EFFICIENT MINING ALGORITHMS IN ENGINEERING

A Dissertation Presented

by

DAN ALEXANDRU PLETEA

Submitted to the Office of Graduate Studies,
University of Massachusetts Boston,
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2013

Computer Science Program
EFFICIENT MINING ALGORITHMS IN ENGINEERING

A Dissertation Presented

by

DAN ALEXANDRU PLETEA

Approved as to style and content by:

________________________________________
Dan A. Simovici, Professor
Chairperson of Committee

________________________________________
Wei Ding, Assistant Professor
Member

________________________________________
Nurit Haspel, Assistant Professor
Member

________________________________________
Catalin Zara, Associate Professor
Member

________________________________________
Dan A. Simovici, Program Director
Computer Science Program

________________________________________
Peter Fejer, Chairperson
Computer Science Department
ABSTRACT

EFFICIENT MINING ALGORITHMS IN ENGINEERING

December 2013

Dan Alexandru Pletea
B.S., Technical University “Gheorge Asachi”, Iasi, Romania

Directed by Professor Dan A. Simovici

Data mining is used in many areas of science and engineering, such as bioinformatics, genetics, finance and electrical engineering. The increased used of data mining brings a lot of new techniques, but with the cost of complexity and specialized problems. In electrical engineering, automated design of circuits is challenging due to the growing scale and complexities of the circuits. Data mining techniques improve the existing solutions, allowing circuit design to be fully automated or with minimal human intervention. Our contribution consists of a method that detects the minimal sets of variables that determine the values of a discrete partially defined function, and in a novel method of decomposition of partially specified index generation functions (PSIGFs). The data mining process can be costly. Therefore, it is important to evaluate the potential payoff of the mining process before the actual mining takes place. We propose a new approach for evaluating the minability of data sets by using compression. The basic idea is that compressible data contain patterns and the existence of these patterns makes the data worth mining.
First of all I dedicate my dissertation work to my family. A special feeling of gratitude to my parents, Dan-Doru and Ariadna-Lucia, who encouraged me to pursue the doctorate degree. I also dedicate this dissertation to my many friends who have supported me throughout the process. I will always appreciate all they have done; for the many hours of proofreading my work and for listening to my ideas.
ACKNOWLEDGEMENTS

First and foremost I want to thank my advisor, Professor Dan A. Simovici. It has been my honor to be his Ph.D. student. He has taught me how to do research, to teach and to better myself both as a scientist and as a person. I appreciate all his ideas, guidance and the countless hours spent with me. I thank my committee members, Wei Ding, Nurit Haspel and Cătălin Zara who provided encouraging and constructive feedback. I want also to thank Professor Octavian Păstrăvanu, who mentored me to complete my first publication.
# TABLE OF CONTENTS

DEDICATION ........................................... v

ACKNOWLEDGEMENTS ................................. vi

LIST OF TABLES ..................................... ix

LIST OF FIGURES ................................... xi

CHAPTER Page

1. INTRODUCTION ................................. 1
   Data and Data Mining ........................... 1
   Engineering Problems in Computer Science .... 3
   Data Compression and Data Mining ............. 5
   Thesis Organization ................................ 6

2. MINING PARTIALLY DEFINED FUNCTIONS ...... 8
   Introduction ..................................... 8
   Determining Sets for Partially Defined Functions ... 10
   An Apriori-like Algorithm for Mining MDSs ...... 12
   Experimental Results ........................... 18
   Entropies Associated with Partial Functions .... 20
   Experimental Results ........................... 36
   Comparison Between the Two Versions ........... 41
   Concluding Remarks ............................ 43

3. DECOMPOSITION OF PARTIALLY SPECIFIED INDEX GENERATING FUNCTIONS .......... 45
   Introduction ..................................... 45
   Partially Specified Index Generation Functions ... 46
   Boolean Matrices and Collections of Sets ....... 47
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intractability of covering a system of a linear equations with a minimum number of solutions</td>
<td>49</td>
</tr>
<tr>
<td>Reduction of Number of Variables for PSIGFs</td>
<td>52</td>
</tr>
<tr>
<td>Experimental Results</td>
<td>56</td>
</tr>
<tr>
<td>Conclusions</td>
<td>58</td>
</tr>
</tbody>
</table>

4. EVALUATING DATA MINABILITY THROUGH COMPRESSION - AN EXPERIMENTAL STUDY | 60
| Introduction | 60 |
| Patterns in Strings and Compression | 62 |
| Random Insertion and Compression | 68 |
| Frequent Items Sets and Compression Ratio | 75 |
| Concluding Remarks | 81 |

5. CONCLUSIONS | 82
| Mining Partially Defined Finite Functions | 82 |
| Decomposition of Partially Specified Index Generating Functions | 83 |
| Evaluating Data Minability Through Compression | 83 |

APPENDIX

A. NOTATIONS AND TERMINOLOGY | 85

B. INDSOL.M FUNCTION | 88

REFERENCE LIST | 90
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Tabular Representation of a Partial Function</td>
<td>3</td>
</tr>
<tr>
<td>2. Tabular Representation of a Partial Function</td>
<td>5</td>
</tr>
<tr>
<td>3. Tabular Representation of a Partial Function</td>
<td>11</td>
</tr>
<tr>
<td>4. Tabular Representation of a Partial Function</td>
<td>16</td>
</tr>
<tr>
<td>5. Partially Defined Function Example</td>
<td>26</td>
</tr>
<tr>
<td>6. Partially Defined Function Example</td>
<td>28</td>
</tr>
<tr>
<td>7. Methods for Computing MDS($f, \ell$)</td>
<td>30</td>
</tr>
<tr>
<td>8. Variables for Computing MDS($f, \ell$)</td>
<td>31</td>
</tr>
<tr>
<td>9. Methods for Compute_ENTROPY($\mathcal{H}(y</td>
<td>X)$)</td>
</tr>
<tr>
<td>10. Variables for Compute_ENTROPY($f, X$)</td>
<td>34</td>
</tr>
<tr>
<td>11. Tabular Representation of a Partial Function</td>
<td>42</td>
</tr>
<tr>
<td>12. PSIGF example for function $f$</td>
<td>47</td>
</tr>
<tr>
<td>13. Functions Used by Algorithm 5</td>
<td>54</td>
</tr>
<tr>
<td>14. Definition table of the function $s_7$</td>
<td>56</td>
</tr>
<tr>
<td>15. Time performance of the algorithm</td>
<td>58</td>
</tr>
<tr>
<td>16. Pattern '001' Prevalence versus the CR$_{JZIP}$</td>
<td>64</td>
</tr>
<tr>
<td>17. Evolution of CR$_{JZIP}(seq_n)$</td>
<td>67</td>
</tr>
<tr>
<td>Table</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>18. Matrix Insertions, Entropy and Compression Ratios</td>
<td>71</td>
</tr>
<tr>
<td>19. Kronecker product and probability distribution for 4 matrices</td>
<td>71</td>
</tr>
<tr>
<td>20. Correlations between $\text{CR}<em>{iZIP}(A</em>{G_n})$ and $\mathcal{H}_P(G_n, k)$</td>
<td>80</td>
</tr>
<tr>
<td>21. Number of association rules at 0.05 support level and 0.9 confidence</td>
<td>81</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The Rymon tree for the $\mathcal{P}({1, 2, 3, 4})$</td>
<td>13</td>
</tr>
<tr>
<td>2.</td>
<td>Dependency of average time on number of tuples</td>
<td>20</td>
</tr>
<tr>
<td>3.</td>
<td>Number of minimal determining set for 8, 16 and 24 variables, as a function of the number of tuples.</td>
<td>21</td>
</tr>
<tr>
<td>4.</td>
<td>Average size of $\text{MDS}(f)$ for 8, 16 and 24 variables, as a function of the number of tuples.</td>
<td>21</td>
</tr>
<tr>
<td>5.</td>
<td>Dependency of average time on number of tuples and limiting factor for 8 variables.</td>
<td>37</td>
</tr>
<tr>
<td>6.</td>
<td>Dependency of average time on number of tuples and limiting factor for 16 variables.</td>
<td>38</td>
</tr>
<tr>
<td>7.</td>
<td>Dependency of average time on number of tuples and limiting factor for 24 variables.</td>
<td>38</td>
</tr>
<tr>
<td>8.</td>
<td>Average size of minimal determining set for 8 variables, as a function of the number of tuples and limiting factor.</td>
<td>39</td>
</tr>
<tr>
<td>9.</td>
<td>Average size of minimal determining set for 16 variables, as a function of the number of tuples and limiting factor.</td>
<td>40</td>
</tr>
<tr>
<td>10.</td>
<td>Average size of minimal determining set for 24 variables, as a function of the number of tuples and limiting factor.</td>
<td>40</td>
</tr>
<tr>
<td>11.</td>
<td>Baseline $\text{CR}_{ZIP}$ Behavior</td>
<td>63</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>12.</td>
<td>Variation of compression rate on the prevalence of the pattern '001'</td>
<td>65</td>
</tr>
<tr>
<td>13.</td>
<td>Dependency of Compression Ratio on Pattern Prevalence</td>
<td>65</td>
</tr>
<tr>
<td>14.</td>
<td>Compression Ratio Behavior of Thue-Morse Sequence</td>
<td>68</td>
</tr>
<tr>
<td>15.</td>
<td>Evolution CR$_{jZIP}$ vs. Shannon Entropy of Probability Distribution</td>
<td>72</td>
</tr>
<tr>
<td>16.</td>
<td>Standard deviation vs. average of the CR$<em>{jZIP}(A</em>{G_n})$ for a number of different permutations of nodes for the same graph.</td>
<td>75</td>
</tr>
<tr>
<td>17.</td>
<td>Standard deviation vs. average of the $H_P(G_n, k)$ of a number of different permutations of nodes for the same graph.</td>
<td>76</td>
</tr>
<tr>
<td>18.</td>
<td>Plots of average CR$<em>{jZIP}(A</em>{G_n})$ (CMP RTIO) and average $H_P(G_n, k)$ (DIST ENT) for randomly generated graphs $G_n$ of equal number of edges with respect to the number of edges. - $n = 60$ and $k = 3$.</td>
<td>77</td>
</tr>
<tr>
<td>19.</td>
<td>Plots of average CR$<em>{jZIP}(A</em>{G_n})$ (CMP RTIO) and average $H_P(G_n, k)$ (DIST ENT) for randomly generated graphs $G_n$ of equal number of edges with respect to the number of edges. - $n = 60$ and $k = 4$.</td>
<td>78</td>
</tr>
<tr>
<td>20.</td>
<td>Plots of average CR$<em>{jZIP}(A</em>{G_n})$ (CMP RTIO) and average $H_P(G_n, k)$ (DIST ENT) for randomly generated graphs $G_n$ of equal number of edges with respect to the number of edges. - $n = 60$ and $k = 5$.</td>
<td>79</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Data and Data Mining

Data Mining is the computational process of discovering patterns in large data sets. It is an interdisciplinary field of Computer Science, using methods from different fields like artificial intelligence, machine learning, statistics and database systems.

In science and engineering computers have become a necessary tool. Besides their role in controlling processes, computers allow accurate data recording. In the last few years, recording and storing data have become financially inexpensive and technically easy, allowing the rapid growth of datasets.

In 2007 scientists from the University of Southern California estimated that humankind had stored more than 295 billion gigabytes (or 295 exabytes) of information. A lot of this data was produced by recordings from different engineering areas. The existence of such large datasets is a motivation to incorporate the data mining techniques in engineering problems and to use data compression in the pre-mining process. In this dissertation, I introduce new optimization techniques inspired by data mining applicable in engineering problems. A novel approach in data mining is presented here: data compression is used in the data pre-mining process to quantify the mining potential of a dataset.

Data mining methods are being used in engineering when conventional approaches are too computationally expensive and unpractical. One of the problems that we have addressed is minimizing the number of inputs that partially defined functions depend on.
These particular types of functions model diverse circuits. When these circuits use fewer inputs, the size of these circuits and the manufacturing and utilization costs decrease substantially. This is the problem that we address and solve in Chapter 2.

Using an Apriori-like algorithm, we present two versions of the proposed method. The Apriori algorithm [AIS93] was developed to determine frequent item sets in transactional databases. The purpose of the Apriori algorithm is to mine a large collection of basket data type transactions for association rules between sets of items with a minimum specified confidence. An example of such association rule is the statement that 90% of transactions that purchase bread and butter will also purchase milk. The Apriori algorithm has one main principle: if an itemset is frequent, then all of its subsets must also be frequent. We used this idea and adapted it to find the determining sets of minimal cardinality, by searching for the possible variable combinations of input variables in a prioritized manner.

Another engineering problem that we solve using data mining is that of finding the optimal decomposition for specific functions encountered in engineering. These functions are called partially specified index generation functions (PSIGFs). The method I propose for decomposing the PSIGFs uses a greedy approach. Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit. Such an approach can be disastrous for some computational tasks, but there are many cases for which it is optimal, such as spanning trees. In Chapter 3 we combine the greedy method with the use of randomness in order to find an optimal decomposition of PSIGFs.

Data Compression has the role of shrinking a file so that it takes less space. The ultimate goal of data compression is to represent a source in digital form with as few bits as possible, while meeting the minimum requirement of reconstruction of the original. There are two main types of data compression: lossy and lossless. A compression method
is lossless if it is possible to reconstruct the exact original data from the compressed version. Lossless compression is called reversible compression, since the original can be recovered perfectly by decompression. A compression method is lossy if it is not possible to reconstruct the original exactly from the compressed version. Data such as multimedia images, audio and video are more easily compressed by lossy compression techniques, while most of the non-multimedia files require lossless compression algorithms. In Chapter 4 we will be using lossless compression.

In the next sections the background of both engineering problems are presented. I also discuss how data compression can be used in data mining.

1.2 Engineering Problems in Computer Science

Partially defined finite functions are studied by both mathematicians and engineers due to their applications, particularly in designing switching circuitry.

Partially defined finite functions are defined in Section 2.2. In a few words, totally defined finite functions are defined on a cartesian product, while partially defined functions are defined on a subset of a cartesian product. An example of a partially defined function is presented in Table 1. The partial function defined on all the pairs \((x_1, x_2)\), where \(x_1, x_2 \in \{0, 1\}\), and ranging in the set \(\{0, 1\}\).

Table 1: Tabular Representation of a Partial Function

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
From the tabular representation of the function (Table 1), we can see that the function presented does not have any value for the pair \((1,0)\). If the function were defined for all four possible values for the pair \((x_1,x_2)\), then the function defined would be a totally defined function.

In Section 2.3 we will use a built-in SQL function. Every implementation of the SQL standard includes at least the following build-in functions: \texttt{sum}, \texttt{avg}, \texttt{max}, \texttt{min} and \texttt{count}. In our research will be using the \texttt{count} function as well as the \texttt{group by} clause.

The \texttt{count} function can be used in several ways, explained in [ST95]:

- \texttt{count}(A) can be used to determine the number of nonull entries under attribute \(A\);
- \texttt{count(distinct A)} computes the number of distinct nonull values that occur under \(A\);
- \texttt{count(*)} determines how many rows exist in a table without duplicate elimination.

In our approach, we will be using a combination of \texttt{count(distinct A)} and \texttt{group by} clause.

The \texttt{group by} clause groups together the tuples of tables based on the common value of an attribute or group of attributes. Once a table has been conceptually partitioned into groups (using \texttt{group by}), the \texttt{select} construct that we used must return one or more atomic pieces of data for every group. Our approach implements the use of the following SQL query in Section 2.3:

\[
\texttt{select count(distinct y) from table group by x_1, x_2};
\]

Partially Specified Index Generating Functions (PSIGFs) are particular cases of the partially defined functions introduced earlier. An example of a PSIGF is presented in Table 2: a partial function defined on a subset of all pairs \((x_1,x_2)\), where \(x_1,x_2 \in \{0,1\}\), and ranging in the set \(\{1,2,3\}\).
Table 2: Tabular Representation of a Partial Function

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

We can see that for each row, $(x_1, x_2)$, the values of $y$ increase by 1. Note that there are no duplicate values of $y$, which make the PSIGFs bijections. As for applications, these functions are used in IP address tables, virus scan circuits, pattern matching and packet filtering. To solve the decomposition problem, we use a greedy randomized optimization.

1.3 Data Compression and Data Mining

Real world data is generally incomplete (i.e. lacks attribute values), noisy (errors, outliers) and inconsistent. This is the reason why, in data mining, data has to be preprocessed prior to applying any algorithms. Data preprocessing consists of several steps: data cleaning, data transformation and data reduction. Data compression has not been used extensively in data mining. Data compression can be regarded as one of the fundamental approaches to data mining, since the goal of the data mining is to “compress data by finding some structure in it” [Man00].

The role of compression in developing parameter-free data mining algorithms for anomaly detection, classification, and clustering was examined in [KLR04]. The size $C(x)$ of a compressed file $x$ is as an approximation of Kolmogorov complexity [CV05] and allows the definition of a pseudo-distance between two files $x$ and $y$ as

$$d(x, y) = \frac{C(xy)}{C(x) + C(y)}.$$  \hspace{1cm} (1.1)
We consider $xy$ the concatenation of the string $x$ with the string $y$. We notice that the value of $d(x,y)$ varies between 0.5 and 1. If the two strings are different, then the value will be closer to 1 and the more similar they are, the closer do they to 0.5. In practice, they will never be exactly either 0.5 or 1.

The compression algorithm that we use is Lempel-Ziv-Welch (LZW); the main idea presented in Chapter 4 works with most of the compression algorithms.

1.4 Thesis Organization

In Chapter 1 we are introducing the topics covered in the dissertation. Terminology and notation are given in Appendix ??, Appendix ?? contains the MATLAB code of the algorithm presented in Chapter 4. The main theme of this dissertation is presenting new and innovative data mining methods to solve engineering problems faster. We also show how we can use data compression as a pre-mining tool.

The contributions to Data Mining are contained in Chapter 2-4 and are summarized as follows.

In Chapter 2 we describe a new method for finding determining sets of minimal cardinality for partially defined functions. Two versions of the algorithm are proposed. The first search is an exhaustive version that finds all the determining sets of minimal cardinality. The second, probabilistic version, is faster and returns a minimal solution most of the time. The probabilistic version has the advantage of introducing a limiting factor to minimize the search space, making this approach suited for bigger problems. The algorithm can be easily parallelized.
In Chapter 3 we present a novel approach to finding the optimal decomposition of partially specified index generation functions (PSIGFs). The method described for decomposing the PSIGFs uses randomness and the greedy method.

In Chapter 4 we show that compression can be used as a tool to evaluate the potential of a data set to produce interesting results in a data mining process. This is a novel and ambitious idea, which has not been approached by the data mining community. The results show that data compression can be used to detect patterns in datasets.

Finally, in Chapter 5 we summarize the content of the dissertation.
CHAPTER 2

MINING PARTIALLY DEFINED FUNCTIONS

2.1 Introduction

In this chapter we describe a method to detect the minimal sets of variables that determine the values of a discrete partially defined function.

Partially defined finite functions are studied by both mathematicians and engineers due to their many technical applications, particularly in designing switching circuitry. They model diverse circuits as logical programmable arrays, or content addressable memory. The performance of such circuits (including wiring complexity, power dissipation, etc.) is heavily influenced by the number of arguments on which the function implemented by the circuit depends.

In this work we evaluate the minimal number of arguments that the function effectively depends on. To this end, two versions of an Apriori-like algorithm [AIS93] are proposed. The exhaustive version performs a prioritized search and returns an optimal solution every time. The probabilistic, faster, version uses the notion of entropy of a partition and introduces a limiting factor to restrict the search, thereby providing the option to reduce running time.

The problem of finding minimal determining sets for partially defined functions has been initially addressed in [Sas08a]. This issue is solved using an algebraic minimization algorithm that applies to functions that depend on a small number of variables. The approach presented in this chapter is distinct and involves techniques inspired by data
mining. Additionally, it has the advantage of not being linked to any value of the input or output radix of the partial function \( f \).

Experimental results are provided for both versions and the results demonstrate the efficiency of the algorithms for functions of 8, 16 and 24 variables. We also examine the effect of the limiting factor on the optimality of the algorithm for different sizes of partial functions.

The proposed method generates various sets of input variables on which a partial function depends effectively using an approach inspired by Apriori, a well-known data mining algorithm developed for determining frequent item sets in transactional databases [AIS93, MT97, ZH05]. The method is based on two main ideas: the partial order that is naturally defined on the set of partitions of a set and the fundamental observation that a superset of a determining set for a partially defined function \( f \) is itself a determining set for \( f \). The dual heredity property of determining sets is used to formulate the algorithm. The potential determining sets are evaluated whether they are a determining set using a SQL query in the first version. In the second version we use a semi-metric among the blocks in the kernel partitions of the subsets of the set of variables.

The rest of the chapter is organized as follows. In Section 2.2, the notion of determining set for a partial function is introduced and properties of these sets are examined. In Section 2.3, we describe the first version of the algorithm. In Section 2.4 the experimental results for the first version are presented. In Section 2.5 we describe the second approach of the algorithm and introduces the notion of limiting factor. In section 2.6 we show the results of the second approach. In section 2.7 a comparison between the two approaches is discussed. Finally, Section 2.8 presents conclusions.
2.2 Determining Sets for Partially Defined Functions

**Definition 2.2.1** Let \( n \) denote the finite set \( \{0, 1, \ldots, n-1\} \). The partial functions studied have as domain a subset of the finite set \( r^n \) and as range a subset of the finite set \( p \).

The positive natural numbers \( r \) and \( p \) are referred to as the *input radix* and the *output radix* of the function, respectively. The set of all such partial functions is denoted by \( PF(r^n, p) \).

If \( f \in PF(r^n, p) \) we denote by \( \text{Dom}(f) \) the domain of \( f \).

A partial function \( f \in PF(r^n, p) \) is specified as a table \( T_f \) having columns labeled by the argument variables \( x_1, \ldots, x_n \) and by the output variable \( y \). If \( f(a_1, \ldots, a_n) = b \) we have in the table \( T_f \) the \((n+1)\)-tuple \( t = (a_1, \ldots, a_n, b) \).

For example, in Table 3 we show a partial function defined on all triplets in \( 3^3 \) that contain at least two non-zero elements, and ranging in the set \( 4 \).

The number of rows of the table that represents a partial function defined on \( r^n \) can range between 0 and \( r^n \). Usually the number of rows of such a table is smaller than \( r^n \) and often this number is much smaller. Tuples \( (a_1, \ldots, a_n) \) that do not belong to the definition domain of \( f \) are considered as “don’t care“ tuples, that is, as input sequences that are unlikely to occur as inputs of the functions, or the output of the function for such inputs is indifferent to the designer.

For a tuple (also referred as row) \( t \) in \( T_f \) and a set of variables \( U \subseteq \{x_1, \ldots, x_n, y\} \) we denote by \( t[U] \) the *projection* of \( t \) on \( U \), that is, the restriction of \( t \) to the set \( U \). If \( U \) consists of one variable we denote the projection \( t[\{z\}] \) just by \( t[z] \).

**Definition 2.2.2** A set of variables \( V \subseteq \{x_1, \ldots, x_n\} \) is a determining set for the partial function \( f \) if for every two tuples \( t \) and \( s \) from \( T_f \), \( t[V] = s[V] \) implies \( t[y] = s[y] \).
In other words, \( V = \{x_{i_1}, \ldots, x_{i_p}\} \) is a determining set for the partial function \( f \) if \( t = (a_1, \ldots, a_n, b) \) and \( s = (c_1, \ldots, c_n, d) \) in \( T_f \) such that \( a_{i_k} = c_{i_k} \) for \( 1 \leq k \leq p \) implies \( b = d \). The collection of determining sets for \( f \) is denoted by \( \text{DS}(f) \).

A set \( V \) is a **minimal determining set for** \( f \) if \( V \) is a determining set for \( f \) and there is no proper subset of \( V \) that is a determining set for \( f \). The set of minimal determining sets of \( f \) is denoted by \( \text{MDS}(f) \). Our main purpose is to present an algorithm that finds the determining sets of minimal cardinality for a given partially specified function.

We introduce a partial order relation "\( \sqsubseteq \)" on the set of partial \( \text{PF}(\mathbb{R}^n, \mathbf{p}) \) by defining \( f \sqsubseteq g \) if \( \text{Dom}(f) \subseteq \text{Dom}(g) \) and \( f(a_1, \ldots, a_n) = g(a_1, \ldots, a_n) \) for every \( (a_1, \ldots, a_n) \). In other words, we have \( f \sqsubseteq g \) if \( g \) is an extension of \( f \) or \( f \) is a restriction of \( g \).
The following simple statement is crucial to the proposed algorithm.

**Theorem 2.2.3** Let \( f \) and \( g \) be two partial functions in \( PF(r^n,p) \). If \( V \in DS(f) \) and \( V \subseteq W \), then \( W \in DS(f) \). Furthermore, if \( f \sqsubseteq g \), then \( DS(g) \subseteq DS(f) \).

**Proof** If \( V \) and \( W \) are two sets of variables such that \( V \subseteq W \) and \( t,s \) are two tuples in \( T_f \), then \( t[W] = s[W] \) implies \( t[V] = s[V] \). Therefore, if \( V \) is a determining set for \( f \) and \( t[W] = s[W] \), it follows that \( t[V] = s[V] \), which implies \( t[y] = s[y] \). Thus, \( W \) is a determining set for \( f \).

For the second part of the theorem, observe that if \( f \sqsubseteq g \) and \( V \in DS(g) \), then \( t[V] = s[V] \) implies \( t[y] = s[y] \), for every \( t,s \in T_f \). Since \( Dom(f) \subseteq Dom(g) \), the same implication holds for any two tuples in \( Dom(f) \), so \( V \in DS(f) \).

Note that if \( f \sqsubseteq g \) and \( V \in MDS(g) \), then there exists \( Z \in MDS(f) \) such that \( Z \subseteq V \).

### 2.3 An Apriori-like Algorithm for Mining MDSs

The algorithm uses Rymon trees \([Rym92, SD08, Try89]\) also known as set enumeration trees, a tool that is useful for the systematic enumeration of the subsets of a finite set \( S \). The subsets of a set \( S \) (which constitute the power set of \( S \), \( \mathcal{P}(S) \)) are listed using a pre-imposed total order on the underlying set \( S \).

The total order on \( S \) is specified by an one-to-one function \( \text{ind} : S \rightarrow \mathbb{N} \), where \( \mathbb{N} \) is the set of natural numbers.

**Definition 2.3.1** A Node’s View

\[
\text{view}(\text{ind},U) = \{ s \in S \mid \text{ind}(s) > \max_{u \in U} \text{ind}(u) \}
\]
Definition 2.3.2 Let $\mathcal{F}$ be a collection of sets closed under inclusion (i.e., if $U \in \mathcal{F}$ and $U \subseteq V$, then $V \in \mathcal{F}$).

The labeled tree $\mathcal{T}$ is a Rymon tree for $\mathcal{F}$ if the root of $\mathcal{T}$ is labeled by the empty set $\emptyset$, and the children of a node labeled $U$ in $\mathcal{T}$ are $\{U \cup \{e\} \in \mathcal{T} | e \in \text{view}(\text{ind}, U)\}$.

In Figure 1 we show the Rymon tree for the complete power set of the set $S = \{1, 2, 3, 4\}$ with the obvious self indexing.

The proposed algorithm takes as input a partially defined function $f$ and outputs a collection of sets with minimum number of variables that $f$ depends on. The algorithm performs a breadth-first search on the Rymon tree for the power-set of the set of variables $E = \{x_1, x_2, \ldots, x_n\}$ of $f$. The search stops when all the sets with the minimum number of
variables that function \( f \) depends on are found; these sets are referred to as determining sets. The minimum number corresponds to the lowest level in the Rymon tree where the first solution set is found since all the levels below have nodes containing sets with a higher number of variables than any level above it.

In Algorithm 1 we denote the breadth first search queue by \( Q \), the array of children of a node \( X \) by \( \text{Child}[X] \), and the tree level of the determining sets by \( d\text{Level} \).

The algorithm is using the following methods:

- \( \text{ENQUEUE}(Q, V) \) inserts node \( V \) in queue \( Q \);
- \( \text{DEQUEUE}(Q) \) removes the first element of queue \( Q \);
- \( \text{LEVEL}(V) \) returns the level of node \( V \) in the tree;
- \( \text{IS\_DSET}(V) \) informs if the set of variables corresponding to node \( V \) is a determining set for the partially defined function \( f \);
- \( \text{ADD}(\mathcal{D}, V) \) add the set of variables corresponding to node \( V \) to \( \mathcal{D} \).

The core of the algorithm is the procedure \( \text{IS\_DSET}(V) \) that has as an input argument a set of variables \( V \) and returns \texttt{true} if \( V \) is a determining set and \texttt{false}, otherwise. In principle, if \( T_f \) is a database table, the implementation of \( \text{IS\_DSET} \) could be done using embedded SQL by running the query:

\[
\text{select count(different y) from } T_f \text{ group by } V.
\]

**Example 2.3.1** Let \( f: 2^3 \rightarrow 3 \) be the partial function specified by Table 4. When we perform the following query:

\[
\text{select count(different y) from } T_f \text{ group by } x1, x2;
\]

the result set returned by the SQL query has 3 elements, \( \{2, 1, 1\} \). So, we can conclude that the partially defined function is not determined by \( \{x_1, x_2\} \).
Algorithm 1: Computing MDS($f$)

**Input:** A partially defined function $f$

**Output:** A collection $\mathcal{D}$ of minimal determining variables sets

begin

dLevel $\leftarrow \infty$

$\mathcal{D} = \emptyset$; ENQUEUE($Q, \emptyset$)

while $Q \neq \emptyset$ do

\hspace{1em} $X \leftarrow$ DEQUEUE($Q$)

\hspace{2em} foreach $V \in \text{Child}[X]$ do

\hspace{3em} ENQUEUE($Q, V$)

\hspace{2em} if $\mathcal{D} = \emptyset$ or $\text{LEVEL}(V) \leq d\text{Level}$ then

\hspace{3em} /* IS\_DSET returns true if the node $V$ corresponds to
\hspace{3em} a determining set */

\hspace{4em} if $\text{IS\_DSET}[V]$ then

\hspace{5em} ADD($\mathcal{D}, V$)

\hspace{5em} if $d\text{Level} = \infty$ then

\hspace{6em} $d\text{Level} = \text{LEVEL}(V)$

\hspace{5em} else

\hspace{6em} break

end
Table 4: Tabular Representation of a Partial Function

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

In the case of the following query:

$$\text{select count(distinct } y) \text{ from } T_f \text{ group by } x_1, x_3;$$

the result set returned by the SQL query has 3 elements, \{1, 1, 1\}. So, we can conclude that the set \{x_1, x_3\} is a determining set.

It is clear that if all values returned for \text{count(distinct } y) are equal to 1, then $V$ is a determining set for $f$. In practice, the overhead entailed by using the database facility impacts negatively on the performance of the algorithm, which leads us to another solution that is based on storing $T_f$ as a file and searching that file.

The procedure $IS\_DSET$ shown in Algorithm 2 makes use of a hash table $MAP$, where the key is determined by the chosen set of variables. The following methods are used:
\textit{GET\_VARIABLES}(V): retrieves the set of variables corresponding to node V

\textit{GET\_VALUES}(\text{tuple}, S): retrieves the values of the variables in S

\textit{ELEMENT}(\text{MAP}, \text{key}): returns the object instance stored in \text{MAP} that contains a certain \text{key}

\textit{GET\_FVALUE}(y): returns the function value of the object instance y

\textit{F}(\text{tuple}): returns the function value of a certain tuple

\textit{ADD}(\text{MAP}, \text{key}, \text{F}(\text{tuple})): adds an object instance containing a key and function value to the MAP.

\textbf{Algorithm 2:} Procedure IS\_DSET(V)

\textbf{Input}: A node containing a subset of the variables set

\textbf{Output}: true if the set is a determining one, false, otherwise

\textbf{begin}

\hspace{10pt} S \leftarrow \text{GET\_VARIABLES}(V)

\hspace{10pt} \textbf{foreach} \hspace{5pt} \text{tuple} \in \text{File} \hspace{5pt} \textbf{do}

\hspace{20pt} \text{key} \leftarrow \text{GET\_VALUES}(\text{tuple}, S)

\hspace{20pt} \textbf{if} \hspace{5pt} \text{key} \in \text{MAP} \hspace{5pt} \textbf{then}

\hspace{30pt} \text{y} \leftarrow \text{ELEMENT}(\text{MAP}, \text{key})

\hspace{30pt} \textbf{if} \hspace{5pt} \text{F}(\text{tuple}) \neq \text{GET\_FVALUE}(y) \hspace{5pt} \textbf{then}

\hspace{40pt} \text{return false}

\hspace{40pt} \text{break}

\hspace{30pt} \textbf{else}

\hspace{40pt} \text{ADD}(\text{MAP}, \text{key}, \text{F}(\text{tuple}))

\hspace{20pt} \textbf{return} \hspace{5pt} \text{true}

\textbf{end}

The following variables are used in the IS\_DSET procedure:
2.4 Experimental Results

We carried out experiments on a Windows Vista 64-bit machine with 8Gb RAM and 2 × Quad Core Xeon Proc E5420, running at 2.50 GHz with a 2×6Mb L2 cache. The program was written in Java 6.

We analyze the results in terms of running time, minimum number of variables of a determining set, and the number of determining sets as functions of the number of tuples in $T_f$.

A program that randomly generates comma separated text files representing partially defined functions with 8, 16 or 24 variables was developed. The values 8, 16 and 24 were chosen based on the experiments made in the related work of T. Sasao [Sas08a].

One hundred files (functions) were randomly generated for each type of partially defined function (with 8, 16, and 24 variables) using an input radix $r = 3$ and an output radix $p = 5$.

Note that a totally defined function with 8 variables and $r = 3$ has $3^8 = 6561$ tuples. In our experiments, we randomly generated up to 1000 tuples for partially defined functions with 8 variables. For functions that depend on 16 and 24 arguments we generated up to
5000 tuples because the number of tuples for completely defined functions with 16 or 24 variables is much higher.

In the experiments, we evaluate the performance of the algorithm with a varying number of tuples. By Theorem 2.2.3, if \((f_1, f_2, \ldots, f_k)\) is a sequence of functions such that

\[ f_1 \sqsubseteq f_2 \sqsubseteq \ldots \sqsubseteq f_k, \]

we have

\[ \text{DS}(f_k) \subseteq \ldots \subseteq \text{DS}(f_2) \subseteq \text{DS}(f_1). \]

In other words, when we start with a partial function \(f_k\) with a small specification table \(T_{f_k}\) and we expend sequentially the specification of the functions, the number of determining sets will decrease where \(k\) is the number of rows of the table \(T_{f_k}\). In the experiments, we are comparing the results for files with 8, 16 and 24 variables and show the averages of the values corresponding to time, minimum number of variables the function depends on, and number of sets with minimum number of elements the function depends on as a function of the number of tuples. In our case, \(k \in \{10, 15, 20, 30, 40, 50, 75, 90, 100, 200\}\). The averages are evaluated over 100 functions within each group of generated functions (8, 16 and 24 variables).

As shown in Figure 2, the running time of the algorithm to find a solution increases with the number of tuples because in most cases, the algorithm needs to search deeper in the Ryman tree. Also, the time increases very fast in rapport to the number of variables. The algorithm performs a breadth-first search and functions with more variables will have trees with a larger branching factor.

Figure 3 shows that the minimum number of variables the function depends on is related to the number of tuples \(k\). As the number of rows increases, the constraints
imposed on the function becomes more extensive, and the minimum number of variables that determine the value of the function increases.

Finally, the experiments also show that the average number of minimal determining sets decreases as we extend the partial functions by introducing more tuples. Figure 4 illustrates this decrease for functions with 8 and 16 variables. The decrease is not as noticeable for functions with 24 variables because these functions have a large number of possible tuples and this behavior can only be observed for a much higher value of $k$ than the maximum used in experiments presented here.

In the next section we propose a probabilistic approach using the notion of entropy of a partition.

### 2.5 Entropies Associated with Partial Functions

Entropy is a probabilistic concept that lies at the foundation of information theory. Generalized entropy was introduced in [HCH67, Dar70]. The entropy can be used to...
Figure 3: Number of minimal determining set for 8, 16 and 24 variables, as a function of the number of tuples.

Figure 4: Average size of MDS($f$) for 8, 16 and 24 variables, as a function of the number of tuples.
quantify the distribution of elements of a set, depending on the set’s partition. In [SD08] a
generalized notion of entropy for partitions, with Shannon entropy as a special case, is
introduced.

A partition on a set \( S \) is a collection
\[
\pi = \{B_1, \ldots, B_m\}
\]
of nonempty, pairwise disjoint sets such that \( \bigcup_{i=1}^{m} B_i = S \). The sets \( B_i \) are referred to as the
blocks of \( \pi \). The set of partitions of \( S \) is denoted by \( \text{PART}(S) \).

If \( \pi, \rho \in \text{PART}(S) \) we say that \( \pi \preceq \rho \) if every block of \( \pi \) is included in a block of \( \rho \);
equivalently, \( \pi \preceq \rho \) if every block of \( \rho \) is a union of blocks of \( \pi \). For \( \pi, \rho \in \text{PART}(S) \) we
say that \( \rho \) covers \( \pi \) if \( \pi \preceq \rho \) and there is no \( \sigma \in \text{PART}(S) \) distinct from \( \pi \) and \( \rho \) such that
\( \pi \preceq \sigma \preceq \rho \). This is denoted by \( \pi \prec \rho \). It can be shown that \( \pi \prec \rho \) if the blocks of \( \rho \)
coincide with the blocks of \( \pi \), with the exception of one block of \( \rho \) which is the union of
two blocks of \( \pi \).

The partially ordered set \((\text{PART}(S), \preceq)\) is actually a lattice. If \( \pi, \rho \in \text{PART}(S) \),
\( \pi = \{B_1, \ldots, B_m\} \) and \( \rho = \{C_1, \ldots, C_n\} \), the greatest lower bound of \( \pi \) and \( \rho \) is the
partition \( \pi \wedge \rho \) given by
\[
\pi \wedge \rho = \{B_i \cap C_j \mid B_i \cap C_j \neq \emptyset\}.
\]

If \( C \) is a subset of \( S \) and \( \pi = \{B_1, \ldots, B_m\} \in \text{PART}(S) \), the trace of \( \pi \) on \( C \) is the
partition consisting of the non-empty sets \( \{C \cap B_1, \ldots, C \cap B_m\} \in \text{PART}(C) \). Unless stated
otherwise, all logarithms are in base 2. In order to avoid the empty sets in the following
computations, we introduce the following function:

\[
\delta(x) = \begin{cases} 
-x \log x & \text{if } x > 0 \\
0 & \text{if } x = 0
\end{cases}
\]
**Definition 2.5.1** Let $S$ be a finite set and let $\pi = B_1, \ldots, B_n$ be a partition of $S$. The Shannon entropy of $\pi$ is the number:

$$H(\pi) = \sum_{i=1}^{m} \delta \left( \frac{|B_i|}{|S|} \right).$$

The Shannon entropy can be used to evaluate the uniformity of the distribution of elements of $S$ in the blocks $\pi$ since the entropy value increases with the uniformity of the distribution of the elements of $S$.

In [SJ02, JS99] it is shown that the entropy of a partition is a dually monotonic function. In other words, if $\pi \preceq \rho$, we have $H(\pi) \geq H(\rho)$.

If $\pi, \rho \in \text{PART}(S)$, $\pi = \{B_1, \ldots, B_n\}$ and $\rho = \{C_1, \ldots, C_n\}$, then the conditional entropy of $\pi$ on $\rho$ is the number

$$H(\pi|\rho) = \sum_{j=1}^{n} \frac{|C_j|}{|S|} H(\pi_{C_j}),$$

that is, the weighted average of the entropies of the traces of $\pi$ on the blocks of $\rho$.

An equivalent expression of the conditional entropy can be obtained as follows:

$$H(\pi|\sigma) = \sum_{j=1}^{n} \left( \frac{|C_j|}{|S|} H(\pi_{C_j}) \right)$$

$$= \sum_{j=1}^{n} \frac{|C_j|}{|S|} \sum_{i=1}^{m} \delta \left( \frac{|B_i \cap C_j|}{|C_j|} \right)$$

$$= \sum_{j=1}^{n} \sum_{i=1}^{m} \frac{|B_i \cap C_j|}{|S|} \log \frac{|B_i \cap C_j|}{|C_j|}$$

$$= \sum_{j=1}^{n} \sum_{i=1}^{m} \frac{|B_i \cap C_j|}{|S|} \log \frac{|B_i \cap C_j|}{|S|} \log \frac{|C_j|}{|S|}$$

$$= \sum_{j=1}^{n} \sum_{i=1}^{m} \frac{|B_i \cap C_j|}{|S|} \log \frac{|B_i \cap C_j|}{|S|} - \sum_{j=1}^{n} \sum_{i=1}^{m} \frac{|B_i \cap C_j|}{|S|} \log \frac{|C_j|}{|S|}$$

$$= \mathcal{H}(\pi \wedge \rho) - \mathcal{H}(\rho).$$
The algorithm proposed in this section uses the conditional entropy of partitions defined as follows. If \( U, V \) are two sets of attributes, the entropy of \( U \) conditioned upon \( V \) is the difference

\[
H(U|V) = H(UV) - H(V).
\] (2.1)

where \( UV \) is the union of the attribute sets \( U \) and \( V \).

The monotonicity of \( H \) implies that the function \( H(\cdot | \cdot) : \text{PART}(\text{Var}_f)^2 \rightarrow \mathbb{R} \) is monotonic increasing in its first argument \( U \).

**Definition 2.5.2** Let \( f \) be a partial function, \( f \in \text{PF}(\mathbb{R}^{n}, p) \) and let \( V \) be a set of variables of \( f, V \subseteq \{ x_1, \ldots, x_n, y \} \). Define the partition \( \pi^V \) of \( \text{Dom}(f) \) by its corresponding equivalence \( \sim_V \), where \( u \sim_V w \) if \( u[V] = w[W] \).

**Definition 2.5.3** The entropy \( H(V) \) of \( V \) is the entropy \( H(\pi^V) \) of the partition \( \pi^V \).

**Lemma 2.5.1** Let \( S \) be a set and let \( C, D \) be two disjoint, non-empty subsets of \( S \). If \( \pi \in \text{PART}(S) \), then

\[
H(\pi_{C \cup D}) \geq \frac{|C|}{|C \cup D|} H(\pi_{C}) + \frac{|D|}{|C \cup D|} H(\pi_{D}).
\]

**Proof** Let \( \pi = \{ B_1, \ldots, B_m \} \in \text{PART}(S) \).

Since the function \( f : [0, 1] \rightarrow \mathbb{R} \) given by \( f(x) = -x \log x \) for \( x \in (0, 1] \) and \( f(x) = 0 \) for \( x = 0 \) is concave, we have

\[
-ap \log p - bq \log q \leq -(ap + bq) \log (ap + bq),
\]

for every non-negative numbers \( a, b \) such that \( a + b = 1 \) and every \( p, q \in (0, 1] \).

If we choose \( a = \frac{|C|}{|C \cup D|}, \quad b = \frac{|D|}{|C \cup D|}, \quad p_i = \frac{|B_i \cap C|}{|C|}, \quad \text{and} \quad q_i = \frac{|B_i \cap D|}{|D|}, \) note that

\[
ap_i + bq_i = \frac{|B_i \cap (C \cup D)|}{|C \cup D|}.
\]
Thus, we obtain the inequality:

\[
- \frac{|C|}{|C \cup D|} \frac{|B_i \cap C|}{|C|} \log \frac{|B_i \cap C|}{|C|} - \frac{|D|}{|C \cup D|} \frac{|B_i \cap D|}{|D|} \log \frac{|B_i \cap D|}{|D|} \\
\leq - \frac{|B_i \cap (C \cup D)|}{|C \cup D|} \log \frac{|B_i \cap (C \cup D)|}{|C \cup D|}.
\]

If we sum up these inequalities for \(1 \leq i \leq n\), the inequality of the lemma follows immediately.

**Theorem 2.5.4** Let \(S\) be a set and let \(\pi, \sigma \in PART(S)\). The conditional entropy \(\mathcal{H}(\pi|\sigma)\) is monotonic relative to \(\sigma\).

**Proof** It suffices to show that if \(\sigma \prec \sigma'\), then \(\mathcal{H}(\pi|\sigma) \leq \mathcal{H}(\pi|\sigma')\). Without loss of generality we can assume that \(\sigma = \{C_1, \ldots, C_{n-2}, C_{n-1}, C_n\}\) and \(\sigma' = \{C_1, \ldots, C_{n-2}, C_{n-1} \cup C_n\}\).

In view of the definition of conditional entropy we need to prove that

\[
\frac{|C_{n-1}|}{|S|} \mathcal{H}(\pi_{C_{n-1}}) + \frac{|C_n|}{|S|} \mathcal{H}(\pi_{C_n}) \leq \frac{|C_{n-1} \cup C_n|}{|S|} \mathcal{H}(\pi_{C_{n-1} \cup C_n}),
\]

and this follows immediately from Lemma 2.5.1.

Note that if \(V\) and \(V'\) are two sets of variables such that \(V \subseteq V'\), then \(\pi^V \leq \pi^{V'}\). Thus, \(V \subseteq V'\) implies \(\mathcal{H}(V) \geq \mathcal{H}(V')\), so the entropy is monotonic with respect to inclusion of attribute sets.

The role of the conditional entropy in detecting determining sets is highlighted by the next result.

**Theorem 2.5.5** Let \(f \in PF(r^d, p)\) be a partial function, and \(U\) a set of variables of \(f\). Then \(X\) is a determining set of \(f\) if and only if \(\mathcal{H}(y|X) = 0\).
Table 5: Partially Defined Function Example

<table>
<thead>
<tr>
<th>x₁</th>
<th>x₂</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Proof Suppose that $X$ is a set of variables such that $\mathcal{H}(y|X) = 0$ and that $\pi^X = \{C_1, \ldots, C_n\}$. Then, by the definition of conditional entropy we have

$$\sum_{j=1}^{n} \frac{|C_j|}{|S|} \mathcal{H}(\pi^y_{C_j}) = 0,$$

which implies $\mathcal{H}(\pi^y_{C_j}) = 0$ for $1 \leq j \leq n$. Thus, each block $C_j$ of $\pi^X$ is included in a block of $\pi^y$, so the values of the variables in $X$ determine the value of the output variable $y$. Therefore, $X$ is a determining set.

Conversely, if $X$ is a determining set, $\pi^X \leq \pi^y$, so

$$\mathcal{H}(y|X) = \mathcal{H}(\pi^y \wedge \pi^X) - \mathcal{H}(\pi^X) = 0,$$

because $\pi^y \wedge \pi^X = \pi^X$.  

Example 2.5.2 Let $f : 3^2 \to 3$ be the partial function specified by Table 5.
The values of $\mathcal{H}(y|U)$ for the subsets $U_1 = \{x_1\}$ and $U_2 = \{x_2\}$ indicate that $U_1$ is a determining set for $f$, while $U_2$ is not.

\[
\begin{align*}
\mathcal{H}(yx_1) &= -\frac{2}{6} \log \frac{2}{6} - \frac{1}{3} \log \frac{1}{3} \approx 1.585 \\
\mathcal{H}(x_1) &= -\frac{2}{6} \log \frac{2}{6} - \frac{1}{3} \log \frac{1}{3} \approx 1.585 \\
\mathcal{H}(y|x_1) &= \mathcal{H}(y,x_1) - \mathcal{H}(x_1) = 0 \\
\mathcal{H}(yx_2) &= -\frac{1}{6} \log \frac{1}{6} - \frac{1}{6} \log \frac{1}{6} \approx 2.585 \\
\mathcal{H}(x_2) &= -\left(\frac{3}{6} \log \frac{3}{6} + \frac{2}{6} \log \frac{2}{6} + \frac{1}{6} \log \frac{1}{6}\right) \approx 1.46 \\
\mathcal{H}(y|x_2) &= \mathcal{H}(yx_2) - \mathcal{H}(x_2) \approx 2.585 - 1.46 \approx 1.125.
\end{align*}
\]

The algorithm 7 uses the conditional entropy $\mathcal{H}(y|X)$ previously described to find determining sets. It starts evaluating $\mathcal{H}(y|X)$ for single element subsets $X = x$ and increases the size of the subsets with each successive iteration. In this way, all possible subsets of variables with size $\alpha$ are evaluated before any subset of $S$ with size $\beta > \alpha$. The algorithm is not redundant because it does not evaluate $\mathcal{H}(y|X)$ for a subset of variables $X$ more than once. For instance, if both subsets $X_1 = \{1\}$ and $X_2 = \{2\}$ are expanded in a subsequent iteration, then the subset $s_3 = \{1,2\}$ common to both expansions will be evaluated once.

The proposed algorithm takes as input a partially defined function $f$ and a limiting factor $\ell$ in the range $(0,1]$ used to reduce the search space. The output is a collection of determining sets for $f$. The algorithm performs a search on the power-set of the set of variables $V = x_1 x_2 \ldots x_n$ of $f$. The limiting factor $\ell$ determines the subsets of variables that are expanded among all subsets within a given size. Namely, $\ell$ is the fraction of subsets $X$ with equal size and lowest $\mathcal{H}(y|X)$ value. When the limiting factor equals 1, all the possible subsets are evaluated until one or more solution sets with equal size are found. The search stops after all the sets if cardinality $|X|$ are evaluated and $\mathcal{H}(y|X) = 0$; these
sets are referred to as determining sets for \( f \). The minimum number found corresponds to the size of the first solution set since the search proceeds with increasing the order of the subsets size and all the remaining subsets that have not been checked have a greater size.

**Example 2.5.3** Table 6 shows another example of a partially defined function.

The values of conditional entropies involved are given by:

\[
\begin{align*}
\mathcal{H}(y|x_1) & = \mathcal{H}(yx_1) - \mathcal{H}(x_1) = 7.229, \\
\mathcal{H}(y|x_2) & = \mathcal{H}(yx_2) - \mathcal{H}(x_2) = 12.381, \\
\mathcal{H}(y|x_3) & = \mathcal{H}(yx_3) - \mathcal{H}(x_3) = 6.703, \\
\mathcal{H}(y|x_4) & = \mathcal{H}(yx_4) - \mathcal{H}(x_4) = 7.229, \\
\mathcal{H}(y|x_3x_1) & = \mathcal{H}(yx_3x_1) - \mathcal{H}(x_3x_1) = 0, \\
\mathcal{H}(y|x_4x_1) & = \mathcal{H}(yx_4x_1) - \mathcal{H}(x_4x_1) = 0.
\end{align*}
\]
When the limiting factor takes a value equal to 0.25, the algorithm only indicates $x_1x_3$ as a determining set for the partially defined function presented in Table 6. Note that $x_1x_4$ is also a determining set for the given function and is part of the solution when the limiting factor takes a value that is at least 0.5.

Table 7 and Table 8 present the methods and variables used in the algorithm.
Table 7: Methods for $\text{ComputingMDS}(f, \ell)$

<table>
<thead>
<tr>
<th>Methods</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD($L,X$)</td>
<td>Inserts set $X$ in the list of sets $L$</td>
</tr>
<tr>
<td>REMOVE($L, idx$)</td>
<td>Removes an element(set) at index $idx$ from the list of sets $L$</td>
</tr>
<tr>
<td>CLEAR($L$)</td>
<td>Removes all elements(sets) from the list of sets $L$</td>
</tr>
<tr>
<td>NEXT($L$)</td>
<td>Gets the next element(set) of the list of sets $L$</td>
</tr>
<tr>
<td>GET($L, idx$)</td>
<td>Gets the element(set) at index $idx$ on the list of sets $L$</td>
</tr>
<tr>
<td>INDEX_MIN($E$)</td>
<td>Gets the index of the first element on the list $E$ with lowest conditional entropy value</td>
</tr>
<tr>
<td>SIZE($X$)</td>
<td>Gets the size of the set $X$</td>
</tr>
<tr>
<td>COLLECTIONADD($\mathcal{D}, X$)</td>
<td>Inserts set $X$ in the collection of determining sets $\mathcal{D}$</td>
</tr>
<tr>
<td>BINOMIAL($l, p$)</td>
<td>Evaluates the binomial coefficient $l$ chooses $p$, which will be multiplied by the limiting factor $\ell$ in order to find the fraction of sets of variables with lowest entropy with a certain size $p$; it takes into consideration the size $l$ of the complete set of variables of $f$</td>
</tr>
<tr>
<td>COMPUTE_ENTROPY($X$)</td>
<td>Evaluates the entropy value of a set of variables $X$</td>
</tr>
</tbody>
</table>
### Table 8: Variables for Computing MDS($f, \ell$)

<table>
<thead>
<tr>
<th>Variables</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>The set of variables of a partially defined function $f$</td>
</tr>
<tr>
<td>SETLIST</td>
<td>A list of sets of variables</td>
</tr>
<tr>
<td>LOW ENTROPY SETLIST</td>
<td>A list of sets of variables with the lowest entropy values among the ones in SETLIST</td>
</tr>
<tr>
<td>ENTRLST</td>
<td>A list of entropy values</td>
</tr>
<tr>
<td>$\ell$</td>
<td>A number in (0,1] corresponding to the limiting factor</td>
</tr>
<tr>
<td>$D$</td>
<td>A collection of determining sets</td>
</tr>
<tr>
<td>set_size</td>
<td>The size of a subset</td>
</tr>
<tr>
<td>dset_founded</td>
<td>Indicates whether at least one determining set of variables was found</td>
</tr>
<tr>
<td>var</td>
<td>A variable of $f$</td>
</tr>
<tr>
<td>$X$</td>
<td>A subset of variables of $S$</td>
</tr>
<tr>
<td>entropy</td>
<td>An entropy value</td>
</tr>
<tr>
<td>$lf$</td>
<td>A fraction of the total number of sets of variables with a certain size that have lowest entropy values</td>
</tr>
<tr>
<td>index</td>
<td>An index of a list</td>
</tr>
</tbody>
</table>
Algorithm 3: Computing MDS \((f, \ell)\)

**Input:** A partially defined function and a limiting factor

**Result:** A collection \(\mathcal{D}\) of determining variables sets

begin
\[
\mathcal{D} \leftarrow \emptyset, \text{set}_\text{size} \leftarrow 1, \text{dset}_\text{found} \leftarrow \text{false}
\]
foreach \(\text{var} \in S\) do
\[
X \leftarrow \text{var}, \text{ADD}(\text{SETLIST}, X)
\]
\[
\text{entropy} \leftarrow \text{COMPUTE_ENTROPY}(X), \text{ADD}(\text{ENTR}_\text{LST}, \text{entropy})
\]
if \(\text{entropy} = 0\) then
\[
\text{COLLECTIONADD}(\mathcal{D}, X), \text{dset}_\text{found} \leftarrow \text{true}
\]
while \(\text{set}_\text{size} \leq \text{SIZE}(S)\) and \(\text{dset}_\text{found} = \text{false}\) do
\[
\ell f \leftarrow \ell \ast \text{BINOMIAL}(\text{SIZE}(S), \text{set}_\text{size})
\]
repeat
\[
\text{index} \leftarrow \text{INDEX_MIN}(\text{ENTR}_\text{LST}), X \leftarrow \text{GET}(\text{SETLIST}, \text{index})
\]
\[
\text{ADD}(\text{LOW}_\text{ENTROPY}_\text{SETLIST}, X)
\]
\[
\text{REMOVE}(\text{SETLIST}, \text{index}), \text{REMOVE}(\text{ENTR}_\text{LST}, \text{index})
\]
\[
\ell f \leftarrow \ell f - 1
\]
until \(\ell f = 0\)
\[
\text{CLEAR}(\text{SETLIST}), \text{CLEAR}(\text{ENTR}_\text{LST})
\]
\[
\text{set}_\text{size} \leftarrow \text{set}_\text{size} + 1
\]
while \(\text{LOW}_\text{ENTROPY}_\text{SETLIST} \neq \emptyset\) do
\[
X \leftarrow \text{NEXT}(\text{LOW}_\text{ENTROPY}_\text{SETLIST})
\]
foreach \(\text{var} \in S\) do
\[
\text{if } X \cup \text{var} \ni \text{SETLIST} \text{ and } \text{var} \ni X \text{ then}
\]
\[
\text{ADD}(\text{SETLIST}, X \cup \text{var})
\]
\[
\text{entropy} \leftarrow \text{COMPUTE_ENTROPY}(X \cup \text{var})
\]
\[
\text{ADD}(\text{ENTR}_\text{LST}, \text{entropy})
\]
if \(\text{entropy} = 0\) then
\[
\text{COLLECTIONADD}(\mathcal{D}, X \cup \text{var})
\]
\[
\text{dset}_\text{found} \leftarrow \text{true}
\]
end

32
Table 9: Methods for COMPUTE\_ENTROPY($\mathcal{H}(y|X)$)

<table>
<thead>
<tr>
<th>Methods</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_XKEYS($v,X$)</td>
<td>Gets the values assigned in a registered vector $v$ of a partial function $f$ to the set of variables $X$</td>
</tr>
<tr>
<td>GET_YXKEYS($v,X$)</td>
<td>Gets the values assigned in a registered vector $v$ of a partial function $f$ to the set of variables $X$ and the function value $y$</td>
</tr>
<tr>
<td>GET($M,k$)</td>
<td>Gets the number of occurrences of the key $k$ in the map $M$</td>
</tr>
<tr>
<td>ASSIGN($M,k,o$)</td>
<td>Maps the key $k$ to the number of occurrences $o$ in the map $M$</td>
</tr>
<tr>
<td>ADD($M,k,o$)</td>
<td>Inserts the mapping with key $k$ and number of occurrences $o$ to the hash structure $M$</td>
</tr>
<tr>
<td>ENTROPY($M$)</td>
<td>Evaluates the entropy of the sets of variables in map $M$</td>
</tr>
</tbody>
</table>

The core of the algorithm is the function COMPUTE\_ENTROPY($f,X$) that has as input arguments a partial function $f$ and a set of variables $X$. This function returns the Shannon Entropy corresponding to $X$. The function is presented in Tables 9 and 10 define the methods and variables used in function COMPUTE\_ENTROPY($f,X$).
Table 10: Variables for COMPUTE_ENTROPY\((f,X)\)

<table>
<thead>
<tr>
<th>Variables</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP(X)</td>
<td>A hash structure that stores objects containing a key corresponding to the values assigned to the set of variables (X) and the number of occurrences of those keys in the map</td>
</tr>
<tr>
<td>MAP(YX)</td>
<td>A hash structure that stores objects containing a key corresponding to the function values - (y) - as well as the corresponding values assigned to the set of variables (X), and the number of occurrences of those keys in the map</td>
</tr>
<tr>
<td>(X)</td>
<td>A subset of the complete set of variables of a partially defined function</td>
</tr>
<tr>
<td>(v)</td>
<td>A registered vector from a partially defined function (f)</td>
</tr>
<tr>
<td>(F)</td>
<td>A partially defined function</td>
</tr>
<tr>
<td>key(X)</td>
<td>The values assigned in (v) to the variables in (X)</td>
</tr>
<tr>
<td>key(YX)</td>
<td>The values assigned in (v) to (y) - the function value - and the variables in (X)</td>
</tr>
<tr>
<td>val</td>
<td>The number of occurrences of a key in a map structure</td>
</tr>
<tr>
<td>entropy(X)</td>
<td>Entropy of a set of variables</td>
</tr>
<tr>
<td>entropy(YX)</td>
<td>Entropy of function values and the corresponding set of variables</td>
</tr>
<tr>
<td>entropy</td>
<td>Entropy corresponding to the set of variables (s)</td>
</tr>
</tbody>
</table>
Algorithm 4: COMPUTE\_ENTROPY(f,X)

**Input:** A partially defined function, a subset $X$ of the complete set of variables of the partially defined function given

**Output:** An entropy value

begin

$\text{foreach } v \in F$ do

$\text{keyX} \leftarrow \text{GET\_XKEYS}(v,X)$

if $\text{keyX} \in \text{MAP\_X}$ then

$\text{val} \leftarrow \text{GET}($\text{MAP\_X}$,\text{keyX})$

$\text{ASSIGN}($\text{MAP\_X}$,\text{keyX},\text{val} + 1)$

else

ADD($\text{MAP\_X}$,\text{keyX},1)

end

$\text{keyYX} \leftarrow \text{GET\_XYKEYS}(v,X)$

if $\text{keyYX} \in \text{MAP\_YX}$ then

$\text{val} \leftarrow \text{GET}($\text{MAP\_YX}$,\text{keyYX})$

$\text{ASSIGN}($\text{MAP\_YX}$,\text{keyYX},\text{val} + 1)$

else

ADD($\text{MAP\_YX}$,\text{keyYX},1)

end

$\text{entropyX} \leftarrow \text{ENTROPY}($\text{MAP\_X}$)$

$\text{entropyYX} \leftarrow \text{ENTROPY}($\text{MAP\_YX}$)$

$\text{entropy} \leftarrow \text{entropyYX} - \text{entropyX}$

return entropy
2.6 Experimental Results

We carried out experiments on a Windows Vista 64-bit machine with 8Gb RAM and 2 × Quad Core Xeon Proc E5420, running at 2.50 GHz with a 2×6Mb L2 cache. The program was written in Java 6.

We analyze the results in terms of running time and minimum number of variables of a determining set found as a function of the number of tuples in $T_f$ and the limiting factor $\ell$.

A program that randomly generates comma separated text files representing partially defined functions with 8, 16 or 24 variables was developed. These values were chosen based on the experiments made in the related work of T. Sasao [Sas08a, Yan91].

One hundred files were randomly generated for each type of partially defined function (with 8, 16, and 24 variables) using an input radix $r = 3$ and an output radix $p = 5$.

Note that a totally defined function with 8 variables and $r = 3$ has $3^8 = 6561$ tuples. In our experiments, we randomly generated 1000 tuples for partially defined functions with 8 variables. For functions that depend on 16 and 24 arguments we generated 5000 tuples because the number of tuples for completely defined functions with 16 or 24 variables is much higher.

In the experiments, we evaluate the performance of the algorithm with a varying number of tuples and limiting factor. By Theorem 2.2.3, if $(f_1, f_2, \ldots, f_k)$ is a sequence of functions such that

$$f_1 \subseteq f_2 \subseteq \ldots \subseteq f_k,$$

we have

$$\text{DS}(f_k) \subseteq \ldots \subseteq \text{DS}(f_2) \subseteq \text{DS}(f_1).$$
In other words, when we start with a partial function $f_k$ with a small specification table $T_{f_k}$ and we expand sequentially the specification of the functions, the number of determining sets will decrease. The experiments compare the results for files with 8, 16 and 24 variables. The results contain averages of the values corresponding to time and number of variables the function depends on as a function of the number of tuples and limiting factor. In our case, $k \in \{10, 20, 30, 50, 90, 100, 200\}$. The averages are evaluated over 100 functions within each group of generated functions (8, 16 and 24 variables).

As shown in Figures 5, 6 and 7, the running time increases with the number of tuples because in most cases, the larger the subset of variables $X$, the greater is the conditional entropy $\mathcal{H}(y|X)$. Likewise, the running time increases with the limiting factor $\ell$ since the search space increases as $\ell$ increases. Also, the time increases very fast with the number of variables. It is clear that the number of subsets evaluated during the search depends on the original number of variables of a partial function $f_1$. 

Figure 5: Dependency of average time on number of tuples and limiting factor for 8 variables.
Figure 6: Dependency of average time on number of tuples and limiting factor for 16 variables.

Figure 7: Dependency of average time on number of tuples and limiting factor for 24 variables.
Finally, Figures 8, 9 and 10 show that the number of variables the function depends on is related to the number of tuples $k$. As $k$ increases, the constraints imposed on the problem become more extensive, and the number of variables that determine the value of the function increases. Notice however that the limiting factor $\ell$ has a remarkable insignificant contribution to the number of variables that a partial function depends on provided by the algorithm. Therefore, the algorithm provides solutions that are optimal or very close to optimal, even when $\ell$ takes an extremely low value. This is an important result since as previously mentioned, the running time for the algorithm to find a solution is highly affected by the limiting factor chosen.
Figure 9: Average size of minimal determining set for 16 variables, as a function of the number of tuples and limiting factor.

Figure 10: Average size of minimal determining set for 24 variables, as a function of the number of tuples and limiting factor.
2.7 Comparison Between the Two Versions

In Table 11, a comparison between the two versions of the algorithm is presented. The results represent the averages over 100 randomly generated partially defined functions for each row of the table. The rows written with bold fonts represent the results for the first version (where the limiting factor does not exist and is marked with “not available” (N/A)).

From the results we can observe that even with a limiting factor, $\ell$, of 1%, the quality of the solution (number of variables the function depends on) is close to the value obtained with the first approach (which always returns the optimal solution), but the second version is up to 40 times faster.
Table 11: Tabular Representation of a Partial Function

| Number of Registered Vectors | Number of $|MDS|$ found | $|MDS|$ | Time(ms) | $\ell\%$ |
|-----------------------------|----------------|--------|----------|---------|
| $10$                        | 2.29           | 117.97 | 13.88    | N/A     |
| $10$                        | 2.52           | 18.21  | 2.27     | 1       |
| $10$                        | 2.43           | 46.19  | 3.96     | 5       |
| $10$                        | 2.36           | 61.36  | 6.68     | 10      |
| $10$                        | 2.33           | 70.21  | 14.6     | 15      |
| $20$                        | 3.17           | 208.48 | 98.84    | N/A     |
| $20$                        | 3.62           | 68.38  | 5.48     | 1       |
| $20$                        | 3.38           | 132.69 | 82.05    | 5       |
| $20$                        | 3.34           | 184.39 | 221.12   | 10      |
| $20$                        | 3.26           | 179.42 | 331.9    | 15      |
| $50$                        | 4.98           | 615.19 | 3774.01  | N/A     |
| $50$                        | 4.99           | 74.74  | 346.83   | 1       |
| $50$                        | 4.99           | 217.28 | 3784.17  | 5       |
| $50$                        | 4.99           | 327.48 | 12017.59 | 10      |
| $50$                        | 4.98           | 397.43 | 23700.24 | 15      |
| $100$                       | 6              | 490.78 | 146017.88| N/A     |
| $100$                       | 6              | 18.21  | 3860.57  | 1       |
| $100$                       | 6              | 46.19  | 50485.98 | 5       |
| $100$                       | 6              | 61.36  | 166445.2 | 10      |
| $100$                       | 6              | 70.21  | 333999.09| 15      |
2.8 Concluding Remarks

In this chapter an algorithm is described that identifies the sets of variables that determine a partially defined function. This algorithm solves this problem using a novel approach. It improves the existing methods [Sas08a, Sas08b] by providing the option of either finding the optimal solution or lowering the search scope and finding a close to optimal solution in a much faster time than the current approaches. Also, it has the advantage of not being linked to any value of the input or output radix of the partial function \( f \).

The first version is based on traversing Rymon trees using a breadth-first search technique. This approach guarantees to find the minimal sets of minimal cardinality that the partially defined function depends on, starting from a tabular specification of the function. This algorithm will be helpful for digital circuit design. Using an SQL query, we determine whether a solution is found. A file implementation using hash sets was done as an alternative, to decrease the time required by querying the database.

The second version uses a clustering technique for discrete functions starting from the conditional entropy that measures the discrepancy between the kernel partitions of these functions. The search and evaluation proceed in increasing order of subset sizes. A limiting factor is used to reduce the search space. When the limiting factor value equals 1 all subsets of the power set of the sets of variables of the partially defined function are evaluated, until the determining sets with smallest size are found. In this case the algorithm determines the minimum number of variables which a partially defined function depends on, as well as all sets of variables with minimum number of elements that the function effectively depends on. The effect caused by the limiting factor on the optimality of the solution is remarkably small. The use of a small limiting factor improves the running time needed to find a solution while still keeping the solution optimal or close to an optimal solution.
Both versions of the algorithm can done in parallel, which would improve the running time considerably.
3.1 Introduction

In this chapter an approach to find the optimal decomposition of partially specified index generation functions (PSIGFs) is described. The method described for decomposing the PSIGFs uses a greedy approach, known for computing the locally optimal choice at each stage. Finding an optimal decomposition is an intractable problem, which motivates this approach.

Studying PSIGFs is necessary due to their important role in circuit design problems. As pointed out by T. Sasao [Sas11, Sas08a, SAS06], these functions are used in implementing address tables for routers and in building terminal access controllers for local area networks. They have other applications in memory patch circuits, electronic dictionaries, password lists, etc. By decomposing the PSIGFs, we can decrease computational complexity and have smaller, more efficient circuits.

The functions studied in this chapter are a special case of the partially defined functions described in Chapter 2. PSIGFs are by definition bijection functions, while the partially defined functions previously described are not bijections. The number of possible values taken by a partially defined function is much smaller than the number of possible inputs/rows, while in the case of PSIGFs by definition each input has a different, consecutive value in the tabular definition.

The rest of the chapter is organized as follows. In Section 3.2, we introduce the notion of partially specified index generation functions and we examine the properties of these
functions. In Section 3.3, we present the mathematical background needed for our proposed method. In Section 3.4 we describe the intractability of the problem. The algorithm that achieves the reduction of the number of variables for PSIGFs is presented in Section 3.5. The experimental results of our approach are shown in Section 3.6. Finally, Section 3.7 presents conclusions.

3.2 Partially Specified Index Generation Functions

A \emph{partial function}, see Definition 2.2.1, between the sets $A$ and $B$ is denoted by $f : A \rightsquigarrow B$. Its definition domain, which is a subset of $A$, is denoted as $\text{Dom}(f)$.

A PSIGF is an injective partial function whose definition domain $\text{Dom}(f)$ is a subset of $\{0, 1\}^k$ and whose set of values is $\{1, \ldots, n\}$, where $n = |\text{Dom}(f)|$. The equality $n = |\text{Dom}(f)|$ together with the injectivity of $f$ means that $f$ is a bijection between $\text{Dom}(f)$ and $\{1, \ldots, n\}$. The members of $\text{Dom}(f)$, written as row vectors in $\{0, 1\}^k$ are referred to as \emph{registered vectors}. Our purpose is to formulate an algorithm that allows us to replace the variables $x_1, \ldots, x_k$ with a smaller number $\alpha$ of linear combinations $y_1, \ldots, y_\alpha$ of these variables (over $\text{GF}(2)$, defined in Section 3.3) defined by linear functions of the form $y_i = g_i(x_1, \ldots, x_k) = x_{ij_1} \oplus \cdots \oplus x_{ij_p}$ such that we can express (for some suitable $F$)

$$f(x_1, \ldots, x_k) = F(g_1(x_1, \ldots, x_k), \ldots, g_\alpha(x_1, \ldots, x_k))$$

for all $(x_1, \ldots, x_k) \in \text{Dom}(f)$. Linear techniques have been used frequently in circuit design [Nec07].

\textbf{Example 3.2.1} For example, for the functions described in Table 12, the following combinations are solutions:

\textbf{Solution 1:} $g_1(x_1, x_3) = x_1 \oplus x_3$ and $g_2(x_2, x_3) = x_2 \oplus x_3$ and the function

$$f(x_1, x_2, x_3) = F(g_1(x_1, x_2), g_2(x_2, x_3))$$
Table 12: PSIGF example for function $f$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

**Solution 2:** $g_1(x_3) = x_3$ and $g_2(x_1) = x_1$ and the function

$$f(x_1, x_2, x_3) = F(g_1(x_3), g_2(x_1))$$

**Solution 2** is possible due to the fact that in Table 12, the variable $x_3$ was chosen such that $x_3 = x_1 \oplus x_2$.

As we shall see in Section 3.4, this is an intractable problem. We propose a greedy algorithm that achieves a satisfactory solution.

### 3.3 Boolean Matrices and Collections of Sets

Let $\text{GF}(2) = (\{0, 1\}, +, \cdot, 0, 1)$ be the two-element Galois field, where the addition is the exclusive-or operation defined by

$$a + b = \begin{cases} 
0 & \text{if } a = b, \\
1 & \text{otherwise}.
\end{cases}$$

It is common to write $a \oplus b$ instead of $a + b$. Vectors from $\text{GF}(2)^p$ will be denoted by bold letters; a vector that has only one component equal to 1, located in the $i^{th}$ place will be denoted by $e_i$. The transpose of a matrix $A$ is denoted by $A^t$. The components of a
vector $x \in \text{GF}(2)^p$ will be denoted by $x_1, \ldots, x_p$. The one-column vector in $\text{GF}(2)^p$ whose components are equal to 1 is denoted by $1_p$; the one-column vector in $\text{GF}(2)^p$ whose components are equal to 0 is denoted by $0_p$.

The set of $n \times k$-matrices over $\text{GF}(2)$ is denoted by $\text{GF}(2)^{n \times k}$. The standard product of two matrices $A \in \text{GF}(2)^{m \times n}$ and $B \in \text{GF}(2)^{n \times p}$, where the elements of $A$ and $B$ are regarded as the numbers 0 and 1 will be denoted by $AB$. The inner product of vectors $x \in \text{GF}(2)^n, y \in \text{GF}(2)^n$, is denoted $xy$.

Suppose that $\text{Dom}(f) = \{m_1, \ldots, m_n\}$, where $f$ is a PSIGF. The problem that we are discussing can be formulated in matrix terms as follows. Given an $n \times k$-matrix over $\text{GF}(2)$,

$$M_f = \begin{pmatrix} m_1 \\ \vdots \\ m_n \end{pmatrix}$$

we look for a minimum number $\alpha$ and a matrix $A = (a_1 \cdots a_\alpha) \in \text{GF}(2)^{k \times \alpha}$ such that for all $i \neq j$ we have $m_i a_p \neq m_j a_p$ for some $p, 1 \leq p \leq \alpha$. This is equivalent to requiring that

$$((m_i + m_j)a_1, (m_i + m_j)a_2, \ldots, (m_i + m_j)a_\alpha) \neq (0,0,\ldots,0),$$

for every pair $(i, j), 1 \leq i < j \leq n$.

Define the difference matrix of $f$ as

$$D_f = \begin{pmatrix} m_1 + m_2 \\ \vdots \\ m_1 + m_n \\ m_2 + m_3 \\ \vdots \\ m_{n-1} + m_n \end{pmatrix} \in \text{GF}(2)^{(n) \times k}.$$
Since the rows of $M_f$ are pairwise distinct, none of the rows of $D_f$ equals $0^t_k$.

If $d \in \text{GF}(2)^k$ is a row vector and $da = 1$ we say that $d$ is *covered by* $a$. Note that every row in $\text{GF}(2)^k$ that contains an odd number of ones is covered by $1_k$.

The problem we are trying to solve can be restated as seeking a matrix $A \in \text{GF}(2)^{k \times \alpha}$ with $\alpha < k$ such that each row of $D_f$ is covered by some column of $A$.

We reduced the problem of finding $\alpha$ linear combinations of the arguments of the function $f : \text{GF}(2)^k \rightarrow \{1, \ldots, n\}$ that can serve as arguments of the function $F : \text{GF}(2)^\alpha \rightarrow \{1, \ldots, n\}$ (with $\alpha < k$) to a matrix problem: given a matrix $D_f \in \text{GF}(2)^{(n/2)^k}$ find a matrix $A \in \text{GF}(2)^{k \times \alpha}$, with $\alpha$ as small as possible, such that each row of the matrix $D_f$ is covered by a column of $A$. This is the MCSL(2) optimization problem, shown in Section 3.4 to be NP-hard. Therefore, we will formulate a greedy algorithm that will provide a reasonable good solution; however, this solution is not guaranteed to be optimal.

To find such a matrix $A$, we need to determine a set of $\alpha$ vectors in $\text{GF}(2)^k$ such each row of $D_f$ is covered by some vector in $A$. By a previous observation, we need to cover only the rows of $D_f$ that contain an even number of ones. If $D_f$ contains rows having an odd number of ones, then we include in $A$ the vector $1_k$.

### 3.4 Intractability of covering a system of a linear equations with a minimum number of solutions

As discussed in the Section 3.1, the optimization problem that we need to solve reduces to the following problem, which we call (MCSL(2)) **min covering solutions for linear systems** in $\text{GF}(2)$.
Find a minimum size set $A$ of vectors in $\text{GF}(2)^k$, such that each equation $d_i x = 1$ is satisfied by at least one vector in $A$. In other words, find a minimum size set $A \subseteq \text{GF}(2)^k$ such that each vector $d_i$ is covered by at least one vector in $A$. To analyze the complexity of the optimization problem $\text{MCSL}(2)$, we consider the following decision problems $\alpha$-$\text{MCSL}(2)$, for every integer $\alpha \geq 1$.

A set of linear equations: $d_1 x = 1, d_2 x = 1, \ldots, d_n x = 1$, where each $d_i \in \text{GF}(2)^k$.

Is there a set $A$ of $\alpha$ vectors in $(\text{GF}(2))^k$, such that each equation $d_i x = 1$ is satisfied by at least one vector in $A$? For $\alpha = 1$, $1$-$\text{MCSL}(2)$ amounts to asking if the given system of linear equations has a solution. This problem can be solved in polynomial time by standard Gaussian elimination. For $\alpha \geq 2$, $\alpha$-$\text{MCSL}(2)$ is NP-complete. This implies that the optimization problem $\text{MCSL}(2)$ is NP-hard.

**Theorem 3.4.1** For $\alpha \geq 2$, $\alpha$-$\text{MCSL}(2)$ is NP-complete.

**Proof** The proof is by a reduction to the graph coloring problem, which is well-known to be NP-complete. We recall that in the $t$-$\text{COLORING}$ problem, the input is a graph $G = (V, E)$, and the question is whether one can color the vertices of $G$ with $t$ colors such that no edge is monochromatic. For any integer $t \geq 3$, $t$-$\text{COLORING}$ is NP-complete (this is one of the NP complete problems in the landmark papers [Kar72, GJ79]).

We reduce $2^\alpha$-$\text{COLORING}$ to $\alpha$-$\text{MCSL}(2)$. Let $G = (V, E)$ be a graph, and let the set of vertices be $V = \{v_1, \ldots, v_n\}$. We build a system of linear equations over $\text{GF}(2)$ as follows. The unknowns are denoted $x_1, \ldots, x_n$ and correspond to the vertices of the graph. For each edge $(v_i, v_j) \in E$, we introduce the equation $x_i + x_j = 1$. We show that the graph $G$ is $2^\alpha$-colorable if and only if the constructed system of equations can be covered with $\alpha$ solutions.
In one direction, suppose there are $\alpha$ vectors $a_1, \ldots, a_\alpha$ (over $\text{GF}(2^n)$) such that each equation is satisfied by at least one of the vectors

$$a_k = \begin{pmatrix} a_{1k} \\ a_{2k} \\ \vdots \\ a_{nk} \end{pmatrix},$$

where $1 \leq k \leq \alpha$. We color each node $v_i$, $1 \leq i \leq n$, with the color $(a_{i1}, \ldots, a_{i\alpha})$ which is a row vector that consists of the $i^{th}$ components of the vectors $a_i$'s. Let us show that no edge is monochromatic. Let $(v_i, v_j)$ be an edge of $G$. The equation $x_i + x_j = 1$ is satisfied by a vector, say, by vector $a_h$. Thus $a_{ih} + a_{jh} = 1$, which implies $a_{ih} \neq a_{jh}$ and therefore the colors of the nodes $x_i$ and $x_j$ are distinct because they differ at least in the $h$-component.

Conversely, suppose $G = (V,E)$ is $2^\alpha$-colorable. We write the color of each node $v_i$ as an $\alpha$-binary $n$-dimensional row vector $c_i = (c_{i1}, \ldots, c_{i\alpha})$ for $1 \leq i \leq n$. The set $A$ consists of $\alpha$ $n$-dimensional vectors $a_1, \ldots, a_\alpha$, defined by

$$a_h = \begin{pmatrix} c_{1h} \\ \vdots \\ c_{nh} \end{pmatrix},$$

for $1 \leq h \leq \alpha$. This set containing $\alpha$ vectors covers the system of linear equations. Let $x_i + x_j = 1$ be an equation. Then, $(v_i, v_j)$ is an edge of $G$, and thus the nodes $v_i$ and $v_j$ have different colors, that is, $c_i \neq c_j$. Thus there exists $h$, $1 \leq h \leq \alpha$, such that $c_{ih} \neq c_{jh}$ and therefore $c_{ih} + c_{jh} = 1$. In other words, the vector $a_h$ satisfies the equation $x_i + x_j = 1$. 

51
3.5 Reduction of Number of Variables for PSIGFs

Let \( f \) be a PSIGF such that \( \text{Dom}(f) \subseteq \text{GF}(2)^k \) taking its values in the set \( \{1, \ldots, n\} \), where \( n = |\text{Dom}(f)| \). We seek to determine a set of functions \( \{g_1, \ldots, g_h\} \), where \( g_i : \text{GF}(2)^k \rightarrow \text{GF}(2) \) for \( 1 \leq i \leq h \), and a function \( F : \text{GF}(2)^h \rightarrow \{1, \ldots, n\} \) with \( h < k \) such that:

1. each function \( g_i \) is linear, that is

\[
g_i(x) = x_{k_1} \oplus x_{k_2} \oplus \cdots \oplus x_{k_{p_i}} \tag{3.1}
\]

for \( x \in \text{Dom}(f) \) and \( 1 \leq i \leq h \), and

2. \( f(x) = F(g_1(x), \ldots, g_h(x)) \) for \( x \in \text{Dom}(f) \subseteq \text{GF}(2)^k \).

If the functions \( F, g_1, \ldots, g_h \) exist having the properties mentioned above, the function \( G : \text{Dom}(f) \rightarrow \text{GF}(2)^h \) given by \( G(x) = (g_1(x), \ldots, g_h(x)) \) is injective since \( f(x) = F(G(x)) \) for \( x \in \text{Dom}(f) \).

Note that each function \( g_i \) can be described by a vector \( g_i \in \text{GF}(2)^k \), where \( (g_i)_j = 1 \) if and only if the variable \( x_j \) occurs in the Expression (3.1) of the function \( g_i \).

**Example 3.5.1** The example given by T. Sasao starts from the function

\( f : \text{GF}(2)^{15} \rightarrow \{1, \ldots, 15\} \) whose definition domain is \( \text{Dom}(f) = \{e_1^i, \ldots, e_1^i\} \) such that \( f(e_1^i) = 16 - i \) for \( 1 \leq i \leq 15 \). The following four functions

\[
\begin{align*}
g_1(x) &= x_1 \oplus x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{13} \oplus x_{15} \\
g_2(x) &= x_2 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11} \oplus x_{14} \oplus x_{15} \\
g_3(x) &= x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{12} \oplus x_{13} \oplus x_{14} \oplus x_{15} \\
g_4(x) &= x_8 \oplus x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12} \oplus x_{13} \oplus x_{14} \oplus x_{15}
\end{align*}
\]
serve as the linear functions that introduce new variables $y_1, y_2, y_3$ and $y_4$.

The function $F$ is $F(y) = i$ for $y \in \mathbb{GF}(2)^4 - \{0\}$, where $i$ is the decimal equivalent of the binary number $y_1y_2y_3y_4$. Each of the functions $g_i$ can be expressed in vector form as $g_i(x) = x^t a$. For instance, for $g_2$ we have

$$a^t = (0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1)$$

The function evensets($D_f$) extracts the rows of the difference matrix that contain an even number of ones. Thus, if the number of rows of the remaining difference matrix $D_f$ is reduced this may result in fewer new variables. Unfortunately, this reduction does not always take place, as it is the case with the index function considered in Example 3.5.1. Let $E$ be the matrix $E = \text{evensets}(D_f)$.

The general idea of the algorithm involves using generating random vectors in $\mathbb{GF}(2)^k$. For each such vector $g$ we compute the set of rows $R_g$ of $E$ that consists of the rows covered by $g$. If $R_g \neq \emptyset$, the row $g$ is added to $G$ and the rows of $R_g$ are removed from $E$. This yields a new matrix $E$ with a smaller number of rows. Otherwise, a new row $g$ is generated until a reduction of the current number of rows of $E$ is obtained. This process is repeated a limited number of times; if this limit is exceeded the algorithm fails. In our tests, the number of times was set to $10^4$.

The function countinter computes for each row $g$ the number of rows of $E$ covered by $g$.

The algorithm developed below uses of the functions shown in Table 13.

We give below the pseudocode of the algorithm that begins with the $n \times k$ matrix of registered vectors of the PSIGF $f : \mathbb{GF}(2)^k \mapsto \{1, \ldots, n\}$ and the targeted number $h$ of functions of the decomposition and produces a $k \times h$ matrix of functions, where each row represents one of the functions $g_i$. 

53
Table 13: Functions Used by Algorithm 5

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>diffmatbool((A))</td>
<td>computes the differences of the rows of (A)</td>
</tr>
<tr>
<td>evenets((D))</td>
<td>removes from (D) those rows having an odd number of 1s</td>
</tr>
<tr>
<td>randrow((n, t))</td>
<td>uses parameters (n) and (t) and generates a row with (n) components; a zero occurs with probability (t).</td>
</tr>
<tr>
<td>nozerorows((A))</td>
<td>returns the number of zero rows of (A)</td>
</tr>
<tr>
<td>prodlin((A, B))</td>
<td>computes (A \ast B), where (A, B) are matrices over (GF(2))</td>
</tr>
</tbody>
</table>
Algorithm 5: Decomposition Algorithm for PSIGF

Input: The matrix \( M_f \) of registered vectors for a PSIGF \( f : \text{GF}(2)^k \sim \{1, \ldots, n\} \), where \( n = |\text{Dom}(f)| \) and a number \( h, h < k \)

Output: Matrix \( S \in \text{GF}(2)^{k \times h} \), the set of functions \( \{g_1, \ldots, g_h\} \) with \( h < k \).

begin

\[ k = \text{size}(M_f, 2) \]
\[ \text{maxtries} = 10000 \]
Compute \( D_f = \text{diffmatbool}(M_f) \in \text{GF}(2)^{p \times k} \)
\[ c = 0 \]
if \( D_f \) contains rows with odd number of 1s then
\[ S(1,:) = \text{ones}(k) \]
\[ c = 1 \]
\[ E = \text{evensets}(D_f) \]
\[ \text{notries} = 0 \]
\[ \text{solutionfound} = 0 \]
while \( \text{solutionfound} = 0 \) and \( \text{notries} \leq \text{maxtries} \) do
\[ \text{notries} = \text{notries} + 1 \]
\[ F = E \]
\[ G = \text{zeros}(l - c, k) \]
\[ j = 1 \]
for \( i = 1 \) to \( l - c \) do
\[ v = \text{randrow}(k, 0.5) \]
\[ u = \text{prodlin}(F, v') \]
\[ z = \text{size}(u, 1) \]
if \( u \neq \text{zeros}(z, 1) \) then
\[ G(j,:) = v \]
\[ j = j + 1 \]
remove from \( F \) the rows \( h \) such that \( F(h,:) == 0 \)
if \( c == 1 \) then
\[ \text{S} = [S; G] \]
else
\[ S = G \]
if \( \text{size}(F, 1) == 0 \) then
\[ \text{return} (S) \]
else
\[ \text{disp}('Failed to find solution') \]
end
end
end
Table 14: Definition table of the function $s_7$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
<th>$s_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

3.6 Experimental Results

For our experiments we used a family of function derived from Sasao’s example in [Sas08a].

**Definition 3.6.1** The Sasao function of order $n$ is the PSIGF function $s_n : GF(2)^n \rightarrow \{1, \ldots, n\}$, where $M_{s_n} = \{e_1', \ldots, e_n'\}$ and $f(e_j') = n + 1 - j$ for $1 \leq j \leq n$.

The decomposition of Sasao functions is expensive because each registered vector contains exactly one unit, so the difference matrix $D_f = \text{diffmatbool}(M_f)$ contains only rows that have an even number of units; thus, the matrix of rows with two units $E$ computed by the algorithm coincides with the matrix $D_f$.

Experiments were run on a Widows 7 machine having a 2 Quad cpu at 2.4GHz with a memory of 3.25 GB.

**Example 3.6.1** For Sasao’s function $s_7 : GF(2)^7 \rightarrow \{1, \ldots, 7\}$ defined by Table 14 the
function call \texttt{indsol}(D_f,3) generates the matrix

\[
S = \begin{pmatrix}
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1
\end{pmatrix}.
\]

in 6 attempts that last approximative 46ms. The corresponding functions are

\[
g_1(x) = x_1 \oplus x_3 \oplus x_4 \oplus x_5,
\]
\[
g_2(x) = x_2 \oplus x_4 \oplus x_5 \oplus x_6,
\]
\[
g_3(x) = x_3 \oplus x_5 \oplus x_6 \oplus x_7.
\]

The function \( F : \text{GF}(2)^3 \rightarrow \text{GF}(2) \) given by

\[
F(g_1(x_1,\ldots,x_7),g_2(x_1,\ldots,x_7),g_3(x_1,\ldots,x_7)) = f(x_1,\ldots,x_7)
\]

can be obtained immediately by observing that the triplets

\((g_1(x_1,\ldots,x_7),g_2(x_1,\ldots,x_7),g_3(x_1,\ldots,x_7))\) are the rows of the matrix \(M_5,S'\). Thus, \( F \) is defined by

<table>
<thead>
<tr>
<th>(y_1)</th>
<th>(y_2)</th>
<th>(y_3)</th>
<th>(F(y_1,y_2,y_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

The average number of tries and time performance (computed over 10 computations in each case) is shown in Table 15. Note that in all our experiments, the algorithm returned
Table 15: Time performance of the algorithm

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of functions</th>
<th>Number of tries</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s7</td>
<td>3</td>
<td>52.8</td>
<td>18.7</td>
</tr>
<tr>
<td>s8</td>
<td>3</td>
<td>414.8</td>
<td>121.8</td>
</tr>
<tr>
<td>s9</td>
<td>4</td>
<td>14.7</td>
<td>15.6</td>
</tr>
<tr>
<td>s10</td>
<td>4</td>
<td>55.9</td>
<td>31.3</td>
</tr>
<tr>
<td>s11</td>
<td>4</td>
<td>64.9</td>
<td>39.0</td>
</tr>
<tr>
<td>s12</td>
<td>4</td>
<td>203.1</td>
<td>109.1</td>
</tr>
</tbody>
</table>

an optimal solution. We can also notice that if the number of registered vectors is \( n \), then every solution requires at least \( \alpha = \lceil \log n \rceil \) functions \( g_1, \ldots, g_\alpha \). For a number of components the time increases with the number of variables of the functions.

Note that the 8-variable function \( s_8 \) is at the limit of the decompositions that involve three functions \( g_i \) and, therefore, there is a significant increase in both the number of tries and the time compared to the function \( s_7 \). When the number of allowed function is increased to 4 a significant decrease of the cost is registered even though the number of arguments increases beyond 8.

### 3.7 Conclusions

The probabilistic greedy algorithm that we propose provides reasonably fast solutions to the decomposition of partially specified index functions. An exhaustive search would not be a viable approach due to time and search space size.
It would be interesting to examine families of functions that can be used in the decomposition process other than linear functions.
4.1 Introduction

In this chapter, our goal is to show that compression can be used as a tool to evaluate the potential of a data set to produce interesting results in a data mining process. The basic idea is that if a dataset displays patterns that occur with a certain regularity, the file containing the dataset will be compressed more efficiently compared to data that has no such characteristics. Thus a pre-processing phase of the mining process should allow to decide whether a data set is worth mining, or compare the interest of applying mining algorithms to several data sets.

Since compression is generally inexpensive and compression methods are well-studied and understood, pre-mining using compression will help data mining analysts focus their efforts on mining resources that can provide a highest payout without an exorbitant cost. Compression has received lots of attention in the data mining literature. As observed by Mannila [Man00], data compression can be regarded as one of the fundamental approaches to data mining, since the goal of the data mining is to “compress data by finding some structure in it”.

The role of compression developing parameter-free data mining algorithms in anomaly detection, classification and clustering was examined in [KLR04]. The size $C(x)$ of a compressed file $x$ is as an approximation of Kolmogorov complexity [CV05] and allows the definition of a pseudo-distance between two files $x$ and $y$ as
\[ d(x, y) = \frac{C(xy)}{C(x) + C(y)} \quad (4.1) \]

Further advances in this direction were developed in [WHM06, KLR07] and [KKH09]. A Kolmogorov complexity-based dissimilarity was successfully used to texture matching problems in [CK10] which have a broad spectrum of applications in areas like bioinformatics, natural languages and music.

We illustrate the use of lossless compression in pre-mining data by focusing on several distinct data mining processes: files with frequent patterns, frequent itemsets in market basket data, and exploring similarity of graphs.

The LZW (Lempel-Ziv-Welch) algorithm was introduced in 1984 by T. Welch in [Wel84] and is among the most popular compression techniques. The algorithm does not need to check all the data before starting the compression and the performance is based on the number of the repetitions and the lengths of the strings and the ratio of 0s/1s or true/false at the bit level. There are several versions of the LZW algorithm. Popular programs (such as Winzip) use variations of the LZW compression. The Winzip/Zip type of algorithms also work at the bit level and not at a character/byte level.

We explore three experimental settings that provide strong empirical evidence of the correlation between compression ratio and the existence of hidden patterns in data. In Section 4.2 we compress binary strings that contain patterns. In Section 4.4 we study the compressibility of adjacency matrix for graphs relative to the entropy of distribution of subgraphs. Finally, in Section 4.5 we examine the compressibility of files that contain market basket data sets.
4.2 Patterns in Strings and Compression

Let $A^*$ be the set of strings on the alphabet $A$. The length of a string $w$ is denoted by $|w|$. The null string on $A$ is denoted by $\lambda$ and we define $A^+ = A^* - \{\lambda\}$.

If $w \in A^*$ can be written as $w = utv$, where $u, v \in A^*$ and $t \in A^+$, we say that the pair $(t, m)$ is an occurrence of $t$ in $w$, where $m$ is the length of $u$.

For $m < p$ the occurrences $(x, m)$ and $(y, p)$ are overlapping if $p < m + |x|$. If this is the case, there is a proper suffix of $x$ that equals a proper prefix of $y$. The number of occurrences of a string $t$ in a string $w$ is denoted by $n_t(w)$. Clearly, we have $\sum \{n_a(w) \mid a \in A\} = |w|$. The prevalence of $t$ in $w$ is the number $f_t(w) = \frac{n_t(w) \cdot |t|}{|w|}$ which gives the ratio of the characters contained in the occurrences of $t$ relative to the total number of characters in the string.

The result of applying a compression algorithm $C$ to a string $w \in A^*$ is denoted by $C(w)$ and the compression ratio is the number

$$\text{CR}_C(w) = \frac{|C(w)|}{|w|}. \quad (4.2)$$

In this section, we use the binary alphabet $B = \{0, 1\}$ and the LZW algorithm or the compression algorithm of the package `java.util.zip`.

We generated random strings of bits (0s and 1s) and computed the compression ratio strings with a variety of symbol distributions. A string $w$ that contains only 0s (or only 1s) achieves a very good compression ratio of $\text{CR}_{jZIP}(w) = 0.012$ for 100K bits and $\text{CR}_{jZIP} = 0.003$ for 500K bits, where $jZIP$ denotes the compression algorithm from the package `java.util.zip`. Figure 14 shows, as expected, that the worst compression ratio is achieved when 0s and 1s occur with equal frequencies.
For strings of small length (less than $10^4$ bits) the compression ratio may exceed 1 because of the overhead introduced by the algorithm. However, when the size of the random string exceeds $10^6$ bits this phenomenon is less noticeable and the compression ratio depends on the prevalence of the bits and is relatively independent on the size of the file. Thus, in Figure 11, the curves that correspond to files of size $10^6$ and $5 \cdot 10^6$ overlap. We refer to the compression ratio of a random string $w$ with an $(n_0(w), n_1(w))$ distribution as the baseline compression ratio.

We created a series of binary strings $\phi_{t,m}$ which have a minimum guaranteed number $m$ of occurrences of patterns $t \in \{0, 1\}^k$, where $0 \leq m \leq 100$. Specifically, we created 101 files $\phi_{001,m}$ for the pattern 001, each containing 100K bits and we generated similar series for $t \in \{01, 0010, 00010\}$. The compression ratio is shown in Figure 12. The compression ratio starts at a value of 0.94 and after the prevalence of the pattern becomes more
frequent than 20% the compression ratio drops dramatically. Results of the experiment are shown in Table 16 and in Figure 13.

Table 16: Pattern '001' Prevalence versus the $CR_{jZIP}$

<table>
<thead>
<tr>
<th>Prevalence of '001' pattern</th>
<th>$CR_{jZIP}$</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>0.93</td>
<td>0.93</td>
</tr>
<tr>
<td>10%</td>
<td>0.97</td>
<td>0.93</td>
</tr>
<tr>
<td>20%</td>
<td>0.96</td>
<td>0.93</td>
</tr>
<tr>
<td>30%</td>
<td>0.92</td>
<td>0.93</td>
</tr>
<tr>
<td>40%</td>
<td>0.86</td>
<td>0.93</td>
</tr>
<tr>
<td>50%</td>
<td>0.80</td>
<td>0.93</td>
</tr>
<tr>
<td>60%</td>
<td>0.72</td>
<td>0.93</td>
</tr>
<tr>
<td>70%</td>
<td>0.62</td>
<td>0.93</td>
</tr>
<tr>
<td>80%</td>
<td>0.48</td>
<td>0.93</td>
</tr>
<tr>
<td>90%</td>
<td>0.31</td>
<td>0.93</td>
</tr>
<tr>
<td>95%</td>
<td>0.19</td>
<td>0.93</td>
</tr>
<tr>
<td>100%</td>
<td>0.01</td>
<td>0.93</td>
</tr>
</tbody>
</table>

We conclude that the presence of repeated patterns in strings leads to a high degree of compression (that is, to low compression ratios). Thus, a low compression ratio for a file indicates that the mining process may produce interesting results.

Compressibility however, does not guarantee that a sequence contains repeated patterns, as shown by the compressibility of the well-known Morse-Thue binary string.
Figure 12: Variation of compression rate on the prevalence of the pattern '001'

Figure 13: Dependency of Compression Ratio on Pattern Prevalence
Definition 4.2.1 Let \( n \in \mathbb{N} \) be a natural number. The Thue-Morse sequence \( s_n = s_0s_1 \ldots s_n \) is a word over the alphabet \( \{0, 1\} \) defined as:

\[
s_i = \begin{cases} 
0 & \text{if } i \text{ has an odd number of 1s in its binary representation} \\
1 & \text{otherwise}, 
\end{cases}
\]

for \( 0 \leq i \leq n \).

For example, we have

\[
s_{10} = (0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0).
\]

It is clear that if \( m, n \in \mathbb{N} \) and \( m \leq n \), \( s_m \) is a prefix of \( s_n \). Thus, the successive Thue-Morse sequences define an infinite sequence.

An equivalent method for defining the Thue-Morse sequence is by starting with 0 and concatenating the complement of the sequence obtained so far. This procedure yields 0, then 01, 0110, 01101001, and so on. Thue-Morse sequences have applications in game theory, fractals and turtle graphics, chaotic dynamical systems, etc. It is known (see [Sal81], for example) that the Thue-Morse sequence is a cube-free sequence, that is, the sequence does not contain substrings of the form \( www \).

We generated the Thue-Morse sequence and stored the sequence of 0s and 1s at the bit level. By using the zip compression from the `java.util.zip` package and the following compression ratios were obtained:

For small values of \( n \) (\( n < 1000 \)), the sequence is incompressible. This happens due to the overhead induced by the compression process. For \( n \) big enough (\( n \geq 2000 \)) the sequence becomes compressible and the compression ratio reaches a low value (of less than 1%) for Thue-Morse sequences longer than 4,000,000 characters. The Thue-Morse sequence for any \( 2^k \) value of \( n \) has the same ratio of 0s and 1s.
Table 17: Evolution of $CR_{jZIP}(seq_n)$

| $n$ | $|seq_n|$ | $CR_{seq_n}$ |
|-----|---------|-------------|
| 5   | 32      | 34          |
| 8   | 256     | 4.625       |
| 10  | 1024    | 1.226       |
| 12  | 4096    | 0.328       |
| 14  | 16384   | 0.0932      |
| 15  | 32768   | 0.0542      |
| 16  | 65536   | 0.0322      |
| 17  | 131072  | 0.0208      |
| 18  | 262144  | 0.0151      |
| 19  | 524288  | 0.012       |
| 20  | 1048576 | 0.010       |
| 21  | 2097152 | 0.010       |
| 22  | 4194304 | 0.009       |
In the previous section, we studied the evolution of $\text{CR}_{jZIP}(w)$ where $w$ was a randomly generated string. If $w$ has the same ratio of 0s and 1s and lacks patterns, we have $\text{CR}_{jZIP}(w) > 1$. Thus, the compressibility of the Thue-Morse sequences shows that even in the absence of repetitions, compression can be used for the detection of patterns.

![Figure 14: Compression Ratio Behavior of Thue-Morse Sequence](image)

4.3 Random Insertion and Compression

For a matrix $M \in \{0, 1\}^{u \times v}$ denote by $n_i(M)$ the number of entries of $M$ that equal $i$, where $i \in \{0, 1\}$. Clearly, we have $n_0(B) + n_1(B) = uv$. For a random variable $V$ which ranges over the set of matrices $\{0, 1\}^{u \times v}$ let $\nu_i(V)$ be the random variable whose values equal the number of entries of $V$ that equal $i$, where $i \in \{0, 1\}$.
Let $C \in \{0, 1\}^{p \times q}$ be a 0/1 matrix and
\[
\mathcal{B} : \begin{pmatrix} B_1 & B_2 & \cdots & B_k \\ p_1 & p_2 & \cdots & p_k \end{pmatrix}
\]
a matrix-valued random variable where $B_j \in \mathbb{R}^{r \times s}$, $p_j \geq 0$ for $1 \leq j \leq k$, and $\sum_{j=1}^{k} p_j = 1$.

**Definition 4.3.1** The random variable $C \leftarrow \mathcal{B}$ obtained by the insertion of $\mathcal{B}$ into $C$ is given by

\[
C \otimes \mathcal{B} = \begin{pmatrix} c_{11} \mathcal{B} & \cdots & c_{1n} \mathcal{B} \\ \vdots & \ddots & \vdots \\ c_{m1} \mathcal{B} & \cdots & c_{mn} \mathcal{B} \end{pmatrix} \in \mathbb{R}^{mr \times ns}
\]

In other words, the entries of $C \leftarrow \mathcal{B}$ are obtained by substituting $c_{ij}$ with the block $c_{ij}B_\ell$, with the probability $p_\ell$.

Note that this operation is a probabilistic generalization of Kronecker or tensor product: if
\[
\mathcal{B} : \begin{pmatrix} B_1 \\ 1 \end{pmatrix},
\]
then $C \leftarrow B$ has as its unique value the Kronecker product $C \otimes B$.

The expected number of 1s in the insertion $C \leftarrow \mathcal{B}$ is

\[
E[\nu_1(C \leftarrow \mathcal{B})] = n_1(C) \sum_{j=1}^{k} n_1(B_j)p_j
\]

When $n_1(B_1) = \cdots = n_1(B_k) = n$, we have $E[\nu_1(C \leftarrow \mathcal{B})] = n_1(C)n$.

In the experiment that involves insertion, we used a matrix-valued random variable such that $n_1(B_1) = \cdots = n_1(B_k) = n$. Thus, the variability of the values of $A \leftarrow \mathcal{B}$ is
caused by the variability of the contents of the matrices $B_1, \ldots, B_k$ which can be evaluated using the entropy of the distribution of $\mathcal{B}$,

$$\mathcal{H}(\mathcal{B}) = -\sum_{j=1}^{k} p_j \log_2 p_j.$$ 

We expected to obtain a strong positive correlation between the entropy of $\mathcal{B}$ and the degree of compression achieved on the file that represents the matrix $C \leftarrow \mathcal{B}$, and the experiments support this expectation.

In a first series of compressions, we worked with a matrix $C \in \{0, 1\}^{106 \times 106}$ and with a matrix-valued random variable

$$\mathcal{B} : \begin{pmatrix} B_1 & B_2 & B_3 \\ p_1 & p_2 & p_3 \end{pmatrix},$$

where $B_j \in \{0, 1\}^{3 \times 3}$, and $n_1(B_1) = n_1(B_2) = n_1(B_3) = 4$. Several probability distributions were considered, as shown in Table 18. Values of $C \leftarrow \mathcal{B}$ had $106^2 \times 3^2 = 101124$ entries.

In Table 18, we had 39% 1s and the baseline compression rate for a binary file with this ratio of 1s is 0.9775. We also computed the correlation between the $CR_{jZIP}$ and the Shannon entropy of the probability distribution and obtained the value 0.9825 for 3 matrices. In Table 19, we did the same experiment but with 4 different matrices of $4 \times 4$. A strong correlation (0.992) was observed between $CR_{jZIP}$ and the Shannon entropy of the probability distribution.

In Figure 15, we have the evolution of $CR_{jZIP}$ on the $y$ axis and on the $x$ axis the Shannon Entropy of the probability distribution for both experiments. We can see clearly the linear correlation between the two.

This experiment proves again that in case of repetitions/patterns the $CR_{jZIP}$ is better than in the case of randomly generated files.
Table 18: Matrix Insertions, Entropy and Compression Ratios

<table>
<thead>
<tr>
<th>Probability distribution</th>
<th>CR_{jZIP}</th>
<th>Shannon Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1,0)</td>
<td>0.33</td>
<td>0</td>
</tr>
<tr>
<td>(1,0,0)</td>
<td>0.33</td>
<td>0</td>
</tr>
<tr>
<td>(0,0,1)</td>
<td>0.33</td>
<td>0</td>
</tr>
<tr>
<td>(0.2,0.2,0.6)</td>
<td>0.77</td>
<td>1.37</td>
</tr>
<tr>
<td>(0.6,0.2,0.2)</td>
<td>0.74</td>
<td>1.37</td>
</tr>
<tr>
<td>(0.33,0.33,0.34)</td>
<td>0.79</td>
<td>1.58</td>
</tr>
<tr>
<td>(0,0.3,0.7)</td>
<td>0.7</td>
<td>0.88</td>
</tr>
<tr>
<td>(0.9,0.1,0)</td>
<td>0.51</td>
<td>0.46</td>
</tr>
<tr>
<td>(0.8,0,0.2)</td>
<td>0.61</td>
<td>0.72</td>
</tr>
<tr>
<td>(0.49,0.25,0.26)</td>
<td>0.77</td>
<td>1.5</td>
</tr>
<tr>
<td>(0.15,0.35,0.5)</td>
<td>0.78</td>
<td>1.44</td>
</tr>
</tbody>
</table>

Table 19: Kronecker product and probability distribution for 4 matrices

<table>
<thead>
<tr>
<th>Probability distribution</th>
<th>CR_{jZIP}</th>
<th>Shannon Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1,0,0)</td>
<td>0.23</td>
<td>0</td>
</tr>
<tr>
<td>(0,1,0,0)</td>
<td>0.23</td>
<td>0</td>
</tr>
<tr>
<td>(0.2,0.2,0.2,0.4)</td>
<td>0.69</td>
<td>0.193</td>
</tr>
<tr>
<td>(0.25,0.25,0.25,0.25)</td>
<td>0.69</td>
<td>2</td>
</tr>
<tr>
<td>(0.4,0,0.2,0.4)</td>
<td>0.53</td>
<td>1.52</td>
</tr>
<tr>
<td>(0.3,0.1,0.2,0.4)</td>
<td>0.65</td>
<td>1.84</td>
</tr>
<tr>
<td>(0.45,0.12,0.22,0.21)</td>
<td>0.61</td>
<td>1.83</td>
</tr>
</tbody>
</table>

Next, we examine the compressibility of binary square matrices and its relationship with the distribution of principal submatrices. A binary square matrix is compressed by
first vectorizing the matrix and then compressing the binary sequence. The issue is relevant in graph theory, where the principal submatrices of the adjacency matrix of a graph correspond to the adjacency matrices of the subgraphs of that graph. The patterns in a graph are captured in the form of frequent isomorphic subgraphs.

There is a strong correlation between the compression ratio of the adjacency matrix of a graph and the frequencies of the occurrences of isomorphic subgraphs of it. Specifically, the lower the compression ratio is, the higher are the frequencies of isomorphic subgraphs and hence the worthier is the graph for being mined.

Let $G_n$ be an undirected graph having $\{v_1, \ldots, v_n\}$ as its set of nodes. The adjacency matrix of $G_n$, $A_{G_n} \in \{0, 1\}^{n \times n}$ is defined as

$$ (A_{G_n})_{ij} = \begin{cases} 
1 & \text{if there is an edge between } v_i \text{ and } v_j \text{ in } G_n \\
0 & \text{otherwise.}
\end{cases} $$
We denote with $\text{CR}_C(A_{G_n})$ the compression ratio of the adjacency matrix of graph $G_n$ obtained by applying the compression algorithm $C$. Define the principal subcomponent of matrix $A_{G_n}$ with respect to the set of indices $S = \{s_1, \ldots, s_k\} \subseteq \{1, 2, \ldots, n\}$ to be the $k \times k$ matrix $A_{G_n}(S)$ such that

$$A_{G_n}(S)_{ij} = \begin{cases} 
1 & \text{if there is an edge between } v_{s_i} \text{ and } v_{s_j} \text{ in } G_n \\
0 & \text{otherwise.}
\end{cases}$$

The matrix $A_{G_n}(S)$ is the adjacency matrix of the subgraph of $G_n$ which consists of the nodes with indices in $S$ along with those edges that connect these nodes. We denote by $\mathcal{P}_n(k)$ the collection of all subsets of $\{1, 2, \ldots, n\}$ of size $k$ where $2 \leq k \leq n$. We have $|\mathcal{P}_n(k)| = \binom{n}{k}$.

Let $(M^k_1, \ldots, M^k_{\ell_k})$ be an enumeration of possible adjacency matrices of graphs with $k$ nodes where $\ell_k = 2^{\frac{k(k-1)}{2}}$. We define the finite probability distribution

$$P(G_n, k) = \left( \frac{n^k_1(G_n)}{|\mathcal{P}_n(k)|}, \ldots, \frac{n^k_{\ell_k}(G_n)}{|\mathcal{P}_n(k)|} \right),$$

where $n^k_i(G_n)$ for $1 \leq i \leq \ell_k$ is the number of subgraphs of $G_n$ with adjacency matrix $M^k_i$.

The Shannon entropy of this probability distribution is:

$$\mathcal{H}_P(G_n, k) = -\sum_{i=1}^{\ell_k} \frac{n^k_i(G_n)}{|\mathcal{P}_n(k)|} \log_2 \frac{n^k_i(G_n)}{|\mathcal{P}_n(k)|}.$$

If $\mathcal{H}_P(G_n, k)$ is low, there are to be fewer and larger sets of isomorphic subgraphs of $G_n$ of size $k$. In other words, small values of $\mathcal{H}_P(G_n, k)$ for various values of $k$ suggest that the graph $G_n$ contains repeated patterns and is susceptible to produce interesting results. Note that although two isomorphic subgraphs do not necessarily have the same adjacency matrix, the number $\mathcal{H}_P(G_n, k)$ is a good indicator of the frequency of isomorphic subgraphs and hence subgraph patterns.
We evaluated the correlation between $\text{CR}_{jZIP}(A_{G_n})$ and $\mathcal{H}_P(G_n,k)$ for different values of $k$.

As expected, the compression ratio of the adjacency matrix and the distribution entropy of graphs are roughly the same for isomorphic graphs, so both numbers are characteristic for an isomorphism type. If $\phi$ is a permutation of the vertices of $G_n$, the adjacency matrix of the graph $G_{\phi}^n$ obtained by applying the permutation is defined by $A_{G_{\phi}^n}$ is given by

$$A_{G_{\phi}^n} = P_{\phi}A_{G_n}P_{\phi}^{-1}.$$  

We compute this adjacency matrix of $A_{G_{\phi}^n}$, the entropy $\mathcal{H}_P(G_{\phi}^n,k)$ the compression ratio $\text{CR}_{jZIP}(A_{G_{\phi}^n})$ for several values of $k$ and permutations.

We randomly generated graphs with $n = 60$ nodes and various number of edges ranging from 5 to 1765. For each generated graph, we randomly produced twenty permutations of its set of nodes and computed $\mathcal{H}_P(G_{\phi}^n,k)$ and $\text{CR}_{jZIP}(A_{G_{\phi}^n})$.

Finally, for each graph we calculated the ratio of standard deviation over average for the computed compression ratios, followed by the same computation for distribution entropies.

The results of this experiment are shown in Figures 16 and 17 against the number of edges. As it can be seen, the deviation over mean of the compression ratios for $n = 60$ does not exceed the number 0.05. Also, the deviation over average of the distribution entropies for various values of $k$ does not exceed 0.006. In particular, the deviation of the distribution entropy for the graphs of 100 to 1500 edges falls below 0.001, which allows us to conclude that the deviations of both compression ratio and distribution entropy with respect to isomorphisms are negligible.

For each $k \in \{3,4,5\}$, we generated randomly 560 graphs having 60 vertices and sets of edges whose size were varying from 10 to 1760. Then, the numbers $\mathcal{H}_P(G_n,k)$ and
Figure 16: Standard deviation vs. average of the $\text{CR}_{J\text{ZIP}}(A_{G_n})$ for a number of different permutations of nodes for the same graph.

$\text{CR}_{J\text{ZIP}}(A_{G_n})$ were computed. Figure 17 captures the results of the experiment. Each plot contains two curves. The first curve represents the changes in average $\text{CR}_{J\text{ZIP}}(A_{G_n})$ for forty randomly generated graphs of equal number of edges. The second curve represents the variation of the average $\mathcal{H}_P(G_n, k)$ for the same forty graphs. The trends of these two curves are very similar for different values of $k$.

Table 20 contains the correlation between $\text{CR}_{J\text{ZIP}}(A_{G_n})$ and $\mathcal{H}_P(G_n, k)$ calculated for the 560 randomly generated graphs for each value of $k$.

### 4.4 Frequent Items Sets and Compression Ratio

A market basket data set consists of a multiset $T$ of transactions. Each transaction $t$ is a subset of a set of items $I = \{i_1, \ldots, i_N\}$. A transaction is described by its characteristic
Figure 17: Standard deviation vs. average of the $H_P(G_n,k)$ of a number of different permutations of nodes for the same graph.

$N$-tuple $t = (t_1, \ldots, t_N)$, where

$$t_k = \begin{cases} 
1 & \text{if } i_k \in t, \\
0 & \text{otherwise}, 
\end{cases}$$

for $1 \leq k \leq N$. The length of a transaction $t$ is $|t| = \sum_{k=0}^{N} t_k$, while the average size of transactions is $\frac{\sum{|t|\text{ in } T}}{|T|}$.

The support of a set of items $K$ of the data set $T$ is the number $\text{supp}(K) = \frac{|\{t \in T | K \subseteq t\}|}{|T|}$. The set of items $K$ is $s$-frequent if $\text{supp}(K) > s$.

The study of market basket data sets is concerned with the identification of association rules. A pair of item sets $(X, Y)$ is an association rule. Its support, $\text{supp}(X \rightarrow Y)$ equals $\text{supp}(X)$ and its confidence $\text{conf}(X \rightarrow Y)$ is defined as

$$\text{conf}(X \rightarrow Y) = \frac{\text{supp}(XY)}{\text{supp}(X)}.$$
Figure 18: Plots of average $\text{CR}_{ZIP}(A_{g_n})$ (CMP RTIO) and average $\mathcal{H}_p(g_n,k)$ (DIST ENT) for randomly generated graphs $g_n$ of equal number of edges with respect to the number of edges. - $n = 60$ and $k = 3$. 
Figure 19: Plots of average $CR_{ZIP}(A_{G_n})$ (CMP RTIO) and average $H_P(G_n, k)$ (DIST ENT) for randomly generated graphs $G_n$ of equal number of edges with respect to the number of edges. - $n = 60$ and $k = 4$. 
Figure 20: Plots of average $CR_{ZIP}(A_{S_n})(\text{CMP RTIO})$ and average $H_p(S_n, k)$ (DIST ENT) for randomly generated graphs $S_n$ of equal number of edges with respect to the number of edges. - $n = 60$ and $k = 5$. 
Table 20: Correlations between $CR_{ZIP}(A_{S_n})$ and $\mathcal{H}_P(S_n,k)$

<table>
<thead>
<tr>
<th>k</th>
<th>Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.92073175</td>
</tr>
<tr>
<td>4</td>
<td>0.920952812</td>
</tr>
<tr>
<td>5</td>
<td>0.919256573</td>
</tr>
</tbody>
</table>

Using the artificial transaction ARMiner generator described in [Cri00], we created a basket data set. Transactions are represented by sequences of bits $(t_1, \cdots, t_N)$. The multiset of $M$ transactions was represented as a binary string of length $MN$ obtained by concatenating the strings that represent transactions.

We generated files with 1000 transactions, with 100 items available in the basket, adding up to 100K bits.

For data sets having the same number of items and transactions, the efficiency of the compression increases when the number of patterns is lower (causing more repetitions). In an experiment with an average size of a frequent item set equal to 10, the average size of a transaction equal to 15, and the number of frequent item sets varying in the set \{5, 10, 20, 30, 50, 75, 100, 200, 500, 1000\}, the compression ratio had a significant variation ranging between 0.20 and 0.75, as shown in Table 21. The correlation between the number of patterns and CR was 0.544. Although the frequency of 1s and baseline compression ratio were roughly constant (at 0.75), the number of patterns and compression ratio were correlated.

Furthermore, there was a strong negative correlation (-0.92) between the compression ratio and the number of association rules indicating that market basket data sets that satisfy many association rules are very compressible.
Table 21: Number of association rules at 0.05 support level and 0.9 confidence

<table>
<thead>
<tr>
<th>Number of Patterns</th>
<th>Frequency of 1s</th>
<th>Baseline compression</th>
<th>Compression ratio</th>
<th>Number of assoc. rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>16%</td>
<td>0.75</td>
<td>0.20</td>
<td>9,128,841</td>
</tr>
<tr>
<td>10</td>
<td>17%</td>
<td>0.73</td>
<td>0.34</td>
<td>4,539,650</td>
</tr>
<tr>
<td>20</td>
<td>17%</td>
<td>0.73</td>
<td>0.52</td>
<td>2,233,049</td>
</tr>
<tr>
<td>30</td>
<td>17%</td>
<td>0.76</td>
<td>0.58</td>
<td>106,378</td>
</tr>
<tr>
<td>50</td>
<td>19%</td>
<td>0.75</td>
<td>0.65</td>
<td>2,910,071</td>
</tr>
<tr>
<td>75</td>
<td>18%</td>
<td>0.75</td>
<td>0.67</td>
<td>289,987</td>
</tr>
<tr>
<td>100</td>
<td>18%</td>
<td>0.75</td>
<td>0.67</td>
<td>378,455</td>
</tr>
<tr>
<td>200</td>
<td>18%</td>
<td>0.75</td>
<td>0.70</td>
<td>163</td>
</tr>
<tr>
<td>500</td>
<td>18%</td>
<td>0.75</td>
<td>0.735</td>
<td>51</td>
</tr>
<tr>
<td>1000</td>
<td>18%</td>
<td>0.75</td>
<td>0.75</td>
<td>3</td>
</tr>
</tbody>
</table>

4.5 Concluding Remarks

The compression ratio of a file can be determined fast and easy, and in many cases offers a cheap way of predicting the existence of embedded patterns in data. Thus, it becomes possible to obtain an approximative estimation of the usefulness of an in-depth exploration of a data set using more sophisticated and expensive algorithms.

The use of compression as a measure of minability is illustrated on a variety of paradigms: graph data, market basket data, etc. Recent investigations show that identifying compressible areas of human DNA is a useful tool for detecting areas where the gene replication mechanisms are disturbed (a phenomenon that occurs in certain genetically based diseases).
Data mining is the process of extracting useful information from big datasets. In the last decade, these datasets have grown rapidly. As a consequence, in many fields like engineering, applying data mining techniques have become a necessity. In this dissertation, I developed three new techniques with applications in engineering and a novel approach to quantifying the mining potential of a dataset using data compression.

5.1 Mining Partially Defined Finite Functions

We present a method to find the minimal cardinality sets of variables that determine the values of a partially defined finite function. Two versions of an Apriori-like algorithm [AIS93] are studied.

The first version is exhaustive, performs a prioritized search and returns an optimal solution. In order to do that, it uses a specific data structure, called Rymon tree. This problem was initially addressed in [Sas08a] and was solved using an algebraic minimization algorithm that applies to functions which depend on a small number of variables. This method guarantees an optimal solution and has the advantage of not being linked to any value of the input or output radix of the partial function $f$.

The second approach is faster and uses the notion of entropy of a partition. The entropy of a partition is used to determine the sets of variables from the Rymon tree, which are likely to reach a solution. We also introduce a limiting factor to restrict the
search. By limiting the search space, the limiting factor allows the user to reduce the running time with little consequences on the quality of the solution.

Each one of the two versions of the algorithm are studied theoretically and experimentally. Time performance is increased as both versions can be parallelized. In comparison to previously existing methods, our approach is faster and at the same time adaptive.

5.2 Decomposition of Partially Specified Index Generating Functions

In Chapter 3 we present a new method for the decomposition of Partially Specified Index Generating Functions (PSIGFs). The PSIGFs are used in implementing address tables for routers and in building terminal access controllers for local area networks. By decomposing the PSIGFs, both the computational complexity and size can be significantly decreased making the circuits more efficient.

We prove that the decompositions of PSIGF is NP-hard problem. This motivates the use of a greedy strategy. We use randomness to achieve reasonably fast solutions to the decomposition of PSIGFs. An exhaustive search would not be a viable approach due to time and search space size.

For the experiments family of functions derived from Sasao’s example in [Sas08a] was used.

5.3 Evaluating Data Minability Through Compression

We present a novel idea, parameter-free method to find the mining potential of a dataset using compression. This is supported by experimental study. The experimental study
shows that compression can be used in the pre-mining process to evaluate the potential of a data set to produce interesting results in a data mining study.

The main contribution of Chapter 4 is the introduction of a parameter-free method that quantifies the prevalence of patterns in a dataset. This method is independent of the compression algorithm.

Experimental results demonstrate an excellent correlation between the compressibility of a dataset in report to a randomly generated file with the same characteristics and the pattern presence in the dataset.
APPENDIX A

NOTATIONS AND TERMINOLOGY

\( f \): a function \( f \)

\( \text{Dom}(f) \): the domain of \( f \)

\( n \): the finite set \( \{0, 1, \ldots, n - 1\} \)

\( r \): input radix

\( p \): output radix

\( \text{PF}(r^n, p) \): set of all partially defined functions

\( T_f \): table

\( t \): tuple or row of \( T_f \)

\( t[U] \): the projection of \( t \) on the set of variables \( U \)

\( t[z] \): \( t[\{z\}] \), where \( z \) is a variable

\( \text{DS}(f) \): the collection of determining sets for the function \( f \)

\( \text{MDS}(f) \): the collection of determining sets of minimal cardinality for the function \( f \)

\( \text{view}(\text{ind}, U) \): a nodes’ view

\( \text{Child}[X] \): the children of a node \( X \)

\( \mathcal{P}(S) \): the power set of \( S \)
PART($S$): the set of partitions of $S$

$\pi$, $\rho$, $\sigma$: partitions

$\mathcal{H}(\pi)$: Shannon entropy of a partition $\pi$

$\mathcal{H}(\pi|\rho)$: conditional entropy of $\pi$ on $\rho$

$\ell$: the limiting factor $\varepsilon(0,1]$

PSIGF: partially specified index generation function

$\text{GF}(2) = (\{0,1\}, +, \cdot, 0, 1)$: the two-element Galois field

$x$: a vector

$xy$: the inner product of vectors $x$ and $y$

$M_f$: the matrix of the function $f$

$D_f$: the difference matrix of the function $f$

MCSL(2): minimum covering solutions for linear systems in $\text{GF}(2)$

$C(w)$: the size in bytes of a compressed string $w$

$\text{CR}_C(w)$: compression ratio of the string $w$, applying compression algorithm $C$

$\text{CR}_{\text{ZIP}}(w)$: compression ratio of the string $w$, applying compression algorithm from the package java.util.zip

$n_t(w)$: the number of occurrences of a string $t$ in a string $w$
\mathcal{B}: \text{probability distribution of the matrices } B_1, \ldots, B_k

\mathcal{G}_n: \text{undirected graph of } n \text{ nodes}
APPENDIX B

INDSOL.M FUNCTION

function [S] = indsol(A,l)
% INDSOL starts from the nxk matrix of registered vectors of a
% PSIGF f and a number l, where l<k, and generates a lxk matrix
% whose l rows give the l functions of the decomposition of f

t = cputime;
k = size(A,2);
maxtries = 10000;
if l >= k
    error('l is too large')
end
D = diffmatbool(A);
E = evensets(D);
c = 0;
if(size(E,1) < size(D,1))
c = 1;
end

notries = 0;
solutionfound = 0;

while((solutionfound == 0) && (notries <= maxtries))
    notries = notries + 1;
    F = E;
    T = zeros(l-c,k);

    j= 1;
    while(j <= l-c)
        v = randrow(k,0.5);
        u = prodlin(F,v');
        z = size(u,1);
        if(~isequal(u,zeros(z,1)))
            T(j,:) = v;
        end
        j = j + 1;
    end
end


j = j + 1;

% G contains the set of rows NOT covered by v
G = [];
p = 1;
for h = 1:size(F,1)
    if(u(h) == 0)
        G(p,:) = F(h,:);
        p = p + 1;
    end
end
if (size(G,1)==0)
    solutionfound = 1;
else
    F = G;
end
end
end

if(c==1)
    S = [ones(1,k);T];
else
    S = T;
end
end

if (solutionfound == 1)
    S
    fprintf('time: %d in %d tries\n', cputime -t,notries)
else
    fprintf('Failed after %d tries', notries)
end
end
REFERENCE LIST


91

