)$ bridging connections possible for each block. These are the reasons why in [57] the total number of inputs is limited to 4, as opposed to 8 as in [93]. We propose a class of ULM circuits that avoids this problem and limits the number of input pins to $n$ by using separate programming bits. Like FPGAs, these ULMs are programmed by serial input to perform a particular function. As in classical ULMs, the functions obtained by permuting inputs and negating inputs (and possibly outputs) are considered to be equivalent. Such ULMs can serve as LUT replacements that require fewer programming bits. Figure 5.1 illustrates the difference between these two approaches for a ULM that can realize all $n$-variable functions.

Since the programming bits are loaded serially in SRAM-based FPGAs, there are no additional inputs required, other than the usual function inputs, which would compete for valuable routing resources. When compared with standard LUTs, our blocks need less time and storage area to reconfigure. This is especially important for emerging architectures in which reconfigurability of FPGAs is essential [6],[46],[92]. The logic block presented in [92] contains memory storage for 4 "contexts", which are the programs for the logic block that can be used interchangeably. In this case, any saving in the number of programming bits is multiplied by 4.

We therefore aim to reduce the length of the description of Boolean functions and to develop logic blocks which can use the reduced descriptions as their programs. We show that for large blocks it is impossible to obtain a significantly more succinct representation than the one used in a RAM-based LUT. However, for smaller (but practically useful) blocks, savings achieved in the number of programming bits needed can be significant, and we explicitly construct ULMs that reach the theoretical minimum.

The rest of the chapter is organized as follows. Section 5.2 gives an overview of the method used, together with the bounds on the number of bits required in the general case. Sections 5.3 and 5.4 provide the explicit designs of ULM.3 and ULM.4 using the proposed methodology. Section 5.5 gives insight into practical issues related to the use of such ULMs, as well as some extensions of the model.

102
5.2 Realization of ULMs

We now describe a procedure for obtaining a class of ULMs with the functionality comparable to LUTs. We exploit the fact that only a subset of all \( n \)-variable functions is sufficient to represent them all if inversions and permutations of signals are available. Further, if there are \( C \) such functions, they can be encoded by \( B = \lceil \log_2(C) \rceil < 2^n \) bits. This alone is not attractive if the ULM circuit is too complex or too slow to be of practical use. In this chapter, we propose such circuits which are inexpensive relative to the LUTs.

5.2.1 Equivalence Classes of Switching Functions

The fact that many functions are equivalent under permutation or inversion of inputs and inversion of outputs allows us to group all functions into equivalence classes. The equivalence under all three of these operations is commonly called npn-equivalence [45]. In FPGAs, we are primarily interested in the restricted notion of np-equivalence, which allows permutations and inversions of inputs only. The output inversions and npn-class will be considered here primarily because the main results can be easier derived using this classification.

The equivalence classes of switching functions have been investigated in early studies of switching functions [42]. Using enumeration techniques of Polya theory, a closed form expression can be derived for the number of equivalence classes, as a function of \( n \), the number of variables. For our purposes, it is sufficient to derive a lower bound on the number of npn-equivalence classes:

\[
C(n) \geq \frac{2^2^n}{n! \cdot 2^n + 2}.
\]  

(5.1)

This bound is obtained as follows. There are at most \( n! \cdot 2^n \) different permutations and negations of inputs and outputs, which defines an upper bound on the class size. The number of classes is then larger than the ratio of the number of all possible functions \( (2^{2^n}) \) and this bound on the class size. The exact number of equivalence classes is larger than this estimate, especially for small \( n \). For example, for \( n = 3 \), there are 14 such classes, while for \( n = 4 \), the number of equivalence classes is 222.

Our concept of the ULM assumes that a number of programming bits are provided that specify which equivalence class is to be realized by the block. For this model, we can derive an estimate on the number of programming bits needed. \( B = \lceil \log_2(C) \rceil \). After applying the
Table 5.1: Number of bits needed to encode all functions of $n$ variables.

<table>
<thead>
<tr>
<th>$n$</th>
<th>npn classes</th>
<th>Bits - npn ULM</th>
<th>Bits - np ULM</th>
<th>Bits - LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>14</td>
<td>4</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>8</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>616</td>
<td>126</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

Stirling approximation [39] for the factorial function and taking a logarithm of Equation 5.1 we obtain:

$$B(n) \geq 2^n - n \log(n) - (n + 1) + n \log(e) - 1/2 \log(2\pi n).$$

or

$$B(n) \geq 2^n - n \log(n) - O(n).$$

This bound rapidly approaches the size of the original lookup table $2^n$. Hence, for large $n$, this type of ULMs does not offer much savings in the number of programming bits.

However, for small $n$, which is of most practical interest, substantial savings can be obtained. For $n = 3$, the total number of bits needed to encode functions realized by a ULM is 4 (as opposed to 8 in LUT.3), while for $n = 4$ and $n = 5$, the minimal number of programming bits $B$ is 8 and 20, respectively. For any number of inputs $n$, up to $C(n)$ classes of functions have to be provided, for which it is sufficient to have $B(n)$ programming bits. For np-equivalence type of device, one programming bit must be added, which can invert the polarity of the output. Table 5.1 gives the number of bits required for the ULM.3 through ULM.5 and compares them with the number of bits needed in the corresponding LUT.

Although the saving in the number of programming bits looks encouraging, the implementation of such ULMs may be much larger and slower than that of LUTs. We now derive ULMs whose implementations are comparable to those of LUTs.

### 5.2.2 Realization of ULMs using BDDs

Each equivalence class can be represented by one function, which is called a class prototype or representative in literature. To generate all functions of $n$ variables in a ULM, it is sufficient to have a block that realizes only the representatives.

We devise effective ULMs by constructing a flexible "supercircuit" that can implement...
all representative functions by using special programmed switches provided in that circuit. Our realization uses the structure of BDDs (Binary Decision Diagrams) [19] to realize a complete set of representative functions. Additional switches, which select the function to be realized, are used to reconfigure the BDD structure. BDDs are chosen because they are a canonical representation of binary functions which can be used in physical implementation. To realize a function given by a BDD, it is sufficient to replace each node by a multiplexer.

The procedure for designing optimal ULMs consists of:

- enumerating all classes of functions.
- realizing each class representative by a BDD.
- creating a superset structure, called a Super BDD, from the union of all BDD representatives.
- providing flexibility in the Super BDD by adding routing resources.
- minimizing the number of routing paths and switches in the Super BDD.
- minimizing the number of programming bits, and
- optimizing circuits that use the programming bits to configure the desired function.

To enumerate function classes for our purposes, it is sufficient to consider only the functions that depend on exactly $n$ variables. The proof that these functions are sufficient is as follows. Assume without loss of generality that the desired function $f$ depends on $n - 1$ variables, $x_1, x_2, \ldots, x_{n-1}$. Then, the logical AND function: $x_n f$ depends on exactly $n$ variables. Since all $n$-variable functions are represented, it follows that we can realize the functions $f$ by assigning the value 1 to $x_n$. Hence, all the functions depending on the smaller number of variables can be implemented as well.

The step of realizing class representatives consists of a straightforward implementation of the BDD construction algorithm. The creation of the Super BDD superset structure is explained in the following two sections, including the simplification of such a structure. The final two optimization steps can be achieved by the standard input encoding techniques [99]. Based on this approach, we explicitly design 3- and 4-input logic blocks.
<table>
<thead>
<tr>
<th>No.</th>
<th>Class representative</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_1x_2x_3$</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>$x_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>$x_1x_2 + x_1x_3$</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>$x_1x_2 + \bar{x}_1\bar{x}_2x_3$</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>$x_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1x_2x_3$</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>$x_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1x_2x_3$</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>$x_1x_2 + x_1x_3 + x_2x_3$</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>$x_1\bar{x}_3 + x_2x_3$</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>$x_1x_2x_3 + x_1\bar{x}_2x_3$</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>$x_1x_2 + x_1x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5.2: Equivalence classes for functions of three variables.

### 5.3 Realization of ULM.3

It is sufficient to enumerate only the npn-equivalent functions, because BDDs describing both a representative of such class and its complement have the same structure, with only the terminal nodes being reversed. For $n = 3$, there are 14 equivalence classes, of which 10 are functions of exactly 3 variables. Table 5.2 shows these classes and the number of functions they represent.

For each representative three-variable function, a BDD can have up to 5 nonterminal nodes. The union of these classes will, therefore, have at most 5 of these nodes, plus an interconnection structure and programming switches needed to program the ULM.3. All 10 representative BDDs are shown in Figure 5.2. The input (control) variables are ordered as: $x_1, x_2, x_3$: we say that the nodes controlled by variable $x_i$ belong to level $i$. The left outgoing edge, $0$-edge, of each node is taken when the input variable is 0, which is indicated by a dashed line. The two successors are referred to as 0- and 1-successor. The values of terminal nodes are not specified, because of the possible inversion of outputs under the (considered) npn-equivalence model; it is assumed that in the canonical case the left terminal node is zero.

#### 5.3.1 Super BDD as a ULM

We can combine the BDDs to create the Super BDD (SBDD.3) in Figure 5.3 that is capable of realizing all 10 class representative functions of three variables. This union structure has 5 nonterminal nodes, which we label as in the sixth BDD in Figure 5.2.

The SBDD.3 is obtained from the representative BDDs by the following transformations.
Figure 5.2: All representative BDDs for three-variable functions.
By enumerating the outgoing edges from each node, a set of possible interconnections is obtained. Sets of outgoing edges are reduced by considering all possible polarities of input variables. At the last stage of optimization, we allow one extension to canonical BDDs, which leads to simpler representation: polarity of the selection variable at node 2 can be changed independently from the selection variable at node 3. The required polarity change is controlled by the switch $S_4$ in Figure 5.3. Other switches in this BDD are used to define the multiplexed connections to outgoing edges. Switch $S_7$ changes the polarity of terminal nodes, i.e., it inverts the function.

The design in Figure 5.3 requires 6 switches if npn-equivalence is assumed. The seventh switch, $S_7$, is needed for the np-equivalence, in which case the terminal nodes can have two possible sets of values. The SBDD.3 can implement functions of two or one variables by assigning some inputs to a constant.

The SBDD.3 can be used as a ULM. Using one bit per switch, the number of programming bits is 7, which gives a saving of one bit compared to LUT.3.

### 5.3.2 Encoding of Programming Bits

It is possible to shorten the function descriptions for the proposed ULM by encoding more compactly all possible configurations of switches. To optimize the encoding of programming bit patterns, we enumerate all possible switch assignments for each representative function. The programming bit settings for each function are listed in Table 5.3. The left path of a programming switch is selected if the corresponding bit is equal to 0. An exception is switch
Table 3.3: Programming bits for three-variable representative functions.

<table>
<thead>
<tr>
<th>Function Number</th>
<th>Programming Switch 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Inverted Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>x_3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x_3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>x_3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x_3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x_3 and x_3</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>one of x_2-x_3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Programming bits for permuted representative functions.

<table>
<thead>
<tr>
<th>Function Number</th>
<th>Programming Switch 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Inverted Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x_2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

$S_1$, which changes the polarity of node 2 in the BDD. If this bit is one, then the 0-successor of node 2 is selected when the input variable $x_2$ is 1. The last column in Table 3.3 indicates if any variable is complemented at the input to the circuit. For function 9, either $x_2$ or $x_3$ should be complemented, but not both.

The desired optimization can be achieved by using more compact input encoding. Since there are 10 programming combinations, 4 bits are sufficient to encode all of them uniquely. However, the objective is to use the simplest logic circuits for this compact encoding. To achieve a simple realization we use the functional composition and reordering of some of the BDDs in Figure 5.2 rather than the standard input encoding approaches [99].

Switches 1 through 4 in Table 3.3 can be encoded separately, because the functions can be decomposed. Since there are 6 combinations of these switches in Table 3.3, three encoding bits are necessary for these four switches, and 5 bits are needed for the whole circuit. However, the optimal ULM.3 should use only 4 bits in total, as shown in Section 5.2.1. To obtain the optimal length encoding, we can have at most 4 programming combinations for these 4 bits. To achieve further reduction, we consider all input permutations for the representative functions. Only five functions can be replaced by reordering the variables, because the other five functions are symmetrical (indicated by bold typeface in Table 3.3). Table
5.4 lists the programming combinations for the permuted representatives that were used to optimize the SBDD.3. The decoder circuit can be simplified if the two functions in Table 5.4 replace the corresponding functions in Table 5.3. Function 10 should be replaced with the permutation which has all programming bits set to one. This permutation has $x_1$ and $x_2$ interchanged, and it implements the function $x_1 x_2 + x_2 x_3 + \bar{x}_1 \bar{x}_2 x_3$. With this function the inversion of the BDD node 2 may or may not be applied, because the variable $x_3$ can be independently inverted in the BDD. Second, function 8 is replaced by an implementation that has variables $x_2$ and $x_3$ interchanged, which leads to the programming vector 100101.

The two replacement BDDs are shown in Figure 5.4. With these replacements, there is a total of 4 different combinations of the programmable switches $S_1$ through $S_4$. These combinations are given in Table 5.5. An efficient encoding that satisfies the constraints in the table is given by:

\[
S_1 = B_0 + B_1 \\
S_2 = S_3 = B_0 \cdot B_1 \\
S_4 = B_1
\]

This allows a simple implementation of the decoder using only two 2-input gates. (Note also that a simple polarity change in switches and/or programming bits $B_0$ and $B_1$ allows us to use simpler NAND and NOR circuits.) The programming bits needed after the minimization is performed are shown in Table 5.6.
### 5.3.3 Implementation Issues

The proposed ULM.3 can be implemented in a straightforward way. In addition to the SBDD.3, the decoder and the programming bit memory are needed. The decoder consists of two 2-input gates, while the programming bits can be kept in a standard SRAM memory. Figure 5.5 shows the overall structure of the ULM.3.

The implementation of ULM.3 can follow the layout of the SBDD.3, which has physically a fairly rectangular shape. Figure 5.6 compares its layout with the LUT.3 logic block. Switches in the ULM.3 correspond to those in SBDD.3. We targeted pass-gate CMOS implementation for most of the logic, as in many LUT implementations [24].

The ULM.3 implementation has advantages with respect to the area. First, only three multiplexers (outlined in bold) in ULM.3 must have the complete functionality: all others perform simpler logic functions, which allows us to decrease the area required without any impact on the speed. Moreover, our ULM avoids a constant overhead of one buffer that each SRAM memory must have when used in LUT configurations [24]. Since there is a possibility of bidirectional current flow when the input to LUT changes, the contents of SRAMs can be erased and an inverter must be added to isolate the SRAM cells. In ULM.3, the memory

<table>
<thead>
<tr>
<th>No.</th>
<th>B₀</th>
<th>B₁</th>
<th>S₅</th>
<th>S₆</th>
<th>Inverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x₂</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x₂</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x₃</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x₃</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x₂</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: Encoding for the first four switches.

Table 5.6: Optimized programming bits.
controls only the gates of transistors and there is no current flow towards SRAMs. Finally, the tree-like structure, which in standard LUT architectures can cause an area overhead in the physical layout, is not present in this block.

The ULM.3 implementation also has some advantages when the delay is considered. Paths of input signals can be kept shorter, which limits the propagation delay through the block. Figure 5.6 shows the critical paths for the two circuits, from input $x_3$ to output $f$.

Both LUT.3 and ULM.3 in Figure 5.6 have been simulated in Spice using 0.8 $\mu$m BNR BATMOS technology [7]. With small topology transistors, both blocks (output buffers were not considered) work equally fast. Even though our ULM.3 block uses one more level of logic than LUT.3, the time critical signal, control variable $x_3$, does not control any pass transistors; instead this signal propagates through the channels of MOS transistors which are set (opened or closed) long before the signal reaches the transistor. The polarity inversion, which is shown in Figure 5.6 as an XOR gate at the output of the circuit is in actual implementation obtained by inverting the terminal nodes: such an inversion does not increase the critical path of the signal. The comparison is summarized in Table 5.7.

One other variation is possible: the block can have the function output available in both its true and complemented form. It would be then left for routing resources to select the proper polarity (or both). In this case, only 4 programming bits would be needed for the block, but adding one more output pin is expensive.

Figure 5.5: ULM.3 implemented as SBDD.3 and a decoder.
5.4 Larger Blocks - ULM.4

The same procedure can be used to design ULMs for larger logic blocks, but the complexity of the process increases rapidly. We illustrate, using the example of ULM.4 the procedure of (computer aided) search for an effective logic block. It is theoretically possible to derive a 4-input block that uses a minimal number of programming bits (8 for npn-equivalent and 9 for np-equivalent ULM). However, since the minimal-length encoding may be too expensive to implement, we investigated trade-offs between the encoding length and the circuit complexity by using non-optimal encodings as well.

There are 222 npn-equivalence classes of 4-variable functions; 208 of them depend on exactly 4 variables. These classes are enumerated in reference [28]. We used this enumeration
Table 5.7: Comparison between LUT.3 and ULM.3.

<table>
<thead>
<tr>
<th></th>
<th>LUT.3</th>
<th>ULM.3</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (bits)</td>
<td>8</td>
<td>5</td>
<td>1 for dual output</td>
</tr>
<tr>
<td>Datapath Mux</td>
<td>7</td>
<td>3</td>
<td>1 invertible</td>
</tr>
<tr>
<td>Program Mux</td>
<td>0</td>
<td>5</td>
<td>can be small</td>
</tr>
<tr>
<td>Decoder</td>
<td>0</td>
<td>1</td>
<td>two 2-input gates</td>
</tr>
<tr>
<td>Program Inv.</td>
<td>0</td>
<td>2</td>
<td>(S_4), (S_7)</td>
</tr>
<tr>
<td>Transistors</td>
<td>78</td>
<td>70</td>
<td>5 transistor RAM</td>
</tr>
<tr>
<td>Delay</td>
<td>1.38 ns</td>
<td>1.31 ns</td>
<td>small width transistors</td>
</tr>
</tbody>
</table>

as an input to a program that generates BDDs for all representative functions. We made these BDDs mutually comparable by using a unique node labeling. The next step was to analyze the connectivity pattern and find outgoing edges from each node in the BDD. Then, a possible Super BDD structure was constructed, which had a number of programmable switches. The remaining steps consisted of minimizing the number of switches, the number of programming bits and the logic needed to perform the encoding function for the switches.

5.4.1 Unifying Representative BDDS

The first step in the procedure for developing the ULM is easy to automate. All the class representatives in \([28]\) are sorted according to the output they produce. There are 208 such functions: we will refer to them as \(f_1\) to \(f_{208}\).

The maximum number of nodes in individual BDDs dictates how many nodes there should be in the SBDD. Theoretically, the largest BDD representing a 1-variable function should have at most 9 nodes, excluding terminal nodes. Starting from the root (level 1) node, the edges can branch as in the full decision tree, except for the fact that there can be only two level-4 nodes. There are 15 BDDs that have this maximal number of nodes.

Unique labeling of nodes is necessary for analyzing the BDDs in a unified way. It is easy to distinguish between two level-2 nodes: they are either 0- or 1-successors of the root node. Also, the two terminal and two level-4 nodes are unique. To label the remaining, level-3 nodes, we use the following scheme. We assign numbers 1 and 2 to terminal 0 and 1 nodes, respectively. Then, a node \(v\) is labelled by combining the labels of its 0- and 1-successors using the expression

\[
label(v) = 2 \cdot label(v.0) + label(v.1)
\]  

(5.2)

Figure 5.7 shows an example of this BDD labeling. The terminal nodes are labeled as 1.
Figure 5.7: Example BDD labels.

and 2, respectively, while all other labels are obtained using the function label. Note that nodes 4 and 5 correspond to the functions \( x \) and \( \bar{x} \), respectively.

This labeling is almost unique. There are \( \binom{3}{2} = 3 \) possible combinations of two different successors of level-3 nodes and only two pairs of successor nodes with the same label. Number 9 can be obtained as \( 9 = 2 \cdot 2 + 5 = 2 \cdot 4 + 1 \), hence the number 9 could label two different nodes, the one with successors \( (2, 5) \) and another with successors \( (4, 1) \). To make the labeling unique, we assign the label 15 to the second of these nodes while keeping the label 9 for the first node, as in Figure 5.7.

To optimize the interconnect, we allow that some level-3 nodes can have both outgoing edges pointing to the same node. Note that this node would not exist in standard BDDs.

When the successor of such node is at level 4 (node 4 or 5), we label the level-3 node as \( T_4 \) or \( T_5 \). An example of node \( T_4 \) is shown in Figure 5.7. Note that when computing a label for a node at level 2 (e.g., node 17 in the figure), the value of \( T_4 \) is 4.

The interconnect patterns can be analyzed with the above unique labeling. For each node \( v \), the set of successors \( S(v) \) is recorded. Since we want to minimize the total interconnect, we first examine if the successor sets can be minimized. We found that since there are many functions, only a few edges can be eliminated by permuting the variables in some of the functions.

Analysis of the structure of the given representative functions shows much regularity. For example, there is no edge 0 (dashed edge) between the root and the level-3 nodes. The goal is to exploit the regularity in the prototype functions to simplify the structure of the
Super BDD.

5.4.2 Generating Super BDD (SBDD.4)

The number of nodes in SBDD.4 was determined in the previous step, and it remains to determine the interconnections. The first step in this process consists of assigning the node labels to the physical nodes in SBDD.4. There are 12 possible labels for the level-3 nodes, but only 4 such nodes are present in the SBDD.4. Therefore, these nodes must be capable of realizing several functions. The functions should be assigned to these nodes, to minimize the total number of switches.

Figure 5.8 shows an outline of one possible SBDD.4. The functions to be realized by level-3 nodes are enumerated inside the oval and the circuits that implement these functions are shown in more detail in the associated dotted boxes. The two level-4 nodes, 4 and 5 (functions $f$ and $\bar{f}$) are used as primitives and we did not expand them further. The two terminal nodes, 0 and 1, are indicated inside the boxes.

The total number of switches to control SBDD.4 is 17, one more than the number of programming bits for LUT.4. One additional bit ($a_1$ in Figure 5.8) was eliminated in the final optimization. Several iterations were made in permuting the variables, to minimize

Figure 5.8: Outline of SBDD.4.
the number of switches. By exchanging variables $x_1$ and $x_2$ in functions $f_{185}$ and $f_{207}$, variables $x_1$ and $x_3$ in $f_{205}$, $f_{206}$, and ordering the first three variables as $x_2, x_3, x_1$ in $f_{201}$, we eliminated two switches, $a_1$ and $d_2$, from the SBDD.4. Elimination of $d_2$ was possible because all four functions needed in L3.4 node can be realized if the left edge always points to node 4. These two optimization steps decreased the number of switches to 15.

The encoding of the switches was analyzed and grouped with respect to the configuration of the first four switches. All the functions are enumerated in Table 5.8. All possible functions assigned to the four L3 nodes are given in columns. While nodes 2 and 3 (called L3.2 and L3.3) have to realize all possible functions, the other two L3 nodes have to realize just a few, which leads to a saving in the number of switches. Thus, the SBDD.4 can be simplified as shown in Figure 5.9. This implementation requires 15 programming switches and hence 15 programming bits.

### 5.4.3 Input Encoding for Programming Bits

To reduce the number of programming bits, we can encode the possible switch settings. To achieve this, we used the input encoding algorithm in NOVA, which is included in the SIS [36] package. Since the minimal-length encoding is expected to be expensive, we considered several other encoding strategies.

The four switches $S_1$ through $S_4$ can be encoded using three bits for 5 possible configurations, corresponding to groups $G_1$ to $G_3$, as shown in Table 5.9. This leads to a simple decoder:

\[
S_1 = B_3, \\
S_2 = B_1
\]
Figure 5.9: Optimized SBDD.4.

Table 5.9: Encoding for first four switches.
\[ S_3 = B_2 \cdot B_3 \]
\[ S_4 = B_2 + B_1 \]

Fourteen programming bits are needed in this arrangement. Note that group \( G_5 \) in Table 5.8 has exactly one function in it (the 4-input XOR), which costs a programming switch \( S_3 \). The edge emanating from this switch is marked as "XOR" in Figure 5.9.

Further reduction in the number of programming bits can be achieved through more careful encoding of functions in each of the existing 5 groups. Notice that this encoding does not affect the speed of the circuit. All the additional circuits are placed between the memory cells and switches, and they are not in the path of the signal.

We were able to remove two more bits in the encoding of the function with less than 10 gates required for all the decoding circuitry in the ULM.4. The encoding of functions is based on sharing bits among the sub-blocks and encoding the XOR function as part of group \( G_1 \). We omit the details of this encoding. The threshold of 10 gates was selected for the decoding circuitry, because with this overhead our ULM.4 is still smaller than LUT.4. Thus, we can realize a block that uses 12 programming bits with no extra expense in the hardware. To go further, to the theoretical limit of 8 bits, one must rely on the general-purpose input encoding programs, combined with pre-encoding. We tried several encoding alternatives, but the circuits that we obtained in this way were too expensive to be used in realistic blocks. It is an open question if there exists a solution that uses less than 12 bits with a reasonable amount of decoding circuitry. Note that all encodings in this section are given for an npn-equivalent/dual-output class of ULMs, and that one more bit should be added for npn-equivalent circuits.

5.4.4 Larger Blocks

The optimization steps required to produce these blocks become costly, as the size of the block increases. A simple alternative is to construct ULMs from a smaller size ULM and LUT, as shown if Figure 5.10. This would produce realistic-size ULM.5 which requires 29 programming bits.

119
5.5 Other Issues

5.5.1 Technology Mapping Using ULMs

The ULM blocks provide the same functionality as lookup tables, but with a restriction on
the ordering and polarity of variables. For recognizing the order and polarity of inputs and
outputs of each block, we have implemented an algorithm based on Generalized Reed-Muller
form matching, as in [94].

Assuming that the interconnection resources in an FPGA allow arbitrary permutations
of inputs to logic blocks, the only remaining constraint is the polarity of input variables
to each block. There can be a polarity disagreement when both polarities of a signal are
needed. Then, some of the ULM-based blocks must be replicated. The study in [57] found,
using the MCNC benchmarks, that their ULM needs both polarities for only 6.6% of the
nodes, and that the required increase in the number of blocks is 9% compared to LUT.3.
Since that study was done using a block with incomplete functionality, the increase in the
number of logic blocks using our block of the same granularity (ULM.3) can be in the
worst case 6.6%. It is sufficient to replicate those blocks with polarity disagreement at the
output. Also, since nodes tend to have smaller fanout when the granularity of the logic
block increases, the results for our larger blocks (ULM.4) can only be better.

Figure 5.10: Construction of larger blocks.
5.5.2 Functionally Incomplete Blocks

The motivation of work in [57] and [93] was to investigate the construction of functionally incomplete blocks. Their blocks implemented 10 of 14 and 201 of 208 representative functions, respectively. It is interesting to note that using standard approaches for designing a functionally complete ULM.3, the best solution requires 5 input pins [88]. The only reason why 4 pins were sufficient in [57] is that the block was incomplete. The price of such complete LUT.3 replacement is obviously too high. Since our ULM.3 is complete and requires a minimum number of programming bits with no area or delay overhead compared to LUT.3, there is no need to consider incomplete blocks for 3-variable functions.

The SBDD.1 construction given here is useful in considering incomplete blocks as well. It is obvious from our ULM.4 that eliminating the 4-input XOR function \( f_{208} \) would remove one switch and one programming bit in the block. Another bit can be saved by considering the node L3.4 in Figure 5.9, for which there are 4 possible functions. Analysis shows that one switch can be eliminated by excluding functions \( f_{185} \) and \( f_{202} \) to \( f_{206} \). Hence, the logic block based on our ULM.4 that implements all but these seven functions requires 12 programming bits, with the decoding circuits consisting of only two 2-input gates. For comparison, the block in [93] realizes the same number of functions, but 8 input pins are required.

One more bit can be removed by excluding two more functions, \( f_{201} \) and \( f_{207} \), but at a price of one more 2-input decoding gate. This incomplete ULM circuit would require 11 bits for dual-output block and 12 bits for single-output np-equivalent block and it would realize 199 of 208 class representatives.

These considerations can lead to useful larger blocks as well. By excluding larger sets of functions, compact ULM circuits can be devised.

5.5.3 Using Other Graph-based Representations - FDD Case

The methodology for constructing ULMs presented here can be used in conjunction with several other Boolean function representations. We consider Functional Decision Diagrams (FDDs), which are the BDD representations of the Reed-Muller form [82]. Compared to BDDs whose nodes can be realized by multiplexers, each node of a FDD can be realized by
BDDs:

FDDs:

Figure 5.11: Comparison of BDDs and FDDs for n=3.
a Boolean function of the type

\[ a \equiv b x. \]

Enumeration of all FDD class representative implementations of \( n = 3 \) and \( n = 4 \) reveals that the FDD implementations are simpler overall. Figure 5.11 illustrates this fact for \( n = 3 \). While the output negation of a function results in a BDD of the same shape (but reversed terminal nodes), the same operation produces two different FDDs, of which the simpler one can be taken as a class representative for a given class.

## 5.6 Concluding Remarks

We presented a class of FPGA logic blocks based on the concept of ULMS, which are functionally complete if the permutations and negations of inputs are provided outside the block. As in SRAM-based FPGAs, we use separate programming bits, which is of advantage in practical FPGAs. Previously considered ULMS require costly additional inputs to the logic blocks.

We also presented a methodology for designing such blocks, using decision diagrams and showed detailed designs of replacements for 3- and 4-input lookup tables. In the case of ULM.3, 5 programming bits are needed, for a block slightly smaller than LUT.3. For ULM.4, several alternatives with different tradeoffs between the number of programming bits and the complexity of the circuit were considered. A circuit that requires 13 bits was devised, such that it is smaller than LUT.4. An especially important problem in the construction of such ULMs is the input encoding problem and we used the domain-specific methods which outperform the standard approaches. Furthermore, while the known ULM circuits considered for application in FPGAs [57, 93] are functionally incomplete, our construction offers the complete functionality at a reasonable cost.

The proposed blocks are particularly interesting for FPGAs that will cater to the emerging area of reconfigurable computing.
Chapter 6

Conclusions

In this dissertation we considered several problems related to the use of spectral methods in CAD in general and RM transform in particular. Our objective was to bring spectral methods closer to engineering practice.

6.1 Field Expansions and Incompletely Specified Functions

We studied the problem of obtaining transforms for partial or incompletely specified functions. We concentrated on field expansions that include both binary and MVL versions of the RM transform. Although the RM transform is the oldest transform used for logic functions, and although incompletely specified functions cannot be avoided in logic synthesis, the problem of computing RM transforms of incompletely specified functions has not been solved adequately.

In this dissertation, we showed that the multivariate finite field polynomial interpolation can be used to produce field expansions for partial functions. We demonstrated that the interpolation approach is the only known approach that can produce the resulting forms that have cost bounded by the input size in polynomial execution time. For that purpose, we created several interpolation algorithms.

In Chapter 3, we presented the dense interpolation algorithm suited for functions with few don't cares. The algorithm incorporates the univariate interpolation in the fast RM transform framework. Consequently, the execution time is comparable to that of the fast RM transform. Next, we presented an algorithm which can produce forms of the size limited by the number of specified points in time $n$ times slower than the fast RM transform. Algo-
rithms which use BDD representations for both the input and the output of the algorithm were derived: extensions to several related forms can be generated as well.

In Chapter 4 we investigated in detail the problem of multivariate polynomial interpolation over finite fields. Based on our characterization of the solution space, we derived efficient interpolation algorithms that are especially useful for interpolations over small finite fields. The interpolation algorithm in the binary case requires only the construction of the ordering relation between the input points, that is, the poset of points. The algorithm then traverses that poset and produces the solution in time quadratic in the number of points given.

Although the poset-based algorithm in Chapter 4 is polynomial in the number of points, and no polynomial-time bounds can be derived for the recursive algorithm from Chapter 3, there are situations in which the latter can be more useful. If the number of points becomes large, the BDD-based algorithm can become less expensive. Its running time is then bounded above by $n^{2n}$, which is less than $n^{2n}$, which is the bound for the poset-based interpolation. Furthermore, there are the indications that the BDD-based algorithm maintains small data structures, although we could not provide a proof on the limit on their size. Additionally, the BDD-based algorithms provide more opportunity to search for more concise forms. We showed that it is possible to combine the knowledge about the interpolation approach, presented here, with the heuristics that seek the simple RM transforms.

A particularly significant result presented in this dissertation is the deterministic algorithm for finite field multivariate polynomial interpolation. We know of no similar result. The exposition of the algorithm is especially suited for interpolation over small finite fields which interest us the most. First, we characterized the solution to the interpolation problem such that the decomposition into smaller problems becomes apparent. Second, using tools of linear algebra, we presented a method to select terms which increase the rank of the multivariate generalized Vandermonde matrix deterministically. Finally, we showed that the search for new terms can be done efficiently. We also offered insight into designing randomized interpolation algorithms which can be interesting in some applications.

Several immediate applications of our algorithm were presented. It was shown that the related forms, and not only the field expansions, may be obtained using our approach. In the MVL case, the fixed polarity RM forms can be produced such that not only the forms become simpler, but the algorithms become faster. Further, the decomposition of
the problem allows us to devise practical parallel and incremental (on-line) interpolations. The latter might find an application in machine, or computational, learning. We consider learning to be related to the circuit realization problem, by which a function which is only partially known has to be implemented. The algorithm that we devised has such learning behavior: it can adjust its output if the data contradictory with the hypothesis that is being built is encountered. For comparison, artificial neural networks, which are used most commonly for such learning applications, cannot provide such behavior: they cannot even provably approximate the functions learned.

6.1.1 Future Work

Many problems with incompletely specified functions are still open in logic synthesis. We relate these problems to the applications in which the transform methods are used, like isomorphism (or classification) of Boolean and MVL functions, detection of various properties, decomposition and synthesis. The goal is to make these applications easier, not harder, with the existence of don’t cares. For now, it seems that the problems only get harder and it would be interesting to see how much progress can be made in that direction.

Our interpolation may have some relevance to using other spectral methods. We see the closest connection to the wavelets: our methods rely on the order between the subspaces at which elements of the base act, and the sole idea of the wavelet is to act over various subsets of the set to be analyzed.

The finite field interpolation algorithm can possibly be improved. The question of existence of an $NC$-class parallel interpolation algorithm may have a better answer than the one presented here. The interpolation algorithm may be extended to the interpolation problem over rational numbers, considered in numerical analysis. The best known algorithms in that area are presented in [31] and [83]: solving the examples provided in those papers is much easier using our algorithm than using the methods provided there.

Another, maybe even more interesting observation, is that the interpolation problem can be given in more general settings than those provided either by us, or by numerical analysts. The interpolation can be defined over $Universal Algebras$ [48], which include much more general algebraic structures than those used here. Our use of partial order may be related to the deeper results in that area [43]. We found no apparent connection to our method, but it became apparent that most of the other methods can be explained
by these results. According to the Universal Algebra view, the algebraic structure has the interpolation property if a form of Chinese remainder theorem holds. Going back to the interpolation methods over fields, one can see that many of these methods can be explained by the use of Chinese remainder theorem: Lagrange and Newton interpolation. FFT and the method by Ben-Or and Tiwari are the obvious examples. No such connection is apparent in our case, and it is open if any further insight can be obtained in this way.

6.2 Universal Logic Modules

We devised a class of ULMs which can be used as a replacement for lookup-table based blocks in FPGAs, and which require a smaller number of programming bits. The bounds on the number of programming bits were established and the procedure for designing optimal blocks with respect to the encoding length was outlined. The optimal and near-optimal modules were designed using this methodology. These modules can be useful in applications where important complexity measures are the length of the description of functions to be implemented and the time to communicate such description.

6.2.1 Future Work

Decision diagrams are not the only representation that can be used in ULM design: the work presented here can be a starting point for investigating ULM modules that use different function representations to achieve the same goal. We are aware of some attempts to use PLA circuits to build ULMs of the same type. It would be interesting to compare these approaches.

The number of programming bits inside the logic blocks is small compared to the total number of programming bits needed in FPGAs. Most of the bits are used for the interconnection. Recent work on reducing the number of programming bits for the interconnection circuitry [32] provided an estimate on the possible savings there. Interestingly, their savings in reducing the permutation possibilities in the interconnect are equal to the savings that we obtain using ULMs. A system-level study which would relate the encoding of logic modules, such as ours, and that of the routing structure would be of interest. Also, the question of synthesis using such logic blocks and routing structures has to be addressed.
Appendix A

Summary of Useful Facts

Relations

We use the notion of a set as a collection of elements. The capital letters are reserved for sets, while the lowercase letters denote the elements. For two sets, $X$ and $Y$, the Cartesian product $X \times Y$ is defined as a set of ordered pairs $(x, y)$ such that $x \in X$ and $y \in Y$. Ordered pairs and the Cartesian product can be generalized to the $n$-tuples, and the product of $n$ sets $S_1 \times S_2 \times \ldots S_n$, respectively.

We recapitulate shortly the facts about the relations that are used throughout the thesis. Binary relation $R$ over a set $S$ is the subset of the Cartesian product

$$R \subseteq S \times S.$$  

Two types of relations deserve special attention as they are used extensively in this thesis. The equivalence relation is symmetric (i.e. $(x, y) \in R \Rightarrow (y, x) \in R$), reflexive (i.e. $(x, x) \in R$) and transitive (i.e. $(x, y) \in R$ and $(y, z) \in R \Rightarrow (x, z) \in R$), while the partial order is antisymmetric, reflexive and transitive. The equivalence relation has a property of partitioning a set into equivalence classes and the partial ordering relation induces an order between the elements of a set. A set together with the partial order is called a partially ordered set, or poset.

The partial order is total or linear if any two elements are ordered, i.e. if they are the members of a relation. For any poset, the linear extension is defined as a total order consistent with the given partial order. One important example of a linear extension is the
**lexicographic order** between $n$-tuples, as a "natural" way to totally order the $n$-dimensional points. The point $a = a_1a_2 \ldots a_n$ precedes point $b = b_1b_2 \ldots b_n$ lexicographically, $a <_L b$, if for each coordinate $i$, starting with the coordinate $i = 1$ either $a_i$ is smaller than $b_i$, or the tuple $a_{i+1}a_{i+2} \ldots a_n$ lexicographically precedes $b_{i+1}b_{i+2} \ldots b_n$.

Graphs represent relations: they consist of a set of *vertices*, $V$, and *edges*, $E$, which is a relation over $V$, i.e., $E \subseteq V \times V$. Directed graph represents relations in general, while undirected represents symmetrical relations: drawings representing the former have arrows on the edges, while those representing the latter do not.

Two graphs are *isomorphic* if there is a one-to-one correspondence between their vertex sets that preserves the adjacency, i.e. the relation between the vertices.

**Finite Fields and Rings**

A *field* is an algebraic structure that consists of a set and two operations that allow the arithmetic over the set. These operations, addition and multiplication, are defined such that they "make sense", i.e. such that the standard properties of these two operations hold. Such properties are the unique invertibility for both addition and multiplication as well as the associativity of these two operations, i.e. $(a + b) + c = a + (b + c)$. together with distributivity property: $(a + b)c = ac + bc$.

All finite fields have the number of elements equal to a prime number, $p$, or some power of it, $p^n$. In the case of a field with a prime number of elements, both operations are defined as usual modulo addition and multiplication. For other finite fields, the operations are defined as multiplication and addition modulo some irreducible polynomial over a simpler field. The usual notation for fields is $GF(p)$, or $GFp$, for the prime cardinality case, and $GF(p^n)$, or $GFp^n$, for the composite case. We use the symbol $GFq$ to denote both cases.

The nonzero elements of $GFq$ form a multiplicative group that is cyclic. Exponential operator in such a group is defined as a repeated application of the multiplication. For all cyclic groups, there is an element $\epsilon$ (called primitive) whose powers $(\epsilon^0, \epsilon^1, \epsilon^2, \ldots, \epsilon^{l-1})$ contain the whole group. Furthermore, for every element $a$ from that group, $a^l = a$, or $a^{q-1} = 1$.

The simplest case is that of a two-element field. Then, the addition is defined as the logical XOR operation, while the multiplication is defined as logical AND. For a 4-element field, the operations are given in Figure 2.1, Chapter 2.
If the restrictions on the multiplication operation are relaxed, then another algebraic structure, that of ring, is introduced. In rings, the equation $ab = 0$ can have solutions different than $a = 0$ or $b = 0$ (or both). Additionally, the multiplication does not have to be commutative. An example of the finite (commutative) ring is a set $0, 1, \ldots, m - 1$, where $m$ can be composite (nonprime) number, together with the operations of addition and multiplication modulo $m$. For example, if $m$ is 6, then 2 multiplied by 3 gives 0, hence the structure is not a field. Notice that all fields are rings.

**Linear Algebra and Matrices**

Vector spaces are defined over arbitrary fields. In addition to scalars, which are the field elements, a set of vectors exists for which only the addition and multiplication by scalars is defined. A subspace of a vector space is a set of vectors closed under vector addition and multiplication by scalars; these two operations form together linear combinations. The basis of a vector space is a set of vectors which generate, or span, the whole space by its linear combinations. Its dimension corresponds to the minimal number of vectors that span the space. A set of vectors is linearly dependent if at least one of the vectors can be expressed as a linear combination of the other vectors. Vectors in $n$-dimensional spaces can be expressed by $n$-tuples of field elements.

A mapping $V$ between two vector spaces is linear mapping if a map of a linear combination of vectors is equal to the same linear combination of mappings $V$ over each vector. Linear mappings from $n$-dimensional to $m$-dimensional space can be expressed by a matrix of size $m \times n$. These mappings can be bijective only if $m = n$, otherwise, some subspaces can map to 0. Such subspaces are called nullspaces, and their dimension is called nullity. Dimension of the image of mapping is called rank; rank is always equal to the maximal number of linearly independent rows of a matrix, which is the same as the number of independent columns. Unique inverse mapping exists for square $n \times n$ matrices of full rank, $n$.

**Algorithms and Complexity**

It is assumed that an algorithm is efficient if its running time is polynomial in the input size. A class of NP-complete problems is believed not to have efficient algorithms, as well as the related NP-hard optimization problems. For these, combinatorial search exponential in the
input size is believed to be the only way to find the optimal solution. For most applications the alternative ways have to be found. The approximations to the optimal solution, which are worse than the optimal solution for only a constant factor are preferred. It has been shown that there are classes of problems for which no approximation algorithm can be efficient, unless NP-complete problems can be solved with polynomial time algorithms [67].

Randomized Algorithms [64], which use a series of random decisions are often simpler and faster than the standard, deterministic algorithms. Many practical algorithms, such as simulated annealing, use the randomization effectively for solving the real-world problems. A randomized algorithm is effective if it can solve the problem in polynomial time, with high probability. The algorithms that always produce result, with the probable polynomial execution time are called Las Vegas algorithms. The algorithms that always terminate in polynomial time, but might not produce a result are called Monte Carlo algorithms.

Parallel Algorithms [72] exploit the existence of many computing units working together. The circuits are one example of parallel machine, another is a collection of Random Access Machines, called PRAM. The efficient parallel algorithms are those that use the polynomial number of processors and the polylogarithmic (a power of a logarithm) time. Such algorithms, which use either a circuit of a PRAM machine, belong to the NC class. Related randomized algorithms belong to the RNC class. Problems for which no effective parallel algorithm exist are P-complete.

Most of the knowledge about the complexity was obtained studying the problems related to the Boolean functions and circuits. The most characteristic example of an NP-complete problem is satisfiability problem, which asks for an assignment of inputs for which the function is 1 (true). This problem is NP-complete for many useful representations of Boolean functions, including BDDs and FDDs. Another fundamental problem separates efficient parallel and serial algorithms: circuit evaluation problem is P-complete and in a way signifies the fact that the circuits are exponentially more compact representations of Boolean functions than the expressions.

Learning

There are the several approaches to machine learning, in both the theory and practice of designing learning algorithms. In areas like pattern recognition, where the goal is to classify an object based on its features (e.g. perform medical diagnosis based on tests), the statistical
formulation seems to be dominant. Other approaches include the games against unknown adversary, or the methods of approximation of functions. Common for most of them is that, based on the observed data for a set of inputs, a Boolean or multiple-valued hypothesis is produced that belongs to some concept set. For example, the concepts are the AND-OR formulas with limited number of terms or polynomials of limited degree. Two theoretical models seem to be dominant. Probably Approximately Correct model algorithms will output a hypothesis that will with a high probability be a good approximation to the concept. The query models assume that the learner can ask a series of questions to deduce the hypothesis compatible with the answers received. Membership queries ask if a hypothesis is true for a specific input, while the equivalence query asks if the complete function is guessed correctly, to which either the positive answer or the counterexample is given. A form of equivalence can be established between various learning models, as shown in [98].

Practitioners use different view on learning. Usually, learning consists of training phase in which a set of inputs is presented to the program (or machine), during which the representation is built. The program should then function correctly for the yet unseen examples (off-training set). Usually, some empirical evaluation of the learning method is sufficient to justify the learning algorithm, at least for a specific application. The most popular realizations of such machines use artificial neural networks.
Appendix B

Proofs on Term Selection

Several auxiliary proofs are included in this appendix. These proofs demonstrate the usefulness of greedy term selection. While these results are not used in the derivation of the interpolation algorithm, the proof methods used can help in understanding the proofs on the algorithm. We consider the $GF3$ case in more detail because it provides most of the worst-case examples.

While other term selection schemes can produce a singular systems for only two points, with $GF3$ case being the worst, greedy term assignment will never fail. When applied to $GF3$, the greedy assignment of terms produces a nonsingular system for any two points.

Claim 1 The greedy algorithm for selecting the terms will produce a nonsingular matrix for any two $n$-tuples over $GF3$.

Proof: Only one exponent will change, in the place where the coordinate in the second point has changed (note that the changed coordinate in the second point must be 2, because of the order). Then, the matrix will have entries:

$$T = \begin{bmatrix} a \cdot 1^1 & a \cdot 1^2 \\ b \cdot 2^1 & b \cdot 2^2 \end{bmatrix}$$

Since neither $a$ nor $b$ can be zero, the determinant of this matrix is $ab - 2ab$, which is always nonzero. □

This claim can be generalized to any field.

Theorem 13 The greedy algorithm for selecting the terms will produce a nonsingular matrix for any two points in a field.
Proof: We start with the coefficient 11...1. Only one coefficient will change for the second point. Suppose that the first point had a coordinate \( s \), and that the second was \( t \). Then we have the following matrix:

\[
T = \begin{bmatrix}
  a \times s^1 & a \times s^2 \\
  b \times t^1 & b \times t^2
\end{bmatrix}
\]

whose determinant is zero if \( s^2 t = st^2 \), or \( s = t \). \( \Box \)

Further, any two points whose term distance is 1 do not give a singular system. If terms for two points differ in one coefficient, and if the values of this coefficient differ by 1, we will call such points neighbors.

**Theorem 14** The system produced by any two neighbor points is nonsingular.

Proof: If the term coefficient that has changed is \( c \), then:

\[
T = \begin{bmatrix}
  a \times s^c & a \times s^{c+1} \\
  b \times t^c & b \times t^{c+1}
\end{bmatrix}
\]

Which is singular if \( s = t \). But, then, the coefficients would be the same. \( \Box \)

This theorem cannot be extended to allow an arbitrary increment in the coefficient. The reason is that the equation \( s^l = t^l \) can have solutions other than \( s = t \) for arbitrary \( l \) over a finite field. But, if there is an arbitrary increment \( l \) to a term coefficient, then all increments between 1 and \( l \) must be present, as well. According to theorem 14, all these neighboring points make mutually nonsingular systems.

One immediate corollary of this theorem is the fact that the greedy selection of points produces a nonsingular system for any three points in any field.

**Theorem 15** Let terms \( p_1, p_2 \) and \( p_3 \) be neighbors. The matrix produced by these three points using the greedy term selection is nonsingular.

Proof: If the terms differ in one coefficient, then the interpolation is univariate. Otherwise assume without loss of generality that the first term (but not the point) consists of all 1s. We can write the system produced by these three points as:

\[
T = \begin{bmatrix}
  a \times u^s & a \times u^2s^2 & a \times u^2s \\
  b \times u^t & b \times u^2t^2 & b \times u^2t \\
  c \times v^w & c \times v^2w^2 & c \times v^2w
\end{bmatrix}
\]

134
The system will be singular if there are three constants $c_1, c_2$ and $c_3$ such that the three rows of a matrix annul. We know that neither $c_3$ nor $c_2$ can be zero, because the two pairs mutually produce a nonsingular system. We can write this system as:

\[ c_1 \cdot aus + c_2 \cdot but + c_3 \cdot cvw = 0 \]
\[ c_1 \cdot aus^2 + c_2 \cdot but^2 + c_3 \cdot cvw^2 = 0 \]
\[ c_1 \cdot au^2s + c_2 \cdot but^2t + c_3 \cdot cv^2w = 0 \]

from which we can eliminate the first constant $c_1$:

\[ c_1 = -\frac{c_2 \cdot but + c_3 \cdot cvw}{aus} \]

Replacing $c_1$ in the second and third equation yields:

\[ c_2 \cdot \left( but^2 - buts \right) + c_3 \cdot \left( cvw^2 - cusw \right) = 0 \]
\[ c_2 \cdot \left( but^2 - but^2 \right) + c_3 \cdot \left( cv^2w - cvw \right) = 0 \]

From the second equation, we obtain that either $c_3$ or $c = a$ equals zero. We know that both of these are impossible: $c_3$ cannot be zero because the system over two points is nonsingular, while $c = a$ implies that the second term coefficient has not changed. ☐

Another case is where all three points lie on one line, i.e., if only one coefficient changes. In this case, too, the system is nonsingular because the interpolation is now univariate. It can be easily shown that our algorithm finds a minimal degree interpolation polynomial for this case. We conclude that all triples of terms, such that one term is a common neighbor of the other two, form a nonsingular system. Since our algorithm always produces such three initial points, we can guarantee that it will produce the correct result for any three points. From Theorem 15 we know that every pair of points that are in an immediate neighborhood of a point produces a nonsingular system. We would like to extend this result to include all points in the system. For this we have the following theorem.

**Theorem 16** Let $p_i$ be a point which has $n$ immediately higher neighbors. Then, these $n + 1$ points together constitute a nonsingular system.
Proof: By induction. For \( n = 3 \), it is shown in Theorem 15 that the system is nonsingular.

Induction step: Assume the theorem is true for \( n = k \) points. Then, there are no constants \( c_1, c_2, \ldots, c_k \) such that the rows annul. We now prove that when a new point is added to a system, there are no such \( k + 1 \) constants. Assume by way of contradiction that there exist such constants which annul the rows. Denote the values of the \( k + 1 \)-st point coordinate by \( w \) and \( z \). By considering the first and last \((k + 1\)-st\) column of the system, we notice that they differ only in the \( k + 1 \)-st coefficient. Hence, the total sum of rows can be expressed solely in terms of the last coordinate. We will write this as:

\[
C_1 \cdot w + c_{k+1} \cdot z = 0
\]

\[
C_1 \cdot w^2 + c_{k+1} \cdot z^2 = 0
\]

where \( C_1 \) is a linear combination of the coefficients \( c_1, c_2, \ldots, c_k \) which does not change in the first and \( k + 1 \)-st column. By the induction hypothesis, \( C_1 \) cannot be zero. From these two equations, we can express:

\[
C_1 = -c_{k+1} \frac{z}{w}
\]

which, when substituted in the second equation yields:

\[
c_{k+1} z \cdot (z - w) = 0
\]

Since the system is nonsingular for \( k \) points, \( c_{k+1} \) cannot be zero; \( z \) and \( w \) must also be nonzero and have different values. Hence, the total system is nonsingular. \( \square \)
Appendix C

Enumeration of 4–Variable Functions by BDDs
Figure C.1: BDDs for 4-variable functions - functions $f_1$ to $f_{36}$
Figure C.2: BDDs for 4-variable functions - functions $f_{37}$ to $f_{72}$
Figure C.3: BDDs for 4-variable functions - functions $f_7$ to $f_{108}$
Figure C.4: BDDs for 4-variable functions - functions $f_{109}$ to $f_{144}$
Figure C.5: BDDs for 4-variable functions - functions $f_{145}$ to $f_{180}$
Figure C.6: BDDs for 4-variable functions - functions $f_{181}$ to $f_{208}$
Bibliography


151


