Recent Progress in the Boolean Domain

Edited by Bernd Steinbach

September 25, 2013

Contents

List of Figures

List of Tables

Preface

Boolean logic and algebra are cornerstones of computing and other digital systems, and are thus fundamental to both theory and practice in Computer Science, Engineering, and many other disciplines. Understanding and developing Boolean concepts and techniques are critical in an increasingly digital world. This book presents recent progress through a variety of contributions by thirty-one authors from the international Boolean domain research community.

The first section of this book addresses exceptionally complex Boolean problems. The reader may well ask "What is an exceptionally complex Boolean problem?" The answer is that there are many and they are diverse. Some are theoretical – some are extremely practical. While every problem has its own defining features, they also have many aspects in common, most notably the huge computational challenges they can pose.

The first challenge considered is identified as the Boolean Rectangle Problem. Like many extremely complex Boolean problems, this problem is easy to state, easy to understand, and easy to tell when you have a solution. It is finding a solution that is the challenge.

The discussion of this problem takes the reader through the description of the problem, its analysis, its formulation in the Boolean domain and from there on to several solutions. While the discussion focuses on a particular problem, the reader will gain a very good general understanding of how problems of this nature can be addressed and how they can be cast into the Boolean domain in order to be solved subsequently by sophisticated and powerful tools such as the Boolean minimizer used in this instance. The discussion of the problem continues with a very insightful comparison of exact and heuristic approaches followed by a study of the role of permutation classes in solving such problems.

Building on the above, the discussion continues to show how the techniques developed in the Boolean domain can be extended to the multiple-valued domain. The presentation again focuses on a single problem, Rectangle-free Four-colored Grids, but as before, the presentation provides broad general insights. The reader is encouraged to consider which techniques transfer easily from the Boolean to the multiple-valued domain and where novel ides must be injected. The discussion is interesting both in terms of how to approach solving the problem at hand and similar problems, and also as an illustration of the use of modern SAT-solvers. Satisfiability (SAT) is a central concept in the theoretical analysis of computation, and it is of great interest and value to see the application of a SAT-solver as a powerful tool for solving an extremely complex Boolean problem. The reader will benefit greatly from this demonstration of another powerful solution technique.

The contribution on Perception in Learning Boolean Concepts approaches the issue of complexity from a very different point of view. Rather than considering complexity in the mathematical or computational sense, the work examines complexity, complexity of Boolean concepts in particular, through a consideration of human concept learning and understanding. This alternate view provides a very different insight into the understanding of the Boolean domain and will aid readers in broadening their conceptual understanding of the Boolean domain.

The use of logic in computation is a broad area with many diverse approaches and viewpoints. This is demonstrated in the presentation on Generalized Complexity of $\mathcal{ALC}$ Subsumption. The discussion considers a variety of concepts from a rather theoretical point of view but also points to the practical implications of those concepts. It also presents yet another view of an extremely complex Boolean problem in terms of algorithmic constructions for the subsumption problem.

The final presentation on exceptionally complex Boolean problems considers encryption and cryptanalysis. The discussion is of considerable interest due to the obvious practical importance of the problem. Readers, even those familiar with the-state-of-the-art for encryption methods, will benefit from the presentation on the concept of algebraic immunity and its computation. The approach presented is also of con-

siderable interest in its use of a reconfigurable computer. The reader should consider this approach as another tool in the computational toolbox for the Boolean domain.

The second section of this book begins with a discussion of low-power CMOS design. CMOS is currently the dominant technology for digital systems and this contribution is of particular significance given the ever-growing demand for low-power devices. After an overview of CMOS design, a number of techniques for power reduction are described. This discussion ends with the key question, "how low can power go?" – an interesting query on its own and also a perfect lead into the discussion of reversibility in the final section of the book.

Design and test of digital devices and systems have been longtime a major motivation for research in the Boolean domain. The combinational logic design contributions in this book treat a variety of topics: the shape of binary decision diagrams; polynomial expansion of symmetric Boolean functions; and the issue of dealing with the don't-care assignment problem for incompletely specified Boolean functions. The final logic design contribution concerns state machine decomposition of Petri nets. Individually, these contributions provide insight and techniques specific to the particular problem at hand. Collectively they show the breadth of issues still open in this area as well as the connection of theoretical and practical concepts.

Testing is the subject of the next two contributions. The first concerns Boolean fault diagnosis with structurally synthesized BDDs. This contribution shows how binary decision diagrams, which are used in quite different contexts in earlier parts of the book, can be adapted to address a significantly different problem in a unique way. The second testing contribution considers techniques in built-in self-test. After reviewing spectral techniques for testing, the discussion centers upon testing of polynomials by linear checks. The reader will gain an appreciation of the relationship between the spectral and the Boolean domains and how fairly formal techniques in the first domain are applied to a very practical application in the second.

The final section of this book addresses topics concerning the connection between two important emerging technologies: reversible and quantum logic circuits. A reversible logic circuit is one where there is

a one-to-one correspondence between the input and output patterns, hence the function performed by the circuit is invertible. A major motivation for the study of reversible circuits is that they potentially lead to low power consumption. In addition, the study of reversible circuits has intensified because of the intrinsic connection of quantum computation to the gate model. The transformations performed by quantum gates are defined by unitary matrices and are thus by definition reversible. Reversible Boolean functions are also central components of many quantum computation algorithms. This section of the book provides novel ideas and is also a very good basis for understanding the challenging problem of synthesizing and optimizing quantum gate realizations of reversible functions.

The section begins with a detailed study of the computational power of a gate referred to as the square root of NOT since two such gates in succession realize the conventional Boolean NOT gate. In addition to describing the computational power of such gates, this contribution provides a very good basis for understanding the connections and differences between reversible Boolean gates and quantum operations.

The Toffoli gate is a key building block in Boolean reversible circuits. The second contribution in this section considers the realization of Toffoli gates using controlled-`NOT` gates and the square root of `NOT` as well as the fourth root of `NOT` gates. The work extends beyond the conventional Toffoli gate to include multiple mixed positive and negative controls, and alternate control functions. The final contribution in this section concerns the quantum realization of pairs of multi-control Toffoli gates. It builds nicely on the work in the two preceding contributions and provides the reader with valuable insight and techniques for the optimization of quantum gate realizations particularly for reversible logic.

I am confident that this book will provide novel ideas and concepts to researchers and students whether or not they are knowledgeable in modern approaches and recent progress in the Boolean domain. I am also confident that study of the work presented here will lead to further developments in Boolean problem solving and the application of such techniques in both theory and practice.

The contributions appearing in this book are extended versions of

works presented at the International Workshop on Boolean Problems held at the Technische Universität Bergakademie Freiberg, Germany on September 19-21, 2012. The 2012 workshop was the tenth in a series of Boolean Problems Workshops held in Freiberg biennially since 1994. Prof. Bernd Steinbach has organized and hosted the workshop since its inception. The Boolean research community is indebted to him for this long-term contribution and for his efforts in organizing and editing this book.

D. Michael Miller

Department of Computer Science
University of Victoria
Victoria, British Columbia, Canada

Foreword

This book covers several fields in theory and practical applications where Boolean models support there solutions. Boolean variables are the simplest variables at all, because they have the smallest possible range of only two different values. In logic applications these values express the truth values *true* and *false*; in technical applications the values of signals *high* and *low* are described by Boolean variables. For simplification, the numbers 0 and 1 are most commonly used as values of Boolean variables.

The basic knowledge in the Boolean domain goes back to the English mathematician, philosopher and logician George Boole as well the American mathematician, electronic engineer, and cryptographer Claude Shannon. The initiator of the very strong increase of Boolean application was Conrad Zuse. He recognized the benefit of Boolean values to avoid errors in large technical systems. In this way he was able to build the first computer.

The benefit of Boolean values is not restricted to computers, but can be utilized for all kinds of control systems in a wide range of applications. The invention of the transistor as a very small electronic switch and the integration of a growing number of transistors on a single chip together with the strong decrease of the cost per transistor was the second important factor for both the substitution of existing systems by electronic ones and the extensive exploitation of new fields of applications. This development is forced by a growing community of scientists and engineers.

As part of this community, I built as electrician control systems for machine tools, developed programs to solve Boolean equations during my studies, contributed to test software for computers as graduated engineer, and taught students as assistant professor for design automation. In 1992, I got a position as full professor at the Technische

Universität Bergakademie Freiberg. Impressed by both the strong development and the challenges in the Boolean domain, I came to the conclusion that a workshop about *Boolean Problems* can be a valuable meeting point for people from all over the world which are working in different branches of the Boolean domain. Hence, I organized the first workshop in 1994 and encouraged by the attendees I continued the organization of the biennial series of such *International Workshops on Boolean Problems* (IWSBP).

The idea for this book goes back to Carol Koulikourdi, Commissioning Editor of Cambridge Scholars Publishing. She asked me one month before the 10th IWSBP whether I would agree to publish this book based on the proceedings of the workshop. I discussed this idea with the attendees of the 10th IWSBP and we commonly decided to prepare this book with extended versions of the best papers of the workshop. The selection of these papers was done based on the reviews and the evaluation of the attendees of the 10th International Workshop on Boolean Problems. Hence, there are many people which contributed directly or indirectly to this book.

I like to thank all of them: starting with the scientists and engineers which have been working hard on Boolean problems and submitted papers about their results to the 10th IWSBP; continuing with the 23 reviewers from eleven countries; the invited speakers Prof. Raimund Ubar from the Tallinn University of Technology, Estonia, and Prof. Vincent Gaudet from the University of Waterloo, Canada; all presenters of the papers; and all attendees for their fruitful discussions of the very interesting presentation on all three days of the workshop. Besides of the technical program, such an international workshop requires a lot of work to organize all necessary things. Without the support by Ms. Dr. Galina Rudolf, Ms. Karin Schüttauf, and Ms. Birgit Steffen, I would not have been able to organize this series of workshops. Hence, I like to thank these three ladies for their valuable hard work very much.

Not only the authors of the papers but often larger group contribute to the presented results. In many cases these peoples are financially supported by grants of many different organizations. Both the authors of the sections of this book and myself thank for this significant support. The list of these organizations, the numbers of the grants,

and the titles of the supported projects is so long that I must forward the interested reader for this information to the proceedings of the 10th IWSBP [297].

I like to emphasize that this book is a common work of many authors. Their names are directly associated to each section and additionally summarized in lexicographic order in the section *List of Authors* starting on page 413 and the *Index of Authors* on page 419. Many thanks to all of them for their excellent collaboration and high quality contributions. My special thank goes to Prof. Michael Miller for his *Preface* which reflects the content of the whole book in a compact and clear manner, Prof. Christian Posthoff for correction of the English text, and Matthias Werner for setting up the LaTeX-project of the book and improving the quality of the book using many LaTeX-tools.

Finally, I like to thank Ms. Carol Koulikourdi for her idea to prepare this book, the acceptance to prepare this scientific book using LaTeX, and for her very kind collaboration. I hope that all readers enjoy to read the book and find helpful suggestions for their own work in the future. It will be my pleasure to talk with many readers on one of the next International Workshops on Boolean Problems or on any other place.

Bernd Steinbach

Department of Computer Science
Technische Universität Bergakademie Freiberg
Freiberg, Saxony, Germany

Introduction

Applications of Boolean variables, Boolean operations, Boolean functions, and Boolean equations are not restricted to computers, but grow in nearly all fields of our daily life. May be that the digitally controlled alarm-clock wake us in the morning; we listen to the sound of digital audio broadcast during our breakfast; we buy the ticket for the train on a ticket vending machine controlled by Boolean values; we use our smart phone that transmits all information by Boolean values for business; look the wrist watch which counts and shows the time based on Boolean data so that we do not miss the end of our work; we get the bill for shopping from a pay machine which calculates the sum of the prices and initiates transmission between our bank account and the bank account of the shop; and in the evening we watch a movie in TV which is also transmitted by Boolean values. Hence, without thinking about the details, our daily life is surrounded with a growing number of devices which utilize Boolean values and operations.

Gorden E. Moore has published in 1965 a paper about the trend of components in integrated circuits which doubles approximately every 12 to 24 months. This observation is known as Moore's Law. Of course, there are many physical limits which take effect against this law, but due to the creativity of scientists and engineers this law is valid in general even now. The exponential increase of the control elements is a strong challenge for all people working in different field influenced by Boolean values.

The International Workshop on Boolean Problems (IWSBP) is a suitable event where people from all over the world meat each other to report new results, to discuss different Boolean problems, and to exchange new ideas. This book documents selected activities and results of the recent progress in the Boolean domain. All sections are written from authors which presented their new results on the 10th IWSBP in September 2012 in Freiberg, Germany.

The most general challenge in the Boolean Domain originates from the exponential increase of the complexity of the Boolean systems as stated in Moore's Law. Chapter 1 of this book deals with this problem using a Boolean task of an unlimited complexity. Chapter 2 applies the found methods to a finite, but unbelievable complex multi-valued problem of more than 10^{195} color patterns. The last open problem in this field was recently solved, too. For completeness, these results are added as Section 2.5 to this book. Basic versions of all other sections of this book are published in the proceedings of the 10th IWSBP [297].

Success in solving special tasks for applications requires a well developed theoretical basis. Chapter 3 of the book contains interesting new results which can be utilized in future applications. The design of digital circuits is the main field where solutions of Boolean problems result in real devices. Seven different topics of this field are presented in Chapter 4. Not all produced devices are free of errors, due to geometrical structures of few nanometers on the chips. Hence, it is a big challenge to test such circuits which consist of millions of transistors as switching elements in logic gates. Chapter 5 deals with these Boolean problems. Following Moore's Law, in the near future single atoms must be used as logic gates. This very strong change of the basic paradigm from classical logic gates to reversible quantum gates requires a comprehensively preparatory work of scientists and engineers. The final Chapter 6 of this book shows recent results in reversible and quantum computing.

A more detailed overview of the content of this book is given in the excellent preface by Prof. Miller. Hence, it remains to wish the readers in the name of all authors pleasure while reading this book and many new insights which are helpful to solve many future tasks.

Bernd Steinbach

Department of Computer Science
Technische Universität Bergakademie Freiberg
Freiberg, Saxony, Germany

Exceptionally Complex Boolean Problems

1. Boolean Rectangle Problem

1.1. The Problem to Solve and its Properties

Bernd Steinbach Christian Posthoff

1.1.1. Motivation and Selection of the Problem

The influence of the Boolean Algebra in almost all fields of our life is growing. Even if we enjoy movies in high quality on the TV or travel by car to our friends, we are strongly supported by the Boolean Algebra.

Where the Boolean Algebra is hidden when we are watching movies? The excellent quality of *High Definition TV* (HDTV) is achieved because both the color information of each pixel of the sequence of pictures and two or more channels of sound information are transmitted by a bit stream that contains only the values 0 and 1. The restriction to this simplest alphabet restricts the influence of noise, and possible transmission errors of one or few bits in a certain period are removed using error correction codes. The number of bits which must be transmitted in each second is approximately equal to 20 millions. Hence, we see our modern TV device is in fact a specialized high-performance computer.

Looking to the second example: we typically do not think about the Boolean Algebra; we simply want to use the car to travel from place A to B. It is a precondition for our travel that the car is located on place A and not taken away be an unfriendly person. An electronic

immobilizer system based on a strongly encoded binary key helps to satisfy this precondition. Next, we have to start and not to kill the engine. This is supported by the engine management system, which calculates in real time the binary information from many sensors and controls the needed amount of gas and the correct trigger moment for each cylinder of the engine. Here the next important computer works in our car. Stopping the car before reaching an obstacle is as important as driving the car. The anti-lock braking system (ABS) helps the driver in critical situations by evaluation of the information from rotation sensors of the wheels and separate control of all four breaks. These are only some selected tasks where the Boolean Algebra inside of computers supports the driver of a car. Further examples are electronic brake-force distribution (EBD), electronic stability control (ESC) or GPS navigation systems. The needed high security is reached in all these systems by the utilization of binary values, which have the highest possible noise immunity.

The benefit of Boolean calculations was discovered by Konrad Zuse [352]. In 1938, he finished his first computer Z1. This event marked the starting point of a gigantic development in Computer Science and its applications. Based on the Boolean Algebra Konrad Zuse and many other engineers strongly improved the technology of computers and other control systems.

Based on the success of the development of digital integrated circuits, Gordon E. Moore published in 1965 an article that included the prediction of an exponential increase of the number of components in digital circuits in fixed periods of time. That means, the performance of electronic devices doubles approximately every one to two years. Evaluating the reached number of transistors of integrated circuits, the truth of the so-called *Moore's Law* can be accepted until now.

This exponential increase of the number of components in electronic devices maps directly onto the complexity of the realized Boolean functions and generates strong challenges for their representation and manipulation. Many different Boolean tasks must be solved directly for the digital circuits. These tasks comprise the design, the analysis, the comparison and the test of digital circuits. Both the available larger memory size and the very high computation power of several computation cores of the *central processing unit* (CPU) or even several

hundreds of cores of the *graphics processing unit* (GPU) make the realization of applications of all fields of our life possible. Altogether this originates in a wide range of different Boolean tasks. It is a common property of these tasks that the number of used Boolean variables and consequently the size of the Boolean function grows more and more into extreme regions.

It is not possible to discuss all these Boolean problems in this book. However, we want to contribute in a generalized manner to the complexity problem in the Boolean domain. In order to do that we select a Boolean task which

1. is easy to understand,
2. requires many Boolean calculations,
3. does not have any restriction in the size, and
4. has a simple solution for each fixed size of the problem.

Slightly increased values of the control parameters of such a problem can cause an extension of the required runtime for the solution process of several orders of magnitude. Finding improvements for the solution process for such a task can be useful for other Boolean tasks.

A task which holds the enumerated requirements is the calculation of the maximal number of edges within a bipartite graph that does not contain any cycle of the length four. This problem of graph theory is equivalent to rectangle-free grids. We explain these two different views of the selected Boolean task in the next two subsections.

1.1.2. The Problem in Context of Graph Theory

Graphs appear in our life without thinking about them, alike the Boolean Algebra discussed in the previous subsection. Basically, a graph G is specified by two sets. The first set contains the vertices $v_i \in V$ of the graph. An example of the elements of this set are the crossings in a city. The second set contains the edges $e_j \in E$.

Completing our example, the roads which connect the crossings, are the elements of this second set for the example of the road net of a city.

A graph can be built in a similar way for the rail net of a railway station. The vertices are in this case the switches, and the edges are the rails between them. From a more general point of view, the graph can be built for a rail net of a whole country or even a continent where the vertices are in this case the cities connected to the rail net, and the edges are the rails between these cities.

The sets of the graph must not necessarily be the representation of real things like streets or rails. In the case of a flight net of an air company, the provided flight connections between selected airports describe the edges E and build together with these airports as set of vertices V the graph $G(V, E)$.

Graphs can be split into directed and undirected graphs, based on the direction of the edges. We will focus on undirected graphs. Furthermore, it is possible to distinguish graphs based on weights associated to the vertices and / or to the edges. We concentrate in this section on unweighted graphs. One more possibility to divide graphs in certain classes is the split of the set of vertices into subsets and the definition of constraints for the edges between these subsets.

The problem we want to study in this section is defined on so-called *bipartite graphs*. In a bipartite graph the set of vertices V is divided into two subsets of vertices V_1 and V_2 with $V = V_1 \cup V_2$. Each edge of a bipartite graph connects one vertex $v_i \in V_1$ with one vertex $v_j \in V_2$. Consequently, there are no edges that connect two vertices of the subset V_1 or two vertices of the subset V_2.

A bipartite graph can be used to describe assignments of items of two sets. As example, we can take the set of bus stations and the set of bus lines in a city. An edge of the bipartite graph describes the assignment that the bus line stops on the connected bus station. The wide range of applications of bipartite graphs becomes visible by a second example. The two sets are in this example the professors of a university and the modules studied by the students. In this case, areas of the experiences of the professors decide about possible modules to

teach. Such a bipartite graph is an important input to solve the difficult problem to build the class schedule for a semester.

A third example brings us finally to the problem to solve. Here we take as first set of vertices the inputs of **NAND**-gates of a circuit and as second set the outputs of the same **NAND**-gates. The **NAND**-gates are chosen due to the property that each Boolean function can be built with this type of gates alone. Each gate merges the logical values of its inputs to the logical value on its output. Hence, the gates of a circuit generate assignments of edges of the bipartite graph due to the internal structure of the gate. In order to reach the needed behavior of the circuit, the outputs of internal gates must be connected by wires with inputs of certain other gates. These wires describe also assignments between the subsets of edges and must be expressed by edges in the bipartite graph.

There are two types of digital circuits: combinatorial circuits and sequential circuits. Both of them can be realized by **NAND**-gates. A bipartite graph, as explained in the last paragraph, is suitable for their description. Assume that a combinatorial circuit must be designed. The associated bipartite graph must not include a cycle of the length 4 because such a cycle causes a memory behavior so that we have a sequential circuit. We do not want to go deeper into the details of digital circuits in this section, but we concentrate on bipartite graphs which do not include any cycle of the length 4.

Now we are prepared to describe formally the problem to solve. The set of vertices V of the bipartite graph is divided into the subsets of vertices V_1 and V_2 with

$$
\begin{aligned}
&V_1 \cup V_2 = V \ , \\
&V_1 \cap V_2 = \emptyset \ , \\
&\qquad m = |V_1| \text{ is the number of vertices in the subset } V_1 \ , \text{ and} \\
&\qquad n = |V_2| \text{ is the number of vertices in the subset } V_2 \ .
\end{aligned}
$$

The number of different graphs $G_{m,n}(V_1, V_2, E)$ with edges from V_1 to V_2 is equal to 2^{m*n}. We select from this complete set of graphs the graphs that do not include a cycle C_4 of the length 4. Such graphs are called *cycle-4-free* and can be expressed by $G_{m,n}^{C_4 f}(V_1, V_2, E)$. The

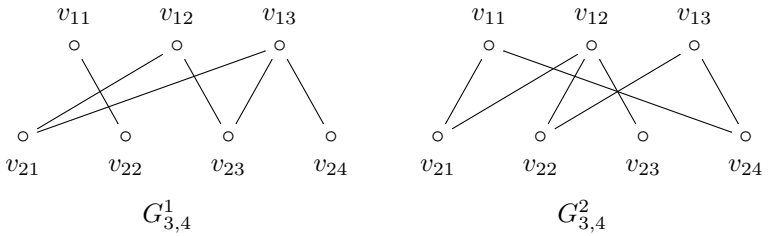


Figure 1.1. Two bipartite graphs: $G^1_{3,4}$ with C_4 and $G^2_{3,4}$ without C_4.

number of edges of a graph $G^{C_4f}_{m,n}(V_1, V_2, E)$ is labeled by:

$$n_e(G^{C_4f}_{m,n}(V_1, V_2, E)) \ .$$

The task to solve consists in finding the maximal number of edges $\mathbf{maxc}_4\mathbf{f}(m, n)$ of all cycle-4-free bipartite graphs $G^{C_4f}_{m,n}(V_1, V_2, E)$ with $m = |V_1|$ and $n = |V_2|$:

$$\mathbf{maxc}_4\mathbf{f}(m, n) = \max_{G^{C_4f}_{m,n}(V_1,V_2,E)} n_e(G^{C_4f}_{m,n}(V_1, V_2, E)) \ . \tag{1.1}$$

This task can be solved for all sets of bipartite graphs with $m > 0$ and $n > 0$. It is a challenge to solve this task because the number of bipartite graphs depends exponentially on $m * n$. The verification of the solution of this task is quite easy because $\mathbf{maxc}_4\mathbf{f}(m, n)$ is a single integer number.

Figure 1.1 shows as example two simple bipartite graphs with $m = 3$ and $n = 4$. The left graph $G^1_{3,4}$ of Figure 1.1 contains six edges $n_e(G^1_{3,4}) = 6$:

$$\{e(v_{11}, v_{22}), e(v_{12}, v_{21}), e(v_{12}, v_{23}), e(v_{13}, v_{21}), e(v_{13}, v_{23}), e(v_{13}, v_{24})\} \ .$$

Four of them

$$\{e(v_{12}, v_{21}), e(v_{12}, v_{23}), e(v_{13}, v_{21}), e(v_{13}, v_{23})\}$$

describe a cycle of the length 4. Hence, the graph $G^1_{3,4}$ does not belong to the set of cycle-4-free bipartite graphs $\left\{G^{C_4f}_{3,4}(V_1, V_2, E)\right\}$:

$$G^1_{3,4}(V_1, V_2, E) \notin \left\{G^{C_4f}_{3,4}(V_1, V_2, E)\right\} \ .$$

The right graph $G^2_{3,4}$ of Figure 1.1 contains seven edges $n_e(G^2_{3,4}) = 7$. Despite the larger number of edges there is no cycle of the length 4 within the bipartite graph $G^2_{3,4}$. Hence, this graph belongs the set of cycle-4-free bipartite graphs $\left\{G^{C_4 f}_{3,4}(V_1, V_2, E)\right\}$:

$$G^2_{3,4}(V_1, V_2, E) \in \left\{G^{C_4 f}_{3,4}(V_1, V_2, E)\right\} \ .$$

It is not possible to add an edge to the graph $G^2_{3,4}$ without loss of the property that it is cycle-4-free. An exhaustive evaluation of all bipartite graphs $G_{3,4}(V_1, V_2, E)$ confirms that $\mathbf{maxc}_4\mathbf{f}(3, 4) = 7$.

1.1.3. Rectangle-free Grids

An alternative to the graphical representation of a bipartite graph $G_{m,n}(V_1, V_2, E)$ is an adjacency matrix. Such a matrix is also called *grid*. We prefer this short term in the rest of Chapter 1.

The subset $V_1 \subset V$ is mapped to the set of rows R of the grid:

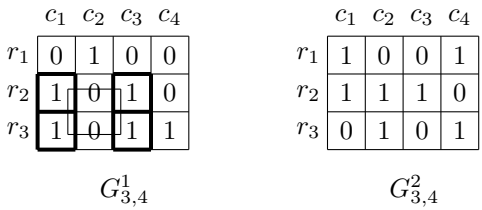
$$v_{1i} \in V_1 \implies r_i \in R \ .$$

Similarly, we map the subset $V_2 \subset V$ to the set of columns C of the grid:

$$v_{2k} \in V_2 \implies c_k \in C \ .$$

As usual for a matrix, the row numbers r_i of the grid grow top down, and the column numbers c_k grow from the left to the right. The elements of the grid are Boolean values. A value 1(0) in the position of row r_i and column c_k means that the edge $e(r_i, c_k) \in E$ belongs (does not belong) to $G_{m,n}(V_1, V_2, E)$.

Figure 1.2 shows the same bipartite graphs $G^1_{3,4}$ and $G^2_{3,4}$ of Figure 1.1 expressed by their grids (adjacency matrices). Four positions of the left grid $G^1_{3,4}$ of Figure 1.2 are framed by thick lines. These four positions describe the four edges of the cycle C_4 which is also called a complete subgraph $K_{2,2}$. It can be seen that these four position are located in the corners of a rectangle. There is a rectangle when all four cross-points of two rows and two columns carry values 1.



$G^1_{3,4}$ $G^2_{3,4}$

Figure 1.2. Grids (adjacency matrices) of two bipartite graphs.

The problem to solve, introduced in the previous subsection, can be expressed for a grid in the following compact manner:

> *What is the largest number of Boolean values* 1 *that can be assigned to the elements of a grid of* m *rows and* n *columns such that not all four corners of each rectangle of any pair of rows and any pair of columns are labeled with the value* 1.

We call this largest number of rectangle-free grids of m rows and n columns $\mathbf{maxrf}(m,n)$ and use the term *Boolean Rectangle Problem* (BRP) as name of the problem.

The grid representation of the bipartite graph emphasizes the Boolean nature of this problem. The cells of the grid contain either the Boolean value 0 or the Boolean value 1. Hence, each cell of the row r_i and the column c_k represents a Boolean variable x_{r_i,c_k}.

There are $m{*}n$ such Boolean variables for the grid $G_{m,n}$. The Boolean function $f_r(\mathbf{x})$ (1.2) is equal to 1 in the incorrect case that Boolean values 1 are assigned to the variables in the corners of the rectangle selected by the rows r_i and r_j and by the columns c_k and c_l:

$$f_r(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l}) = \\ x_{r_i,c_k} \wedge x_{r_i,c_l} \wedge x_{r_j,c_k} \wedge x_{r_j,c_l} \ . \qquad (1.2)$$

The conditions of the Boolean Rectangle Problem for a grid $G_{m,n}$ are met when the function $f_r(\mathbf{x})$ (1.2) is equal to 0 for all rectangles which

can be expressed by:

$$\bigvee_{i=1}^{m-1} \bigvee_{j=i+1}^{m} \bigvee_{k=1}^{n-1} \bigvee_{l=k+1}^{n} f_r(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l}) = 0 \ . \tag{1.3}$$

It can be verified that the function $f_r(\mathbf{x})$ (1.2) is equal to 0 for all 18 possible rectangles of the grid $G^2_{3,4}$ of Figure 1.2. Hence, $G^2_{3,4}$ is a rectangle-free grid $G^{rf}_{3,4}$.

The solution of the Boolean equation (1.3) is the set of all rectangle-free grids of m rows and n columns. The number of values 1 in such a grid is $n_1(G_{m,n})$. Counting the values 1 in the grids of Figure 1.2, we get as example $n_1(G^1_{3,4}) = 6$ and $n_1(G^2_{3,4}) = 7$. It is our final aim to find the value: $\mathbf{maxrf}(m,n)$ which is the maximal number of values 1 of all rectangle-free grids of m rows and n columns:

$$\mathbf{maxrf}(m,n) = \max_{G^{rf}_{m,n}} n_1(G^{rf}_{m,n}) \ . \tag{1.4}$$

It should be mentioned that there is another problem [124] which is strongly related to the Boolean Rectangle Problem. It is the Zarankiewicz Problem which is named after the Polish mathematician Kazimierz Zarankiewicz. Zarankiewicz has contributed more than 60 years ago to this problem. The first sentence in the abstract of [249] is:

> Zarankiewicz, in problem P 101, *Colloq. Math.*, **2** (1951), p. 301, and others have posed the following problem: Determine the least positive integer $k_{\alpha,\beta}(m,n)$ so that if a 0,1-matrix of the size m by n contains $k_{\alpha,\beta}(m,n)$ ones then it must have a α by β submatrix consisting entirely of ones.

One difference between the Boolean Rectangle Problem (BRP) and the Zarankiewicz Problem (ZP) is that these problems evaluate opposite sides of the same problem. The BRP searches for the *largest* number of values 1 in the grids of the size m by n which does *not contain* any rectangle (this is a submatrix 2 by 2) consisting entirely of ones. The ZP searches for the *smallest* number of values 1 in the grids

of the size m by n which *must contain* a submatrix 2 by 2 consisting entirely of ones. Hence,

$$\mathbf{maxrf}(m,n) = k_{2,2}(m,n) - 1 \ .$$

Furthermore, the Zarankiewicz Problem is more general in the sense that not only rectangles as submatrices of the size 2 by 2 but also submatrices of larger sizes are taken into account.

1.1.4. Estimation of the Complexity

The 0-pattern of a grid is specified by the values of the Boolean variables x_{r_i,c_k}. There are $m * n$ Boolean variables for the grid $G_{m,n}$. Hence, in order to find the maximal number $\mathbf{maxrf}(m,n)$ of values 1 of all grids $G_{m,n}$,

$$n_{gp}(m,n) = 2^{m*n} \tag{1.5}$$

different grid patterns must be evaluated.

The function (1.2) must be equal to 0 for each possible rectangle. The number of all possible rectangles depends on the number of rows m and the number of columns n of a grid $G_{m,n}$. Each pair of rows generates together with each pair of columns one possible rectangle. Hence,

$$n_r(m,n) = \binom{m}{2} * \binom{n}{2} \tag{1.6}$$

rectangles must be verified.

For each of the n_{gp} grid patterns all n_r rectangles (1.6) must be evaluated. The number of all rectangles n_{ar} (1.7) which must be evaluated is equal to the product of n_{gp} (1.5) and n_r (1.6).

$$n_{ar}(m,n) = 2^{m*n-2} * m * n * (m-1) * (n-1) \tag{1.7}$$

The evaluation of these n_{ar} rectangles of grid patterns takes on an Intel Quad Core i7 processor 1.5 hours for $m = n = 6$, but already 2.7 years for $n = m = 7$. However, we are interested to solve BRP for significantly larger sizes of the grid. A challenge of Chapter 2 requires as subproblem to solve the BRP for the grid of the size $m = n = 18$.

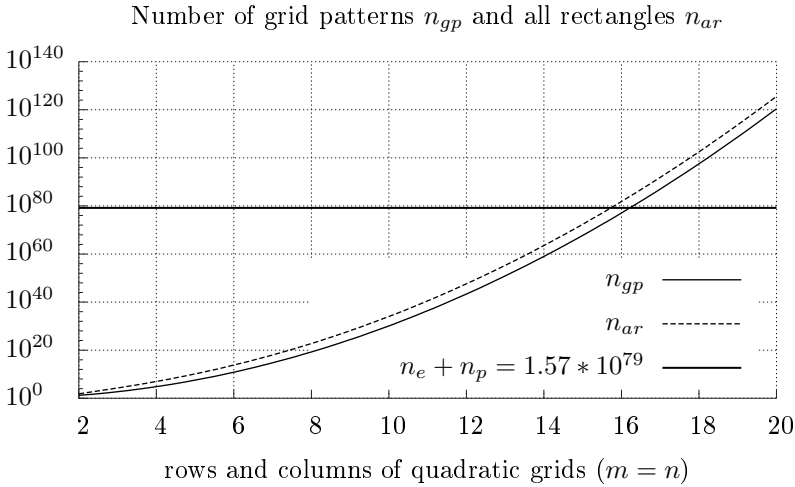


Figure 1.3. Number of grid patterns n_{gp} (solid line) and all included rectangles (dashed line) for quadratic grids using a logarithmic scale of the vertial axis.

The estimated time for the evaluation of $n_{ar}(18, 18)$ is approximately 8.7 * 10^{84} years [333].

In order to get an impression of the complexity of the BRP, Figure 1.3 shows both the number of different grid patterns n_{gp} and the number of all rectangles n_{ar} for quadratic grids in comparison to the number of all electrons n_e and all protons n_p of the whole universe (thick horizontal line) in the range of 2 to 20 rows and columns. This, with slightly growing values of the numbers of rows m and columns n, extremely growing number of grid patterns can be restricted to classes of grid patterns. The exchange of any pair of rows or any pair of columns does not change the existing rectangles characterized by (1.2). Both $m!$ permutations of rows and $n!$ permutations of columns of the grid do not change both the number of assigned values 1 and the number of rectangles consisting entirely of the value 1. Hence, there is a potential of improvements of $m! * n!$ which grows from 518.400 for the grid $G_{6,6}$ to $4 * 10^{31}$ for the grid $G_{18,18}$. We will show in the following section how this potential of improvements can be utilized.

1.2. Search Space Restriction

Bernd Steinbach Christian Posthoff

1.2.1. Basic Approach: Complete Evaluation

The task to solve consists in finding the maximal number $\mathbf{maxrf}(m,n)$ of values 1 which can be assigned to the grid $G_{m,n}$ of m rows and n columns without violating the rectangle condition (1.3). The maximal number $\mathbf{maxrf}(m,n)$ must be found when all 2^{m*n} different grid patterns are evaluated. The evaluation of each grid pattern can be split into two subtasks:

1. verify whether the function $f_i(\mathbf{x})$, which is associated to the grid pattern $G^i_{m,n}$, satisfies Equation (1.3), and

2. count the number of function values 1 of $f_i(\mathbf{x})$ for the decision whether $f_i(\mathbf{x})$ belongs to the set of maximal grid patterns.

These two subtasks can be solved in an algorithm in two different orders. Each of these orders has an advantage. In case of the order VERIFY followed by COUNT, the second subtask can be omitted if $f_i(\mathbf{x})$ does not satisfy the rectangle condition (1.3). Vice versa, in case of the order COUNT followed by VERIFY, it is not necessary to verify whether $f_i(\mathbf{x})$ satisfies the rectangle condition (1.3) if the number of values 1 of $f_i(\mathbf{x})$ is smaller than a known number of values 1 of another $f_j(\mathbf{x})$, which satisfies the rectangle condition (1.3).

In this basic approach we use the order of the subtasks as shown in the enumeration given above. There are again two different possibilities to solve the first subtask. Due to the finite numbers of m rows and n columns the set of grid patterns $\left\{G^i_{m,n} | i = 1, \ldots, 2^{m*n}\right\}$ is also a finite set. Hence, the $m*n$ function values of each associated function $f_i(\mathbf{x})$ can be substituted into Equation (1.3), and it can be checked whether the rectangle condition is satisfied for $f_i(\mathbf{x})$. However, this is a very time-consuming procedure due to the large number 2^{m*n} of different functions.

The solution of a Boolean equation is a set of binary vectors. Hence, a Boolean equation of $m*n$ variables divides the set of all 2^{m*n} vectors of the whole Boolean space B^{m*n} into two disjoint subsets. Each solution vector of $m*n$ Boolean values that solves Equation (1.3) describes one function $f_i(\mathbf{x})$ of this Boolean space. This set is the solution of the first subtask.

The Boolean function $f_r(\mathbf{x})$ (1.2) describes the incorret case that all grid values in the corners of the rectangle selected by the rows r_i and r_j and by the columns c_k and c_l are equal to 1. The conditions of the Boolean rectangle problem on a grid $G_{m,n}$ are met when the function $f_r(\mathbf{x})$ (1.2) is equal to 0 for all rectangles which can be expressed by the restrictive equation (1.3). The set of solutions of a Boolean equation remains unchanged if the functions on both sites are replaced by their negated function. Using this transformation, we get the characteristic equation (1.8) from (1.3). On the left-hand side of (1.8) the Law of De Morgan is applied which changes all given disjunctions into conjunctions.

$$\bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} \bigwedge_{k=1}^{n-1} \bigwedge_{l=k+1}^{n} \overline{f_r(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l})} = 1 \ . \qquad (1.8)$$

Equation (1.8) can be efficiently solved using XBOOLE [240], [302]. XBOOLE is a library of more than 100 operations. The basic data structure of XBOOLE is the list of ternary vectors (TVL). The operations of XBOOLE can be used in programs written in the languages C or C++. A simple possibility to use XBOOLE is a program called XBOOLE Monitor. This program can be downloaded and used without any restrictions for free. The URL is:

`http://www.informatik.tu-freiberg.de/xboole` .

Algorithm 1.1 describes how the complete evaluation of a grid $G_{m,n}$ can be done using XBOOLE operations. In lines 1 to 11 the set of all rectangle-free grids is calculated. Based on these data this algorithm determines in line 12 the number of solutions of (1.8) and in line 13 the number of ternary vectors needed to represent all solutions. The ratio between these two numbers provides an insight into the power

Algorithm 1.1 CompleteEval(m, n)

Require: number of rows m and number of columns n of the grid $G_{m,n}$
Ensure: **maxrf**(m, n): maximal number of assignments of values 1 in the rectangle-free grid $G_{m,n}$
Ensure: n_{sol}: number of the solutions of (1.8)
Ensure: n_{tv}: number of ternary vectors of the solutions of (1.8)

```
 1: all ← ∅
 2: all ← CPL(all)                                   ▷ complement
 3: for i ← 1 to m − 1 do
 4:     for j ← i + 1 to m do
 5:         for k ← 1 to n − 1 do
 6:             for l ← k + 1 to n do
 7:                 all ← DIF(all, f_r(i, j, k, l))   ▷ difference
 8:             end for
 9:         end for
10:     end for
11: end for
12: n_sol ← nbv(all)                          ▷ number of binary vectors
13: n_tv ← NTV(all)                           ▷ number of ternary vectors
14: h ← CEL(all, " − 11")                     ▷ change elements
15: maxrf(m, n) ← 0
16: for i ← 1 to n_tv do
17:     tv ← STV(h, i)                        ▷ select ternary vector
18:     n_1 ← SV_SIZE(tv)                     ▷ set of variables: size
19:     if maxrf(m, n) < n_1 then
20:         maxrf(m, n) ← n_1
21:     end if
22: end for
```

of XBOOLE. The lines 14 to 22 determine the wanted maximal value **maxrf**(m, n).

All used XBOOLE operations are indicated by capital letters. The small letters of the `nbv()` function in line 12 indicate that this is not a basic XBOOLE function. This function calculates the number of binary vectors of all solutions using some more elementary XBOOLE operations. The XBOOLE operation `NTV` in line 13 simply counts the number of ternary vectors.

$$
\begin{aligned}
\overline{f_{r1}(\mathbf{x})} \wedge \overline{f_{r2}(\mathbf{x})} \wedge \overline{f_{r3}(\mathbf{x})} &= 1 \\
1 \wedge \overline{f_{r1}(\mathbf{x})} \wedge \overline{f_{r2}(\mathbf{x})} \wedge \overline{f_{r3}(\mathbf{x})} &= 1 \\
(1 \setminus f_{r1}(\mathbf{x})) \wedge \overline{f_{r2}(\mathbf{x})} \wedge \overline{f_{r3}(\mathbf{x})} &= 1 \\
((1 \setminus f_{r1}(\mathbf{x})) \setminus f_{r2}(\mathbf{x})) \wedge \overline{f_{r3}(\mathbf{x})} &= 1 \\
((1 \setminus f_{r1}(\mathbf{x})) \setminus f_{r2}(\mathbf{x})) \setminus f_{r3}(\mathbf{x}) &= 1
\end{aligned} \tag{1.9}
$$

The idea for the procedure to solve Equation (1.8) becomes visible by the following simplified transformation. The added 1 in the second line of (1.9) is a constant 1 function that describes the whole Boolean space. This function is created in Algorithm 1.1 in the first two lines such that the complement operation `CPL` is executed to the previously initialized 0-function. Lines 3 to 6 and 8 to 11 describe the nested loops as required in (1.8). The difference operation `DIF` excludes in each application as shown in the last three lines of (1.9) one incorrect rectangle pattern from the remaining solution set.

It is an advantage of the chosen approach that only the number of values 1 in the solution vectors of (1.8) must be counted. The representation of the solutions by ternary vectors restricts this effort even more. One ternary vector with d dash elements ('-') represents 2^d binary vectors.

Only the binary vector which is created by the substitution of all dashes by values 1 can belong to the set maximal grid patterns. The XBOOLE operation `CEL` is used in line 14 of Algorithm 1.1 for this transformation, which excludes the explicit count of the values 1 in $2^d - 1$ vectors. Within the same operation all given elements '0' are changed into elements '-' in order to exclude them from counting the covered positions.

In the loop over n_{tv} ternary vectors in lines 16 to 22, the XBOOLE operation `STV` selects each ternary vector once for the next evaluation in line 17. The number of values 1 in this vector is counted by the XBOOLE operation `SV_SIZE` in line 18. The searched value $\mathbf{maxrf}(m, n)$ is initialized in line 15 with the value 0 and replaced by a larger number n_1 of found values 1 in the solution vectors as shown in lines 19 to 21 of Algorithm 1.1.

Table 1.1. Maximal assignments of **maxrf**(m, n) values 1 to a rectangle-free grid up to 6 rows calculated by complete evaluation

number of			values 1			complete solution set		time in milliseconds		
m	n	n_v	**maxrf**	all	ratio	n_{sol}	n_{tv}	calc.	count	all
2	2	4	3	4	0.750	15	4	0	0	0
2	3	6	4	6	0.666	54	12	0	0	0
2	4	8	5	8	0.625	189	32	0	0	0
2	5	10	6	10	0.600	648	80	0	0	0
2	6	12	7	12	0.583	2,187	192	0	0	0
2	7	14	8	14	0.571	7,290	448	16	0	16
2	8	16	9	16	0.563	24,057	1,024	31	0	31
2	9	18	10	18	0.555	78,732	2,304	47	0	47
2	10	20	11	20	0.550	255,879	5,120	94	31	125
2	11	22	12	22	0.545	826,686	11,264	141	109	250
2	12	24	13	24	0.542	2,657,205	24,576	297	608	905
2	13	26	14	26	0.538	8,503,056	53,248	656	3,198	3,854
2	14	28	15	28	0.536	27,103,491	114,688	1,404	17,972	19,376
2	15	30	16	30	0.533	86,093,442	245,760	3,167	92,836	96,003
2	16	32	17	32	0.531	272,629,233	524,288	7,348	568,279	575,627
3	2	5	4	6	0.666	54	12	0	0	0
3	3	9	6	9	0.666	334	68	16	0	16
3	4	12	7	12	0.583	1,952	326	16	0	16
3	5	15	8	15	0.533	10,944	1,485	32	0	32
3	6	18	9	18	0.500	59,392	6,580	110	46	156
3	7	21	10	21	0.476	313,856	28,451	390	843	1,233
3	8	24	11	24	0.458	1,622,016	120,048	1,872	16,896	18,768
3	9	27	12	27	0.444	8,224,768	494,537	9,578	558,840	568,418
4	2	8	5	8	0.625	189	12	0	0	0
4	3	12	7	12	0.583	1,952	68	16	0	16
4	4	16	9	16	0.563	18,521	326	16	0	16
4	5	20	10	20	0.500	165,120	1,485	32	0	32
4	6	24	12	24	0.500	1,401,445	6,580	110	46	156
5	2	10	6	10	0.600	648	80	16	0	16
5	3	15	8	15	0.533	10,944	1,457	31	0	31
5	4	20	10	20	0.500	165,120	18,769	265	312	577
5	5	25	12	25	0.480	2,293,896	216,599	4,103	79,513	83,616
6	2	12	7	12	0.583	2,187	192	0	0	0
6	3	18	9	18	0.500	59,392	6,418	94	31	125
6	4	24	12	24	0.500	1,401,445	130,521	2,184	27,317	29,501

Table 1.1 summarizes the experimental results for grids up to 6 rows restricted by a memory size of 2 Gigabytes and executed on a PC using a single core of the CPU i7-940 running on 2.93 GHz. This computer was used for all experiments which are described in this subsection. The first three columns of Table 1.1 specify the values of rows m, columns m, and Boolean variables n_v of the evaluated grids. The main results are the maximal numbers of values 1 of rectangle-free grids in the column **maxrf**. The values in the column *ratio* are calculated by:

$$ratio = \frac{\mathbf{maxrf}(m,n)}{all} \ ,$$

where $all = m * n$. It can be seen that this ratio decreases for growing numbers of rows or columns of the grid.

The sizes of the complete solution sets give an impression of the enormous complexity of the problem to be solved. The benefit of the utilization of lists of ternary vectors as main data structure in XBOOLE [240] becomes visible comparing the values in the columns of all correct grids n_{sol} and the number of ternary vectors n_{tv} required to store these sets.

The Boolean function of the simplest grid of 2 rows and 2 columns depends on 4 variables and has $2^4 = 16$ function values. $n_{sol} = 15$ is the number of function values 0 associated to correct grids; only that grid is excluded in which all four positions (all for edges of the bipartite graph) are assigned to the value 1. These 15 solutions are expressed by $n_{tv} = 4$ ternary vectors. The ratio between n_{sol} and n_{tv} reaches nearly three orders of magnitude for growing numbers of rows and columns of the grid.

The power of XBOOLE becomes visible looking at the time required for the calculation. All 272,629,233 correct grid patterns of $G_{2,16}$ were calculated within only 7.348 seconds. It is an advantage that not all of these patterns must be counted to find the searched value of $\mathbf{maxrf}(m,n)$, but only the elements of 524,288 ternary vectors. Despite this advantage it takes 568.279 seconds for the counting procedure for $G_{2,16}$.

The basic approach is limited by the available memory to store the solution of Equation (1.8). In [295] we have published an recursive

approach in which the limit of space is compensated by additional time. In the same publication the needed time is reduced by parallel algorithms using the message passing interface MPI [229].

1.2.2. Utilization of Rule Conflicts

The strategy of the basic approach was the restriction of the set of all 2^{m*n} grid patterns to the correct patterns and the detection of patterns of maximal numbers of values 1 out of them. The very large search space of 2^{m*n} can be divided into $1 + m * n$ subspaces that contain grid patterns of a fixed number of values 1 in the range from zero values 1 to $m * n$ values 1. In Subsection 1.2.3 we will utilize the ordered evaluation of these subspaces.

Here we restrict ourselves to the subtask of finding correct grid patterns of a fixed number of values 1. The evaluation of such a subspace can be realized using a recursive algorithm that generates all permutations of the fixed number of values 1.

Solving this subtask, the search space can be restricted even more. It is not necessary to assign further values 1 to a partially filled grid that contradicts the rectangle-free rule (1.3). Combining the recursive assignment of values 1 with the rule check for conflicts regarding a newly assigned value 1 allows an immediate backtrack in the recursion in the case of a rule conflict. In this way many incorrect grid patterns must neither be constructed nor checked for the rectangle condition.

Algorithm 1.2 generates recursively all permutations of grid assignments of n_1 values 1 for a grid $G_{m,n}$ with $m \geq 2$ rows and $n \geq 2$ columns which satisfy rectangle-free rule (1.3). All three parameters *level*, *next*, and *nop* are equal to 0 in the case of the initial call.

The variable *level* carries the value of the level of the recursion. On each level a single value 1 is assigned to one grid position. Hence, the value of the variable *level* is equal to the number of values 1 assigned to the grid. The grid contains $pmax = m * n$ grid positions which are numbered consecutively in the range from 0 to $m * n - 1$.

Algorithm 1.2 GenPerm($m, n, n_1, level, next, nop$)

Require: number of rows m of the grid $G_{m,n}$
Require: number of columns n of the grid $G_{m,n}$
Require: number of values 1, $n_1 > 0$ that must be assigned to the grid $G_{m,n}$
Require: number *level* of already assigned values 1
Require: position *next* for the next assignment
Ensure: *nop*: the number of all rectangle-free permutations of n_1 values 1 in the grid $G_{m,n}$

```
1:  pmax ← m * n
2:  pos ← next
3:  while pos ≤ pmax − n1 + level do
4:      repeat
5:          if pos > next then
6:              G[(pos − 1)/n, (pos − 1) mod n] ← 0          ▷ reset to 0
7:          end if
8:          G[pos/n, pos mod n] ← 1                          ▷ assign a value 1
9:          conflict ← crc((pos/n), (pos mod n))             ▷ check condition
10:         pos ← pos + 1                                    ▷ select next grid position
11:     until pos > pmax − n1 + level OR conflict = false
12:     if conflict = true AND pos ≥ pmax − n1 + level then
13:         return nop                                       ▷ no further rectangle-free grid
14:     end if
15:     if level < n1 − 1 then
16:         nop ← GenPerm(m, n, n1, level + 1, pos, nop)     ▷ recursion
17:     else if conflict = false then
18:         nop ← nop + 1                                    ▷ correct grid pattern of n1 values 1
19:     end if
20: end while
21: return nop                                               ▷ nop correct grid patterns of n1 values 1
```

The variable *next* specifies the position in this range where the next value 1 must be assigned to the grid. Algorithm 1.2 changes this position using the internal variable *pos*. In order to simplify the mapping of the value *pos* to the associated row and column, 0-based index values are used in Algorithm 1.2. The result of the integer division (pos/n) is equal to the row index $(0, \ldots, m-1)$ selected by the value of *pos*, and the result of the remainder of this operation ($pos \mod n$) indicates the associated column index $(0, \ldots, n-1)$.

Algorithm 1.2 initializes the variables *pmax* and *pos* in lines 1 and 2, realizes the main task within the while-loop in lines 3 to 20, and returns the number *nop* of rectangle-free grid patterns of n_1 values 1 in line 21.

In case of the initial call of Algorithm 1.2, the while-condition in line 3 is satisfied for $n_1 \leq m * n$, but the if-condition in line 5 is not satisfied. Hence, a value 1 is assigned to the grid position of *pos* in line 8. All rectangles which contain this assigned value 1 in one of their corners are checked by the function $\mathrm{crc}((pos/n), (pos \bmod n))$ whether the rectangle-free rule (1.3) is satisfied. Independent on the result of this check the next position *pos* of the grid is selected in line 10 of Algorithm 1.2.

The utilization of the rule conflict is implemented in line 11 of Algorithm 1.2. If there is a rule conflict ($conflict = true$), further additional assignments of values 1 cannot satisfy the rectangle-free rule (1.3) and will be omitted, because the repeat-until-loop in lines 4 to 11 is repeated.

If position *pos* does not reach the limit given in line 11, alternative assignments of the last assigned value 1 exist. Due to the enlarged value of *pos* in line 10 the if-condition in line 5 becomes true so that the last assigned value 1 is deleted in line 6, and the next grid position is assigned to a value 1 in line 8. The repeat-until-loop in lines 4 to 11 will be left, the value 1 is assigned in line 8 without causing a conflict regarding the rectangle-free rule (1.3).

In the case that both the last assignment of a value 1 causes a conflict regarding the rectangle-rule (1.3) and the required number of values 1 n_1 cannot be reached, the if-statement in lines 12 to 14 breaks the recursion and returns the number *nop* of so far found rectangle-free assignments of n_1 values 1 to the grid.

Algorithm 1.2 must be recursively called for each rectangle-free grid that includes less than n_1 values 1. This is controlled by the if-condition in line 15 and executed in line 16. If $level = n_1 - 1$, the conflict-free assignment of n_1 values 1 is completed and will be remembered by the increment of *nop* in line 18. All rectangle-free patterns of n_1 values 1 are found due to the while-loop of Algorithm 1.2.

Table 1.2. Recursive generation of all grids $G_{5,5}$ containing 10, 11, 12 or 13 values 1

	10 values 1		11 values 1		12 values 1		13 values 1	
	level	conflicts	level	conflicts	level	conflicts	level	conflicts
	0	0	0	0	0	0	0	0
	1	0	1	0	1	0	1	0
	2	0	2	0	2	0	2	0
	3	48	3	39	3	33	3	30
	4	700	4	508	4	373	4	289
	5	3,980	5	3,980	5	2,600	5	1,691
	6	16,584	6	13,584	6	13,584	6	7,956
	7	61,695	7	37,895	7	28,235	7	28,235
	8	203,777	8	103,973	8	52,353	8	32,553
	9	489,410	9	243,106	9	105,898	9	39,290
			10	340,260	10	154,716	10	57,684
					11	96,480	11	39,744
							12	6,264
number of all grids		3,268,760		4,457,400		5,200,300		5,200,300
grids without any conflict		388,440		108,000		7,800		0
time in milliseconds		172		125		93		32

Table 1.2 summarizes some experimental results and reveals the benefits of the explained restricted recursive algorithm. It is not necessary to generate all permutations of grids of a fixed number of values 1. After the assignment of 3 values 1 (labeled by level 3 in Table 1.2) 30 or more assignments of the fourth value 1 cause a conflict with the rectangle-free rule (1.3). Hence, in these cases the recursive invocation of the function **GenPerm**() with the next higher level can be omitted. In this way the search space is strongly reduced without any loss of solutions. On each higher level the same property is utilized and restricts the search space even more.

From Table 1.1 it is known that the basic approach needs 83,616 milliseconds to calculate $\mathbf{maxrf}(5,5) = 12$. The efficiency of Algorithm 1.2 becomes visible by the required time to calculate all rectangle-free patterns for fixed numbers n_1 of values 1 for the same grid of $m = 5$

Algorithm 1.3 OrderedEval(m, n)

Require: number of rows m and number of columns n of the grid $G_{m,n}$

Ensure: **maxrf**(m, n): maximal number of assignments 1 in the rectangle-free grid $G_{m,n}$

1: $n_1 \leftarrow 3$ ▷ there cannot be a conflict for 3 assigned values 1
2: **repeat**
3: $n_1 \leftarrow n_1 + 1$ ▷ select next grid position
4: $nop \leftarrow \text{GenPerm}(m, n, n_1, 0, 0, 0)$ ▷ evaluation for n_1
5: **until** $nop = 0$ ▷ no rectangle-free grid for n_1
6: **maxrf**$(m, n) \leftarrow n_1 - 1$ ▷ largest number of n_1 with $nop > 0$
7: **return maxrf**(m, n)

rows and $n = 5$ columns. The needed time to calculate all rectangle-free patterns decreases despite a growing number of all grid patterns in the subspace of a fixed number of values 1.

1.2.3. Evaluation of Ordered Subspaces

There is a monotone order within the subspaces of fixed numbers of values 1. Each grid pattern that contains $n_1 + 1$ values 1 can be constructed by an additional assignment of a single value 1 to a grid pattern that contains n_1 values 1. Hence, if there is a rectangle-conflict in the grid of n_1 values 1, at least the same conflict occurs in the constructed grid of $n_1 + 1$ values 1.

Theorem 1.1. *If no Boolean grid pattern of n_1 values satisfies the rectangle-rule* (1.3)*, no Boolean grid pattern of more than n_1 values 1 can be rectangle-free.*

Proof. The conclusion follows from the premise because all values 1 which do not satisfy the rectangle-rule (1.3) remain in the grid that is extended by one or more values 1. □

The iterative Algorithm 1.3 utilizes Theorem 1.1. Using the function GenPerm() shown in Algorithm 1.2 of Subsection 1.2.2, successively

Table 1.3. Iterative generation of grids $G_{5,5}$ and $G_{6,6}$ for fixed numbers of values 1; all time measurements are given in milliseconds

number	$G_{5,5}$			number	$G_{6,6}$		
of 1s	correct	incorrect	time	of 1s	correct	incorrect	time
0	1	0	0	0	1	0	0
1	25	0	0	1	36	0	0
2	300	0	0	2	630	0	0
3	2,300	0	0	3	7,140	0	0
4	12,550	100	0	4	58,680	225	0
5	51,030	2,100	16	5	369,792	7,200	31
6	156,500	20,600	15	6	1,837,392	110,400	203
7	357,100	123,600	47	7	7,274,880	1,072,800	1,279
8	582,225	499,350	94	8	22,899,240	7,361,100	3,105
9	627,625	1,415,350	171	9	56,508,480	37,634,800	9,048
10	388,440	2,880,320	172	10	106,441,776	147,745,080	20,997
11	108,000	4,349,400	125	11	146,594,016	454,211,280	38,080
12	7,800	5,192,500	93	12	138,031,200	1,113,646,500	52,635
13	0	5,200,300	32	13	79,941,600	2,230,848,000	54,350
				14	23,976,000	3,772,234,200	41,870
				15	2,769,120	5,565,133,440	25,678
				16	64,800	7,307,807,310	13,837
				17	0	8,597,496,600	7,145
complete time:			765				268,258

all subspaces for a fixed number n_1 of values 1 are evaluated in line 4 of Algorithm 1.3. There cannot be a rectangle-conflict for 3 or less values 1. Hence, the variable n_1 is initialized with the value 3 in line 1 of Algorithm 1.3. Due to the monotony property, the repeated invocation for a successively incremented number n_1 of values 1 within the repeat-until-loop in lines 2 to 5 is finished when a completely evaluated subspace does not contain any rectangle-free grid indicated by $nop = 0$. The searched value $\mathbf{maxrf}(m, n)$ is equal to the largest number of n_1 with $nop > 0$ which is assigned in line 6 of Algorithm 1.3.

Table 1.3 shows experimental results of Algorithm 1.3 for the grids $G_{5,5}$ and $G_{6,6}$. We added the values of rectangle-free grids for the cases $n_1 = 0, \ldots, 3$ for completeness. The other values in the columns *correct* are the results *nop* of the function GenPerm() called in line 4 of Algorithm 1.3. The wanted result $\mathbf{maxrf}(m, m)$ is indicated as bold

number in the columns *number of 1s*. The numbers of grid patterns which do not satisfy the rectangle-rule (1.3) are shown in the columns *incorrect*. These values are calculated by

$$\binom{m*n}{n_1} - nop(n_1) \ . \tag{1.10}$$

These numbers of incorrect grid patterns grow both absolutely and relatively with regard to the correct grid patterns for growing numbers of values 1 in the grid patterns.

The comparison of the results in Tables 1.1 and 1.3 reveals the efficiency of Algorithm 1.3. The time to find the value $\mathbf{maxrf}(5,5) = 12$ is reduced from 83,616 milliseconds for the complete evaluation using Algorithm 1.1 to 765 milliseconds for the successive evaluation of ordered subspaces using Algorithm 1.3. This is an improvement factor of 109.3.

In addition to the achieved speed-up the BRP could be solved for larger grids using Algorithm 1.3. Restricted by the available memory the largest solvable quadratic grid of the complete evaluation using Algorithm 1.1 is $G_{5,5}$. The successive evaluation of ordered subspaces using Algorithm 1.3 allows to find the value $\mathbf{maxrf}(6,6) = 16$. That means that the evaluated set of grid pattern is improved from 2^{5*5} to 2^{6*6} so that a $2^{11} = 2,048$ times larger set of grid patterns could be successfully evaluated.

1.2.4. Restricted Evaluation of Ordered Subspaces

It is the aim of our efforts to find the maximal value $\mathbf{maxrf}(m,n)$ of values 1 of grid patterns which does not violate the rectangle-free condition (1.3). For this purpose the number of such maximal rectangle-free grids is not required. When we evaluate a subspace of a fixed number of values 1, it is sufficient to know whether there is either at least one or no rectangle-free grid pattern.

If there is at least one rectangle-free grid pattern of n_1 values 1, the subspace with $n_1 \leftarrow n_1 + 1$ must be evaluated. In the case that a subspace of n_1 values 1 does not contain any allowed grid pattern,

no larger subspace can contain a rectangle-free grid pattern due to Theorem 1.1. Hence, the implementation of the restricted evaluation of subspaces requires only a slightly change of Algorithm 1.2 into Algorithm 1.4.

Algorithm 1.3 can be reused such that the invocation of

$$\text{GenPerm}(m, n, n_1, level, next, nop)$$

(Algorithm 1.2) is replaced by the invocation of

$$\text{GenPermRestrict}(m, n, n_1, level, next, nop)$$

(Algorithm 1.4).

Algorithm 1.4 extends Algorithm 1.2 only in line 3 by the additional condition: AND $nop = 0$. This additional condition allows the termination of Algorithm 1.4 when the first rectangle-free grid pattern of n_1 values 1 is found. The benefit of this approach is that only the subspace of the smallest number of values 1 without any rectangle-free grid pattern must be completely evaluated.

Table 1.4 shows the experimental results. The numbers of calculated grids are given in the columns *calculated* of Table 1.4. The comparison between the *complete* case of Subsection 1.2.3 with the *restricted* case of this subsection for the grid $G_{6,6}$ shows the strong improvement. The needed time is concentrated to the subspace of the smallest number of values 1 for which no rectangle-free grid pattern exists. The restricted ordered evaluation of subspaces reduces the runtime for the grid $G_{6,6}$ by a factor of 36.6 in comparison to the already improved approach of the complete ordered evaluation of subspaces.

Using the suggested restricted ordered evaluation of subspaces it was possible to find the value $\mathbf{maxrf}(7, 7) = 21$. The evaluated set of grid patterns is extended from 2^{5*5} to 2^{7*7} in comparison with the basic approach; a set could be successfully evaluated that is

$$2^{24} = 16,777,216 = 1.678 * 10^7$$

times larger.

We used the program that implements the approach of the restricted ordered subspaces of Subsection 1.2.4 for solving the BRP of grids

Algorithm 1.4 GenPermRestrict$(m, n, n_1, level, next, nop)$

Require: number of rows m, number of columns n of the grid $G_{m,n}$, number of values 1, $n_1 > 0$ that must be assigned to the grid $G_{m,n}$, number *level* of already assigned values 1, and position *next* for the next assignment

Ensure: *nop*: number of all rectangle-free permutations of n_1 values 1 in the grid $G_{m,n}$

```
1:  pmax ← m * n
2:  pos ← next
3:  while pos ≤ pmax − n1 + level AND nop = 0 do
4:      repeat
5:          if pos > next then
6:              G[(pos − 1)/n, (pos − 1) mod n] ← 0          ▷ reset to 0
7:          end if
8:          G[pos/n, pos mod n] ← 1                          ▷ assign a 1
9:          conflict ← crc((pos/n), (pos mod n))     ▷ check condition
10:         pos ← pos + 1                    ▷ select next grid position
11:     until pos > pmax − n1 + level OR conflict = false
12:     if conflict = true AND pos ≥ pmax − n1 + level then
13:         return nop                  ▷ no further rectangle-free grid
14:     end if
15:     if level < n1 − 1 then
16:         nop ← GenPerm(m, n, n1, level + 1, pos, nop)   ▷ recursion
17:     else if conflict = false then
18:         nop ← nop + 1          ▷ correct grid pattern of n1 values 1
19:     end if
20: end while
21: return nop              ▷ nop correct grid patterns of n1 values 1
```

$G_{m,n}$. We restricted the runtime in this experiment to about 10 minutes and the number of rows to $m = 9$. The results are shown in Table 1.5.

1.2.5. Analysis of the Suggested Approaches

We investigated several restrictions of the search space to solve the Boolean rectangle problem (BRP). The simple task of counting the

Table 1.4. Restricted iterative generation of grids $G_{6,6}$ and $G_{7,7}$ for fixed numbers of values 1; the time is measured in milliseconds

	$G_{6,6}$					$G_{7,7}$	
number	complete		restricted		number	restricted	
of 1s	calculated	time	calculated	time	of 1s	calculated	time
0	1	0	1	0	0	1	0
1	36	0	1	0	1	1	0
2	630	0	1	0	2	1	0
3	7,140	1	1	0	3	1	0
4	58,680	5	1	0	4	1	0
5	369,792	34	1	1	5	1	0
6	1,837,392	193	1	0	6	1	0
7	7,274,880	873	1	0	7	1	0
8	22,899,240	3,144	1	0	8	1	0
9	56,508,480	9,129	1	0	9	1	0
10	106,441,776	21,171	1	0	10	1	0
11	146,594,016	38,361	1	0	11	1	0
12	138,031,200	52,982	1	3	12	1	0
13	79,941,600	54,774	1	2	13	1	0
14	23,976,000	42,197	1	2	14	1	47
15	2,769,120	26,664	1	1	15	1	47
16	64,800	14,183	1	132	16	1	31
17	0	7,293	0	7,262	17	1	31
					18	1	16
					19	1	6,630
					20	1	44,819
					21	1	26,660
					22	0	1,638,837
complete time:		271,006		7,404			1,717,118

number of values 1 required to find the value of $\mathbf{maxrf}(m, n)$ is very time-consuming. In the basic approach this task is restricted to 6.84 percent of the grid patterns $G_{5,5}$ that satisfy the rectangle-free condition. Utilizing the ternary representation of the rectangle-free grid patterns even only 0.65 percent of all grid patterns must be really counted. Hence, the utilization of XBOOLE reduces the runtime by a factor of more than 100. Restricted by the memory space of 2 GB, the largest solved grid is $G_{5,5}$ for which $2^{5*5} \approx 3.4 * 10^7$ different grid patterns exist.

As a second complete approach we combined the utilization of rule

Table 1.5. Maximal numbers of values 1 in grids $G_{m,n}$ of m rows and n columns calculated within about 10 minutes using the approach of restricted evaluation of ordered subspaces

m	n																	
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	4	6	7	8	9	10	11	12	13	14	15	16						
4	5	7	9	10	12	13	14	15	16	17								
5	6	8	10	12	14	15	17	18										
6	7	9	12	14	16	18	19											
7	8	10	13	15	18	21												
8	9	11	14	17	19													
9	10	12	15	18														

conflicts with the evaluation of ordered subspaces. Here we replaced the counting procedure by the generation of grid patterns of a fixed number of values 1. Using Theorem 1.1 the evaluation of subspaces with more than $\mathbf{maxrf}(m,n)+2$ values 1 can be completely omitted. The utilization of rule conflicts restricts the effort to decide whether the evaluated grid pattern is rectangle-free or not because one detected conflict is sufficient to exclude the pattern from the solution set. The joint use of these two methods reduces the runtime for the grid $G_{5,5}$ by a factor of more than 100. Using this approach $\mathbf{maxrf}(6,6) = 16$ could be calculated for the quadratic grid $G_{6,6}$ for which $2^{6*6} \approx 6.9 * 10^{10}$ different grid patterns exist.

As final restriction of the search space we evaluated the grid patterns of a subspace of a fixed number of values 1 only until the first rectangle-free grid is found. In this way in less than 10 minutes $\mathbf{maxrf}(7,7) = 21$ could be calculated for the quadratic grid $G_{7,7}$ for which $2^{7,7} \approx 5.6 * 10^{14}$ different grid patterns exists.

Taking into account the achieved improvement of more than seven orders of magnitude, we can conclude that the restriction of the search space is a powerful method. However, the restricted resources of the computers and the exponential increase of the number of different grid pattern depending on the numbers of rows and columns limit the application of this approach.

1.3. The Slot Principle

Bernd Steinbach Christian Posthoff

1.3.1. Utilization of Row and Column Permutations

We learned in Section 1.2 both the benefit of recursively generated grid patterns and the time limit of this procedure due to the exponential complexity. It is our aim to improve this time-consuming process without loss of the exact solution of $\mathbf{maxrf}(m, n)$ by utilization of permutations of rows and columns. One possible approach is the application of the *slot principle*. Before we give a formal definition of this term within a grid, we explain both the procedure to create a slot and the benefits of a slot for the remaining recursive generation of grid patterns.

We assume that a grid pattern is partially filled with values 1 without violating the rectangle-free rule (1.3). In such a grid we can count the numbers of values 1 for each row. This horizontal checksum must be maximal for one or several rows. We select in the set of rows with the maximal horizontal checksum one row and exchange its position with the top row r_1 of the grid. The result of the rectangle-free rule (1.3) remains unchanged for the grid patterns before and after this permutation of rows.

A result of this exchange of rows is a unique maximal number of values 1 in the first row r_1. As next we swap the columns of the grid such that all values 1 of the row r_1 are located in the leftmost columns. This permutation of columns does not change neither the horizontal checksums of the grid nor the result of the rectangle-free rule (1.3). We call the horizontal checksum of row r_1 *slot width* sw_1. In this way we have created the slot s_1 that is the leftmost subgrid of sw_1 columns c_1 to c_{sw_1}. The *head row* of the slot is completely filled with values 1.

There are two benefits of the slot s_1 for the remaining recursive generation of grid patterns:

1. not more than a single value 1 can be assigned in the rows r_i with $1 < i \leq m$ to the elements $x_{r_i,c_1}, \ldots, x_{r_i,c_{sw_1}}$ without violating the rectangle-free rule (1.3) of the grid, and

2. no value 1 can be assigned in the first row right of the slot s_1 to the grid elements $x_{r_1,c_k}, \ldots, x_{r_1,c_n}$, where $k = sw_1 + 1$.

Further slots s_{i+1}, $i \geq 1$, can be created within the columns of the grid which are not covered by the slots $s_1, \ldots, s_i$. Values 1 located in the slots $s_1, \ldots, s_i$ are not used for the selection of the maximal horizontal checksum of s_{i+1}. The head row of slot s_{i+1} is located in row r_{i+1}. All elements in the slot s_{i+1} above its head row are equal to 0.

Definition 1.1 (slot, slot width, head row)**.** *A **slot** is a set of columns c_k of a grid $G_{m,n}$ which meet the following properties:*

1. *each of the n columns is assigned exactly to one slot of the grid,*

2. *the columns of a slot are located within a closed interval,*

3. *the slots $s_i, i \geq 1$ are enumerated from the left to the right within the grid,*

4. *the number of columns in the slot s_i is called **slot width** sw_i,*

5. *the slots are ordered based on their slot width such that $sw_i \geq sw_j$ for $i < j$,*

6. *all sw_i elements of the slot s_i in the row r_i are equal to 1; this row is called **head row** of the slot, and*

7. *all elements of the slot s_i in rows r_j, $j < i$ are equal to 0.*

Figure 1.4 shows how the slots of a grid $G_{4,5}$ given in Figure 1.4 (a_1) will be constructed. The horizontal checksums are determined by counting the values 1 in a row of the grid and represented at the right of the grid in Figure 1.4 (a_1). The maximal horizontal checksum 3 appears twice in rows r_2 and r_3. Each of these rows can be chosen as head row. We select row r_2 and exchange it with row r_1 as shown

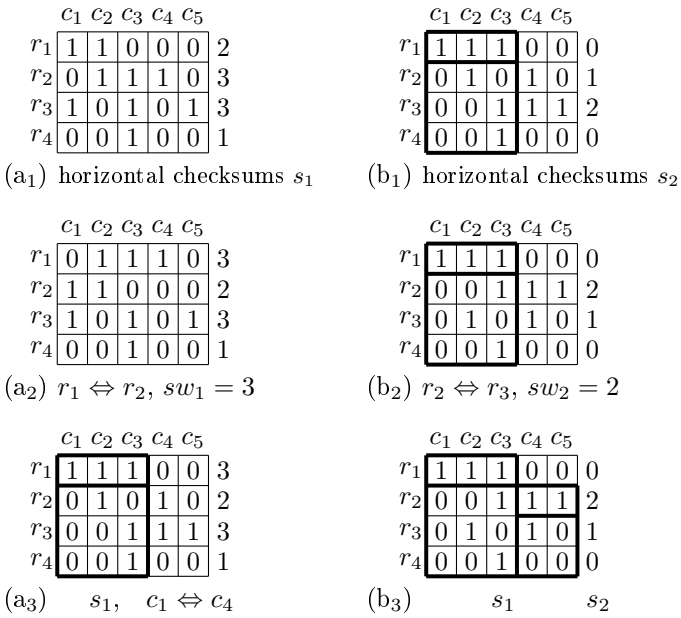


Figure 1.4. Creating slots within a grid $G_{4,5}$: (a_i) slot s_1, (b_i) slot s_2.

in Figure 1.4 (a_2). The slot width $sw_1 = 3$ for the slot s_1 is equal to the horizontal checksum of the new row r_1. As final step to create the slot s_1 the columns must be swapped such that all values 1 in the head row of slot 1 are located at the left of the grid. This is realized by exchanging the columns c_1 and c_4. The thick lines in Figure 1.4 (a_3) emphasize the slot s_1 with a slot width of $sw_1 = 3$ and its head row.

The same procedure is used to create the slot s_2. As can be seen in Figure 1.4 (b_1), the horizontal checksums are only counted in the remaining columns c_4 and c_5. The row with the maximal horizontal checksum 2 must be used as row r_2 for slot s_2 with a slot width of $sw_2 = 2$. Therefore the complete rows r_2 and r_3 are exchanged as shown in Figure 1.4 (b_2). In this case the head row of slot s_2 is ordered as required as closed interval of values 1. Hence, no further modifications are necessary. Both the slot s_1 and the slot s_2 have been emphasized by thick lines in Figure 1.4 (b_3).

1.3.2. The Head of Maximal Grids

The head row uniquely identifies a slot by the values 1 which are not elements of a slot of a lower number. All head rows together specify the *head of a grid.*

Definition 1.2 (head of a grid)**.** *The head of a grid is built by all elements 1 of the head rows of all slots of a grid.*

We are searching for grids which contain $\mathbf{maxrf}(m,n)$ values 1. Such grids require maximal heads.

Theorem 1.2. *A maximal head of the grid $G_{m,n}$ contains n values 1 distributed over all n columns of the grid and the rows r_i, $1 \leq i \leq n_s$, where n_s is the number of slots.*

Proof. The number of values 1 of the grid $G_{m,n}$ cannot be maximal in the case that the head of the grid contains less than n values 1. At least one additional value 1 can be assigned to each column of the grid which does not belong to the grid head because a single value 1 in a column of a grid cannot violate the rectangle-free rule (1.3).

Such additional values 1 cause either extensions of the given slots or new slots such that the head of the maximal head of the grid contains n values 1 distributed over all n columns of the grid. □

Due to Theorem 1.2, a maximal head of a grid $G_{m,n}$ contains a single value 1 within each of the n columns. There are

$$n_{soc} = \binom{m}{1}^n = m^n \tag{1.11}$$

different grid patterns which contain exactly a single value 1 within each of the n columns. These m^n grids can be transformed into an extremely smaller number of maximal heads using the allowed permutations of rows and columns.

The number of all grid heads n_{gh} is equal to the sum of all partitions

$P(n,k)$ of the n columns into k parts:

$$n_{gh} = \sum_{k=1}^{n} P(n,k) \ . \tag{1.12}$$

Algorithm 1.5 utilizes the slot concept to find the maximal number **maxrf**(m,n) of values 1 of all rectangle-free grids $G_{m,n}$. The main task of Algorithm 1.5 is the generation of all grid heads which can be extended to rectangle-free grids $G_{m,n}$. The used function

`ExtendSlotBodies`(n_{max}, n_a)

solves the subtask of finding the maximal assignment of values 1 to the bodies of the slots.

All possible maximal grid heads are constructed by Algorithm 1.5 directly within the grid $G_{m,n}$. This grid of m rows and n columns must be initialized with values 0 on all positions. The recursive Algorithm 1.5 is called on the top level by `GenMaxGrid`$(1,n,1,0)$. The first parameter 1 means that the slot number 1 with row r_1 as head row must be constructed. The second parameter n means that the first generated slot has a slot width of sw of all n columns. The third parameter 1 means that the slot begins in the column number 1, and the last parameter 0 is used as initial value of n_{max}. The variable n_{max} holds at each point of time the so far known maximal number of values 1 within a rectangle-free grid of the explored size.

The break of the recursion of Algorithm 1.5 is organized in lines 1 to 6. The so far generated grid head cannot be extended by an additional slot when the condition of line 1 is satisfied. A maximal grid head is generated if both the condition of line 1 and the condition of line 2 are satisfied. In this case, the maximal extension of the bodies of the slots is calculated in line 3. The so far found maximal number n_{max} of values 1 is returned to the next higher level of the recursion in line 5.

The **for**-loop in lines 8 to 11 creates a single slot head of the requested slot width sw in the row of the index hr starting in the column of the

Algorithm 1.5 GenMaxGrid$(hr, sw, first1, n_{max})$

Require: hr: index of the head row
Require: sw: slot width
Require: $first1$ first column to which a value 1 must be assigned
Require: n_{max}: so far known maximal number of assignments 1
Require: n_a: number of assignments 1, initialized with n
Require: m: number of rows of the grid $G_{m,n}$
Require: n: number of columns of rows of the grid $G_{m,n}$
Require: $G_{m,n}$ in which all elements are initialized with values 0
Ensure: **maxrf**(m, n)

```
1:  if hr > m then                                   ▷ no more head row
2:      if first1 > n then                           ▷ slots cover all columns
3:          n_max ← ExtendSlotBodies(n_max, n_a)
4:      end if
5:      return n_max
6:  end if
7:  last1 ← first1
8:  for i ← 1 to sw do
9:      G_{m,n}[hr, last1] ← 1                       ▷ create slot head
10:     last1 ← last1 + 1
11: end for
12: if last1 ≤ n then                                ▷ next slot head
13:     nsw ← Minimum(n − last1 + 1, sw)
14:     n_max ← GenMaxGrid(hr + 1, nsw, last1, n_max)
15: else                                             ▷ extend slot bodies
16:     n_max ← ExtendSlotBodies(n_max, n_a)
17: end if
18: for last1 ← first1 + sw − 1 downto 1 do
19:     G_{m,n}[hr, last1] ← 0          ▷ reduce slot head by one column
20:     if last1 > first1 then                       ▷ next grid head
21:         nsw ← Minimum(n − last1, last1 − first1)
22:         n_max ← GenMaxGrid(hr + 1, nsw, last1, n_max)
23:     end if
24: end for
25: maxrf(m, n) ← n_max
26: return maxrf(m, n)
```

index $first1$. If the number of the last column of the created slot is less than the number of columns n then the next slot will be created

by a recursive invocation of Algorithm 1.5 using:

- the row $hr + 1$ as index of the head row,
- the maximal next slot width nsw that satisfies item 5 of Definition 1.1, and
- the next unused column of the index $last1$ as first column of the next slot, and
- the so far known maximal number n_{max} of values 1 within a rectangle-free grid.

The next slot width nsw is determined in line 13 of Algorithm 1.5 as the minimum of the last used slot width sw and the number of remaining columns $n - last1 + 1$. If the created slots cover all columns the completed grid head is used for finding maximal grids in the function `ExtendSlotBodies`(n_{max}, n).

In the **for**-loop in lines 18 to 24 the recursive creation of all further possible maximal grid heads is organized. In order to do this the slot head that belongs to the level of recursion is reduced by one column in line 19 of Algorithm 1.5. The recursion must stop when the actual slot width is reduced to 0 as realized in the **if**-statement in line 20 of Algorithm 1.5. The decremented value of $last1$ causes different next slot widths nsw within the loop. The recursive invocation of Algorithm 1.5 in line 24 assures the generation of all maximal grid heads.

The variable n_{max} carries the searched value $\mathbf{maxrf}(m, n)$ if the **for**-loop in lines 18 to 24 is finished on the highest level of the recursion. Hence, this n_{max} is assigned to $\mathbf{maxrf}(m, n)$ in line 25 of and returned in line 26 as result of the Algorithm 1.5.

Figure 1.5 shows the sequence of all five maximal grid heads of the grid $G_{4,4}$. It can be seen how the slot of index i is reduced successively to allow the creation of the slot of index $i + 1$. The slot width of the slot of index $i + 1$ is restricted by both the slot width of the slot of index i and by the remaining columns.

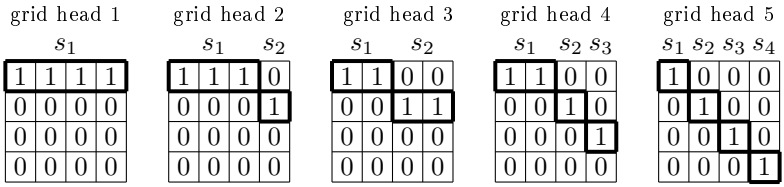


Figure 1.5. Sequence of all maximal grid heads of $G_{4,4}$.

The creation of the grid heads solves only a subtask of maximal rectangle-free grids. However, this utilization of permutations of both rows and columns strongly restricts the number of grids which must be evaluated.

Table 1.6 shows that all $2.05891 * 10^{44}$ possible assignments of a single value 1 to each column of a grid $G_{30,30}$ can be transferred into only $5,604$ grid heads by permutations of rows and columns. Hence, the grid heads save in this case the evaluation of more than $2 * 10^{44}$ grids. Algorithm 1.5 creates all grid heads for grids up to $m = 30$ rows and $n = 30$ columns in less than one second. It should be mentioned that Algorithm 1.5 is not restricted to square grids but it can be used to generate all grid heads for any number of rows and columns very fast.

1.3.3. The Body of Maximal Grids

Figure 1.5 shows that the values 1 of each head of a maximal grid separate values 0 in the north-east of the grid from values 0 in the south-west of the grid. The north-east region is empty in the special case of grid head 1. Due to item 7 of Definition 1.1, no value 1 can be assigned within the north-est region above the head of a grid. This property strongly reduces further construction efforts.

In order to find $\mathbf{maxrf}(m, n)$ of the grid all heads must be extended as much as possible by values 1. The number of values 0 in the north-east region above the head of a quadratic grid grows from 0 for the grid head of a single slot to $n * (n-1)/2$ for the grid head of the main diagonal strip. Hence, up to $2^{n*(n-1)/2}$ assignments of values 1 can be skipped which is $8.8725 * 10^{130}$ for one grid head of the grid $G_{30,30}$.

Table 1.6. Grid heads of all quadratic grids up to $m = 30$ rows and $n = 30$ columns with a single value 1 in each column calculated with Algorithm 1.5

m	n	number of all m^n grids	number of grid heads	time in milliseconds
1	1	1	1	1
2	2	4	2	1
3	3	27	3	1
4	4	256	5	1
5	5	3,125	7	1
6	6	46,656	11	2
7	7	823,543	15	2
8	8	16,777,216	22	3
9	9	387,420,489	30	3
10	10	$1.00000 * 10^{10}$	42	4
11	11	$2.85312 * 10^{11}$	56	4
12	12	$8.91610 * 10^{12}$	77	5
13	13	$3.02875 * 10^{14}$	101	8
14	14	$1.11120 * 10^{16}$	135	10
15	15	$4.37894 * 10^{17}$	176	16
16	16	$1.84467 * 10^{19}$	231	19
17	17	$8.27240 * 10^{20}$	297	25
18	18	$3.93464 * 10^{22}$	385	31
19	19	$1.97842 * 10^{24}$	490	41
20	20	$1.04858 * 10^{26}$	627	53
21	21	$5.84259 * 10^{27}$	792	73
22	22	$3.41428 * 10^{29}$	1,002	109
23	23	$2.08805 * 10^{31}$	1,255	122
24	24	$1,33374 * 10^{33}$	1,575	162
25	25	$8.88178 * 10^{34}$	1,958	201
26	26	$6.15612 * 10^{36}$	2,436	267
27	27	$4.43426 * 10^{38}$	3,010	334
28	28	$3.31455 * 10^{40}$	3,718	422
29	29	$2.56769 * 10^{42}$	3,718	529
30	30	$2.05891 * 10^{44}$	5,604	687

It remains the south-west region below the grid head for additional assignments of values 1 to find maximal grids. We call this region *body of a grid*.

Definition 1.3 (body of a slot, body of a grid)**.** *All elements below the head row of a slot are called* ***body of the slot****. The* ***body of the grid*** *is the union of all bodies of the slots of the grid.*

The number of values 1 within a rectangle-free slot s_i is limited by $sw_i + m - i$.

Theorem 1.3. *A rectangle-free grid cannot contain more than a single value 1 within any row of the body of a slot.*

Proof. The head row of a slot contains the value 1 in each column. Two values 1 in any body row of a slot violate together with the fitting values 1 in the head of the slot the rectangle-free rule (1.3). □

Theorem 1.3 provides a further strong restriction of the effort to construct maximal rectangle-free grids. The slot s_i, $i \geq 1$, with a slot width sw_i of a grid of m rows contains $(m - i)$ body rows and therefore $sw_i * (m - i)$ body elements. According to Theorem 1.3 at most $(m - i)$ values 1 can be assigned in the body of this slot. Hence, the number of possible assignments of values 1 to the body of the slot s_i can be reduced by a factor of at least $2^{(sw_i-1)*(m-i)}$ which is equal to $1.4663 * 10^{253}$ for a single slot of the grid $G_{30,30}$.

The extreme reduction can be improved furthermore because the columns within a slot can be exchanged. Hence, the topmost value 1 within the body of a slot can be located in the leftmost column of the slot. In this way sw_i possible assignments of this topmost value 1 are reduced to a single assignment. Additional assignments of values 1 below this topmost assignment can be located in the leftmost column of the slot, too. However, two values 1 in one column of a slot can cause, together with two values 1 within a single column of another slot, a violation of the rectangle-free rule (1.3).

Such a violation can be avoided by moving the value 1 in the lower row into a so far unused column of the slot. Usable target columns are such columns of the slot which contain no value 1 in the body. Due to the possible permutation of slot columns we use in this case the leftmost possible slot column which additionally restricts the possible assignments.

Algorithm 1.6 ExtendSlotBodies(n_{max}, n_a)

Require: n_{max}: so far known maximal number of assignments 1
Require: n_a: number of rectangle-free assignments 1
Require: G: grid $G_{m,n}$ of m rows and n columns, initialized by a maximal grid head
Ensure: se: vector that contains the index of the last column of each slot
Ensure: la: matrix initialized by values -1, for last assignments of values 1 within the slot bodies
Ensure: n_{max}: so far known maximal number of assignments 1
Ensure: n_s: the number of slots of the grid $G_{m,n}$

1: **for** $k \leftarrow 1$ to n **do**
2: $se[k] \leftarrow -1$ ▷ initial value of slot ends
3: **end for**
4: **for** $i \leftarrow 1$ to m **do**
5: **for** $k \leftarrow 1$ to n **do**
6: $la[i,k] \leftarrow -1$ ▷ initial value of last assignments
7: **end for**
8: **end for**
9: $i \leftarrow 1$ ▷ row index
10: $n_s \leftarrow 0$ ▷ number of slots
11: **while** $(i < m)$ and $(se[i] < n)$ **do**
12: **if** $i = 1$ **then** ▷ left slot
13: $se[i] \leftarrow 0$
14: **else** ▷ other slots
15: $se[i] \leftarrow se[i-1]$
16: **end if**
17: **while** $(i \leq m)$ and $(se[i] \leq n)$ and $(G[i, se[i]] = 1)$ **do**
18: $se[i] \leftarrow se[i] + 1$
19: **end while**
20: $n_s \leftarrow n_s + 1$
21: **end while**
22: $n_{max} \leftarrow$ `MaxFillUp`$(2, 1, n_{max}, n_a)$
23: **return** n_{max}

The utilization of these sources for improvements requires an easy access to information about the columns on the right-hand side of each slot and the rightmost assignments within the slots. Algorithm 1.6 prepares this information which are used in the recursive Algorithm

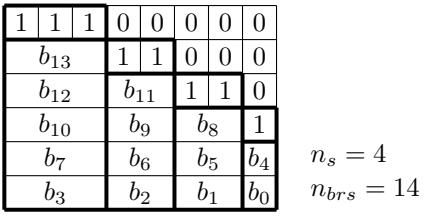


Figure 1.6. Enumeration of the body rows in the grid $G_{6,8}$ with 4 slots.

1.7. The vector *se* is initialized by values -1 in the **for**-loop in lines 1 to 3. Similarly, the matrix *la* is initialized by values -1 in the **for**-loops in lines 4 to 8. The information about the end of the slots is taken from the generated grid head of $G_{m,n}$. Within the **while**-loop in lines 11 to 21 both the values of the slot ends *se* and the number of slots *nos* are assigned.

An initial value $se[i]$ is assigned in the alternative in lines 12 to 16. The required value of $se[i]$ is built by counting the values 1 of the slot i of $G_{m,n}$ in the **while**-loop in the lines 17 to 19. Finally, the function `MaxFillUp`$(2, 1, n_{max}, n_a)$ of the recursive Algorithm 1.7 is called.

The first parameter 2 of the function `MaxFillUp`$(2, 1, n_{max}, n_a)$ selects row 2 as the topmost body row of the first slot. The second parameter 1 specifies the slot in which the additional value 1 can be assigned. The third parameter is the maximal number of values 1 known so far for the rectangle-free grid $G_{m,n}$. The final parameter n_a is the number of assigned values 1 in the actually evaluated grid $G_{m,n}$.

Algorithm 1.7 assigns as much as possible values 1 in the body of the grid. This algorithm must find maximal additional assignments of values 1 to a given grid head. This is achieved in a smart restricted manner based on Theorem 1.3. The number of body rows of all slots n_{brs} is equal to

$$n_{brs} = \frac{n_s * (n_s - 1)}{2} + n_s * (m - n_s) \ , \tag{1.13}$$

where n_s is the number of slots.

Each body row of each slot can contain at most one single value 1. The body rows b_i can be ordered as shown in Figure 1.6. As basic

step we assign all binary numbers from 0 to $2^{n_{brs}} - 1$ to these body rows using the row index b_i as shown in Figure 1.6. These values 1 are assigned to the leftmost column of the associated body row of a slot. If such an assignment causes a conflict with the rectangle-free rule (1.3) then we try to solve this conflict by moving one assigned value 1 within its body row to the right.

Algorithm 1.7 recursively realizes the assignment of the $2^{n_{brs}}$ binary numbers to the body rows of the slots. The core of this recursion is separated in Algorithm 1.8. The invocation of Algorithm 1.8 in line 4 of Algorithm 1.7 ensures that the upper body rows of the slots carry only values 0. For the example of Figure 1.6 the body rows in the sequence $b_{13}, b_{12}, \ldots, b_0$ are filled with values 0. The alternative of lines 5 to 9 of Algorithm 1.7 selects the first column c of the actual slot.

Within the **repeat-until**-loop in lines 10 to 33, a single value 1 is assigned to the body row of a slot in line 11 selected by the level of recursion. The first value 1 is assigned to the body row b_0 in the example of Figure 1.6. Such an assignment is counted by n_a in line 12. The used column is stored in the matrix la in line 13. The value 1 assigned to the grid in line 11 can cause for slots s_i, $i > 1$, a rectangle-free conflict which is checked by the function `crc`(r, c) in line 14. The check for rectangle-free conflicts is restricted in this function to rectangles which contain the position (r, c) of the new assigned value 1.

In the case that the rectangle-free condition (1.3) is violated by the assigned value 1, this assignment is withdrawn in lines 15 to 17. In such a case the next column to the right within the body row of the slot is selected for an alternative assignment of the value 1 in line 18.

If there is no possible column, as checked in line 19, the recursion will be broken in line 20. Otherwise the next possible assignment of a value 1 within the body row of the slot is induced by the **continue**-statement for the **repeat-until**-loop in line 22. The break of the recursion in line 20 can occur on a high level of the recursion and restricts the necessary evaluation of the search space significantly.

In the case that the rectangle-free condition (1.3) is not violated, a new

Algorithm 1.7 MaxFillUp(r, s, n_{max}, n_a)

Require: all variables as introduced in Algorithm 1.6
Require: r: index of the row of the actual recursion
Require: s: index of the slot of the actual recursion
Ensure: n_{max}: the so far known maximal number of assignments 1

```
1: if r > m then                                   ▷ break of the recursion
2:     return n_max
3: end if
4: n_max ← NextSlot(r, s, n_max, n_a)
5: if s = 1 then          ▷ select the column for the next assignment
6:     c ← 1
7: else
8:     c ← se(s − 1) + 1
9: end if
10: repeat
11:     G[r, c] ← 1                                          ▷ assign value 1
12:     n_a ← n_a + 1
13:     la[r, s] ← c                                  ▷ remember the column
14:     if s > 1 and crc(r, c) then                     ▷ rectangle conflict?
15:         G[r, c] ← 0                              ▷ replace assigned 1 by 0
16:         n_a ← n_a − 1
17:         la[r, s] ← −1                                ▷ restore initial value
18:         c ← c + 1                                          ▷ next column
19:         if (c > la[r − 1, s] + 2) or (c > se[s]) then
20:             return n_max                          ▷ no more possible column
21:         else
22:             continue                  ▷ try assignment in next column
23:         end if
24:     end if
25:     if n_a > n_max then                        ▷ larger rectangle-free grid
26:         n_max ← n_a
27:     end if
28:     n_max ← NextSlot(r, s, n_max, n_a)
29:     G[r, c] ← 0                                  ▷ replace assigned 1 by 0
30:     n_a ← n_a − 1
31:     la[r, s] ← −1                                    ▷ restore initial value
32:     c ← c + 1                                              ▷ next column
33: until (c > la[r − 1, s] + 2) or (c > se[s])
34: return n_max
```

maximal assignment of values 1 can be found. This will be checked in line 25. In the successful case the new maximal value of assigned values 1 is remembered in line 26. Thereafter, the recursive assignment of a value 1 in the remaining slots is initiated in line 28. If a conflict with the rectangle-free rule happens in this part of the recursion, an alternative assignment in the actual body row is prepared in lines 29 to 32 as in the case of a direct conflict in lines 15 to 18.

Table 1.7. Early break of the recursion in Algorithm 1.7 due to the rectangle-free condition for the Grids $G_{8,8}$, $G_{9,9}$, and $G_{10,10}$

number of assigned values 1	number of detected conflicts for grid $G_{8,8}$	$G_{9,9}$	$G_{10,10}$
10	0	0	0
11	137	0	0
12	1117	275	0
13	5298	2634	565
14	14924	16066	6689
15	40351	73595	48796
16	82225	227264	262336
17	153122	541870	1187941
18	206705	1267637	4202340
19	193196	2200083	11511211
20	117481	3836481	25336356
21	105632	5063619	53319506
22	87276	5327652	86332581
23	35345	3771029	145781391
24	6266	1854612	191922194
25	698	1212354	220620600
26	0	814065	186319424
27	0	294390	115718047
28	0	56279	53227679
29	0	5597	27424287
30	0	332	15174190
31	0	0	6060258
32	0	0	1444707
33	0	0	175683
34	0	0	10824
35	0	0	635
36	0	0	0

Table 1.7 reveals the benefit of the early break of the recursion in cases of violations of the rectangle-free condition using the quadratic grid $G_{8,8}$, $G_{9,9}$, and $G_{10,10}$ as example. A violation of the rectangle-free condition (1.3) cannot occur when values 1 are only assigned to the grid head. Any two additionally assigned values 1 cannot cause a violation of the rectangle-free condition (1.3), because these two values 1 are either in different body rows of the same slot or in body rows of different slots. Three additional values can can cause together with a value 1 of the slot $i+1$ a violation of the rectangle-free condition (1.3) when two of them are located in the same column of slot i, and the third value 1 is assigned to a body row of slot $i+1$. Due to this property, the first break of the recursion appears for $n+3$ assigned values 1. The break of the recursion on this very early level strongly restricts the search space.

Table 1.7 shows that the number of breaks of the recursion grows for increased numbers of assigned values 1. Both the already restricted search space and the larger number of values 1 which may cause a conflict are the reasons that the number of conflicts decreases when the number of assigned values 1 surpasses a given limit. The maximal numbers of values 1 of rectangle-free grids are $\mathbf{maxrf}(8,8) = 24$, $\mathbf{maxrf}(9,9) = 29$, and $\mathbf{maxrf}(10,10) = 34$ (see Table 1.8). Table 1.7 shows that each additional assignment of a value 1 to grids with these values $\mathbf{maxrf}(m,n)$ causes a break of the recursion on the deepest level.

Algorithm 1.8 organizes the recursive invocation of Algorithm 1.7 for all body rows of a grid. The number of body rows of the slots grows from 1 to $n_s - 1$ in the rows from 2 to the number of slots n_s. Figure 1.6 shows that the grid of $n_s = 4$ slots contains in row 2 the single body row b_{13}, in row 3 two body rows b_{12} and b_{11}, and in row 4 three body rows b_{10}, b_9 and b_8. The first summand of (1.13) describes this number of body rows.

All rows r_i with $i > n_s$ contain n_s body rows. The second summand of (1.13) expresses this part of the body rows. In Figure 1.6 these are the four body rows b_7, b_6, b_5, and b_4 in row 5 and the four body rows b_3, b_2, b_1, and b_0 in row 6, respectively. Therefore the recursion must be distinguished between the range of the grid head and the remaining other rows of the grid. Algorithm 1.8 makes this decision in line 1.

Algorithm 1.8 NextSlot(r, s, n_{max}, n_a)

Require: variables as introduced in Algorithm 1.7
Ensure: recursive invocation of function `MaxFillUp`(r, s, n_{max}, n_a) for all body rows of all slots of a grid
Ensure: n_{max}: the so far known maximal number of assignments

```
 1: if r ≤ n_s then                                  ▷ within the head rows
 2:     if r − s < 2 then                            ▷ first slot in the next row
 3:         n_max ← MaxFillUp(r + 1, 1, n_max, n_a)
 4:     else                                         ▷ next slot in the same row
 5:         n_max ← MaxFillUp(r, s + 1, n_max, n_a)
 6:     end if
 7: else                                             ▷ below the head rows
 8:     if n_s − s < 2 then                          ▷ first slot in the next row
 9:         n_max ← MaxFillUp(r + 1, 1, n_max, n_a)
10:     else                                         ▷ next slot in the same row
11:         n_max ← MaxFillUp(r, s + 1, n_max, n_a)
12:     end if
13: end if
14: return n_max
```

The recursion for the rows of the grid head is organized in lines 2 to 6 of Algorithm 1.8. If there is no more body row to the right of the actual body row, Algorithm 1.7 is invoked for the first slot row of the next row in line 3; otherwise the next slot to the right in the same row is used in line 5 of Algorithm 1.7.

The recursion for the rows below the head rows of the grid is organized in lines 8 to 12 of Algorithm 1.8. The index of the actual slot and the number of slots are used in this case for the decision whether the Algorithm 1.7 is invoked for the next slot of the same row in line 11 or the first slot of the next row in line 9.

1.3.4. Experimental Results

Both the achieved speedup and the extended successfully evaluated grid size are measurements for the improvement. Using the approach of the search space restriction of Section 1.2, it was possible to calcu-

Table 1.8. **maxrf**(m,n) of quadratic rectangle-free grids utilizing the slot principle; the time is measured in milliseconds

m	n	n_v	size of search space	**maxrf**(m,n)	time
2	2	4	$1,6000000 * 10^1$	3	0
3	3	9	$5,1200000 * 10^2$	6	32
4	4	16	$6,5536000 * 10^4$	9	32
5	5	25	$3,3554432 * 10^7$	12	47
6	6	36	$6,8719476 * 10^{10}$	16	62
7	7	49	$5,6294995 * 10^{14}$	21	468
8	8	64	$1,8446744 * 10^{19}$	24	17,207
9	9	81	$2,4178516 * 10^{24}$	29	503,522
10	10	100	$1,2676506 * 10^{30}$	34	25,012,324

late **maxrf**$(7,7) = 21$ for a grid $G_{7,7}$ within 1,717,118 milliseconds. Table 1.8 shows that the utilization of the slot principle with Algorithms 1.5, 1.6, 1.7, and 1.8 reduces this runtime to only 468 milliseconds. Hence, the suggested slot principle reduces the runtime by a factor of 3,669.

Furthermore, using the slot principle, **maxrf**$(10,10) = 34$ could be calculated for the grid $G_{10,10}$. The search space of $2^{7*7} = 5,63 * 10^{14}$ for the grid $G_{7,7}$ could be successfully extended to the search space of $2^{10*10} = 1,27 * 10^{30}$ which is an extension by a factor of $2.2518 * 10^{15}$.

There can be several different maximal grids of a fixed size with the same value **maxrf**(m,n). Figure 1.7 on page 50 depicts in the upper part examples for maximal quadratic grids from $G_{2,2}$ to $G_{10,10}$. Both the emphasized slot structure and the distribution of the additional values 1 in the bodies of the slots facilitate the comprehension of the slot approach.

As a second experiment we calculated the values **maxrf**(m,n) for grids up to 25 rows and 25 columns. We have chosen this number of rows and columns due to the restricted width of Table 1.9 over the full width of the page. All results of Table 1.9 are calculated within seconds or few minutes.

The reached improvement can be measured by the maximal size of

Table 1.9. **maxrf**(m,n) of grids $G_{m,n}$ calculated by the algorithms 1.5, 1.6, 1.7, and 1.8 which utilize the slot principle

n	m																								
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
3	4	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
4	5	7	9	10	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
5	6	8	10	12	14	15	17	18	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
6	7	9	12	14	16	18	19	21	22	24	25	27	28	30	31	32	33	34	35	36	37	38	39	40
7	8	10	13	15	18	21	22	24	25	27	28	30	31	33	34	36	37	39	40	42	43	44	45	46
8	9	11	14	17	19	22	24	26	28	30	32	33	35	36	38	39	41	42	44	45	47	48	50	51
9	10	12	15	18	21	24	26	29	31	33	36													
10	11	13	16	20	22	25	28	31	34															
11	12	14	17	21	24	27	30	33																
12	13	15	18	22	25	28	32	36																
13	14	16	19	23	27	30	33																	
14	15	17	20	24	28	31	35																	
15	16	18	21	25	30	33	36																	
16	17	19	22	26	31	34	38																	
17	18	20	23	27	32	36	39																	
18	19	21	24	28	33	37	41																	
19	20	22	25	29	34	39	42																	
21	22	24	27	31	36	42	45																	
22	23	25	28	32	37	43	47																	
23	24	26	29	33	38	44	48																	
24	25	27	30	34	39	45	50																	
25	26	28	31	35	40	46	51																	

the search space for which **maxrf**(m,n) could be calculated. The maximal successfully evaluated search space based on the search space restriction of Section 1.2 is $2^{7*7} = 2^{49} = 5.6295 * 10^{14}$. The maximal successfully evaluated search space in this limited approach of the slot principle is $2^{8*25} = 2^{200} = 1.60694 * 10^{60}$. Hence, the utilization of the slot principle extends the successfully evaluated search space by a factor of $2.8545 * 10^{45}$. Figure 1.7 shows at the bottom a maximal rectangle-free grid $G_{8,25}$.

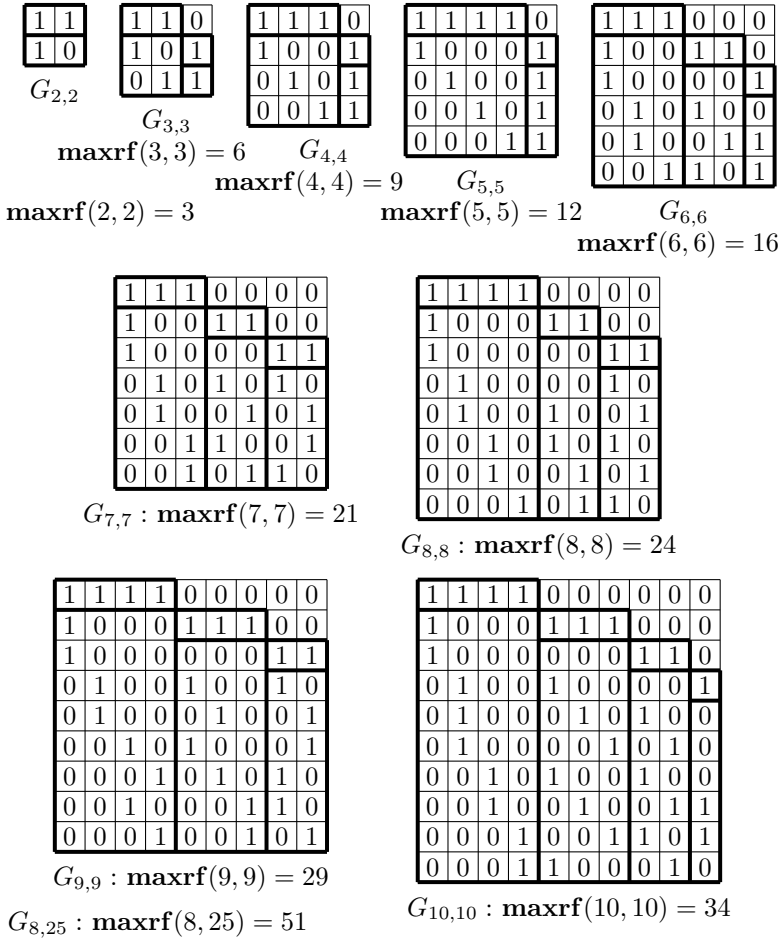


Figure 1.7. Examples of maximal rectangle-free grids $G_{2,2}$ to $G_{10,10}$ and $G_{8,25}$ with emphasized slots.

1.4. Restricted Enumeration

Matthias Werner

1.4.1. The Overflow Principle

In Section 1.2 we have seen a recursive algorithm that computes the maximum number of values 1 among the rectangle-free grids $G_{m,n}$ up to $m{=}n{=}7$. The utilization of the slot principle in Section 1.3 could extend the successfully evaluated grids $G_{m,n}$ up to $m{=}n{=}10$. In order to find $\mathbf{maxrf}(m,n)$, many "different" assignments must be tested. But what makes an assignment different from another one? Since the rectangle-free condition (1.3) does not depend on the order of the columns and rows, one has to consider only those assignments which are distinct with respect to column and row permutations. In terms of bipartite graphs, permuting vertices in one of the two disjoint sets yields just an isomorphic graph.

Table 1.10. Number of assignments modulo permutations

m	n	none ($2^{m\cdot n}$)	row	row & column
8	8	1.845×10^{19}	5.099×10^{14}	8.182×10^{11}
11	11	2.659×10^{36}	6.841×10^{28}	7.880×10^{23}
12	12	3.230×10^{43}	4.731×10^{34}	8.291×10^{28}

Table 1.10 gives a comparison for the number of assignments with respect to permutations. The numbers for assignments modulo row permutations can be computed directly and will be given later in this section. For the number of assignments modulo row and column permutations a direct formula is still unknown, since it involves partition numbers. The series up to $m{=}n{=}12$ can be found on the website [225] of *The On-Line Encyclopedia of Integer Sequences*.

Each assignment has to be checked further for rectangles which requires $\binom{n}{2}\binom{m}{2}$ additional operations for a single assignment.

Table 1.11. Time estimation to compute $\mathbf{maxrf}(m,n)$

$m=n$	all grids evaluated	permutations utilized
8	1.71×10^5 a	66 h 44 min
11	9.55×10^{22} a	2.83×10^{10} a
12	1.15×10^{30} a	4.29×10^{15} a

Table 1.11 shows the theoretical times to calculate $\mathbf{maxrf}(m,n)$ for quadratic grids without (with) utilization of row and column permutations. These estimated times assume a single CPU with 2.67 GHz and 1 instruction per cycle. Even the evaluation of the grid $G_{8,8}$ needs almost 67 h to compute $\mathbf{maxrf}(8,8)$ with taking advantage of permutation classes. This section will give an exact algorithm solving $\mathbf{maxrf}(8,8)$ in just 1 s and $\mathbf{maxrf}(11,11)$ in 7 days.

The Boolean matrix $k(m,n)$ of the grid can be represented as a list of positive numbers a_i in binary notation. Because the permutation of rows yields an isomorphic solution, $a_{i+1} \geq a_i$ is assumed. In this way, no solutions will be created which would be actually the same by just permuting the rows of them. Running $a_i \in \{1,\ldots,2^n-1\}$ generates all bit patterns for the i-th row with n bits. Step by step, every row is incremented by value 1 always beginning with the last row.

When a_m reaches the limit 2^n-1 then a_{m-1} is incremented by value 1 and a_m is reset to the new value of a_{m-1}. When a_{m-1} *overflows*, then a_{m-2} is incremented and a_{m-1}, a_m are reset to the new value of a_{m-2}. After each change, the whole assignment must be completely verified.

If the binary matrix $k(m,n)$ does not contain any rectangle, a feasible solution has been found. The algorithm will finish when a_1 overflows. Because of that overflow principle we will call this the Overflow Algorithm. Algorithm 1.9 gives a simple implementation in pseudo-code, without the verification and optimality tests.

The values $\{1 \leq a_i < 2^n\}$ can be represented as nodes sorted in a row where the most right node begins with $a_i = 1$. A matrix of bit patterns is obtained, where the current state of the algorithm represents a path

Algorithm 1.9 Overflow principle as pseudo code

Require: m: number of rows of the grid $G_{m,n}$
Require: n: number of columns of rows of the grid $G_{m,n}$
Ensure: $G_{m,n}$ with **maxrf**(m,n) is constructed by $a_1, i = 1,\ldots,m$
1: $a_1 \leftarrow 1$
2: $i \leftarrow 1$
3: **while** $a_1 < 2^n$ **do** ▷ algorithm will finish, when a_1 overflows
4: **for** $i = i{+}1,\ldots,m$ **do** ▷ reset all successors of a_i to a_i
5: $a_i \leftarrow a_{i-1}$
6: **end for**
7: **while** $a_m < 2^n{-}1$ **do** ▷ a_m runs through $\{a_{m-1},\ldots,2^n{-}1\}$
8: $a_m \leftarrow a_m + 1$ ▷ yields an instance for $k(m,n)$
9: **end while**
10: **repeat**
11: $i \leftarrow i - 1$ ▷ finds i where a_i is not at the limit yet
12: **until** $a_i < 2^n{-}1$ or $i{=}1$
13: $a_i \leftarrow a_i + 1$
14: **end while**

as illustrated in Figure 1.8. Due to the row order only steps to the south and west are allowed. For the complexity, the total number of paths has to be computed.

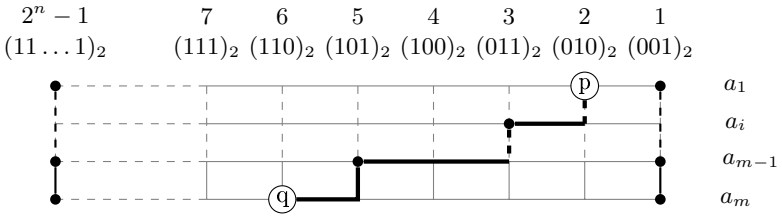


Figure 1.8. Paths of patterns taken by the Overflow Algorithm.

Theorem 1.4. *Let $\{a_1,\ldots,a_m\}$ be a set of positive numbers with $a_i \in \{1,\ldots,T\}$ as well as $T := T(n) \geq 1$ and $a_i \leq a_{i+1}$, $i \in \{1,\ldots,m\}$. The number $N(m,n,a_1,T)$ of paths from a_1 to T is:*

$$N(m,n,a_1,T) = \binom{T - a_1 + m}{m}\,. \tag{1.14}$$

Proof. Let us consider a block of the matrix of bit patterns (Figure 1.8) with m rows, $a_1 = p$, $a_m = q$, and the alphabet $\Gamma = \{W, S\}$. W and S describe a step to the west, and a step to the south, respectively. A word must contain $(q - p =: j) \times W$ and $(m-1) \times S$. Hence the number of words with the length $q - p + m - 1$ is $\binom{m+j-1}{j}$. Since each path can be identified by its words with S-W combinations, we obtain the result by applying Pascal's rule:

$$N(m, n, a_1, T) = \sum_{j=0}^{T-a_1} \binom{m+j-1}{j} = \binom{T-a_1+m}{m} . \qquad (1.15)$$

□

In our case $T = 2^n - 1$ is given, thus n should be minimal ($m \geq n$ was already assumed). If the adjacency matrix is long enough, then there is a direct result. Given $m \geq \binom{n}{2}$ then the exact value of the Zarankiewicz function easily can be computed (for proof see [333, p. 23] while [124, p. 130] gives a generalisation for $Z_{r,s}(m, n)$):

$$\mathbf{maxrf}(m, n) = Z_2(m, n) - 1 = m + \binom{n}{2}, \quad m \geq \binom{n}{2} . \qquad (1.16)$$

For $m < \binom{n}{2}$ the Overflow Algorithm comes into the play. The number of all paths given in Theorem 1.4 is the indicator for the complexity of the algorithm. One way to reduce the value of $N(m, n, a_1, T)$ could be to increase the start value a_1 of the first row as much as possible. It turns out that there are many optimal solutions. A lot of paths can be initially skipped. The question is how far the algorithm can be pushed forward without losing optimality? Corollary 1.1 provides a simple but good estimation of (1.14). It will be used for examining the runtime improvements, when the start value of a_1 is increased.

Corollary 1.1 (see [333, p. 8])**.** *If the most significant bit in the first row a_1 is shifted from the 0-th to the p-th position, the runtime improves by:*

$$\frac{c_p(m, n)}{c_0(m, n)} := \frac{N(m, n, 2^p, 2^n - 1)}{N(m, n, 1, 2^n - 1)} = \prod_{i=1}^{m} \frac{2^n - 2^p + m - i}{2^n - 1 + m - i} . \qquad (1.17)$$

Let be $\beta(p, m, n) = \left(1 - \frac{2^p - 1}{2^n - 2}\right)^m$. For $p \in [0, n]$ one can show that $\beta(p, m, n) \leq \frac{c_p(m,n)}{c_0(m,n)}$. If m grows slower than $\sqrt{2^n}$, the absolute error

$\Delta(p,m,n) := \left|\frac{c_p(m,n)}{c_0(m,n)} - \beta(p,m,n)\right| = \frac{c_p(m,n)}{c_0(m,n)} - \beta(p,m,n)$ *converges to 0 and, hence,* $\beta(p,m,n)$ *converges to* $\frac{c_p(m,n)}{c_0(m,n)}$ *for* $p \in [0,n]$.

Figure 1.9 shows the absolute error $\Delta(p,m,n)$, which occurs by estimating $\frac{c_p(m,n)}{c_0(m,n)} = \frac{N(m,n,2^p,2^n-1)}{N(m,n,1,2^n-1)}$ with $\beta(p,m,n)$. The error becomes visible when m is much higher than n, as $\Delta(p,36,6)$ illustrates well. In the opposite case, $\Delta(p,6,6)$ gives a deviation of not more than 2 percent. The estimation with $\beta(p,m,n)$ already becomes almost accurate for a grid $G_{10,10}$.

Figure 1.10 shows the relative runtime behavior of the Overflow-Algorithm, where a_1 starts at 2^p instead of $2^0 = 1$. The approximation quality of $\beta(p,m,n)$ is shown as well. The estimation $\beta(p,m,n)$ converges to $\frac{c_p(m,n)}{c_0(m,n)}$ under the conditions of Theorem 1.1. The bigger n the later the runtime acceleration takes impact due to the influence of the ratio $\frac{2^{p+1}-2^p}{2^n} = \frac{2^p}{2^n}$. Small bit positions would do almost nothing to reduce the amount of paths. For $m = n = 18$ the theoretical runtime improves noticeable when the most significant bit is moved at least to the 10th position.

With the help of Corollary 1.1 the considerably simpler derivation of the relative runtime estimation can be utilized for the maximum of the relative runtime acceleration:

$$\begin{aligned} p_{\max}(m,n) &:= \arg\max\left\{\left|\tfrac{\mathrm{d}}{\mathrm{d}p}\beta(p,m,n)\right|\right\} \\ &= \log_2\left(\tfrac{2^n-1}{m}\right) \\ &= n - \log_2(m) - \log_2(1-2^{-n}) \\ &\approx n - \log_2(m)\ . \end{aligned} \tag{1.18}$$

For instance, the value of the maximum acceleration of $\beta(p,18,18)$ is $p_{\max}(18,18) = 13.830069\ldots \approx 13.83$. The exact maximum acceleration given by $\arg\max\left\{\left|\frac{\mathrm{d}}{\mathrm{d}p}\frac{c_p(m,n)}{c_0(m,n)}\right|\right\}$ for $m = n = 18$ is $13.830121\ldots$ In Subsection 1.4.3 we will utilize (1.18).

In Algorithm 1.9 we have omitted the verification for rectangles. This test involves $\binom{m}{2}$ comparisons on the set of the m numbers a_i. Two numbers a and b form a rectangle, if they have at least two 1 bits in common. A fast check that `c=a&b` contains two or more 1 bits is:

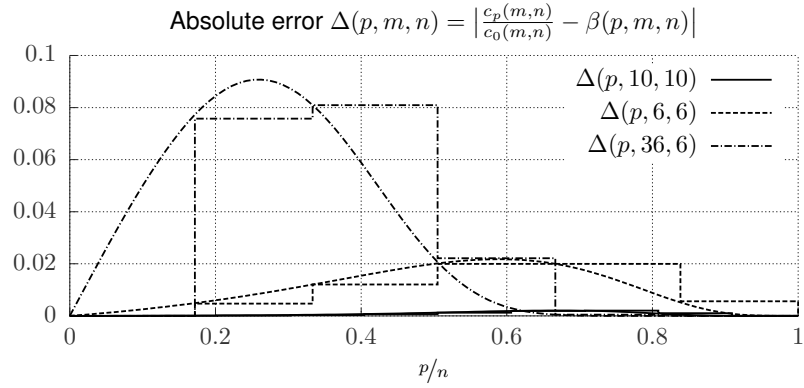


Figure 1.9. Absolute error $\Delta(p,m,n)$ by estimating $\frac{c_p(m,n)}{c_0(m,n)}$ with the help of $\beta(p,m,n)$.

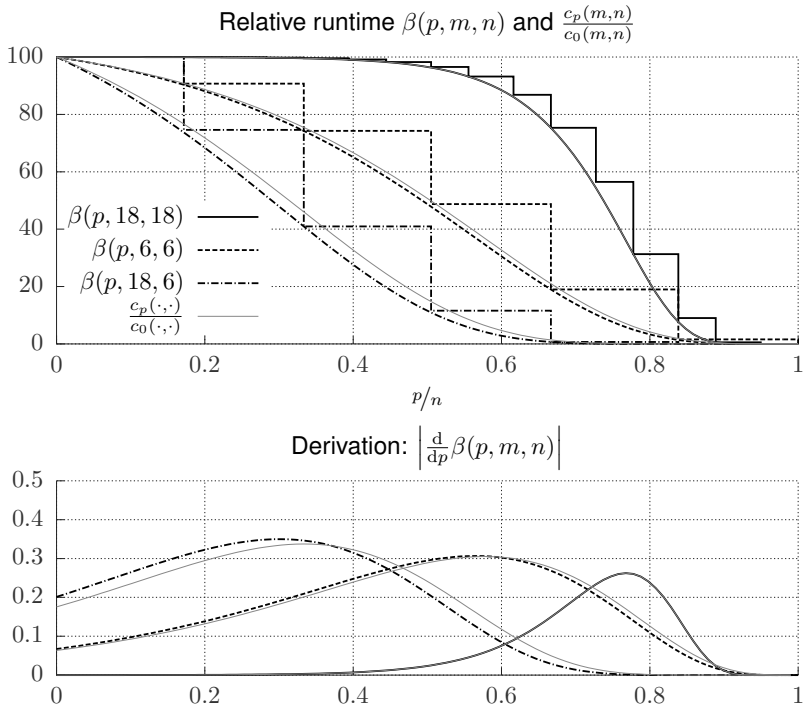


Figure 1.10. Relative runtime improvement with $a_1 = 2^p$ and $p \in [0,n)$.

```
bool has_rectangle := c && ((c & (~c + 1)) != c);
```

where '&' is the bitwise AND operator, '$\sim$' the unary bitwise complement, and '&&' the logical AND operator.

1.4.2. Strategies for Improvements

The implemented Overflow Algorithm works without recursion and uses several improvements to speed up the runtime. If a row a_i is incremented then a_{i+1} has to be reset to a_i (since a_{i+1} caused an overflow before). If a_i has two or more 1 bits then the assignment $a_{i+1} \leftarrow a_i$ would yield a rectangle. So the second bit of a_{i+1} is shifted up by one position and the following bits are reset to 0:

$$\begin{array}{c|c} a_i & 0\ 1\ 0\ 0\ 1\ 0 \\ a_{i+1} & 0\ 1\ 0\ 1\ 0\ 0 \end{array} \qquad \begin{array}{c|c} a_i & 0\ 1\ 0\ 1\ 0\ 1 \\ a_{i+1} & 0\ 1\ 1\ 0\ 0\ 0 \end{array}$$

Figure 1.11. Rectangle-free incrementing a_{i+1} with respect to a_i

The assignment can be also skipped if $a_i \geq (1100\ldots0)_2$, $i < m$, because a_{i+1} then would always have the first two bits in common with a_i.

Another improvement utilizes the estimation of the maximum value $\mathbf{maxrf}(m-i,n)$ for a given $i \in \{1,\ldots,m-1\}$ in the remaining subgrid $(m-i,n)$ by an upper bound $z(m-i,n)$. Let be $e_{\max}$ the current maximum, b_{i-1} the accumulated bit count from previous rows and i the current row. If $b_{i-1} + z(m-i,n) \leq e_{\max}$ then this assignment can be skipped. a_i or the previous rows do not have enough bits to improve the local maximum. Now $z(m,n)$ can return the optimal value, if it is already known or it computes an upper bound instead. Theorem 1.5 uses [333, p. 25].

Theorem 1.5. *If $m \geq n > 2$ then the following inequality is satisfied:*

$$\mathbf{maxrf}(m,n) \leq \frac{1}{2}\left(m + \sqrt{m^2 + 4 \cdot mn(n-1)}\right) . \tag{1.19}$$

Proof. The number of 1 bits of row a_i, namely the pattern size, is

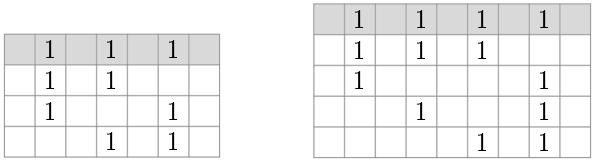


Figure 1.12. Recipes for creating higher patterns from 2-bit patterns.

denoted by $d_i \in \{1, \ldots, n\}$. So $\mathbf{maxrf}(m,n) = \sum_{i=1}^{m} d_i$ gives the total number of 1 bits in $k(m,n)$. To prove the upper bound an ideal pattern size $\sigma \in [1,n]$ has to be computed. If $m = m_0 = \binom{n}{2}$ then $\mathbf{maxrf}(m_0,n) = 2 \cdot m_0$ due to (1.16). Every row of the Boolean matrix $k(m_0,n)$ has two 1 bits. For $m < m_0$, rows from $k(m_0,n)$ can be combined to create higher patterns. Since $k(m_0,n)$ is rectangle-free, no rectangles can occur in $k(m,n)$. A pattern with size σ acquires $\binom{\sigma}{2}$ rows from $k(m_0,n)$. Since every row will have the same pattern size σ, the following condition must be satified: $m \cdot \binom{\sigma}{2} \stackrel{!}{=} m_0 = \binom{n}{2}$. Solving this for positive σ yields: $\sigma = \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{n(n-1)}{m}}$. Hence, we obtain the upper bound by assigning σ to all rows:

$$\mathbf{maxrf}(m,n) \leq m \cdot \sigma = \frac{m}{2} + \sqrt{mn(n-1) + \frac{m^2}{4}} \; . \tag{1.20}$$

□

Figure 1.12 illustrates the creation of higher pattern sizes using 2-bit patterns as explained in the proof of Theorem 1.5. For instance, a 3-bit pattern acquires three 2-bit patterns. The upper bound of (1.20) was found first by I. Reiman [248]. For $n \leq m \leq 14$, the bound is optimal or overestimates the number of edges by only one. Yet it gives a noticeable impact on the runtime of the algorithm, since more branches have to be traversed by the algorithm. Thus, we will take the exact maximum value of a subgrid, unless otherwise specified.

1.4.3. Applying Heuristics

Figure 1.13 shows the solution structure of the last optimum found by the Overflow Algorithm. It constructs a nice bow pattern of the most

1	·	·	1	·	1	1	1	·	46
2	·	·	1	1	·	·	·	1	49
3	·	1	·	·	1	·	·	1	73
4	·	1	·	1	·	1	·	·	84
5	·	1	1	·	·	·	·	·	96
6	1	·	·	·	·	1	·	1	133
7	1	·	·	1	1	·	·	·	152
8	1	·	1	·	·	·	·	·	160
9	1	1	·	·	·	·	1	·	194

1	·	·	1	·	1	1	1	·	·	92
2	·	·	1	1	·	·	·	1	1	99
3	·	1	·	·	·	1	·	1	·	138
4	·	1	·	·	1	·	·	·	1	145
5	·	1	·	1	·	·	1	·	·	164
6	1	·	·	·	·	·	1	1	·	262
7	1	·	·	·	·	1	·	·	1	265
8	1	·	·	1	1	·	·	·	·	304
9	1	1	1	·	·	·	·	·	·	448

Figure 1.13. Last obtained optimum for $G_{9,8}$, $G_{9,9}$.

significant bits. The Slot Algorithm in [333, p. 15] uses the so-called Slot Architecture that is inspired by B. Steinbach's work in [294]. The Overflow Algorithm seems to produce that Slot Architecture by itself, i.e., subdividing the matrix in groups (slots). Figure 1.13 shows instances with a slot configuration "$(4, 3, 2)$", which is defined by the size of grouped most significant bits in rows 6-9, 3-5, and 1-2.

Another interesting result can be observed on the first row a_1 and its first significant 1 bit. Its position p seems to fit $p_{\max}(m, n) \approx n - \log_2(m)$ of (1.18). Now the key idea is to fix a_1 to a pattern where the leading 1 bit is placed at $\lfloor n - \log_2(m) \rfloor$. However, the number of bits of a_1 must be known, most likely their positions are important too.

The optimal pattern of a_1 as well as the whole optimal Slot Architecture is still unknown. So the pattern of a_1 is estimated with the help of the ideal pattern size σ from Theorem 1.5. It is assumed that the pattern size of a_1 is $\sigma(s) := \lfloor \sigma + s \rfloor$ with $s \in [0, 1)$. s is a weight for choosing the next higher pattern size that must also exist in the optimal solution (assuming that its set of pattern sizes is connected). Figure 1.14 illustrates some possible configurations of the first three rows depending on σ.

$$
\begin{array}{c|cccccc|}
a_1 & . & . & 1 & . & 1 & . \\
a_2 & . & . & 1 & 1 & . & . \\
a_3 & . & 1 & . & . & . & .
\end{array}
\qquad
\begin{array}{c|cccccccc|}
a_1 & . & . & 1 & . & 1 & 1 & . & . \\
a_2 & . & . & 1 & 1 & . & . & 1 & . \\
a_3 & . & 1 & . & . & . & . & . & .
\end{array}
\qquad
\begin{array}{c|cccccccccc|}
a_1 & . & . & 1 & . & 1 & 1 & 1 & . & . & . \\
a_2 & . & . & 1 & 1 & . & . & . & 1 & 1 & . \\
a_3 & . & 1 & . & . & . & . & . & . & . & .
\end{array}
$$

Figure 1.14. Configurations of first three rows where a_1 is a fixed pattern.

1.4.4. Experimental Results

Applying the restrictions from the heuristics mentioned in Subsection 1.4.3 a significant speed-up is achieved as illustrated in Table 1.12. However, accuracy no longer can be hold on other certain dimensions. For instance, the most significant bit p of a_1 for the grids $G_{12,8}$ and $G_{12,9}$ is $p_{\max}(m,n)-1$. So only a suboptimal solution is found with the suggested heuristic. Conversely, the grid $G_{5,5}$ has an optimal solution with p at $p_{\max}(5,5)+1$ (yet an optimal solution with $p = p_{\max}(5,5)$ exists). The following questions arise from the applied heuristic ($\hat{a}_1$ refers to the first row of an optimal solution):

- What is the highest leading bit position for $\hat{a}_1$?
- Does $\hat{a}_1$ really have a pattern with size $\sigma(s)$ for given $s \in [0,1)$?
- Which of the other bit positions in $\hat{a}_1$ are possible at all?
- How many rows are in the first group (slot)? (We assumed two rows for the first group, where a_2 can be elevated and a_3 already is in the next slot.)

Table 1.12. Number of solutions and runtime (Intel X5650 2.67 GHz), $\sigma = \sigma(s),\ s = 0.3$

maxrf(m,n)		full run		#sol.	a_1 fixed		#sol.	σ
$(7,7)$	$= 21$	3 ms	(3 ms)	1	0.5 ms	(0.1 ms)	1	3
$(14,7)$	$= 31$	11 s	(2 s)	27 641	400 ms	(1.7 ms)	282	2
$(8,8)$	$= 24$	1 s	(72 ms)	16 596	14 ms	(1 ms)	128	3
$(9,9)$	$= 29$	23 s	(9 s)	4464	354 ms	(171 ms)	144	3
$(10,10)$	$= 34$	18 min	(34 s)	32 838	1.6 s	(1.6 s)	1	3
$(11,11)$	$= 39$	7 d 4 h	(9 h)	1 168 996	34 min	(6 min)	432	4
$(12,12)$	$= 45$	–	–	–	12 h 28 min	(5 h 7 min)	72	4

Table 1.12 gives a comparison of the exact and the heuristic variant of the Overflow Algorithm. In addition to the number of optimal solutions, the complete runtime and the time to find the optimal solution (in brackets) are enumerated. The pattern size for a_1 is given in case of the heuristic variant $\sigma(s)$. An optimal solution was found for the

Table 1.13. Comparison of relative runtime - theoretical and *experimental* results for the grid $G_{10,10}$

p	1	2	3	4	5	6	7	8
$\frac{N(10,10,2^p,2^n-1)}{N(10,10,1,2^n-1)}$	99.0 %	97.1 %	93.4 %	86.3 %	73.6 %	53.1 %	26.7 %	5.8 %
$\beta(p,10,10)$	99.0 %	97.1 %	93.4 %	86.3 %	73.5 %	52.9 %	26.5 %	5.7 %
full run	100.4 %	100.4 %	99.9 %	93.7 %	64.4 %	53.9 %	88.3 %	5.2 %
first solution	99.0 %	99.7 %	101.7 %	88.8 %	71.0 %	65.9 %	–	–

grid $G_{12,12}$ after just 5 h (see Table 1.11), where $\hat{a}_1$ consists of four 1 bits with the most significant 1 bit at $p = 8$.

Table 1.13 compares the experimental runtime improvement, where the most significant 1 bit of a_1 starts[1] at $p \in 0, \ldots, n-1$. So the base value 100% refers to the full runtime of the Overflow Algorithm starting with $p = 0$, i.e., $a_1 = 2^p = 1$. With $p = 6$ the algorithm needs 53.9 % of the time compared to $p = 0$. For $p > 6$ there is no optimal solution. Since the maximum could not be found at all, the branches cannot be discarded as early as for $p \leq 6$, where the maximum was found quite fast after 34 s. Thus the relative runtime increases back to 88.3 % for $p = 7$. For $p = 8$ the branches become very lightweight and only a few number of paths are visited at all.

Table 1.14. Runtime by utilizing $z(m,n)$ to estimate the Zarankiewicz function for subgrids.

	upper bound	exact
$\mathbf{maxrf}(9,9)$	5 min 49 s	23 s
$\mathbf{maxrf}(10,10)$	37 h 19 min	18 min

The developed exact algorithm solves the fast growing Zarankiewicz problem in a non-recursive manner and uses simple structure information for speed-up. In comparison to [292] the Overflow Algorithm calculates $\mathbf{maxrf}(7,7) = 21$ in 3 ms instead of 28 min. Table 1.14 shows that just the estimation of the maximum of the remaining subgrids by an upper bound can have a noticeable time impact. Even the smallest overestimation of the maximum of the subgrids yields signif-

[1] a_1 is not fixed, but only starts at $a_1 = 2^p$ and is incremented until $(1011\ldots1)_2$ is exceeded.

icant time differences. Of course, the exact maximum of a subgrid helps to discard a time-consuming branch earlier.

Based on the complexity observations combined with the solution structure of the Overflow Algorithm we introduced heuristics at the expense of accuracy in certain dimensions. Yet an enormous speed-up could be achieved and the grid $G_{12,12}$ could be solved correctly in just 12.5 hours. This illustrates the performance gain when the pattern of the first row in the adjacency matrix is known. Moreover, the heuristics give a clue how to create such a pattern or at least a range of patterns for the first row. It will be a future task to discover these important patterns.

The source code of the Overflow Algorithm and the Slot Algorithm which we did not discuss here is available on my website, see [333].

1.5. Permutation Classes

Bernd Steinbach Christian Posthoff

1.5.1. Potential of Improvements and Obstacles

A grid pattern that satisfies the rectangle-free condition (1.3) does not loose this property when any pair of rows or columns are swapped. Vice versa, a rectangle-free conflict of a grid pattern cannot be removed by swapping any pair of rows or columns.

In order to use this property we take the set of all possible patterns and define for this set a relation ρ as follows:

Definition 1.4. *$p_1 \rho p_2$ holds for two patterns p_1 and p_2 when p_1 can be transformed into p_2 by a sequence of permutations of entire rows or columns.*

This relation is

- *reflexive*: if 0 permutations are applied, then the pattern is transformed into itself;
- *symmetric*: if a pattern p_1 is transformed into a pattern p_2 by a given sequence of permutations, then the inverse sequence will transform p_2 into p_1;
- *transitive*: if there are two sequences of permutations, the first transforming p_1 into p_2, the second p_2 into p_3, then these two sequences can be combined to one sequence which is used for a direct transformation of p_1 into p_3.

These properties define the relation ρ as an equivalence relation, and therefore the set of all grids will be divided into equivalence classes.

Definition 1.5. *A permutation class of grid patterns is an equivalence class of the relation ρ of Definition 1.4.*

Table 1.15. Maximal number $m!*n!$ of patterns of permutation classes of quadratic grids of m rows and n columns

m	n	$m!$	$n!$	$m!*n!$
2	2	2.00E+00	2.00E+00	4.00E+00
3	3	6.00E+00	6.00E+00	3.60E+01
4	4	2.40E+01	2.40E+01	5.76E+02
5	5	1.20E+02	1.20E+02	1.44E+04
6	6	7.20E+02	7.20E+02	5.18E+05
7	7	5.04E+03	5.04E+03	2.54E+07
8	8	4.03E+04	4.03E+04	1.63E+09
9	9	3.63E+05	3.63E+05	1.32E+11
10	10	3.63E+06	3.63E+06	1.32E+13
11	11	3.99E+07	3.99E+07	1.59E+15
12	12	4.79E+08	4.79E+08	2.29E+17
13	13	6.23E+09	6.23E+09	3.88E+19
14	14	8.72E+10	8.72E+10	7.60E+21
15	15	1.31E+12	1.31E+12	1.71E+24
16	16	2.09E+13	2.09E+13	4.38E+26
17	17	3.56E+14	3.56E+14	1.27E+29
18	18	6.40E+15	6.40E+15	4.10E+31
19	19	1.22E+17	1.22E+17	1.48E+34
20	20	2.43E+18	2.43E+18	5.92E+36

The number of elements of an equivalence class can be between 1 (all grid points equal to 0 or equal to 1) and $m!*n!$ (each swapping of a row or a column results in a new grid pattern). The number of elements is smaller than $m!*n!$ when the same sequence of values 0 and 1 appears in more than one row or more than one column. The restriction to a single representative for all grid patterns of a permutation class is a strong potential for improvements. Table 1.15 shows how many different grid patterns can be represented by a single grid pattern. This potential for improvements increases for growing sizes of grids.

This potential for improvements is at the same time an obstacle because each of these permutation classes can contain a very large number of grid patterns. In order utilize the permutation classes of grid patterns we must answer the following questions:

1. How can permutation classes be utilized for the calculation of **maxrf**(m, n)?

2. How can a single grid pattern be chosen as a representative of the class?

1.5.2. Sequential Evaluation of Permutation Classes

An answer to the first question asked at the end of Subsection 1.5.1 gives the simple Algorithm 1.10. This algorithm utilizes the property of an equivalence relation that the whole Boolean space is completely divided into disjoint permutation classes of grid patterns. Hence, each grid pattern belongs to one single permutation class.

Algorithm 1.10 Sequential Evaluation of Permutation Classes (SEPC)

Require: TVL agp of all grid patterns $G_{m,n}$
Ensure: TVL $racgp$ representatives of all correct grid patterns

```
1:  racgp ← ∅
2:  while NTV(agp) > 0 do                    ▷ number of ternay vectors
3:      rgp ← STV(agp, 1)                    ▷ select first ternary vector
4:      rgp ← CEL(rgp, agp, "010")           ▷ substitute all '-' by '0'
5:      if RecangleFree(rgp) then
6:          racgp ← UNI(racgp, rgp)          ▷ store correct representative
7:      end if
8:      pcgp ← GeneratePermutationClass(rgp)
9:      agp ← DIF(agp, pcgp) ▷ remove grids of the permutation class
10: end while
```

Algorithm 1.10 takes advantage of permutation classes in three ways.

- The first benefit is the sequential evaluation of permutation classes. Hence, at each point of time only the grid patterns of a single permutation class must be handled.

- The second benefit is that only one representative is checked for the rectangle-free rule in line 5 of Algorithm 1.10. All other members of the permutation class are generated in line 8 of Algorithm 1.10 and excluded from further calculations in line 9.

- The third benefit is that only the single chosen representative is stored in line 6 of Algorithm 1.10.

Each grid pattern can be chosen as the representative of the permu-

tation class. Algorithm 1.10 uses as the representative grid pattern (*rpg*) simply that grid pattern which results from the first ternary vector in the list of all grid patterns (*apg*) which are not yet evaluated and in which dash elements are replaced by values 0 in line 2. The difference (`DIF`) in line 9 of Algorithm 1.10 removes all patterns of the permutation class of grid patterns (*pcgp*) from all grid patterns (*apg*) which have not yet been evaluated. Hence, each grid pattern is taken into account only once.

Unfortunately, Algorithm 1.10 has two drawbacks with regard to the required memory space:

1. The number of grid patterns of a permutation class grows exponentially depending on the size of the grid.

2. The grid patterns of a permutation class are binary vectors which have in most cases a Hamming distance lager than 1. Hence, these vectors cannot be merged well into a smaller number of ternary vectors. Consequently, the difference operation in line 9 of Algorithm 1.10 splits the remaining search space into a very large number of ternary vectors.

For these reasons Algorithm 1.10 is restricted by the available memory. In the case of 2 GB memory this limit is reached already for a grid of $m = 6$ rows and $n = 6$ columns.

1.5.3. Iterative Greedy Approach

An iterative greedy approach can overcome the obstacle of memory requirements. The basic idea of this approach is that a rectangle-free grid $G_{m,n}$ with maximal number of values 1 also includes a rectangle-free grid $G_{m-1,n-1}$ having a maximal number of values 1. We know that this assumption must not be true. However, the restriction to rectangle-free grids $G_{m-1,n-1}$ with maximal number of values 1 reduces the search space and guaranties a subset of correct solutions.

The first four lines of Algorithm 1.11 are used for initialization of the wanted maximal number n_{max} of values 1, the maximal grid $G_{1,1}$,

Algorithm 1.11 Iterative Greedy Grid Selection (IGGS)

Require: k number of rows and columns of a quadratic grid $G_{k,k}$
Ensure: n_{max} maximal number of values 1 in the grid $G_{k,k}$

```
 1: n_max ← 1
 2: G_{1,1}[1,1] ← 1                                   ▷ initial grid
 3: smgp[1] ← G_{1,1}    ▷ set of maximal grid patterns stored as TVL
 4: i ← 2
 5: while i ≤ k do
 6:     sgp[i] ← RectangleFreeGrids(smgp[i − 1])
 7:     smgp[i] ← ∅
 8:     while NTV(sgp[i]) > 0 do          ▷ number of ternay vectors
 9:         gp ← STV(sgp[i], 1)           ▷ select first ternary vector
10:         gp ← CEL(gp, sgp[i], "010")     ▷ substitute all '-' by '0'
11:         n_1 ← NumberOfValuesOne(gp)
12:         if n_1 ≥ n_max then
13:             urgp ← UniqueRepresentative(gp)
14:             if n_1 > n_max then       ▷ larger number of values 1
15:                 smgp[i] ← urgp
16:                 n_max ← n_1
17:             else                ▷ same maximal number of values 1
18:                 smgp[i] ← UNI(smgp[i], uggp)
19:             end if
20:         end if
21:         sgp[i] ← DIF(sgp[i], gp)          ▷ remove evaluated grid
22:     end while
23:     i ← i + 1
24: end while
```

the assignment of this grid to the first element $smgp[1]$ of an array of TVLs, and the index i that controls the iteration. Both the number of rows and the number of columns of the evaluated grid are increased by 1 within the while-loop in lines 5 to 24 of Algorithm 1.11.

The function RectangleFreeGrids calculates the set of rectangle-free grid patterns $sgp[i]$ of the grid $G_{i,i}$ which contain known maximal grid patterns of the grid $G_{i-1,i-1}$ in the top left part. This function restricts the introduced $2i-1$ Boolean variables by means of the rectangle-free condition (1.3). The set $smgp[1]$ includes the single existing maximal grid pattern of $G_{1,1}$. In later iterations $smgp[i-1]$

1	1
1	0

1	0
1	1

1	1
0	1

Figure 1.15. Maximal assignments of the value 1 to the grid $G_{2,2}$ where the top left position is initially assigned to the value 1.

contains only representatives of found maximal permutation classes. In this way the required memory is strongly restricted.

The task of the while-loop in lines 8 to 22 is the selection of representatives of maximal grid patterns out of the set of rectangle-free grid patterns $sgp[i]$. These selected grid patterns are stored in $smgp[i]$ that is initialized as empty set before this loop. In order to limit the memory space, each grid pattern of $smgp[i]$ is separately selected in lines 9 and 10, evaluated in lines 11 to 20, and removed from the set $smgp[i]$ in line 21.

As first step of the evaluation the number n_1 of values 1 in the grid pattern is counted in the function of line 11. If this number is smaller than n_{max} then this grid pattern is excluded from further evaluations in line 21. Otherwise, a unique representative grid pattern $urgp$ of the dedicated permutation class is calculated in line 13. This mapping of the grid pattern to the unique representative reduces the number of grid patterns which must be evaluated in the next round of iteration.

If the number n_1 of values 1 is larger than the so far known maximal number n_{max} of values 1 then $smgp[i]$ is replaced by the new maximal representative grid pattern $urgp$ in line 15 and the larger number n_1 of values 1 is stored as new value of n_{max} in line 16. Otherwise, the representative grid pattern $urgp$ is included into the set $smgp[i]$.

Algorithm 1.11 finds in line 6 of the first iteration 7 of 15 rectangle-free grid patterns $G_{2,2}$ where the top left position is initially assigned to the value 1. The maximal number of values 1 assigned to the rectangle-free grid $G_{2,2}$ is equal to 3. Hence, four grid patterns with less than three values 1 are not included into the set $smgp[2]$ due to the decision in line 12. Figure 1.15 shows the set of maximal grid patterns $G_{2,2}$ that contain the value 1 in the top left position due to the initialization in line 2.

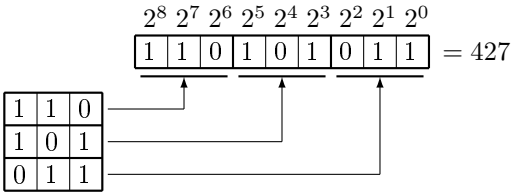


Figure 1.16. Decimal equivalent of a grid pattern.

It can be seen in Figure 1.15 that these three grid patterns belong to the same permutation class. The grid pattern in the middle of Figure 1.15 can be built by permutation of the rows in the leftmost grid pattern, and the rightmost grid pattern is the result of the permutation of the columns of the leftmost grid pattern. For larger grids this benefit grows exponentially depending on the size of the grid. Hence, a unique representative of grid patterns strongly restricts the effort of the next round of iteration. Algorithm 1.11 hides this task in the function UniqueRepresentative(gp) in line 13. In the next two sections we present both one exact and one soft computing approach for this task.

1.5.4. Unique Representative of a Permutation Class

Algorithm 1.10 of Subsection 1.5.2 generates all grid patterns of a permutation class in the function GeneratePermutationClass and can use therefore an arbitrarily chosen representative grid pattern. Algorithm 1.11 of Subsection 1.5.3 takes advantage of the sequential evaluation of few grid patterns. This approach requires the direct mapping of the actually evaluated grid pattern to a unique representative of the dedicated permutation class.

Generally, each grid pattern of a permutation class can be chosen as representative. Hence, a rule must be defined for the unique selection of a representative of a permutation class. A simple possibility for the unique selection of the representative of a permutation class is the utilization of the decimal equivalent of a grid pattern.

A grid pattern of m rows and n columns can be mapped onto a binary

Algorithm 1.12 Representative of a Permutation Class (RPC)

Require: TVL *gp* of a single grid pattern of the grid $G_{m,n}$ of m rows and n columns
Ensure: TVL *urgp* of the unique representative grid pattern of the dedicated permutation class of *gp*
1: $spgp \leftarrow$ PERMUTEROWS(gp, m, n)
2: $spgp \leftarrow$ PERMUTECOLUMNS$(spgp, m, n)$
3: $urpc \leftarrow$ SELECTREPRESENTATIVE$(spgp)$
4: **return** *urpc*

vector of the length $m * n$ where the top left corner of the grid is mapped to the bit with the value 2^{m*n-1} followed by the bits of the first row, the bits of the remaining rows in the given natural order until the bit of the bottom right corner of the grid with the value 2^0. Figure 1.16 illustrates how the decimal equivalent 427 of a grid pattern of 3 rows and 3 columns is built. A simple possibility to define a representative of a permutation class is the use of the grid pattern which belongs to the decimal equivalent of the the highest value.

This simple selection of the representative requires the knowledge of all grids of a permutation class. However, only a single grid pattern of a permutation class is known in Algorithm 1.11 for the invocation of the function UNIQUEREPRESENTATIVE(*gp*) in line 13. Hence, it is necessary to generate temporarily all patterns of a permutation class of the given grid size. Algorithm 1.12 solves this problem such that at first the set of all permuted grid patterns *spgp* is calculated by the functions PERMUTEROWS(gp, m, n) and PERMUTECOLUMNS$(spgp, m, n)$ in lines 1 and 2. The function SELECTREPRESENTATIVE(*spgp*) calculates as final step for each grid pattern of the permutation class the decimal equivalent and returns the grid pattern of the highest decimal equivalent as the wanted representative.

Algorithm 1.12 can be used as function UNIQUEREPRESENTATIVE(*gp*) in the iterative greedy grid search of Algorithm 1.11. The results of the combination of these algorithms confirms the strong benefit of the iterative greedy approach to find the maximal number of assignments of values 1 to a grid. Up to the grid $G_{4,4}$ only a single permutation class of maximal grid patterns exists. These representatives are shown in Figure 1.17. That means, in case of the grids $G_{4,4}$ only one of all

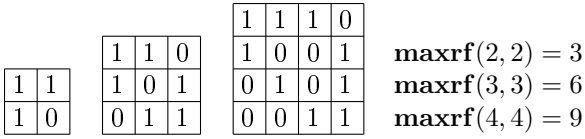


Figure 1.17. Representatives of permutation classes of maximal assignments of values 1 to the grids $G_{2,2}$, $G_{3,3}$, and $G_{4,4}$.

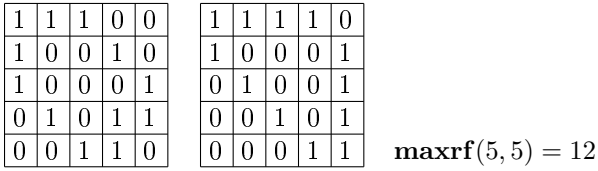


Figure 1.18. Representatives of permutation classes of maximal assignments of values 1 to the grid $G_{5,5}$.

$2^{16} = 65,536$ grid patterns carries the needed information.

Two different maximal permutation classes were calculated within 1.3 seconds for assignments of values 1 to the grid $G_{5,5}$. Figure 1.18 shows theses representatives. The reduction in terms of needed memory space to represent the maximal grid patterns decreases from $2^{25} = 33,554,432$ down to 2.

Another interesting observation: there is only one single maximal permutation class of assignments of the value 1 to the grid $G_{6,6}$. The representative grid of this permutation class is shown in Figure 1.19.

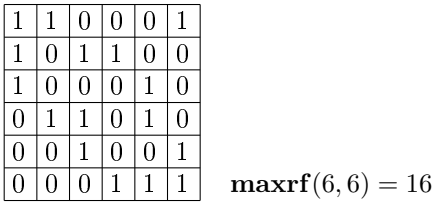


Figure 1.19. Representative of the permutation class of maximal assignments of values 1 to the grid $G_{6,6}$.

From all $2^{36} = 68,719,476,736 = 6.9 * 10^{10}$ grid patterns of the grid $G_{6,6}$ only a single grid pattern is needed as representative. This is a strong reduction in terms of required memory. However, it takes already 338 seconds to find this single representative of the maximal permutation class of the grid $G_{6,6}$ even though the calculation of all rectangle-free grid patterns using the iterative greedy approach needs only 11 milliseconds. The reasons for this strong difference are:

1. based on the two representatives of maximal grid patterns $G_{5,5}$ all extended rectangle-free patterns of the grid $G_{6,6}$ can be represented by 60 ternary vectors,

2. only three of these grid patterns have a maximal number of 16 values 1, but

3. $6! * 6! = 720 * 720 = 518,400$ grid patterns $G_{6,6}$ of the permutation class must be generated in order to select the single unique representative.

From this analysis we can conclude that Algorithm 1.12 has both strong benefits and a strong drawback. The benefits are:

1. a single representative of $G_{m,n}$ represents the large number of $m! * n!$ grid patterns so that both the required calculation time for the next larger grid $G_{m+1,n+1}$ and the needed memory to store these grid patterns is drastically reduced,

2. the permutation classes can be sequentially evaluated so that the memory to store all patterns of a single permutation class is sufficient.

The drawback of Algorithm 1.12 is that all $m! * n!$ grid patterns of a single maximal permutation class must be generated in order to select the representative using the maximal value of the decimal equivalent. Taking into account that the number of grid patterns of a single permutation class grows from grid $G_{m-1,n-1}$ to grid $G_{m,n}$ by a factor of $m * n$, the limit for the application of the algorithm RPC is reached for the grid $G_{6,6}$ or, depending on the computer, for grids which are a little bit larger.

Algorithm 1.12 can be modified such that the calculation of the decimal equivalent and its evaluation is executed immediately when a new permuted grid pattern of the class is generated. In this way only a single grid pattern must be stored in the modified RPC algorithm at each point in time so that the memory problem is solved. However, the time problem remains because all $m! * n!$ grid patterns of a permutation class must be sequentially evaluated.

1.5.5. Direct Mapping to Representatives

In Subsection 1.5.4 we learned that it is practically not possible to generate and store all grid patterns of a permutation class for grids larger than approximately $G_{6,6}$. The sequential evaluation of all $m! * n!$ grid patterns of a permutation class overcomes the memory problem. However, due to the exponential increase of the number of grid patterns of a permutation class, the time required for their evaluation limits practical solutions for a similar size of grids. Hence, we need an algorithm that allows us to map a known grid pattern of a permutation class directly to a unique representative.

There are two important questions which must be answered to solve this task.

1. How the representative grid pattern of a permutation class is defined?

2. Which information is constant for rows and columns of a grid independent on any permutation of rows and columns?

The answer to the first question is that we can choose each grid pattern of a permutation class as representative. However, we must use exactly the same representative for each given grid pattern of the same permutation class. Hence, the rules which specify the representative can be expressed by an algorithm that maps a given grid pattern to the representative of the class by certain permutations of rows and columns.

The answer to the second question is that the horizontal checksum of

a row does not change for any permutation of columns and the vertical checksum of a column does not change for any permutation of rows.

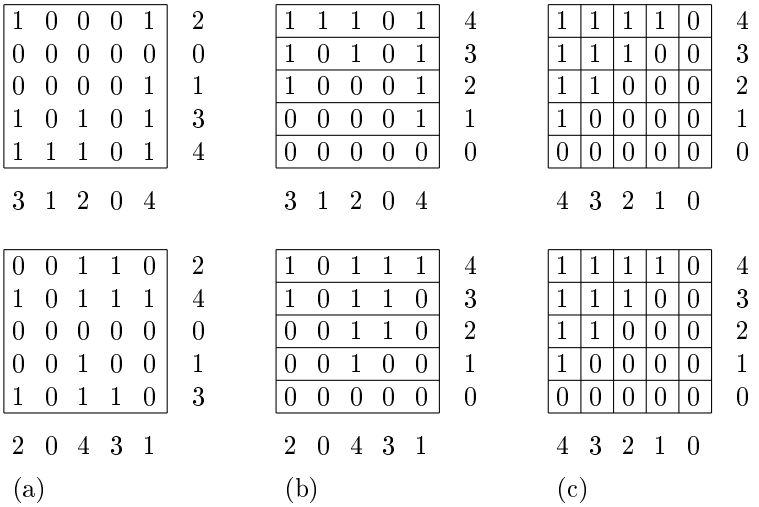


Figure 1.20. Mapping onto a representative by permutations controlled by descending checksums for rows and columns for two grid patterns $G_{5,5}$: (a) two given grid patterns, (b) grid patterns after row permutations, (c) final grid patterns after column permutations.

The main idea of such a mapping algorithm of any grid pattern to the representative of the dedicated permutation class is the ordering of the rows of the grid pattern top down in descending order of the horizontal checksums and the ordering of the columns of the grid pattern from the left to the right in descending order of the vertical checksums. When we assume that both all horizontal checksums and all vertical checksums are different then the permutation of rows and columns in descending order of their checksums creates a unique representative grid pattern. Figure 1.20 shows how two grid patterns $G_{5,5}$ of the same permutation class are mapped onto their representative. The checksums of rows are annotated on the right-hand sides of the grids, and the checksums of the columns are labeled below the corresponding columns.

In a first step, the rows are ordered top down with regard to descend-

(a)

1	1	0	0	0	1	3
1	0	1	1	0	0	3
1	0	0	0	1	0	2
0	1	1	0	1	0	3
0	0	1	0	0	1	2
0	0	0	1	1	1	3

(b)

1	1	0	0	0	1	3
1	0	1	1	0	0	3
0	1	1	0	1	0	3
0	0	0	1	1	1	3
1	0	0	0	1	0	2
0	0	1	0	0	1	2

Figure 1.21. Mapping of a grid pattern $G_{6,6}$ using checksums of rows in descending order: (a) given grid, (b) unique row intervals.

ing horizontal checksums. Intervals defined by the same horizontal checksum are indicated by horizontal lines in Figure 1.20 (b). In the next step the columns are ordered from the left to the right with regard to descending vertical checksums. Again, intervals defined by the same vertical checksum are indicated be vertical lines in Figure 1.20 (c). After these two permutation steps each element of the given grid patterns is assigned to a unique position in the grid. Hence, we have created a grid pattern which we define and use as representative of the permutation class. Algorithm 1.13 realizes this procedure.

A special feature of the example shown in Figure 1.20 is that no checksum of the rows and no checksum of the columns occurs more than once. Generally, the checksums of several rows or columns can be equal to each other. In this case it is not possible to create directly a grid pattern where each interval consists of a single row or a single column. However, intervals of several rows with the same horizontal checksum and intervals of columns with the same vertical checksum can also be built in a unique way.

In order to study the most general case of grid patterns, we consider grid patterns with equal checksums for the rows or the columns. Figure 1.21 shows a given grid pattern (a) and the resulting grid pattern (b) after permuting the rows such that intervals in descending order are built. This interval border remains fixed in all further steps of permutations to the final representative grid pattern. The permutations are executed by swapping neighboring rows.

Column intervals are built based on the vertical checksums of the

1	1	0	0	0	1	3
1	0	1	1	0	0	3
0	1	1	0	1	0	3
0	0	0	1	1	1	3
1	0	0	0	1	0	2
0	0	1	0	0	1	2
2	2	2	2	2	2	
1	0	1	0	1	1	

(a)

1	0	0	1	1	0	2 1
1	1	0	0	0	1	2 1
0	1	1	0	1	0	2 1
0	0	1	1	0	1	2 1
1	0	1	0	0	0	2 0
0	1	0	1	0	0	2 0
2	2	2	2	2	2	
1	1	1	1	0	0	

(b)

Figure 1.22. Mapping of a grid pattern $G_{6,6}$ using checksums of columns in descending order: (a) given grid ordered according to row intervals, (b) unique intervals of rows and columns.

created row intervals in the next step. All the checksums of the upper row interval are equal to 2, the checksums of the lower interval have the values 0 and 1. We arrange the columns from the left to the right in such a way that the columns with the values 1 come first, the columns with the value 0 second. When we consider the column checksums of the initial grid, we have the value 3 on the left side (columns 1 - 4) and the value 2 in the columns 5 and 6. Figure 1.22 shows both the additionally created column intervals and all checksums of the intervals.

The new intervals allow a more detailed differentiation between rows of fixed row intervals and columns of fixed column intervals. We calculate separate checksums for the intervals and order them according to the order of the intervals. As can be seen in Figure 1.22 (b), horizontal checksums are restricted to the fixed column intervals and vertical checksums are restricted to the fixed row intervals.

New intervals of rows or columns are created when there are different checksums of intervals where a horizontal checksum of a left interval dominates all checksums of intervals located to the right. Similarly, a vertical checksum of an upper interval dominates all checksums of intervals located thereunder. This procedure of creating smaller intervals of rows and columns is repeated iteratively within one loop while differences in the checksums exist.

In the special case of Figure 1.22 the checksums for each interval are

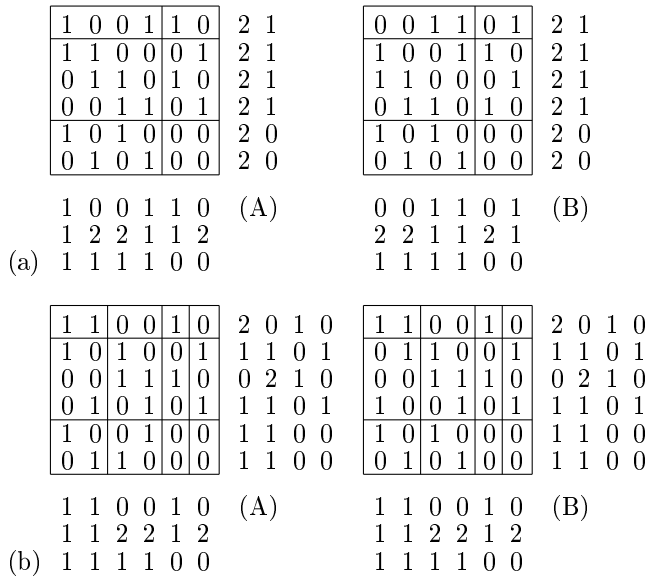


Figure 1.23. Mapping of two grid patterns $G_{6,6}$ of the same permutation class for different topmost intervals: (a) new row interval, (b) subsequent creation of column intervals.

already equal to each other for both the rows and columns. Hence, the rows of the upper interval (the first four rows) are equivalent to each other with regard to their checksum vectors in each column interval, therefore each of them can be chosen as content of a separate interval. An analog statement is true for the columns of the left interval.

Algorithm 1.13 (MRPC) separates in such a case the topmost row as a single interval and creates new column intervals based on the new checksums. Thereafter, the leftmost column is separated as a single column interval, and new row intervals based on the new checksums are created, respectively. In order to demonstrate that different selections out of a set of rows with identical checksum vectors lead to the same representative, we show in parallel the further transformation steps for the selection of the first row as case (A) and the selection of the fourth row as case (B). Figure 1.23 shows the additionally created intervals for both cases.

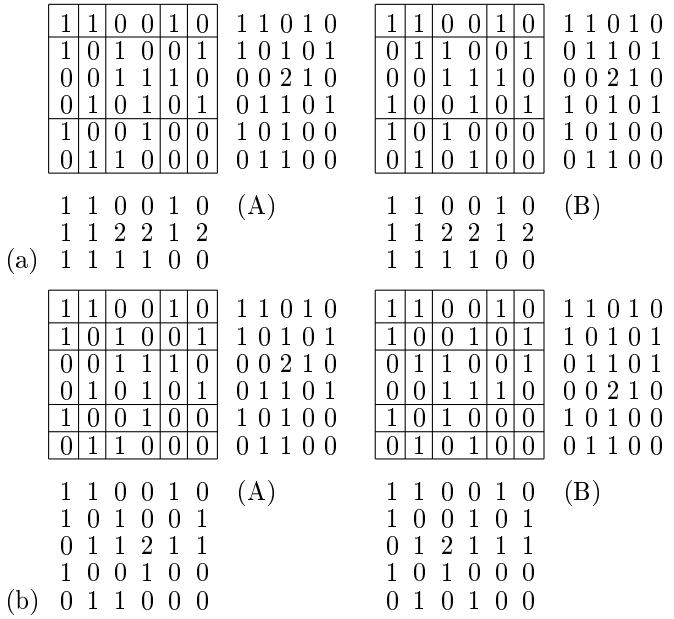


Figure 1.24. Mapping of two grid patterns $G_{6,6}$ of the same permutation class for different topmost intervals: (a) new column interval, (b) subsequent creation of row intervals.

As mentioned above, next the first column is selected from the equivalent columns of the left interval. Based on this additionally fixed column interval further row intervals can be created uniquely. Figure 1.24 shows the results of these steps for both grids of Figure 1.23 (b).

Only the third row interval remains with more than one row. Based on the different values in the second column of this interval (see Figure 1.24 (b)) these rows can be separated in fixed row intervals. In case of grid (A) of Figure 1.24 (b), the third and fourth row must be swapped because $G[3,2] < G[4,2]$; the result is shown in Figure 1.25 (a).

Due to different values in the second row of the third column interval, the last column interval of more than one column can be separated into two column intervals of one column each (see Figure 1.25). Here it is necessary to swap the third and fourth column of grid (B) of Figure

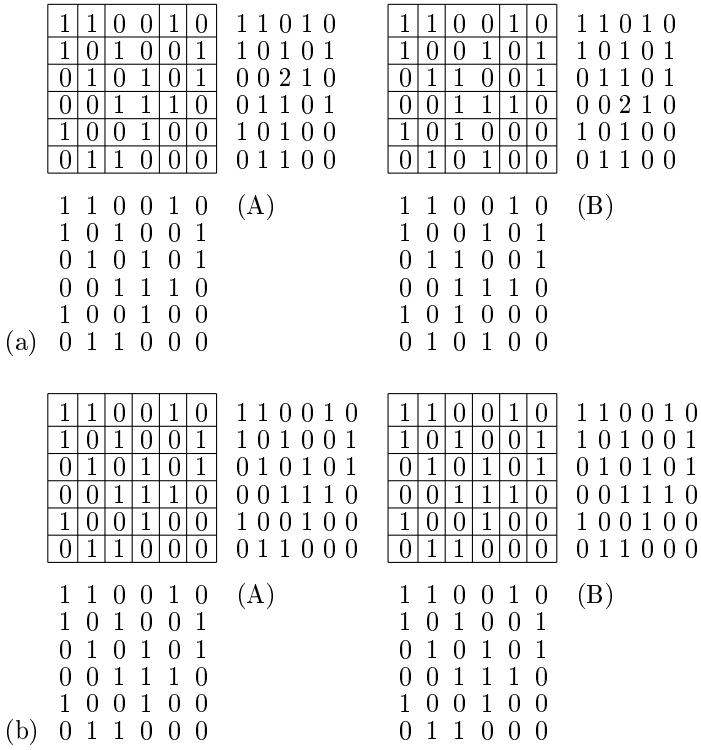


Figure 1.25. Mapping of two grid patterns $G_{6,6}$ of the same permutation class for different topmost intervals: (a) completely fixed row intervals, (b) completely fixed column intervals: this is the unique representative.

1.25 (a), because $G[2,3] < G[2,4]$; the result of this step is shown in Figure 1.25 (b). After this step, both all rows and all columns are separated from each other by intervals and define in this way the *representative of the permutation class of grid patterns*. Figure 1.25 (b) shows identical grid patterns which are constructed from different grid patterns (A) and (B) of Figure 1.23 (a) which are taken from the same permutation class of grids.

The main steps of the algorithm MRPC are summarized in Algorithm 1.13. The matrix *rim* stores the checksums of the row intervals and

Algorithm 1.13 Mapping onto Representatives of Permutation Classes (MRPC)

Require: TVL gp of a single grid pattern of the grid $G_{m,n}$ of m rows and n columns
Ensure: TVL $urgp$ of unique representative grid pattern of the dedicated permutation class of gp

```
 1: rim ← InitRm(m, n)                    ▷ initialize row interval matrix
 2: cim ← InitCm(m, n)                 ▷ initialize column interval matrix
 3: while any interval of rows or columns > 1 do
 4:     while checksums specify smaller intervals do
 5:         ⟨gp, rim⟩ ← CreateRowIntervals(gp, cim)
 6:         ⟨gp, cim⟩ ← CreateColumnIntervals(gp, rim)
 7:     end while
 8:     if any interval of rows > 1 then
 9:         ⟨gp, rim⟩ ← DefineRowInterval(gp, cim)
10:         ⟨gp, cim⟩ ← CreateColumnIntervals(gp, rim)
11:     end if
12:     if any interval of columns > 1 then
13:         ⟨gp, cim⟩ ← DefineColumnInterval(gp, cim)
14:         ⟨gp, rim⟩ ← CreateRowIntervals(gp, rim)
15:     end if
16: end while
17: urpc ← gp
18: return urpc
```

is initialized in line 1. Similarly, the matrix cim stores the checksums of the column intervals and is initialized in line 2. The actions in the while-loop in lines 3 to 16 are repeated until all row intervals have a height of 1 and all column intervals have a width of 1.

The inner while-loop in lines 4 to 7 creates in a sequence first new row intervals and thereafter new column intervals based on different dominating checksums. The functions of lines 5 and 6 extend the dedicated interval matrix by new checksums and exchange required rows or columns in the grid pattern gp.

The function

$$\textsc{CreateRowIntervals}(gp, cim)$$

works as shown in Figure 1.21, and the function

$$\textsc{CreateColumnIntervals}(gp, rim)$$

proceeds as shown in Figure 1.22.

In the case that neither rows nor columns of intervals can be separated based on their checksums, the function

$$\textsc{DefineRowInterval}(gp, cim)$$

creates a row interval of one row in line 9 of Algorithm 1.13. An additional condition for this step is that at least on row interval consists of more than one row. Commonly, the row interval matrix rim is extended with the created row interval. An example of this step is shown in Figure 1.23 (a). Due to the new checksums of row intervals, the function

$$\textsc{CreateColumnIntervals}(gp, rim)$$

builds additional column intervals in line 10. This step is illustrated in Figure 1.23 (b).

Similarly, the function

$$\textsc{DefineColumnInterval}(gp, rim)$$

builds a new single column interval within an interval of more than one column with identical checksums in line 13 of Algorithm 1.13. An example for this step is shown in Figure 1.24 (a). Function

$$\textsc{CreateRowIntervals}(gp, cim)$$

in line 14 creates possible consecutive row intervals as shown in Figure 1.24 (b).

The while-loop in lines 3 to 16 terminates when each element of the grid pattern gp belongs exactly to one row interval and to one column interval. Necessary permutations of rows and columns change the given grid pattern into a pattern that is used as representative of the permutation class. Figure 1.25 (b) shows that in the explored example the same representative is created for the different grid patterns (A)

and (B) of Figure 1.23 (a) which are elements of the same permutation class.

It must be mentioned that Algorithm 1.13 (MRPC) belongs to the class of soft-computing algorithms. This algorithm allows to find representatives of extremely large permutation classes but cannot guarantee that a unique representative is constructed for each member of the permutation class. Vice versa, in the case that Algorithm 1.13 (MRPC) creates the same representative for a given set of grid patterns, all these grid patterns belong to the permutation class, certainly. Hence, the benefits of Algorithm 1.13 (MRPC) can be measured by the number of grid patterns which are mapped to the same representative of a permutation class of grids. In detail we evaluate this quality in the next section of experimental results.

1.5.6. Soft-Computing Results

We implemented Algorithm 1.11 of the iterative greedy approach of subsection 1.5.3 and used Algorithm 1.13 (MRPC) as implementation of the function UNIQUEREPRESENTATIVE(gp) in line 13 of Algorithm 1.11 for the direct mapping onto representatives of a permutation class. Table 1.16 summarizes the results of this combined soft-computing approach.

The first three columns of Table 1.16 are labeled by m, n, and v which means that the evaluated grid has m rows and n columns so that $v = m*n$ Boolean variables are necessary to store the assignments of values 1 to the grid elements. It can be seen that the number of needed variables ranges from 4 to 400. The runtime of about 5 seconds for the last iteration step reveals that the complexity of 2^{400} does not restrict the calculation of even larger grids. We did not continue the calculation for grids larger than $G_{20,20}$ because in the context of chapter 2 we are interested in the results for the grid $G_{18,18}$.

We evaluated in this experiment only grids with the same number of rows and columns. Generally, grids of each size can be evaluated using the applied approach. Again, the restriction to quadratic grids is caused by the focus on the grid $G_{18,18}$ that has the same number

Table 1.16. Results of the iterative greedy approach in combination with the direct mapping onto a representative of a permutation class.

m	n	n_v	recursively defined maximal assignments	grids	classes	ratio v/ass.	time in seconds
2	2	4	3	3	1	0.750	0.001
3	3	9	6	1	1	0.667	0.171
4	4	16	9	3	1	0.563	0.176
5	5	25	12	7	2	0.480	0.196
6	6	36	16	3	1	0.444	0.189
7	7	49	21	1	1	0.429	0.192
8	8	64	24	28	4	0.375	0.418
9	9	81	28	108	12	0.346	3.149
10	10	100	33	48	8	0.330	1.612
11	11	121	38	51	11	0.314	1.483
12	12	144	43	142	31	0.299	3.860
13	13	169	49	85	33	0.290	3.449
14	14	196	56	4	3	0.285	1.023
15	15	225	61	63	15	0.271	3.932
16	16	256	67	180	8	0.262	11.024
17	17	289	74	16	1	0.256	2.704
18	18	324	81	3	1	0.250	2.272
19	19	361	88	6	4	0.244	3.246
20	20	400	96	12	2	0.240	5.370

of rows and columns.

The main results are given in columns 4 to 6 of Table 1.16. The meaning of this columns must be explained precisely. The first row summarizes the exact solution for the grid $G_{2,2}$. There are exactly three grid patterns (column 5) in which three elements (column 4) carry the value 1 having the initial grid value $G[1,1] = 1$. Figure 1.15 shows these grids which belong to the same permutation class so that a single representative (column 6) is constructed by Algorithm 1.13. This representative is the leftmost grid of Figure 1.15.

The results for all grids $G_{m+1,n+1}, m \geq 2, n \geq 2, n = m$ are calculated based on the results of $G_{m,n}$. Hence, only grids of the known maximal

assignments of values 1 of the grid $G_{m,n}$ are extended to associated grids $G_{m+1,n+1}$ with a maximal number of values 1. This greedy approach must not define exact maximal assignments because less assignments of values 1 in a grid $G_{m,n}$ can allow more assignments of values 1 in the added row and column to construct the grid $G_{m+1,n+1}$.

The numbers of recursively defined maximal grid patterns and permutation classes in columns 5 and 6 of Table 1.16 reveal an interesting *self-adjustment behavior* of the used iterative greedy approach. Generally, there are more rectangle-free grid patterns and permutation classes which do not have the maximal number of assignments of values 1. We verified by another approach that the number of assignments given in Table 1.16 for the grids from $G_{2,2}$ until $G_{8,8}$ are the largest possible rectangle-free assignments of values 1. Hence, in this range the values of column 4 of Table 1.16 are equal to $\mathbf{maxrf}(m,n)$. The significantly larger number of rectangle-free grids $G_{9,9}$ is caused by the fact that no maximal rectangle-free grid $G_{9,9}$ includes a maximal rectangle-free grid $G_{8,8}$. Vice versa, the smaller number of rectangle-free grids $G_{14,14}$ or $G_{17,17}$ in comparison to the grids $G_{13,13}$ or $G_{16,16}$ indicates the return to maximal grids.

Table 1.16 shows in column 7 the ratio between all variables v which is equal to all grid positions and the maximal number of rectangle-free assignments of values 1. This ratio decreases from 0.75 for $G_{2,2}$ to 0.25 for $G_{18,18}$. This ratio 0.25 for the grid $G_{18,18}$ is very important for the task to solve in Chapter 2.

Finally, we evaluate the advantage of the direct mapping onto representatives of a permutation class by Algorithm 1.13 (MRPC) as implementation of function UNIQUEREPRESENTATIVE(gp) in line 13 of Algorithm 1.11 (IGGS). Algorithm 1.13 reduces the number of found grids in most cases to a smaller number of representatives of permutation classes as can be seen by comparing the columns 5 and 6 of Table 1.16. Only in the case of a single grid pattern no smaller number of representatives can be found. Algorithm 1.11 belongs to the class of soft-computing algorithms because the aim of maximal assignments of values 1 cannot be reached in each sweep of the iteration. Using the soft-computing algorithms IGGS and MRPC, the strong lower bound of $\mathbf{maxrf}(20,20) = 96$ in the gigantic search space of $2^{400} = 2.58 * 10^{120}$ could be calculated within few seconds using a

normal PC.

The achieved improvement (the quotient of the values in columns 5 and 6 of Table 1.16) is in the range between 1 for the grid $G_{2,2}$ and 22.5 for the grid $G_{16,16}$. On the first glance these reductions do not seem to be strong. However, due to the recursive greedy approach we have to multiply these quotients so that Algorithm 1.13 (MRPC) contributes an overall improvement of $1.41816 * 10^{11}$ to find the rectangle-free grid patterns of $G_{20,20}$. Without the utilization of Algorithm 1.13 (MRPC) $2.83632 * 10^{11}$ rectangle-free grid patterns must be calculated and stored instead of only 2 of these grid patterns calculated by Algorithm 1.13. Taking into account that 50 bytes are needed to store a single solution vector of 400 bits, 14.1816 Terabytes are necessary to store the iteratively calculated solutions of rectangle-free grids $G_{20,20}$. Assuming furthermore, each of the $2.83632 * 10^{11}$ rectangle-free grids can be calculated within 1 millisecond, then it takes more than 9 years to create all of them. We see, without the utilization of the direct mapping onto representatives of permutation classes it is practically impossible to solve the considered task using normal resources.

2. Four-Colored Rectangle-Free Grids

2.1. The Problem to Solve and its Complexity

Bernd Steinbach Christian Posthoff

2.1.1. Extension of the Application Domain

Many practical problems innately belong to the Boolean domain. Boolean problems have the characteristic property that their variables can attain only two values. Such pairs of values can be, e.g.,

- the light in the room is *on* or *off*,
- the train drives at a switch to the left or to the right,
- the book is in a bookstore available or not available,
- the answer to a question is *yes* or *no*,
- a paper is accepted or not accepted for publication in a journal,
- one more picture can be stored on the memory card of a camera, and so on.

A small number of Boolean problems were presented in Chapter 1 to motivate the Boolean rectangle problem. Many other Boolean problems are explored in the other chapters of this book. Hence, there is a very wide field in which Boolean methods can directly be used to solve the given problem.

In this chapter we extend the field of problems in which Boolean methods can be utilized in order to find a solution. The variables of this extended domain do not only hold two values but a finite number of different values. Such variables are called multiple-valued variables. Very often these problems are summarized in parts of *Discrete Mathematics*.

An example of multiple-valued variables is the assignment of one of a small number of frequencies to the hotspots of a mobile net. Neighbored hotspots must use different frequencies in order to reduce the mutual perturbations. This problem leads to the well-known problem that nodes of a graph must satisfy certain restrictions. We will study a similar problem in this chapter.

All multiple-valued problems can be mapped into the Boolean domain because each multiple-valued variable holds only a finite number of different values, and k Boolean variables suffice to represent up to 2^k different values of a multiple-valued variable. Basically, the table to encode up to 2^k different values by k Boolean variables can be arbitrarily chosen. However, the same table must be used to map solutions from the Boolean domain back into the multiple-valued domain.

One general approach to solve a multiple-valued problem consists in three steps:

1. encode the multiple-valued variables by Boolean variables using a unique code,
2. solve the problem in the Boolean domain,
3. decode the results stored in the Boolean domain back into the multiple-valued domain using the same coding table in the reverse direction.

2.1.2. The Multiple-valued Problem

This chapter deals with a special graph-coloring problem. Examples for applications of graph-coloring were already introduced in Chapter 1. Further applications of graph-coloring are explained in [186]. Now

we extend the range of graph-coloring. We found this problem in [98] published on a web page. The short definition of the problem in this paper is as follows.

> A two-dimensional *grid* is a set $G_{m,n} = [m] \times [n]$. A grid $G_{m,n}$ is *c-colorable* if there is a function $\chi_{m,n} : G_{m,n} \to [c]$ such that there are no rectangles with all four corners of the same color.

In comparison with [98] we exchanged in this definition the variables m and n to get a natural alphabetic order of m rows and n columns.

In order to explain the problem in a way which is easy to understand we start with a simple bipartite graph $G(V, E)$. The set of all vertices V of the bipartite graph G is divided into two subsets V_1 and V_2 with $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$. It is a special property of each bipartite graph G that each edge $e \in E$ connects one vertex $v_{1i} \in V_1$ with one vertex $v_{2k} \in V_2$. There are no edges between vertices of the subset V_1 or the subset V_2.

Only complete bipartite graphs are taken into consideration. That means that each vertex of V_1 is connected with each vertex of V_2. Figure 2.1 shows as example two different complete bipartite graphs of three vertices in the subset $V_1 \subset V$ and four vertices in the subset $V_2 \subset V$.

Each edge of the complete bipartite graph must be colored using one of four colors. The four colors red, green, blue, and yellow are mapped to four-valued variables of the edges using the encoding table of Figure 2.1. This table additionally shows different styles to represent the color of an edge in a black-white picture of the graph.

The left graph $G^1_{3,4}$ of Figure 2.1 contains:

- two red (1) edges: $\{e(v_{11}, v_{21}), e(v_{12}, v_{23})\}$
- three green (2) edges: $\{e(v_{11}, v_{22}), e(v_{12}, v_{22}), e(v_{13}, v_{21})\}$
- four blue (3) edges: $\{e(v_{11}, v_{23}), e(v_{11}, v_{24}),$ $e(v_{13}, v_{23}), e(v_{13}, v_{24})\}$, and
- three yellow (4) edges: $\{e(v_{12}, v_{21}), e(v_{12}, v_{24}), e(v_{13}, v_{22})\}$.

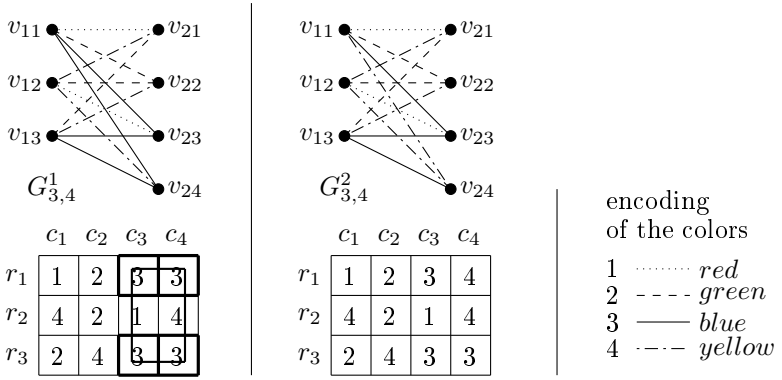


Figure 2.1. Edge colorings of two complete bipartite graphs $G^1_{3,4}$ and $G^2_{3,4}$ using four colors: incorrect graph $G^1_{3,4}$ that violates the rectangle-free condition and rectangle-free graph $G^2_{3,4}$.

The graph $G^1_{3,4}$ contains a complete subgraph $K_{2,2}$ in which all edges between two vertices of V_1 and two vertices of V_2 are colored by the same color. These are the blue-colored edges $e(v_{11}, v_{23})$, $e(v_{11}, v_{24})$, $e(v_{13}, v_{23})$, and $e(v_{13}, v_{24})$. Such a monochromatic cycle of four edges violates the color condition. Hence, the graph $G^1_{3,4}$ must be excluded from the set of wanted graphs. Such a coloring of a complete bipartite graph is called incorrect.

The graphs $G^1_{3,4}$ and $G^2_{3,4}$ differ only in the color of the edge $e(v_{11}, v_{23})$. The graph $G^2_{3,4}$ does not contain any monochromatic subgraph $K_{2,2}$. Hence, such a rectangle-free coloring of a complete bipartite graph is called correct.

An alternative to the graphical representation of a bipartite graph $G_{m,n}$ is an adjacency matrix. Such a matrix is also called a *grid*. We prefer this short term in the rest of this chapter.

The subset $V_1 \subset V$ is mapped to the set of rows R, and the subset $V_2 \subset V$ is mapped to the set of rows C. The row r_i of such a grid indicates the vertex $v_{1i} \in V_1$, and the column c_k refers to the $v_{2k} \in V_2$. As usual for a matrix, the row numbers r_i of the grid grow top down, and the column numbers c_k grow from the left to the right. The

elements of the grid are the colors indicated by associated integers. A value $c \in \{1, \ldots, 4\}$ in the position of row r_i and column c_k means that the edge $e(r_i, c_k)$ is colored with the given color c.

Figure 2.1 shows the grids (adjacency matrices) below the graphical representation of the bipartite graphs $G^1_{3,4}$ and $G^2_{3,4}$. Four positions of the left grid $G^1_{3,4}$ of Figure 2.1 are framed by thick lines. These four positions describe the four edges of a complete subgraph $K_{2,2}$ that violates the color condition. It can be seen that these four positions are located in the corners of a rectangle. There is a monochromatic rectangle when all four rectangle positions are labeled by the same values.

Due to this special locations of the four edges of the monochromatic subgraph $K_{2,2}$ in the corners of a rectangle we can substitute the term *color condition* by the term *rectangle-free condition*. Hence, the problem to solve is:

> *Are there rectangle-free grids of a certain size which are completely colored using four colors.*

The result of a comprehensive mathematical analysis of this problem in [98] are:

- upper bounds of grid sizes which can be colored rectangle-free using four colors, and
- lower bounds of grid sizes which cannot be colored rectangle-free using four colors.

There are six grids for which it is unknown whether the grid can be colored rectangle-free using four colors. Table 2.1 is a part of a table given in [98], and indicates the so far unsolved grids by bold letters U. We are going to solve all this problems in this chapter.

Four of these so far unsolved grids differ in the size of rows and columns only by one. A four-colored rectangle-free grid $G_{18,18}$ can be restricted to:

Table 2.1. Sizes of grids which are four-colorable rectangle-free (C), not four-colorable rectangle-free (N), or where it is unknown whether a rectangle-free four-coloring exists (**U**)

	11	12	13	14	15	16	17	18	19	20	21	22
11	C	C	C	C	C	C	C	C	C	C	C	N
12	C	C	C	C	C	C	C	C	C	C	**U**	N
13	C	C	C	C	C	C	C	C	C	C	N	N
14	C	C	C	C	C	C	C	C	C	C	N	N
15	C	C	C	C	C	C	C	C	C	C	N	N
16	C	C	C	C	C	C	C	C	C	C	N	N
17	C	C	C	C	C	C	**U**	**U**	N	N	N	N
18	C	C	C	C	C	C	**U**	**U**	N	N	N	N
19	C	C	C	C	C	C	N	N	N	N	N	N
20	C	C	C	C	C	C	N	N	N	N	N	N
21	C	**U**	N	N	N	N	N	N	N	N	N	N
22	N	N	N	N	N	N	N	N	N	N	N	N

- several four-colored rectangle-free grids $G_{17,18}$ by removing any row
- several four-colored rectangle-free grids $G_{18,17}$ by removing any column, or
- several four-colored rectangle-free grids $G_{17,17}$ by removing of both any row and any column.

Hence, a four-colored rectangle-free grid $G_{18,18}$ will solve the coloring problem for four of the open grid sizes. Subsections 2.2, 2.3, and 2.4 describe our steps to find solutions for this problem.

The other two unsolved grids are $G_{12,21}$ and $G_{21,12}$. A solution for one of these grids is after a rotation by 90 degrees also a solution of the other grid. The properties of this grid require other steps to find a solution. Section 2.5 extends our basic results of [293] and shows, how special properties could be utilized to find also four-colored rectangle-free grids $G_{12,21}$.

2.1.3. Multiple-valued Model

The four colors can be represented by the four values $\{1, 2, 3, 4\}$. The value of the grid in the row r and the column c can be modeled by the four-valued variable $x_{r,c}$. One rectangle of the grid is selected by the rows r_i and r_j and by the columns c_k and c_l. The color condition for the rectangles can be described using the following three operations:

1. *equal (multiple-valued):*

$$x \equiv y = \begin{cases} 1 & \text{if } x \text{ is equal to } y \\ 0 & \text{otherwise} \end{cases} , \tag{2.1}$$

2. *and (conjunction, Boolean):*

$$x \wedge y = \begin{cases} 1 & \text{if both } x \text{ and } y \text{ are equal to 1} \\ 0 & \text{otherwise} \end{cases} , \tag{2.2}$$

3. *or (disjunction, Boolean):*

$$x \vee y = \begin{cases} 0 & \text{if both } x \text{ and } y \text{ are equal to 0} \\ 1 & \text{otherwise} \end{cases} . \tag{2.3}$$

The function $f_{ec}(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l})$ (2.4) depends on four four-valued variables and has a Boolean result that is true if the colors in the corners of the rectangle selected by the rows r_i and r_j and by the columns c_k and c_l are equal to each other.

$$\begin{aligned} &f_{ec}(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l}) = \\ &\quad ((x_{r_i,c_k} \equiv 1) \wedge (x_{r_i,c_l} \equiv 1) \wedge (x_{r_j,c_k} \equiv 1) \wedge (x_{r_j,c_l} \equiv 1)) \vee \\ &\quad ((x_{r_i,c_k} \equiv 2) \wedge (x_{r_i,c_l} \equiv 2) \wedge (x_{r_j,c_k} \equiv 2) \wedge (x_{r_j,c_l} \equiv 2)) \vee \\ &\quad ((x_{r_i,c_k} \equiv 3) \wedge (x_{r_i,c_l} \equiv 3) \wedge (x_{r_j,c_k} \equiv 3) \wedge (x_{r_j,c_l} \equiv 3)) \vee \\ &\quad ((x_{r_i,c_k} \equiv 4) \wedge (x_{r_i,c_l} \equiv 4) \wedge (x_{r_j,c_k} \equiv 4) \wedge (x_{r_j,c_l} \equiv 4)) \end{aligned} \tag{2.4}$$

The condition that in the four corners of the rectangle selected by the rows r_i and r_j and by the columns c_k and c_l not only one of the four colors $\{1, 2, 3, 4\}$ will appear is

$$f_{ec}(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l}) = 0 . \tag{2.5}$$

Table 2.2. Encoding of four colors x by two Boolean variables a and b

x	a	b
1	0	0
2	1	0
3	0	1
4	1	1

For the whole grid $G_{m,n}$ we have the four-valued rectangle condition:

$$\bigvee_{i=1}^{m-1} \bigvee_{j=i+1}^{m} \bigvee_{k=1}^{n-1} \bigvee_{l=k+1}^{n} f_{ec}(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l}) = 0 \ . \tag{2.6}$$

2.1.4. Boolean Model

The four colors can be represented by the four values $\{1, 2, 3, 4\}$. Two Boolean variables are needed to encode these four values. Table 2.2 shows the used mapping.

Using the encoding of Table 2.2, a complete model of the problem within the Boolean domain can be created. The function (2.7) depends on eight Boolean variables and has a Boolean result that is true if the color is the same in the four corners of the rectangle selected by the rows r_i and r_j and by the columns c_k and c_l.

$$\begin{aligned} f_{ecb}(a_{r_i,c_k}, b_{r_i,c_k}, a_{r_i,c_l}, b_{r_i,c_l}, a_{r_j,c_k}, b_{r_j,c_k}, a_{r_j,c_l}, b_{r_j,c_l}) = \\ (\overline{a}_{r_i,c_k} \wedge \overline{b}_{r_i,c_k} \wedge \overline{a}_{r_i,c_l} \wedge \overline{b}_{r_i,c_l} \wedge \overline{a}_{r_j,c_k} \wedge \overline{b}_{r_j,c_k} \wedge \overline{a}_{r_j,c_l} \wedge \overline{b}_{r_j,c_l}) \vee \\ (a_{r_i,c_k} \wedge \overline{b}_{r_i,c_k} \wedge a_{r_i,c_l} \wedge \overline{b}_{r_i,c_l} \wedge a_{r_j,c_k} \wedge \overline{b}_{r_j,c_k} \wedge a_{r_j,c_l} \wedge \overline{b}_{r_j,c_l}) \vee \\ (\overline{a}_{r_i,c_k} \wedge b_{r_i,c_k} \wedge \overline{a}_{r_i,c_l} \wedge b_{r_i,c_l} \wedge \overline{a}_{r_j,c_k} \wedge b_{r_j,c_k} \wedge \overline{a}_{r_j,c_l} \wedge b_{r_j,c_l}) \vee \\ (a_{r_i,c_k} \wedge b_{r_i,c_k} \wedge a_{r_i,c_l} \wedge b_{r_i,c_l} \wedge a_{r_j,c_k} \wedge b_{r_j,c_k} \wedge a_{r_j,c_l} \wedge b_{r_j,c_l}) \end{aligned} \tag{2.7}$$

The first conjunction of (2.7) describes the case that all four corners of the selected rectangle carry the color red ($x = 1$). Equivalent properties for the colors green ($x = 2$), blue ($x = 3$), and yellow

($x = 4$) are specified in the rows 2, 3, and 4 of the function f_{ecb} (2.7), respectively.

The index *ebc* of the function f_{ecb} (2.7) means: *Equal Colors, Binary encoded*. The rectangle-free condition of a grid $G_{m,n}$ for a single rectangle of the rows r_i, r_j and the columns c_k, c_l for all four colors is:

$$f_{ecb}(a_{r_i,c_k}, b_{r_i,c_k}, a_{r_i,c_l}, b_{r_i,c_l}, a_{r_j,c_k}, b_{r_j,c_k}, a_{r_j,c_l}, b_{r_j,c_l}) = 0 \ . \quad (2.8)$$

All conditions of the four-color problem of a grid $G_{m,n}$ are achieved when the function f_{ecb} (2.7) is equal to 0 for all rectangles which can be expressed by

$$\bigvee_{i=1}^{m-1} \bigvee_{j=i+1}^{m} \bigvee_{k=1}^{n-1} \bigvee_{l=k+1}^{n} f_{ecb}(a_{r_i,c_k}, b_{r_i,c_k}, a_{r_i,c_l}, b_{r_i,c_l}, a_{r_j,c_k}, b_{r_j,c_k}, a_{r_j,c_l}, b_{r_j,c_l}) = 0 \ . \quad (2.9)$$

2.1.5. Estimation of the Complexity

The four colors of the grid element in the row r_i and the column c_k are specified by the values of Boolean variables a_{r_i,c_k} and b_{r_i,c_k}. Due to these two Boolean variables of a single grid element, the number m of rows r_i, and the number n of columns c_k, the overall number of Boolean variables required to encode all different four-colored grids $G_{m,n}$ is equal to $2 * m * n$. Hence, the numbers of different color patterns are

$$n_{cp}(m, n) = 2^{2*m*n} \quad (2.10)$$

for a grid $G_{m,n}$ that is completely colored by four colors. Hence, the numbers of different four-colored grids for the studied sizes are

$$G_{12,21}: \quad n_{cp}(12, 21) = 2^{2*12*21} = 2^{504} = 5.23742 * 10^{151} \ , \quad (2.11)$$

$$G_{18,18}: \quad n_{cp}(18, 18) = 2^{2*18*18} = 2^{648} = 1.16798 * 10^{195} \ . \quad (2.12)$$

Obviously, these are extremely large numbers. Therefore the generation of all color patterns for the grids $G_{12,21}$ and $G_{18,18}$ is not an option for the solution of these problems, also including the fact that the grid $G_{18,18}$ is 10^{43}-times more complex than the grid $G_{12,21}$.

The problem is even more difficult due to the number of rectangles which must be evaluated for each color pattern. The number of all possible rectangles depends on the number of rows m and the number of columns n of a grid $G_{m,n}$. Each pair of rows and each pair of columns generates one possible rectangle. Hence,

$$n_r(m,n) = \binom{m}{2} * \binom{n}{2} = \frac{m(m-1)}{2} * \frac{n(n-1)}{2} \tag{2.13}$$

rectangles exist for a grid $G_{m,n}$. The number of rectangles which must be evaluated for the studied grids are:

$$G_{12,21}: \quad n_r(12,21) = \binom{12}{2} * \binom{21}{2} = 66 * 210 = 13,860 \ , \tag{2.14}$$

$$G_{18,18}: \quad n_r(18,18) = \binom{18}{2} * \binom{18}{2} = 153 * 153 = 23,409 \ . \tag{2.15}$$

The number of rectangles does not depend on the number of used colors so that $n_r(m,n)$ for the studied four-valued grids (2.13) is the same as for the Boolean rectangle problem (1.6).

The number of all rectangles n_{ar}^{4c} for all four-colored grids of a certain size is equal to

$$n_{ar}^{4c}(m,n) = \binom{m}{2} * \binom{n}{2} * 2^{2*m*n} \ . \tag{2.16}$$

The value of $n_{ar}(m,n)$ characterizes how many subtasks must be solved in order to find all color patterns which satisfy the rectangle condition (2.9) for the grid $G_{m,n}$. Hence, $n_{ar}(m,n)$ is a measure for the complexity of the task to be solved. The number of all rectangles which must be evaluated for all different grid patterns of the studied grids are:

$$G_{12,21}: \quad n_{ar}^{4c}(12,21) = 66 * 210 * 2^{504} = 7.2591 * 10^{155} \ , \tag{2.17}$$

$$G_{18,18}: \quad n_{ar}^{4c}(18,18) = 153 * 153 * 2^{648} = 2.7341 * 10^{199} \ . \tag{2.18}$$

The number of rectangles $n_r(m,n)$ of a grid $G_{m,n}$ does not depend on the number of used colors. Hence, $n_r(m,n)$ for the studied four-valued grids (2.13) is the same as for the Boolean Rectangle Problem (1.6).

However, the number of all rectangles $n_{ar}(m, n)$ depends on the number of colors. The value 2 in the exponent of (2.10) indicates the number of Boolean variables to encode the four colors. Therefore, the number of all rectangles which must be evaluated for all four-colored grids is $n_{gp} = 2^{m*n}$ times larger than the number of all rectangles for a grid of the same size that contains Boolean values.

2.2. Basic Approaches and Results

Bernd Steinbach Christian Posthoff

2.2.1. Solving Boolean Equations

In order to make progress, we investigated more properties of the problem. The details of our explorations have been published in [295]. Here we summarize the main results.

The Boolean equation (2.9) can be completely solved for small grid sizes. The representation by ternary vectors of XBOOLE [240], [302] and [298] helps to restrict the required memory.

Table 2.3 shows the detailed results for grids of $m = 2, \ldots, 7$ rows, and $n = 2$ columns. The column labeled by v gives the number of Boolean variables of the Boolean equation (2.9). The next column enumerates the number of *ternary vectors* required to express the *correct solutions* given in the fifth column. The benefit of the ternary representation is obvious. It is easy to calculate all solutions, because XBOOLE uses orthogonal ternary vectors.

The difference between the number of all possible color patterns n_{cp} (2.10) and the number of correct solutions is equal to the number of *incorrect patterns*. The column *incorrect fraction* in Table 2.3 gives the percentage of these incorrect patterns.

For the simplest grid $G_{2,2}$ almost all of the $n_{cp} = 2^8 = 256$ color patterns are correct solutions. Only the four patterns specified by the function (2.7) are incorrect. The incorrect fraction of rectangle-free four-color patterns of $G_{2,2}$ is equal to 1.56%. This percentage grows to more than 25% for the grid $G_{7,2}$. From this observation we learn that there can be a four-colored rectangle-free grid $G_{m,n}$ with the property that at least one of the grids $G_{m+1,n}$ or $G_{m,n+1}$ is not four-colorable in a rectangle-free way. This border is achieved for a four-colored rectangle-free grid $G_{m,n}$ with a fraction of incorrect patterns of 0.75. The practical results of Table 2.3 confirm the theory of [98].

Table 2.3. Solutions of the Boolean equation (2.9) for m rows and n columns calculated with XBOOLE

			number of					
m	n	n_v	ternary vectors	correct solutions	incorrect patterns	incorrect fraction	memory in KB	time in seconds
2	2	8	24	252	4	1.56 %	6	0.002
3	2	12	304	3,912	184	4.49 %	61	0.004
4	2	16	3,416	59,928	5,608	8.56 %	675	0.006
5	2	20	36,736	906,912	141,664	13.51 %	7,248	0.029
6	2	24	387,840	13,571,712	3,205,504	19.11 %	76,509	0.379
7	2	28	4,061,824	201,014,784	67,420,672	25.12 %	801,259	4.383

2.2.2. Utilization of Permutations

The limit in the previous approach was the required memory of about 800 Megabytes to represent the correct color patterns of the grid $G_{7,2}$ which could be calculated in less than 5 seconds. To break the limitation of memory requirements we exploited some heuristic properties of the problem:

1. Knowing one single correct solution of the four-color problem, $4! = 24$ permutations of this solution with regard to the four colors are also correct solutions.

2. The permutation of rows and columns of a given correct solution pattern creates another pattern that is a correct solution, too.

3. A nearly uniform distribution of the colors in both the rows and the columns increases the fraction of four-colored rectangle-free grids.

Hence, in [295] we confined ourselves to the calculations of solutions with a *fixed uniform distribution of the colors in the top row and in the left column*. We restricted the calculation in this experiment to 12 columns and 2 Gigabytes of available memory.

Table 2.4. Selected solutions of the Boolean equation (2.9) with fixed uniform distribution of the colors in the top row and in the left column

			number of		memory	time in
m	n	n_v	ternary vectors	correct solutions	in KB	seconds
2	2	8	1	4	1	0.000
2	12	48	6,912	2,361,960	1,365	0.023
3	2	12	1	16	1	0.002
3	8	48	4,616,388	4,616,388	424,679	7.844
4	2	16	1	64	1	0.002
4	5	40	674,264	12,870,096	113,135	1.000
5	2	20	1	256	1	0.002
5	4	40	573,508	12,870,096	133,010	0.824
6	2	24	4	960	2	0.003
6	3	36	15,928	797,544	11,367	0,020
7	2	28	16	3,600	4	0.004
7	3	42	183,152	104,93,136	95,565	0.314
8	2	32	64	3,600	14	0.005
8	3	48	2,152,819	136,603,152	910,656	4.457
9	2	36	256	50,625	52	0.007
10	2	40	768	182,250	153	0.008
15	2	60	147,456	104,162,436	29,090	0.386
19	2	76	6,553,600	14,999,390,784	1,292,802	21.378

Table 2.4 shows some results of this restricted calculation selected from a table given in [295]. Figure 2.2 depicts the four selected four-colored rectangle-free grids of the first row in Table 2.4. Using the same memory size, the number of Boolean variables could be enlarged from $n_v = 28$ for $G_{7,2}$ to $n_v = 76$ for $G_{19,2}$. That means, by utilizing properties of the four-color problem mentioned above, we have solved problems which are $2^{48} = 281,474,976,710,656$ times larger than before, but again the available memory size restricts the solution of larger four-colored rectangle-free grids.

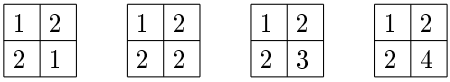


Figure 2.2. Four-colored grids $G_{2,2}$ of the first row in Table 2.4.

2.2.3. Exchange of Space and Time

The function f_{ecb} (2.7) describes the incorrect patterns of a single rectangle. The number of rectangles $n_r(m,n)$ (2.13) of a grid $G_{m,n}$ is specified by all possible pairs of rows and all possible pairs of columns. This number controls the sweeps of the loop in algorithms which evaluate these rectangles. Color patterns which violate the rectangle-free condition (2.8) for a single rectangle must be excluded from all patterns of the Boolean space B^{2*m*n} which can be calculated using the `DIF`-operation of XBOOLE [240], [302], and [298].

The main data structure of XBOOLE is an orthogonal list of ternary vectors (TVL). Each ternary vector consists of $2*m*n$ elements '0', '1', and '–' for the Boolean space B^{2*m*n}. Two different binary vectors which are equal to each other in $(2*m*n)-1$ positions can be merged into a single ternary vector that contains a dash element in the position of given different binary values. A ternary vector of d dash elements expresses 2^d binary vectors. Hence, a TVL can be used as a compact representation of a set of binary vectors. The benefit in terms of required memory for the TVLs can be seen in Table 2.3. The XBOOLE-operation `C=DIF(A,B)` calculates the *set difference* $C = A \setminus B = A \cap \neg B$ for the given TVLs `A` and `B` such that as much as possible dash elements of `A` remain in the result `C`.

We use the DIF-operation of XBOOLE to solve the four-color problem of grids. We assume that `aps` is the *actual partial solution* which is initialized by the whole Boolean space B^{2*m*n}. Algorithm 2.1 finds all rectangle-free four-color patterns when unrestricted memory space can be used.

Algorithm 2.1 RF4CG with unrestricted space requirements

Require: n_r the number of rectangles of a grid
Require: $f_{ecb}[i]$ Boolean rectangle condition for four colors
Ensure: all four-colored rectangle-free patterns of the evaluated grid

```
1: aps ← ∅
2: aps ← CPL(all)                          ▷ complement
3: for i ← 1 to n_r do
4:     aps ← DIF(aps, f_ecb[i])            ▷ difference
5: end for
```

It is an important drawback of Algorithm 2.1 that the size of *aps* typically extremely increases up to a certain index i and decreases later on. Therefore, we implemented an algorithm that exchanges space against time.

The partial solution vectors of *aps* are logically connected by disjunctions. Hence, *aps* can be split by (2.19) into two parts:

$$aps = aps_0 \vee aps_1 \ . \tag{2.19}$$

The main idea is to solve the problem sequentially for aps_0 and aps_1, and combine both solutions at the end. If necessary, this approach can be recursively applied. The decision about the split of *aps* can be controlled by the size of *aps* itself. Hence, it is possible to restrict the used memory space to an appropriate size. The restriction of the memory size has to be compensated by a certain amount of computation time for the administration of the split and merge operations.

There are several implementations of the suggested approach. One of them uses one stack *aps_stack* to store aps_1 and another stack *position_stack* to store the value of the index i that belongs to the executed split. Two stacks with the check operation EMPTY(), and the access operations PUSH() and POP() are used as storage for the intermediate results. The function $(aps_0, aps_1) \leftarrow \text{split}(aps)$ splits the TVL *aps* into aps_0 and aps_1 having approximately the same size.

Algorithm 2.2 uses the value of *SplitLimit* to decide which TVL *aps* must be split into two parts. The `NTV`-operation of XBOOLE returns the *number of ternary vectors* of a given TVL. This operation is used in line 5 of Algorithm 2.2 to get the number of ternary vectors of *aps* as basis of the decision about the execution of a split. Decisions depending on the the number of ternary vectors of *aps*, the content of the stack, and the index of the loop are used in lines 10 to 27 to control the required actions.

Both Algorithm 2.1 and Algorithm 2.2 solve the same task to find all four-colored rectangle-free patterns of the evaluated grid using the same basic approach as shown in lines 3 and 4 of both algorithms. Algorithm 2.1 fails to solve the problem if the program runs out of memory. Such a break can not occur in Algorithm 2.2. Controlled by the value of *SplitLimit* only disjoint subsets of all solutions are

Algorithm 2.2 RF4CG with restricted space requirements

Require: n_r number of rectangles of a grid
Require: $f_{ecb}[i]$ Boolean rectangles conditions for four colors
Require: *SplitLimit* number of ternary vectors which causes a split of *aps*
Ensure: all four-colored rectangle-free patterns of the evaluated grid

1: $aps \leftarrow \emptyset$
2: $aps \leftarrow \texttt{CPL}(all)$ ▷ complement
3: **for** $i \leftarrow 1$ to n_r **do**
4: $aps \leftarrow \texttt{DIF}(aps, f_{ecb}[i])$ ▷ difference
5: **if** $\texttt{NTV}(aps) > SplitLimit$ **then**
6: $(aps, aps_1) \leftarrow \text{split}(aps)$
7: aps_stack.PUSH(aps_1)
8: $position\_stack$.PUSH(i)
9: **end if**
10: **if** $\texttt{NTV}(aps) = 0$ **then** ▷ number of ternary vectors
11: **if** aps_stack.EMPTY() **then**
12: **return** *aps* ▷ no solution
13: **else**
14: $aps \leftarrow aps\_stack$.POP()
15: $i \leftarrow position\_stack$.POP()
16: **end if**
17: **else**
18: **if** $i = n_r$ **then**
19: appendToFileOfSolutions(*aps*) ▷ solutions
20: **end if**
21: **if** aps_stack.EMPTY() **then**
22: **return** *aps* ▷ solutions
23: **else**
24: $aps \leftarrow aps\_stack$.POP()
25: $i \leftarrow position\_stack$.POP()
26: **end if**
27: **end if**
28: **end for**

calculated during each interval of time. The function

appendToFileOfSolutions(*aps*)

appends found solutions to an external file.

Table 2.5. Four-colored rectangle-free grid patterns using Algorithm 2.2 canceled after the calculation of the first correct solution

			number of		maximal	time in
m	n	n_v	ternary vectors	correct solutions	stack size	seconds
12	2	48	337	6,620	3	0.011
12	3	72	147	2,423	22	0.029
12	4	96	319	7,386	30	0.056
12	5	120	236	1,188	47	0.095
12	6	144	181	1,040	61	0.147
12	7	168	231	627	69	0.216
12	8	192	109	413	81	0.287
12	9	216	72	227	79	0.398
12	10	240	34	103	87	0.516
12	11	264	40	109	88	0.645
12	12	288	112	293	82	0.806
12	13	312	51	81	81	1.054
12	14	336	82	1,415	97	1.290
12	15	360	1	1	80	2.361
12	16	384	2	3	81	426.903

We applied Algorithm 2.2 to grids of 12 rows, used a fixed value $SplitLimit = 400$, and cancelled the calculation when the first solutions were found. Table 2.5 shows the results.

Exchanging space against time allows us again an extreme improvement: solutions for four-colored rectangle-free grids which are modeled with up to 384 Boolean variables were found instead of 76 variables in the second (already improved) approach. This means that the approach of exchanging space and time for the four-colored rectangle-free grids allows the solution of problems which are

$$2^{308} = 5.21481 * 10^{92}$$

times larger than before. An approach to reduce the required time is given by parallel computing [296].

2.3. Power and Limits of SAT-Solvers

Bernd Steinbach Christian Posthoff

2.3.1. Direct Solutions for Four-colored Grids

It is the aim of the satisfiability problem (short SAT) to find at least one assignment of Boolean variables such that a Boolean expression in conjunctive form [290] becomes true. We tried to find four-colored rectangle-free grid patterns using the best SAT-solvers from the SAT-competitions of the last years. The equation (2.9) can easily be transformed into a SAT-equation by negation of both sides and the application of De Morgan's Law to the Boolean expression on the left-hand side. In this way we get the required conjunctive form (2.20) for the SAT-solver.

$$\bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} \bigwedge_{k=1}^{n-1} \bigwedge_{l=k+1}^{n} \overline{f_{ecb}(a_{r_i,c_k}, b_{r_i,c_k}, \overline{a_{r_i,c_l}, b_{r_i,c_l}, a_{r_j,c_k}, b_{r_j,c_k}, a_{r_j,c_l}, b_{r_j,c_l}})} = 0 \ . \tag{2.20}$$

Table 2.6 shows the required time to find the first rectangle-free solution for the quadratic four-colored grids $G_{12,12}$, $G_{13,13}$, $G_{14,14}$, and $G_{15,15}$ using the SAT-solvers *clasp* [114], *lingeling* [31], *plingeling* [31], and *precosat* [31].

The power of SAT-solvers becomes visible by comparing the successfully solved grid sizes. The largest grid $G_{12,16}$ that has been solved with the method of Subsection 2.2.3 by exchanging space and time depends on 308 Boolean variables. All four SAT-solvers found a solution for the grid $G_{15,15}$ which needs 450 Boolean variables. Hence, the number of additional variables is $450 - 308 = 142$; the successfully evaluated search space is increased by the factor of $2^{142} = 5.575 * 10^{42}$, a very strong improvement. One source of this improvement is that SAT-solvers are focused to find a single assignment to the Boolean variables that satisfies the given equation while the aim of our previous

Table 2.6. Time to solve quadratic four-colored grids using different SAT-solvers

number of			time in minutes:seconds.milliseconds			
m	n	n_v	clasp	lingeling	plingeling	precosat
12	12	288	0:00.196	0:00.900	0:00.990	0:00.368
13	13	338	0:00.326	0:01.335	0:04.642	0:00.578
14	14	392	0:00.559	0:03.940	0:02.073	0:00.578
15	15	450	46:30.716	54:02.304	73:05.210	120:51.739

approaches is the calculation of all solutions. Only some SAT-solvers are able to calculate iteratively several or even all solutions.

From the utilization of the SAT-solvers we learned that

1. SAT-solvers are powerful tools which are able to calculate a four-colored rectangle-free grid up to $G_{15,15}$,

2. it was not possible to calculate a four-colored rectangle-free grid larger than $G_{15,15}$ within a reasonable period of time.

The reasons for the second statement are in the first place that the search space for the four-colored rectangle-free grid $G_{16,16}$ is already $2^{62} = 4.61 * 10^{18}$ times larger than the search space for the four-colored rectangle-free grid $G_{15,15}$, and secondly that the fraction of four-colored rectangle-free grids is even more reduced for the larger grid.

Figure 2.3 shows the four-colored grid $G_{15,15}$ found by the SAT-solver *clasp* within 46.5 minutes.

2.3.2. Restriction to a Single Color

Taking into account on the one hand both the power and the limit of SAT-solvers, and on the other hand the strong complexity of four-colored rectangle-free grids, a divide-and-conquer approach may facili-

2	4	1	3	1	2	4	3	4	4	1	3	2	1	3
1	2	3	1	1	2	3	2	3	4	4	4	4	2	3
4	2	1	2	2	1	1	3	2	3	3	4	3	4	4
3	4	3	4	3	4	1	4	1	3	1	4	1	2	2
1	3	4	4	2	3	1	2	4	2	4	2	3	1	2
2	2	3	3	4	1	2	4	4	1	1	2	3	3	4
4	3	2	2	1	4	2	2	4	1	3	1	4	3	1
3	1	2	3	4	4	1	1	3	4	3	2	2	4	1
1	4	4	3	2	3	2	1	1	3	2	1	4	2	4
4	3	4	3	3	2	3	4	2	4	2	2	1	1	1
4	4	3	1	4	3	2	1	2	1	3	3	2	1	2
3	2	1	1	4	3	3	3	1	2	2	4	2	3	1
2	1	2	1	3	3	4	4	2	2	1	1	4	2	3
1	1	1	2	3	4	4	3	3	1	4	2	1	2	4
2	3	4	2	1	1	4	1	3	3	2	3	1	4	2

Figure 2.3. Four-colored rectangle-free grid $G_{15,15}$ found by the SAT-solver *clasp-1.2.0* in about 46.5 minutes.

tate the solution of the four-colored grid $G_{17,17}$ or even the grid $G_{18,18}$. The divide step restricts to single color. At least one fourth of the grid positions must be covered by the first color without contradiction to the rectangle-free condition. When such a partial solution is known, the same fill-up step must be executed for the second color taking into account the already fixed positions of the grid. This procedure must be repeated for the remaining two colors.

The advantage of this approach is that a single Boolean variable describes whether the color is assigned to a grid position or not. The function f_{ecb} (2.7) which describes equal colors in the corners of a rectangle can be simplified to f^s_{ecb} (2.21) for a single color in the corners of the row r_i and r_j and the columns c_k and c_l in the divide-and-conquer approach.

$$f^s_{ecb}(a_{r_i,c_k}, a_{r_i,c_l}, a_{r_j,c_k}, a_{r_j,c_l}) = a_{r_i,c_k} \wedge a_{r_i,c_l} \wedge a_{r_j,c_k} \wedge a_{r_j,c_l} \quad (2.21)$$

The index *ebc* of the function f_{ecb} (2.7) means as before: *Equal Colors, Binary encoded*, and the letter s indicates that the function describes

the condition only for a *single* color. The rectangle-free conditions for a single color of a grid $G_{m,n}$ are satisfied when the function f^s_{ecb} (2.21) is equal to 0 for all rectangles which can be expressed by:

$$\bigvee_{i=1}^{m-1} \bigvee_{j=i+1}^{m} \bigvee_{k=1}^{n-1} \bigvee_{l=k+1}^{n} f^s_{ecb}(a_{r_i,c_k}, a_{r_i,c_l}, a_{r_j,c_k}, a_{r_j,c_l}) = 0 \ . \qquad (2.22)$$

Equation (2.22) depends on $18 * 18 = 324$ Boolean variables for the grid $G_{18,18}$. Knowing that the four-colored grid $G_{13,13}$ depends on $13 * 13 * 2 = 338$ and that a four-colored rectangle-free pattern could be calculated in about 1 second by several SAT-solvers, it may be expected that the 1-color rectangle problem for the grid $G_{18,18}$ can be easily solved by each SAT-solver. Using De Morgan's Law, equation (2.22) can be converted into the SAT-formula:

$$\bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} \bigwedge_{k=1}^{n-1} \bigwedge_{l=k+1}^{n} \overline{f^s_{ecb}(a_{r_i,c_k}, a_{r_i,c_l}, a_{r_j,c_k}, a_{r_j,c_l})} = 1 \ . \qquad (2.23)$$

Most of the SAT-solvers calculate only a single solution. One possible solution of (2.23) is the assignment of values 0 to all variables of this equation. Exactly this solution was immediately found by the SAT-solver for the grid $G_{18,18}$ in our experiment. Of course, in the case that no value 1 is assigned to the grid, the first color does not violate the rectangle condition. However, we are need solutions in which a maximal number of variables is equal to 1.

There is only one possibility to eliminate this restriction of SAT-solvers. Some of the SAT-solvers are able to calculate several or even all solutions of a given SAT-equation. One of them is *clasp* [114]. The needed patterns for one color with 0.25 % or more values 1 can be selected by counting the values 1 in the found solutions of the SAT-solver. However, both the created extremely large output file of the SAT-solver and the required time for counting the assigned values 1 are a strong drawback of this approach.

The function f^s_{ecb} (2.21) is monotonously rising. Hence, if there is a needed solution with 81 assignments of the first color for the grid

$G_{18,18}$ then for each of these solutions all

$$\sum_{i=0}^{80} \binom{324}{i} = 3.36329 * 10^{77}$$

color patterns are also solutions of the SAT-equation. Hence, such a huge amount of waste must be calculated first by the SAT-solver and thereafter eliminated by a time-consuming counting procedure. Hence, we must conclude that the SAT-approach is *not* suitable to solve the assignment problem of a single color to the large rectangle-free grid $G_{18,18}$.

2.4. Cyclic Color Assignments of Four-Colored Grids

Bernd Steinbach Christian Posthoff

2.4.1. Sequential Assignment of a Single Color

As mentioned in Subsection 2.3.2, a divide-and-conquer approach may facilitate the solution of the four-colored grid $G_{17,17}$ or even the grid $G_{18,18}$ because such an approach reduces the complexity. The divide step restricts us to a single color. At least one fourth of the grid positions must be covered by the first color without contradiction to the rectangle-free condition. Based on such a partial solution, the same fill-up step must be executed for the next color taking into account the already fixed positions of the grid. This procedure must be repeated for all four colors.

The advantage of this approach is that a single Boolean variable describes the fact whether the color is assigned to a grid position or not. Such a restriction to one half of the needed Boolean variables drastically reduces the search space from $2^{2*18*18} = 1.16 * 10^{195}$ to $2^{18*18} = 3.41 * 10^{97}$ for the grid $G_{18,18}$.

The function f^s_{ecb} (2.21) which describes equal colors with Boolean variables for a single color in the corners of the row r_i and r_j and the columns c_k and c_l is given in Subsection 2.3.2. This function is reused n_r times in (2.22) to describe the rectangle-free condition for all rectangles of a grid. Unfortunately, the associated SAT-equation (2.23) cannot be solved in a reasonable period of time.

Using the soft-computing approach that combines the iterative greedy approach of Subsection 1.5.3 and the utilization of permutations by direct mapping to representatives of Subsection 1.5.5 we found as important experimental result of Subsection 1.5.6 one permutation class for the Grid $G_{18,18}$ that contains the needed 0.25 % of assignments of the first color. Figure 2.4 shows the representative of the found

single color solution of the grid $G_{18,18}$ which contains 81 of 324 color 1 tokens.

1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0
1	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0
0	1	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0
0	1	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0
0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0	0	0	1	0	0	0	1	0
0	0	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	1
0	0	0	0	1	1	0	1	0	0	1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1
0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1
0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1	0
0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0
0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0
0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0

Figure 2.4. Rectangle-free grid $G_{18,18}$ where one fourth of all positions is colored with a single color.

As next step of the divide-and-conquer approach the color pattern of Figure 2.4 must be extended by 81 assignments of the color 2. The 81 already assigned tokens of the color 1 simplifies this task because the number of free Boolean variables is $324 - 81 = 243$ for the grid $G_{18,18}$. However, this simplification is not sufficient for the application of the SAT-solver in a reasonable period of time, because

$$\sum_{i=0}^{80} \binom{243}{i} = 7.72567 * 10^{65}$$

unwanted solutions must be calculated and later on excluded.

Alternatively the soft-computing approach that combines the iterative

r_1	s_1	t_1	r_2
t_4	u_1	u_2	s_2
s_4	u_4	u_3	t_2
r_4	t_3	s_3	r_3

Figure 2.5. Cyclic quadruples in the grid $G_{4,4}$.

greedy approach of Subsection 1.5.3 and the utilization of permutations by direct mapping to representatives of Subsection 1.5.5 can be adapted to the modified problem. However, our effort to fill up the grid $G_{18,18}$ of Figure 2.4 with the second color again in 81 grid positions failed. This results from the fact that the freedom for the choice of the positions is restricted by the assignments of the first color. We learned from this approach that it is not enough to know a correct coloring for one color; these assignments must not constrain the assignment of the other colors.

2.4.2. Reusable Assignment for Four Colors

The smallest restrictions for the rectangle-free coloring of a grid by four colors are given when the number of assignments to the grid positions is equal for all four colors. For quadratic grids $G_{m,n}, m = n$, with an even number of rows m and columns n, quadruples of all grid positions can be chosen which contain all four colors. There are several possibilities of such selections of quadruples. One of them is the cyclic rotation of a chosen grid position by 90 degrees around the center of the grid. Figure 2.5 illustrates this possibility for a simple grid $G_{4,4}$. The quadruples are labeled by the letters r, s, t, and u. The attached index specifies the element of the quadruple.

In addition to the rectangle-free condition (2.23) for the chosen single color we can require that this color occurs exactly once in each quadruple. This property can be expressed by two additional rules. For the corners of the grid of Figure 2.5, for instance, we model as first rule the requirement

$$f_{requ}(r_1, r_2, r_3, r_4) = 1 \ , \tag{2.24}$$

with

$$f_{requ}(r_1, r_2, r_3, r_4) = r_1 \vee r_2 \vee r_3 \vee r_4 \ , \tag{2.25}$$

so that at least one variable r_i must be equal to 1. As second rule, the additional restriction

$$f_{rest}(r_1, r_2, r_3, r_4) = 0 \ , \tag{2.26}$$

with

$$\begin{aligned} f_{rest}(r_1, r_2, r_3, r_4) = {} & (r_1 \wedge r_2) \vee (r_1 \wedge r_3) \vee (r_1 \wedge r_4) \vee \\ & (r_2 \wedge r_3) \vee (r_2 \wedge r_4) \vee (r_3 \wedge r_4) \end{aligned} \tag{2.27}$$

prohibits that more than one variable r_i is equal to 1.

The SAT-equation (2.23) can be extended by these additional rules for all quadruples. The function $f_{requ}(r_1, r_2, r_3, r_4)$ (2.25) on the left-hand side of Equation (2.24) has the required disjunctive form. Hence, this expression can be directly used as clause of the SAT-formula (2.23). However, the function $f_{rest}(r_1, r_2, r_3, r_4)$ (2.27) on the left-hand side of Equation (2.26) is an expression in disjunctive form. The set of solutions of a Boolean equation remains unchanged when the expressions on both sides are negated:

$$\overline{f_{rest}(r_1, r_2, r_3, r_4)} = 1 \ , \tag{2.28}$$

with

$$\begin{aligned} \overline{f_{rest}(r_1, r_2, r_3, r_4)} = {} & (\overline{r}_1 \vee \overline{r}_2) \wedge (\overline{r}_1 \vee \overline{r}_3) \wedge (\overline{r}_1 \vee \overline{r}_4) \wedge \\ & (\overline{r}_2 \vee \overline{r}_3) \wedge (\overline{r}_2 \vee \overline{r}_4) \wedge (\overline{r}_3 \vee \overline{r}_4) \ . \end{aligned} \tag{2.29}$$

De Morgan's Law is used to transform the disjunctive form of (2.27) into the needed conjunctive form of (2.29). If the six disjunctions of (2.29) are added as additional clauses to the SAT-formula (2.23) then only one of the four variables r_i can be equal to 1 in the solutions.

Cyclic reusable rectangle-free color patterns for a quadratic grid $G_{m,n}$ with $m = n$ rows and columns are solutions of the SAT-equation:

$$\begin{aligned} & \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} \bigwedge_{k=1}^{m-1} \bigwedge_{l=k+1}^{m} \overline{f^s_{ecb}(x_{r_i,c_k}, x_{r_i,c_l}, x_{r_j,c_k}, x_{r_j,c_l})} \wedge \\ & \qquad \bigwedge_{quad=1}^{m^2/4} f_{requ}[quad] \wedge \bigwedge_{quad=1}^{m^2/4} f_{rest}[quad] = 1 \ , \end{aligned} \tag{2.30}$$

where the index *quad* indicates the quadruple.

For each quadruple one clause of (2.24) and six clauses for the six disjunctions of (2.28) are added in (2.30) to the clauses of (2.23). The complete SAT-equation (2.30) for this cyclic single color approach contains

$$\begin{aligned} n_{clauses(18,18)} &= \binom{18}{2} * \binom{18}{2} + \frac{18^2}{4} * 7 \\ &= 153 * 153 + \frac{324 * 7}{4} = 23,976 \end{aligned} \tag{2.31}$$

clauses for the grid $G_{18,18}$.

The solution of such a SAT-formula for a quadratic grid of even numbers of rows and columns must assign exactly one fourth of the variables to 1. Such a solution can be used rotated by 90 degrees for the second color, rotated by 180 degrees for the third color, and rotated by 270 degrees for the fourth color without any contradiction.

We generated the cnf-file of this SAT-formula which depends on 324 Boolean variables and contains 23,976 clauses for the grid $G_{18,18}$. The Boolean variables are labeled in the cnf-file by integer numbers. We assigned the 324 Boolean variables row-by-row starting in the top left corner of the grid with the variable 1. Hence, the top right corner of the grid $G_{18,18}$ belongs to the variables 18, the bottom left corner to the variable 307, and bottom right corner to the variable 324. The cnf-format format requires after possible line of comments the line:

```
p cnf 324 23976
```

in which the numbers of varibles and the number of clauses are specified. Each following line contains one of the 23,976 disjunctions (clauses) of the conjunctive form. The end of a clause is indicated by a number 0 at the end of the line. The variables of a clause are separated by a space. A minus-sign in front of the number of a variable indicates a negated variable of the clause.

The first clause

```
-1 -2 -19 -20 0
```

describes the rectangle of the top two rows and the left two columns. If all these four variables are equal to 1 then this disjunction is equal to

```
c clasp version 2.0.0
c Reading from stdin
c Solving...
c Answer:  1
v -1 -2 3 4 -5 -6 7 -8 -9 -10 -11 -12 -13 -14 15 -16 17 -18 -19 -20 -21 -22
v -23 -24 25 -26 -27 -28 29 30 -31 -32 -33 34 -35 36 -37 -38 -39 -40 41 -42
v -43 -44 -45 46 -47 48 -49 50 -51 -52 53 -54 -55 56 -57 58 -59 -60 -61 -62
v 63 -64 -65 66 -67 -68 -69 -70 -71 -72 73 -74 -75 76 -77 78 -79 -80 -81
v -82 -83 -84 -85 86 -87 -88 -89 90 91 92 -93 -94 -95 -96 -97 98 -99 -100
v -101 -102 -103 -104 -105 106 107 -108 109 -110 -111 -112 113 -114 115 -116
v 117 -118 -119 -120 -121 -122 -123 -124 -125 -126 -127 -128 -129 -130 131
v 132 -133 -134 -135 -136 -137 -138 -139 -140 141 142 -143 -144 -145 146
v -147 -148 -149 150 151 -152 -153 154 -155 -156 157 -158 -159 -160 -161
v -162 -163 164 165 -166 167 -168 -169 -170 -171 -172 -173 -174 -175 -176
v -177 -178 -179 180 -181 -182 -183 -184 -185 -186 -187 188 189 190 -191
v -192 -193 -194 195 -196 -197 198 -199 -200 -201 202 203 -204 -205 206 -207
v -208 209 -210 211 -212 -213 -214 -215 -216 217 -218 -219 -220 -221 -222
v -223 -224 -225 -226 -227 228 229 -230 231 -232 -233 -234 -235 -236 -237
v -238 -239 240 -241 -242 243 -244 245 -246 -247 -248 -249 -250 251 -252
v -253 -254 255 -256 -257 -258 -259 -260 261 -262 -263 -264 265 266 -267
v 268 -269 -270 -271 -272 273 -274 -275 276 -277 278 -279 -280 -281 282 -283
v -284 -285 -286 -287 -288 -289 290 -291 -292 -293 -294 -295 -296 -297 -298
v 299 -300 -301 302 303 -304 -305 -306 307 -308 309 -310 -311 -312 -313 -314
v -315 316 317 -318 -319 -320 -321 -322 -323 -324 0
s SATISFIABLE

c Models    :  1+
c Time      :  212301.503s
               (Solving:  212300.96s 1st Model:  212300.96s Unsat:  0.00s)
c CPU Time  :  211867.158s
```

Figure 2.6. Cyclic reusable single color solution of the grid $G_{18,18}$ calculated by the SAT-solver *clasp-2.0.0-st-win32*.

0 and the whole SAT-formula cannot be satisfied. This is the wanted result, because color-1 token form in this case a rectangle in the upper left range of the grid $G_{18,18}$.

We tried to find a solution using the improved version SAT-solver *clasp*. This SAT-solver *clasp-2.0.0-st-win32* found the first cyclic reusable solution for the grid $G_{18,18}$ after 2 days 10 hours 58 minutes 21.503 seconds. Figure 2.6 depicts the output of the SAT-solver.

The solution of Figure 2.6 can converted into grid $G_{18,18}$ of Boolean values. A minus-sign in front of a variable in this solution specifies a value 0 of the associated cell, and otherwise a value 1 must be assigned to the associated cell. Figure 2.7 shows this solution for the first color of the grid $G_{18,18}$.

0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	1	0	1
0	0	0	0	1	0	0	0	0	1	0	1	0	1	0	0	1	0
0	1	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0
1	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
0	1	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0	0	1
0	0	0	1	1	0	0	1	0	0	1	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0
0	0	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

Figure 2.7. Cyclic reusable coloring of grid $G_{18,18}$.

Using the core solution of Figure 2.7 we have constructed the 4-colored grid $G_{18,18}$ of Figure 2.8 by three times rotating around the grid center by 90 degrees each and assigning the next color.

Many other four-colored rectangle-free grids can be created from the solution in Figure 2.8 by permutations of rows, columns, and colors. Several four-colored rectangle-free grids $G_{17,18}$ originate from the four-colored rectangle-free grid $G_{18,18}$ by removing any single row; and by removing any single column, we get four-colored rectangle-free grids $G_{18,17}$. Obviously, several so far unknown four-colored rectangle-free grids $G_{17,17}$ can be selected from the four-colored rectangle-free grid $G_{18,18}$ of Figure 2.8 by removing both any single row and any single column.

The sign '+' in the line '`c Models :  1+`' of Figure 2.6 indicates that there are further cyclic solutions. These solutions can be found by the

2	4	1	1	4	2	1	3	3	4	4	2	2	2	1	3	1	3
4	2	4	3	3	4	1	3	2	2	1	1	2	4	2	1	3	1
2	4	2	2	1	4	3	4	2	1	3	1	3	1	4	3	1	2
4	1	3	1	3	3	2	4	1	3	4	1	4	2	2	3	4	2
1	3	4	1	4	1	2	3	2	3	2	2	3	1	4	2	4	1
1	1	2	3	2	3	3	1	4	2	2	4	4	2	4	1	1	3
1	4	4	4	1	3	1	3	1	2	3	2	4	3	3	4	2	2
3	4	2	3	1	1	2	2	3	3	3	4	2	4	1	1	4	4
3	1	4	2	2	1	1	2	4	1	4	2	1	3	2	3	3	4
2	1	1	4	1	3	4	2	3	2	4	3	3	4	4	2	3	1
2	2	3	3	2	4	2	1	1	1	4	4	3	3	1	4	2	1
4	4	2	1	1	2	4	1	4	3	1	3	1	3	2	2	2	3
1	3	3	2	4	2	2	4	4	2	3	1	1	4	1	4	3	3
3	2	4	2	3	1	4	4	1	4	1	4	3	2	3	2	1	3
4	2	1	4	4	2	3	2	1	3	2	4	1	1	3	1	3	2
4	3	1	2	3	1	3	1	3	4	2	1	2	3	4	4	2	4
3	1	3	4	2	4	3	3	4	4	1	3	2	1	1	2	4	2
1	3	1	3	4	2	2	4	4	1	1	3	4	2	3	3	2	4

Figure 2.8. Cyclic four-colored rectangle-free grid $G_{18,18}$.

SAT-solver. However, it will take a lot of time because the SAT-solver needs already almost two and a half day to find the first solution.

We suggested in [291] a *reduced cyclic model* which allows us to reduce the number of needed Boolean variables for the first color of the grid $G_{18,18}$ form 324 to 162. In the same paper we suggested one more advanced approach which is the *knowledge transfer*.

The knowledge transfer utilizes the property that

1. a quadratic cyclic four-colored rectangle-free grid $G_{m,m}$ contains a grid $G_{m-2,m-2}$ with the same properties that is surrounded by additional row on the top and on the bottom and additional columns on both sides,
2. due to the exponentially grow of the search space, a SAT-solver

Table 2.7. Cyclic reusable rectangle-free grids $G_{18,18}$ colored with 81 tokens of the first color calculated by knowledge transfer from $G_{16,16}$ with 64 tokens satisfying the same properties.

days	$G_{16,16}$	$G_{18,18}$
2	9,900	0
4	20,000	0
6	31,500	4
8	41,700	8
10	51,400	24
20	111,300	85
30	173,900	120
40	238,400	152
50	297,500	180
60	357,200	208

finds solutions of a grid $G_{m-2,m-2}$ much faster than a solution for the grid $G_{m,m}$, and

3. a known solution for the a grid $G_{m-2,m-2}$ can be used to fix a large number of variables in the SAT-equation for the grid $G_{m,m}$ so that the solution for the grid $G_{m,m}$ will be found much faster by a SAT-solver.

We used both advanced approaches in an experiment over 60 days to calculate more cyclic reusable rectangle-free grids $G_{18,18}$ colored with 81 tokens of the first color. Table 2.7 shows the results of this experiment. It can be seen that only a very small fraction of the cyclic reusable rectangle-free grids $G_{16,16}$ can be extended to grids $G_{18,18}$ with the same property.

The approach of cyclic reusable assignments of tokens can be applied to quadratic four-colored grids of an odd number of rows and columns, too. The difference between quadratic girds of an even number of rows and columns and quadratic girds of an odd number of rows and columns is that the number of grid elements is a multiple of four in the even case, but the number of grid elements is equal to $4 * k + 1$ for odd grids. Figure 2.9 shows that the construction of the quadruples can also be used for odd grids.

The four corners of the grid are used as the first quadruple. Further

r_1	s_1	t_1	u_1	r_2
u_4	v_1	w_1	v_2	s_2
t_4	w_4	x_1	w_2	t_2
s_4	v_4	w_3	v_3	u_2
r_4	u_3	t_3	s_3	r_3

Figure 2.9. Cyclic quadruples in the grid $G_{5,5}$.

quadruples use the next position in clockwise direction. The central grid position cannot be assigned to a quadruple of positions. Hence, the rectangle condition must be satisfied for the central position regarding each of the four colors, and the SAT-equation (2.30) must be extended by a clause that consists only of the variable of the central position of the grid. In case of Figure 2.9 this clause is equal to x_1.

We generated SAT-equation for odd grids from $G_{3,3}$ to $G_{17,17}$. The SAT-solver *clasp* found the first cyclic reusable solution for the $G_{15,15}$ in less than 0.6 seconds but could not solve this task for grid $G_{17,17}$ within more than one month. Therefore we adapted the advanced approaches of the reduced cyclic model and the knowledge transfer to odd grids and run an experiment over 60 days with the aim to find cyclic reusable rectangle-free grids $G_{17,17}$ colored with 73 tokens of the first color. Table 2.8 shows the results of this experiment.

In comparison with Table 2.7, it can be seen that the numbers of cyclic reusable rectangle-free grid $G_{15,15}$ which are found in the same period of time are two orders of magnitude larger than the corresponding grids $G_{16,16}$. However, no cyclic reusable rectangle-free grid $G_{17,17}$ was found with this advanced approach within 60 days. Hence, it remains the open problem whether there is a cyclic reusable rectangle-free grid $G_{17,17}$.

Table 2.8. Cyclic reusable rectangle-free grids $G_{17,17}$ colored with 73 tokens of the first color calculated by knowledge transfer form $G_{15,15}$ with 57 tokens showing the same properties.

days	$G_{15,15}$	$G_{17,17}$
2	1,455,000	0
4	2,960,000	0
6	4,435,000	0
8	6,030,000	0
10	7,625,000	0
20	16,150,000	0
30	23,200,000	0
40	29,645,000	0
50	36,600,000	0
60	43,970,000	0

2.5. Four-Colored Rectangle-Free Grids of the Size 12×21

Bernd Steinbach Christian Posthoff

2.5.1. Basic Consideration

The grid $G_{12,21}$ consists of 12 rows and 21 columns. Hence, $12 * 21 = 252$ elements must be colored with tokens of the given four colors. Due to the pigeonhole principle a rectangle-free assignment of at least $252/4 = 63$ grid elements with one selected color is a necessary condition for a four-colored rectangle-free grid $G_{12,21}$.

A first simple approach is the unique distribution of the 63 tokens of one color to the 21 columns of the grid. For this assumption $63/21 = 3$ elements of each column of the grid $G_{12,21}$ have to be colored with the selected color. At the first glance such an assignment fits well to the rectangle-free coloring of the grid $G_{12,21}$ with four colors. The remaining $12 - 3 = 9$ elements of each column can be colored using each of the other three colors on three positions, too.

However, the evaluation of the color assignments in the rows of the grid $G_{12,21}$ avoids this suggested simple approach. Dividing the necessary 63 tokens of one color by the 12 row results in 9 rows of 5 tokens and 3 rows of 6 tokens of the selected color. It must be verified whether a rectangle-free coloring of a grid $G_{12,6}$ for one color exists that holds the following conditions:

1. one row of the grid $G_{12,6}$ contains 6 tokens of the selected color,

2. each of the 6 columns of the grid $G_{12,6}$ contains 3 tokens of the selected color,

3. the $3*6 = 18$ tokens of the selected color hold the rectangle-free condition (2.9) for $m = 12$ rows and $n = 6$ columns.

	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1					
3	1					
4		1				
5		1				
6			1			
7			1			
8				1		
9				1		
10					1	
11					1	
12						1

Figure 2.10. Assignment of color 1 tokens to the grid $G_{12,6}$ based on the proof of Theorem 2.6.

Theorem 2.6. *There does not exist a rectangle-free coloring of the grid $G_{12,6}$ of 12 rows and 6 columns with $3 * 6 = 18$ tokens colored by a single color such that one row is completely covered with tokens of the selected color and each of the 6 columns contains 3 tokens of the this color.*

Proof. The exchange of any pair of rows or any pair of columns within each grid does not change the property that the grid satisfies the rectangle-free condition. Hence, without loss of generality the first row can be chosen to assign one token of the selected color to each of the 6 positions. For the same reason the required further two tokens with the same color in the first column can be located in the rows number 2 and 3. Figure 2.10 shows these assignments using tokens of the color 1.

In order to satisfy the rectangle-free condition no further token of the given color can be assigned to row 2 and row 3. As shown in Figure 2.10, the required assignments of three tokens in the columns 2, 3, 4, and 5 fill up the rows 4 to 11. The rectangle-free condition prohibits each further assignment of a token of the used color to the rows 2 to 11 to the grid shown in Figure 2.10 and permits only one additional token of this color in row 12.

	1	2	3
1	1		
2	1		
3	1		
4	1		
5		1	
6		1	
7		1	
8		1	
9			1
10			1
11			1
12			1

Figure 2.11. Grid $G_{12,3}$ of disjoint assignments of color 1 tokens to four rows in three columns.

Hence, column 6 only contains two tokens of the selected color and does not satisfies the requirement of three tokens. Each further assignment of a token of the selected color to any element of the grid shown in Figure 2.10 violates the rectangle-free condition. □

According to Theorem 2.6, no four-colored rectangle-free grid $G_{12,21}$ will exist that contains in each column each color three times. Taking into account that each of the three necessary rows of six tokens of the same color causes one column of only two tokens of this color, three columns of four tokens of this color must be used.

There are several possibilities to assign four tokens of the same color to the three necessary columns. Due to the rectangle-free condition, it is not correct that these tokens overlap in more than one row. The three columns of four tokens of the same color cause the smallest restriction for the remaining columns when these tokens do not overlap in any row.

Despite of the maximal freedom for further assignments of tokens of the same color, the chosen distribution of tokens as shown in Figure 2.11 restricts the maximal number of columns in which three tokens of the same color can be assigned.

Theorem 2.7. *The grid $G_{12,3}$ of Figure 2.11 can be extended to a rectangle-free grid that includes at most 16 additional columns of three tokens of the same color. Without violating the rectangle-free condition additional columns can contain only in one token of the same color.*

Proof. Figure 2.12 shows a correct rectangle-free grid $G_{12,19}$ that holds the conditions of Theorem 2.7. Without loss of generality only tokens of the color 1 are used in this proof. Due to the four tokens in column 1, the rectangle-free condition restricts the assignments in rows 1 to 4 to one token in each of columns 4 to 19. Analog restrictions must hold for the interval of rows 5 to 8 caused by the second column and for the interval of rows 9 to 12 caused by the third column, respectively. Hence, the three assignments of tokens in columns 4 to 19 must be done in the three row intervals 1 to 4, 5 to 8, and 9 to 12.

The rectangle-free condition constrains the assignment to one token in each column 4 to 19 and each of the three row intervals. Furthermore, tokens in one row of one row interval require tokens in different rows of the other row intervals in order to satisfy the rectangle-free condition. Hence, the assignment of tokens in the first row is restricted to four columns due to the four rows in the second and third row interval.

The well-ordered pattern of tokens in the interval of rows 1 to 4 in Figure 2.12 can be reached by permutations of columns 4 to 19. The used assignment of tokens in Figure 2.12 in the interval of rows 5 to 8 in a downstairs order is one simple possible selection. This pattern can be reached by permutations of columns indicated by tokens in the same row in the upper row interval.

The rule to be satisfied is that each assignment in the interval of rows 1 to 4 is combined with one assignment in the interval of rows 5 to 8. These $4 * 4 = 16$ combinations restrict the maximal number of columns with three tokens under the restriction of the assignments in the first three columns to 16 additional columns. This proves the first assertion.

The assignments in the last interval of rows 9 to 12 must cover each row and each column in intervals 4 to 7, 8 to 11, 12 to 15, and 16

	1			4				8				12				16			
1	1			1	1	1	1												
2	1							1	1	1	1								
3	1											1	1	1	1				
4	1															1	1	1	1
5		1		1				1				1				1			
6		1			1				1				1				1		
7		1				1				1				1				1	
8		1					1				1				1				1
9			1	1					1					1					1
10			1		1			1							1			1	
11			1				1			1		1					1		
12			1			1					1		1			1			

Figure 2.12. Assignment of three color 1 tokens to each of the columns from 4 to 19 of the grid $G_{12,19}$ based on Theorem 2.7 using the Latin square 1 without permutations of rows or blocks.

to 19 exactly once. There are several possible assignments in this range which hold the rectangle-free condition. One of them is shown in Figure 2.12.

An exhaustive evaluation of the tokens of the grid $G_{12,19}$ in Figure 2.12 proves that each pair of rows is covered by two tokens in one of the 19 columns. Other possible assignments do not change this property. This proves the second assertion that without violating the rectangle-free condition additional columns can contain only one token of the same color. □

Using permutations of rows and columns the pattern in the upper 8 rows and the left 3 columns of Figure 2.12 can be constructed. The rectangle-free condition in the remaining region can be achieved by means of Latin squares [175]. Figure 2.13 shows the four reduced Latin squares of the size 4×4 which have a fixed natural order of the letters in the first row and the first column. These reduced Latin squares are labeled by the values $0, \ldots, 3$ based on the letters of the main diagonal in natural order. The mapping rules between a Latin square and the four 4×4 blocks, which are indicated by thick lines in the bottom right of Figure 2.12 are as follows:

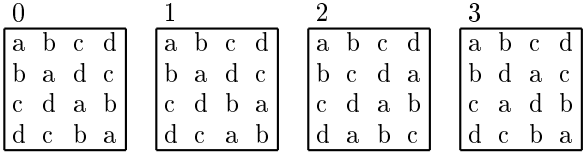


Figure 2.13. All four reduced Latin squares $0, \ldots, 3$ of the size 4×4.

- one Latin square is used to assign the tokens of one color to all four 4×4 blocks in the range of rows 9 to 12 and columns 4 to 19,
- each letter a indicates one token of this color in the leftmost block,
- each letter b indicates one token of this color in the second block from the left,
- each letter c indicates one token of this color in the third block from the left, and
- each letter d indicates one token of this color in the rightmost block.

All correct patterns of the selected color can be constructed based on the four Latin squares of Figure 2.13 and permutations restricted to the last four rows and complete 4×4 blocks in these rows. Figure 2.12 shows the result of the mapping of Latin square 1 without permutations of rows or blocks.

The grid $G_{12,19}$ of Figure 2.12 contains $3 * 4 + 16 * 3 = 12 + 48 = 60$ tokens. Is it possible to extend this grid to a rectangle-free grid $G_{12,21}$ that contains the necessary 63 tokens of one color? Each of the additional two columns 20 and 21 can contain only one token of the same color due to Theorem 2.7. Hence, such a grid $G_{12,21}$ contains only $60 + 2 = 62$ tokens, and does not reach the required limit of 63 tokens of a single color for a possible completely four-colored rectangle-free grid $G_{12,21}$.

This observation, however, will not prove that a four-colored rect-

1			1	1		1	1	1											
1									1	1	1	1							
1													1	1	1	1			
1																	1	1	1	1
	1		1		1				1				1				1			
	1					1				1				1				1		
	1						1				1				1				1	
	1							1				1				1				1
		1		1	1					1					1					1
		1				1			1							1			1	
		1						1			1		1					1		
		1					1					1		1			1			

Figure 2.14. Rectangle-free grid $G_{12,21}$ that contains 63 tokens of one color.

angle-free grid $G_{12,21}$ does not exist. One of the columns 4 to 19 of Figure 2.12 can be replaced by three columns which contain two tokens of the selected color each. Using column 4 for this split we get the required

$$3 * 4 + 3 * 2 + 15 * 3 = 12 + 6 + 45 = 63$$

tokens of one color to construct a completely four-colored rectangle-free grid $G_{12,21}$. Figure 2.14 shows such a grid $G_{12,21}$ that contains a rectangle-free assignment of 63 color 1 tokens.

2.5.2. Grid Heads of all Four Colors

The columns of the grid can be classified based on the number of tokens of one color. We call the six columns of four or two tokens of the same color the *head of the grid*. The remaining 15 columns of 3 tokens of the same color are called *body of the grid*.

Each column of the grid must be completely filled by tokens of the four colors. The possible combinations are restricted by the required numbers of tokens of one color which can be equal to 2, 3, or 4. Four of these digits must result in the number of the 12 rows. This can be

1	1	2	2	1	1	2
2	1					2
3	1					2
4	1					2
5	2	1	2	1	2	1
6		1			2	
7		1			2	
8		1			2	
9	2	2	1	2	1	1
10			1	2		
11			1	2		
12			1	2		

Figure 2.15. Rectangle-free grid $G_{12,6}$ that merges the heads of two colors.

achieved by

$$3+3+3+3=12\ ,$$
$$2+4+3+3=12\ ,\ \text{or}$$
$$2+2+4+4=12\ .$$

The body of the grid $G_{12,21}$ can be colored using three tokens of each of the four colors in each of the 15 columns. Based on this assumption the head of the grid must contain in each column two tokens of two colors and four tokens of the other two colors. Figure 2.15 shows the grid head that contains four-token, and two-token columns of the first two colors within six columns.

It can be seen in Figure 2.14 that the rows which contain 6 tokens of one color include two tokens of columns with two tokens. The grid head shown in Figure 2.15 contains the tokens of the colors 1 and 2 of the two-token columns within rows 1, 5, and 9. Hence, these three rows must contain six color 1 tokens, six color 2 tokens, five color 3 tokens, and five color 4 tokens. Using this assignment, the sum of necessary tokens in these rows is equal to 22 but only 21 columns exist. Therefore, the grid head of Figure 2.15 cannot be extended to a four-colored rectangle-free grid $G_{12,21}$.

Therefore, each row of the four-colored rectangle-free grid $G_{12,21}$ must contain six tokens of one color and five tokens of each of the other three

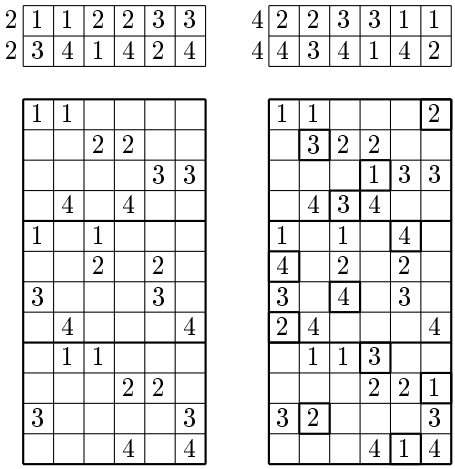


Figure 2.16. Steps to construct a rectangle-free grid $G_{12,6}$ that merges the heads of all four colors: (a) assignment of two-token columns, (b) necessary extension of (a) with one token of the four-token columns.

colors. That means the tokens of the two-token columns of the four colors must be assigned to different rows.

The construction of the grid head of six columns of all four colors is the key to our solution of the four-colored rectangle-free grid $G_{12,21}$. This construction requires complicated entanglements of the assignments of two-token, or four-token columns of the four colors. We explain our sequence of decisions supported by several figures and one table starting with Figure 2.16.

From the trivial merging of the grid heads of two colors in Figure 2.15 we learned that the tokens of the two-token columns of different colors must be assigned to different rows. There are three two-token columns of each of the four colors. These 12 columns can be merged into the 6 columns of the grid head such that two two-token columns of different colors are assigned to one column of the grid head.

We decided further that the triples of two-token columns overlap in

one column. The array on top of Figure 2.16 (a) indicates the columns which are used for two tokens of the specified color. It can be seen that the first three colors can be assigned in consecutive ranges in a cyclic manner. Consequently, the columns 2, 4, and 6 must be used as two-token columns of color number 4.

Figure 2.16 (a) shows that a natural order is used for the rows in which tokens of the two-token columns are located:

color:	1	-	rows:	1,	5,	9,
color:	2	-	rows:	2,	6,	10,
color:	3	-	rows:	3,	7,	11,
color:	4	-	rows:	4,	8,	12.

It is known from the construction of the single-color grid head that the four-token columns of one color do not overlap with the two-token columns of the same color. Based on the selected two-token columns on top of Figure 2.16 (a) the four-token columns of the grid head must be chosen as shown in the array on top of Figure 2.16 (b). Each of the six columns of the grid head is either used for a two-token column or for a four-token column for each of the four colors.

As shown in Figure 2.11, each row must contain exactly one token of the four-token columns. Figure 2.16 (a) shows that up to now each row leaves four elements of the grid head free. Hence, each of the four colors must be assigned exactly once to these four free elements in each row of Figure 2.16 (a). It can be seen from the array on top of Figure 2.16 (b) that this rule can be satisfied in each row by one of the four colors only in a single position. These necessary positions are added in Figure 2.16 (b) and labeled by thick boarders.

The necessary tokens of four-token columns are assigned based on the evaluation of the available colors of four-token columns in the free positions of the rows. A check of these tokens labeled by thick boarders in Figure 2.16 (b) shows that each of the two colors of the four-token columns of the array on top of Figure 2.16 (b) is assigned exactly once to each column of the grid head. The first column of the array on top of Figure 2.16 (b) specifies as example that this column must contain four tokens of the color 2 and four tokens of the color 4. The necessary tokens are assigned in this column as follows: color 2 to row 8 and color 4 to row 6.

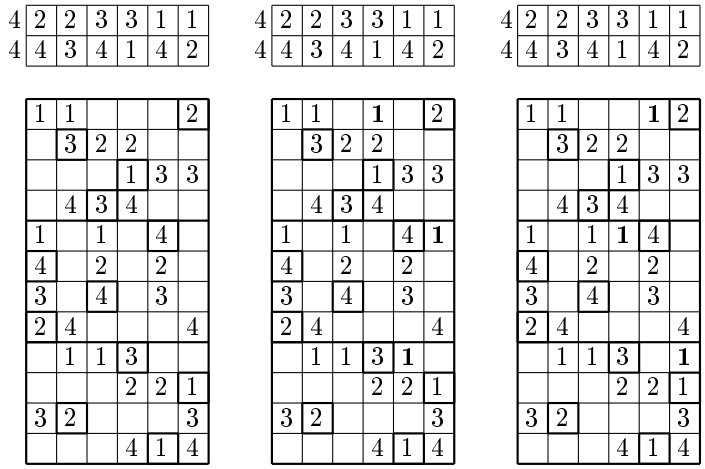


Figure 2.17. Alternative assignments to construct rectangle-free grid $G_{12,6}$ that merges the heads of all four colors: (a) necessary assignments of four-token columns, (b) first choice of color 1, (c) second choice of color 1.

It is known from Figure 2.14 that each two-token column overlaps in two rows of the grid head with two different four-token columns of the same color. There are two possible positions for these assignments in each row used for the two-token columns. Figure 2.17 shows on the right two grid heads the possible assignments of color 1 tokens of four-token columns by bold numbers 1. Figure 2.17 (a) repeats the necessary assignments of tokens from Figure 2.16 (b) for direct comparisons.

Figure 2.17 (a) shows that row 1 is used for tokens of color 1 of two-token columns and includes empty positions for tokens of color 1 of four-token columns in the columns 4 and 5. Bold numbers 1 in the first row of Figure 2.17 (b) and (c) depict these two possible assignments, and this entails further assignments of the color 1 in four-token columns in rows 5 and 9. Free positions for color 1 in row 5 are given in columns 4 and 6 of Figure 2.17 (a). Column 4 is already occupied by the color 1 token in row 1 so that the color 1 token of the four-token column must be assigned in row 5 to column 6 as shown

in Figure 2.17 (b). It remains row 9 that includes color 1 tokens of two two-token columns. Figure 2.17 (a) shows two free positions in columns 5 and 6 which can be used for color 1 tokens of four-token columns. Column 6 is already occupied by the color 1 token in row 5 so that the color 1 token of the four-token column must be assigned in row 9 to column 5 as shown in Figure 2.17 (b).

Figure 2.17 (c) shows the single alternative to this assignment of color 1 tokens of four-token columns to rows which are used for color 1 tokens of two-token columns. The chosen color 1 token in row 1 and column 5 bans the color 1 token in row 9 and column 5. Hence, the color 1 token of four-token column 6 must be assigned in row 9. This color 1 token bans the color 1 token in row 5 so that color 1 token of four-token column 4 must be assigned to row number 5.

Figure 2.17 (b) and (c) shows the alternative assignments of color 1 tokens to the rows 1, 5, and 9 which are used for color 1 tokens of two-token columns. A detailed analysis reveals that two alternative assignments exist for

- color 2 tokens of four-token columns in the rows 2, 6, and 10,
- color 3 tokens of four-token columns in the rows 3, 7, and 11,
- color 4 tokens of four-token columns in the rows 4, 8, and 12.

Table 2.9 enumerates these alternative assignments for each color in two triples of rows. Bold numbers in the right six columns indicate the color of the token that overlaps with the same color of a two-token column. Each of these alternative assignments for one color can be combined with each alternative assignment for all other colors. Hence, there are 16 combinations. For a further analysis Boolean variables s_1, s_2, s_3, and s_4 are introduced such that the index indicates the associated color. The value 0 of these variables indicates the left assignment of these tokens in the rows from 1 to 4.

The rows 1, 5, and 9 of Figure 2.17 (b) and (c) contain two empty elements. These elements must be filled with tokens of four-token columns which are missing in these rows. The chosen (bold) color 1 tokens entail necessary assignments of the missing tokens. As can be

Table 2.9. Alternative assignments in four-token columns of the grid head.

		selection				column					
color	row	s_1	s_2	s_3	s_4	1	2	3	4	5	6
1	1	0						3	**1**	4	
1	5	0					2		3		**1**
1	9	0				4				**1**	2
1	1	1						4	3	**1**	
1	5	1					3		**1**		2
1	9	1				2				4	**1**
2	2		0			**2**				4	1
2	6		0				3		1		**2**
2	10		0			4	**2**	3			
2	2		1			4				1	**2**
2	6		1				**2**		3		1
2	10		1			**2**	3	4			
3	3			0		2	**3**	4			
3	7			0			2		**3**		1
3	11			0				**3**	1	4	
3	3			1		4	2	**3**			
3	7			1			**3**		1		2
3	11			1				4	**3**	1	
4	4				0	**4**				1	2
4	8				0			3	1	**4**	
4	12				0	2	3	**4**			
4	4				1	2				**4**	1
4	8				1			**4**	3	1	
4	12				1	**4**	2	3			
		restrictions									
2		1		0		*					
4		0		1		*					
2		0			1		*				
3		1			0		*				
3		0	0					*			
4		1	1					*			
1			0		0				*		
3			1		1				*		
1			1	1						*	
4			0	0						*	
1				0	1						*
2				1	0						*

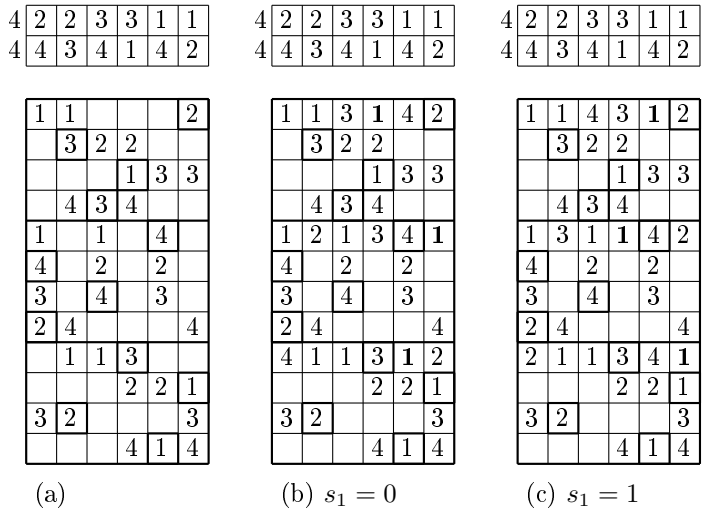


Figure 2.18. Extended alternative assignments of color 1 tokens and consecutive color 2, color 3, and color 4 tokens to construct a rectangle-free grid $G_{12,6}$ that merges the heads of all four colors: (a) necessary assignments of four-token columns, (b) first choice of color 1, (c) second choice of color 1.

seen in the array on top of Figure 2.18 (b), column 5 can contain only tokens of the colors 1 or 4 as element of four-token columns. The bold valued 1 in row 1 indicates that color 1 is already used as element of a four-token column. Hence, a color 4 token must be assigned to the column 4 of row 1 as shown in Figure 2.18 (b). This color 4 token bans a further color 4 token in row 1 and column 3 of Figure 2.18 (b). The missing color 3 in row 1 of Figure 2.18 (b) is permitted for the empty position in column 3 so that the first row of Figure 2.18 (b) is completely filled and satisfies all conditions.

Based on equivalent conditions the color 3 token must be assigned to the column 4 of row 5 entailed by the assignment of a color 2 token to the column 2 of row 5. Figure 2.18 (b) shows the final assignment for the rows 1, 5, and 9. The necessary consecutive assignments in row 9 are first a color 2 token in column 6 followed by a color 4 token in column 1. Analog conditions entail the unique assignment in the rows 1, 5, and 9 of the second choice of the value 1 as shown Figure

2.18 (c).

The four-column tokens of all other colors i are uniquely assigned for both cases of the selection variable s_i. The last six columns in the upper part of Table 2.9 show these entailed assignments by normal digits with or without a light-gray background.

The two different assignments in the rows belonging to two-color columns of one color were chosen independent on the analog assignments for the other colors. For that reason $2^4 = 16$ grid heads can be constructed. Do these 16 grid head satisfy all conditions of the grid head? We will show as final step to construct the grid head that the answer to this question is *NO*.

Each column of the grid head combines two two-token columns and two four-token columns. Hence, a grid head is not correct when one column contains more than four tokens of one color. Table 2.9 shows that certain values of four-token columns must be assigned to a column of the grid head independent on the value of the selection variable s_i. These values are marked by light-gray backgrounds in Table 2.9. This means, for example, within the rows 1, 5, and 9:

- one color 3 token is assigned to the column 4,
- one color 4 token is assigned to the column 5, and
- one color 2 token is assigned to the column 6

independent on the value of s_1. These fix assignments cause restrictions for the values of the selection variables s_i.

The evaluation of possible assignments in the first column of the grid head results in two restrictions. It is known from Figure 2.16 (b) that a color 2 token must be assigned to row 8 of the first column. As indicated by bold numbers 2 in Table 2.9 for column 1 of the grid head, the second color 2 token must be assigned to the first column independent on the value of s_2. The two light-gray marked values 2 of Table 2.9 associated to column 1 of the grid head requires the third color 2 token that must be assigned to the first column independent on the value of the selection variable s_4. There are two more values

2 belonging to the column 1 of the grid head. These values will be chosen by $s_1 = 1$ and $s_3 = 0$. Hence, the solution of the Boolean equation $s_1 \wedge \overline{s}_3 = 1$ describes a pattern of the grid head that includes five color 2 tokens in the first column. Consequently, the combinations of the grid head must satisfy the restriction

$$s_1 \wedge \overline{s}_3 = 0 \ .$$

Analog evaluation for the color 4 in the first column of the grid head lead to the restriction

$$\overline{s}_1 \wedge s_3 = 0 \ .$$

Table 2.9 enumerates these restrictions in the lower part for all six columns of the grid head. A $*$ indicates the evaluated column of the grid head regarding the value of the color given in the first column of Table 2.9.

Correct combinations of the assignments to the grid head must satisfy the system of Boolean equations (2.32) which are found by the complete analysis as explained above for all columns.

$$\begin{aligned}
s_1 \wedge \overline{s}_3 &= 0 \\
\overline{s}_1 \wedge s_3 &= 0 \\
\overline{s}_1 \wedge s_4 &= 0 \\
s_1 \wedge \overline{s}_4 &= 0 \\
\overline{s}_1 \wedge \overline{s}_2 &= 0 \\
s_1 \wedge s_2 &= 0 \\
\overline{s}_2 \wedge \overline{s}_4 &= 0 \\
s_2 \wedge s_4 &= 0 \\
s_2 \wedge s_3 &= 0 \\
\overline{s}_2 \wedge \overline{s}_3 &= 0 \\
\overline{s}_3 \wedge s_4 &= 0 \\
s_3 \wedge \overline{s}_4 &= 0
\end{aligned} \tag{2.32}$$

The system of Boolean equations can be simplified to the Boolean equation:

$$(s_1 \oplus s_3) \vee (s_1 \oplus s_4) \vee \overline{(s_1 \oplus s_2)} \vee \overline{(s_2 \oplus s_4)} \vee \overline{(s_2 \oplus s_3)} \vee (s_3 \oplus s_4) = 0 \ . \tag{2.33}$$

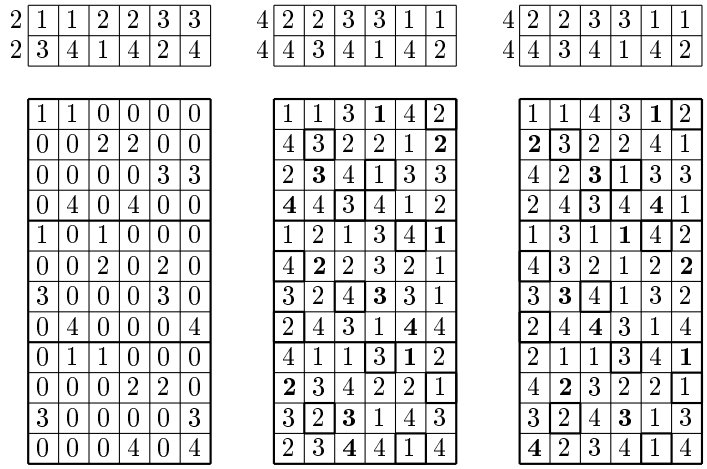


Figure 2.19. Rectangle-free grid $G_{12,6}$ of all four colors: (a) assignment of two-token columns, (b) first extension of (a) with four-token columns for the solution $\mathbf{s} = (0100)$, (c) second extension of (a) with four-token columns for the solution $\mathbf{s} = (1011)$.

This equation has only two solutions:

$$(s_1, s_2, s_3, s_4) = \{(0,1,0,0), (1,0,1,1)\} \ . \tag{2.34}$$

Hence, there are only two grid heads which satisfy all requirements. Figure 2.19 shows these grid heads which are constructed based on Figure 2.16 (b), the content of Table 2.9, and the solution of the Boolean equation (2.33).

2.5.3. Extended Models to Solve the Grid Using a SAT-solver

The SAT-formula (2.20) describes the recangle-free condition of four colors for grids of m rows and n columns. This Boolean equation can be adapted to (2.35) for the grid $G_{12,21}$. The SAT-formula (2.35) depends on $12 * 21 * 2 = 504$ Boolean variables and contains $\binom{12}{2} *$

$\binom{21}{2} * 4 = 55,440$ clauses.

$$\bigwedge_{i=1}^{11} \bigwedge_{j=i+1}^{12} \bigwedge_{k=1}^{20} \bigwedge_{l=k+1}^{21} \overline{f_{ecb}(a_{r_i,c_k}, b_{r_i,c_k}, a_{r_i,c_l}, b_{r_i,c_l}, a_{r_j,c_k}, b_{r_j,c_k}, a_{r_j,c_l}, b_{r_j,c_l})} = 1 \ . \tag{2.35}$$

Using the grid head of Figure 2.19 (b) or (c) $12 * 6 * 2 = 144$ Boolean variables can be used with fixed values. The $6 * 2$ additional clauses for the first row of Figure 2.19 (b) are, e.g.,

$$\overline{a}_{1,1} \wedge \overline{b}_{1,1} \wedge \overline{a}_{1,2} \wedge \overline{b}_{1,2} \wedge \overline{a}_{1,3} \wedge b_{1,3} \wedge \overline{a}_{1,4} \wedge \overline{b}_{1,4} \wedge a_{1,5} \wedge b_{1,5} \wedge a_{1,6} \wedge \overline{b}_{1,6} \ .$$

These 144 clauses of single variables simplify the problem by a factor of $2^{144} \approx 2.23 * 10^{43}$.

SAT-solvers are able to find a solution for a SAT-formula with a large number of Boolean variables if many solutions exist. Recall from Table 2.6, the SAT-solver *clasp* found a solution for the grid $G_{14,14}$ of 392 free variables within 0.559 seconds. The number of free variables of the grid $G_{12,21}$ with a fix grid head of 6 columns is equal to $12*15*2 = 360$. Hence, due to the further simplification, it could be expected that the used SAT-solver `clasp-2.0.0-st-win32` finds the solution very quickly. However, the additional restrictions of the grid head exclude so many color patterns that after a runtime of one month no solution was found.

In the next experiment we tried to guide the SAT-solver using our knowledge of the structure of a possible solution. The 15 columns from number 7 to number 21 must contain exactly three tokens of each of the four colors. Hence, it is incorrect that the same color occurs four times in these columns. $\binom{12}{4} = 495$ clauses of eight variables are needed to specify this requirement of one color in one column of the grid body.

We generated a SAT-formula that contains the 55,440 clauses for the rectangle rule (2.35), the 144 clauses of the constant values of the grid head and additionally $\binom{12}{4} * 15 * 4 = 29,700$ clauses for the color restriction in the body of the grid $G_{12,21}$. Unfortunately, the SAT-solver did not find a solution for this more precise SAT-formula of 504 Boolean variables and 85,284 clauses within one month.

Due to the very large runtime of the SAT-solver, we restricted the search space of the problem in the next experiment even more. We mapped the potential solution of the body of the grid $G_{12,21}$ for the first color as shown in Figure 2.14 to the entangled position based on both grid heads of Figure 2.19 (b) and (c). The SAT formula of 504 Boolean variables contains with this extension 85,374 clauses, but the free variables are restricted to $15 * 9 * 2 = 270$. The SAT-solver solves this restricted problem for the grid head of Figure 2.19 (b) within only 0.608 seconds and for the grid head of Figure 2.19 (c) within only 0.069 seconds, respectively. However, the answer of the SAT-solver was in both cases `UNSATISFIABLE`.

Does this result mean that no four-colored rectangle-free grid $G_{12,21}$ exists? Our answer to this question is *NO*. We explored both unsatisfiable SAT problems more in detail. The embedded grid heads of Figure 2.19 (b) and (c) can be transformed by permutation of rows and columns in such a way that the patterns of the color 1 is equal to the pattern of this color shown in the first six columns of Figure 2.14. The pattern of the color 1 in the remaining columns 7 to 21 and rows 1 to 8 can be transformed into the pattern of this range given in Figure 2.14 using only permutations of columns. Hence, it remains a region of several choices of color 1 in the region of columns 7 to 21 and rows 9 to 12.

Due to these alternatives we removed the 15 assignments of color 1 tokens in this region and get a SAT formula of 504 Boolean variables and 85,344 clauses that includes 300 free variables. It took 343.305 seconds to solve this problem for the grid head of Figure 2.19 (b) and 577.235 seconds for the grid head of Figure 2.19 (c) using the SAT-solver `clasp-2.0.0-st-win32`. Figure 2.20 shows these solutions of four-colored rectangle-free grids $G_{12,21}$ for both grid heads of Figure 2.16. Hence, we found solutions for this problem for both grid heads.

2.5.4. Classes of Rectangle-free Grids $G_{12,21}$

Both permutations of rows and columns of the four-colored rectangle-free grids $G_{12,21}$ of Figure 2.20 generate other four-colored rectangle-free grids $G_{12,21}$. Such a set of grids constructed from one given four-

(a)

1	1	3	1	4	2	1	1	1	3	4	3	2	4	2	3	2	4	3	4	2
4	3	2	2	1	2	1	3	3	2	2	1	3	4	4	4	1	1	3	2	4
2	3	4	1	3	3	2	4	4	1	1	1	1	4	3	3	3	2	4	2	2
4	4	3	4	1	2	4	2	1	1	4	2	3	3	1	2	3	2	4	1	3
1	2	1	3	4	1	4	4	3	1	3	2	4	1	4	3	2	1	2	2	3
4	2	2	3	2	1	1	3	4	3	1	4	4	2	1	2	3	4	1	3	2
3	2	4	3	3	1	3	1	2	2	4	1	4	3	2	1	4	2	3	1	4
2	4	3	1	4	4	3	3	4	2	3	2	2	2	3	4	4	1	1	1	1
4	1	1	3	1	2	2	4	2	4	1	3	3	2	3	1	4	3	2	4	1
2	3	4	2	2	1	4	2	1	3	3	4	1	3	2	4	1	3	2	4	1
3	2	3	1	4	3	2	2	3	4	2	4	2	1	1	1	1	3	4	3	4
2	3	4	4	1	4	3	1	2	4	2	3	1	1	4	2	2	4	1	3	3

(b)

1	1	4	3	1	2	2	2	3	1	4	4	3	1	2	4	3	1	3	4	2
2	3	2	2	4	1	4	2	1	4	2	1	4	3	1	3	3	1	2	4	3
4	2	3	1	3	3	2	4	2	4	2	4	3	2	3	3	4	1	1	1	1
2	4	3	4	4	1	1	3	3	1	3	4	2	2	2	2	1	4	4	1	3
1	3	1	1	4	2	1	1	1	2	3	2	3	2	4	4	4	3	2	3	4
4	3	2	1	2	2	3	3	2	1	1	1	1	4	3	2	2	4	3	4	4
3	3	4	1	3	2	4	4	3	3	2	3	2	1	1	1	1	2	4	2	4
2	4	4	3	1	4	1	3	2	2	1	3	4	4	1	3	4	2	1	3	2
2	1	1	3	4	1	2	4	4	3	4	2	1	4	3	1	2	3	1	2	3
4	2	3	2	2	1	3	1	4	3	1	2	4	1	2	4	3	2	4	3	1
3	2	4	3	1	3	3	1	4	4	3	1	2	3	4	1	2	4	2	1	2
4	2	3	4	1	4	4	2	1	2	4	3	1	3	4	2	1	3	3	2	1

Figure 2.20. Four-colored rectangle-free grids $G_{12,21}$: (a) extension of the grid head of Figure 2.19 (b); (b) extension of the grid head of Figure 2.19 (c)

colored rectangle-free grids $G_{12,21}$ is an equivalence class. A conjecture of the long runtime of the SAT-solver without finding a solution is that only a small subset of color patterns satisfies the rectangle-free condition. This conjecture implies the interesting question, how many different classes of four-colored rectangle-free grids $G_{12,21}$ exist for the grid heads of Figure 2.19.

This problem can be solved in such a way that all solutions of the finally constructed SAT-formula of 504 variables and 85,344 clauses for both grid heads are calculated and evaluated regarding their equiva-

lence class. The complete calculation takes only 453.880 seconds for the grid head of Figure 2.19 (b) and 745.700 seconds for the grid head of Figure 2.19 (c). There are 38,926 different solutions for each of these grid heads.

These 38,926 different solutions for each of these grid heads can be classified regarding

- the four Latin squares used in the region of rows 9 to 12 and columns 7 to 21 of Figure 2.14
- the permutations of the three blocks in these rows and columns 10 to 21, and
- the permutations of the three rows 10 to 12.

This taxonomy originates $4 * 3! * 3! = 4 * 6 * 6 = 144$ permutation classes (equivalence classes) of four-colored rectangle-free grids $G_{12,21}$ for each grid head. The result of our detailed evaluation is that for each of both grid heads 100 of these permutation classes are empty and only the remaining 44 permutation classes contain equivalence classes of four-colored rectangle-free grids $G_{12,21}$.

Figure 2.21 shows the distribution of the found 44 permutation classes regarding the possible permutations of the four Latin squares for the grid head of Figure 2.19 (b). The distribution of the 38,926 different equivalence classes of four-colored rectangle-free grids $G_{12,21}$ for grid head of Figure 2.19 (c) is shown in Figure 2.22.

The values in the eight tables of Figures 2.21 and 2.22 indicate the number of different assignments of color 2, color 3, and color 4 tokens for a certain fixed assignment of color 1 tokens. A value 0 in these tables means that no four-colored rectangle-free grid $G_{12,21}$ exists for the chosen assignment of the color 1 tokens. The unsatisfiable SAT-formulas which extends the grid heads of Figure 2.19 (b) and (c) by all assignments of the color 1 tokens in the columns from 7 to 21 as shown in Figure 2.14 belong to the class of the top left corner in Table (b1) of Figure 2.21 or Table (a2) of Figure 2.22.

Finally, the fraction of four-colored rectangle-free grids $G_{12,21}$ of all

(a1) numbers of permutation classes for the Latin square 0

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	875	0	0	0	0	16
132	0	16	0	70	0	0
213	15720	0	0	0	0	875
312	0	0	0	0	0	0
231	0	875	0	16	0	0
321	0	0	0	0	0	0

(b1) numbers of permutation classes for the Latin square 1

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	0	184	0	16	0	0
132	10	0	0	0	0	48
213	0	5488	0	185	0	0
312	0	0	0	0	0	0
231	822	0	0	0	0	10
321	0	0	0	0	0	0

(c1) numbers of permutation classes for the Latin square 2

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	185	0	2	0	0	10
132	0	185	0	16	0	0
213	5488	0	0	0	0	822
312	0	2	32	0	20	0
231	0	10	0	48	0	0
321	0	0	0	20	0	0

(d1) numbers of permutation classes for the Latin square 3

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	5488	0	0	0	0	184
132	0	16	20	48	0	0
213	822	0	0	0	0	10
312	0	0	2	0	0	32
231	0	184	32	10	4	0
321	0	0	0	0	4	4

Figure 2.21. Number of permutation classes of four-colored rectangle-free grids $G_{12,21}$ for the grid head of Figure 2.19 (b).

(a2) numbers of permutation classes for the Latin square 0

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	0	70	0	16	0	0
132	16	0	0	0	0	875
213	0	0	0	0	0	0
312	875	0	0	0	0	15720
231	0	0	0	0	0	0
321	0	16	0	875	0	0

(b2) numbers of permutation classes for the Latin square 1

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	48	0	0	0	0	10
132	0	16	0	184	0	0
213	0	0	0	0	0	0
312	0	185	0	5488	0	0
231	0	0	0	0	0	0
321	10	0	0	0	0	822

(c2) numbers of permutation classes for the Latin square 2

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	0	16	0	185	0	0
132	10	0	0	0	2	185
213	0	0	20	2	32	0
312	822	0	0	0	0	5488
231	0	20	0	0	0	0
321	0	48	0	10	0	0

(d2) numbers of permutation classes for the Latin square 3

permutation of	permutation of blocks					
rows	123	132	213	312	231	321
123	0	48	0	16	20	0
132	184	0	0	0	0	5488
213	32	0	0	0	2	0
312	10	0	0	0	0	822
231	4	0	4	0	0	0
321	0	10	4	184	32	0

Figure 2.22. Number of permutation classes of four-colored rectangle-free grids $G_{12,21}$ for the grid head of Figure 2.19 (c).

color patterns of this grid size can be determined. The number of all four-colored rectangle-free grids $G_{12,21}$ is equal to the sum of all 12! permutations of the row and 21! permutations of the columns of the 38,926 permutation classes of two grid heads:

$$n_{rf4c}(12,21) = 2 * 12! * 21! * 38,926 = 1.90524 * 10^{33} \ . \qquad (2.36)$$

The number of different color patterns n_{cp} of the grids $G_{12,21}$ is equal to $5.23742 * 10^{151}$ (2.11). Hence, the ratio of four-colored rectangle-free grids $G_{12,21}$ regarding all color patterns of this grid size is equal to

$$\frac{1.90524 * 10^{33}}{5.23742 * 10^{151}} = 3.6377 * 10^{-119} \ . \qquad (2.37)$$

Hence, almost all four-colored grids $G_{12,21}$ do not satisfy the rectangle-free condition (2.9).

3. Theoretical and Practical Concepts

3.1. Perceptions in Learning Boolean Concepts

Ilya Levin Gabi Shafat

Hillel Rosensweig Osnat Keren

3.1.1. Boolean Concept Learning

The section deals with *Human Concept Learning* (HCL). Learning concept by humans may utilize various approaches to study how concepts are learned. Specifically, the perception and the recognition of Boolean concepts by humans are considered.

Concepts are the atoms of thought and they are therefore at the nucleus of cognition science [105]. People begin to acquire concepts from infancy and continue to acquire and plan new concepts throughout their entire lives [192, 193]. One way to create a new concept is by utilizing existing concepts in different combinations. One of the problems in learning concepts is determining the concepts' subjective difficulty. An important aspect of a concept learning theory is the ability to predict the level of difficulty in learning different types of concepts.

There are classes of concepts of a special interest. One of the most popular and significant class of concepts of special interest is a class of

Boolean concepts. Boolean concepts are concepts that are formed by Boolean functions of existing concepts which themselves can be not Boolean. The modern technological environment has been intensively digitized during the last two decades. The environment mainly has a Boolean nature and, naturally, puts human learning of Boolean concepts on the research agenda. Human concept learning of Boolean concepts is called Boolean Concept Learning (BCL).

The difficulty of learning some Boolean concepts by humans can be referred to as the cognitive complexity of the corresponding concepts. One of the important research fields of BCL concerns the factors that determine the subjective difficulty of the concepts. This research field addresses the question why some concepts are simple to learn, while others seem to be quite difficult?

In general, the problem of complexity of Boolean concepts was associated with the synthesis of digital systems [131]. The complexity measures were purely technological and mainly related to hardware implementations of corresponding Boolean concepts. Examples of technological tasks, sensitive to Boolean complexity, may be the quantity of computer hardware, its reliability, testability, level of the power dissipation etc. Nowadays, new criteria of complexity of Boolean concepts have moved to the center of the research agenda due to the intensive dissemination of computer means into the everyday life. Not only computer specialists are involved in Boolean concepts evaluation, but also psychologists, sociologists and cognitive scientists. The new criteria address the human perception of concepts, and in particular - Boolean concepts.

Recent technology trends put the human personality in the center. A contemporary problem is actually how the human personality develops its abilities to exist in a new informational society full of intellectual artifacts. Scientists have become interested in a simply formulated question: are there properties of Boolean functions that would allow a human to effectively (i.e., with minimal efforts) learn (understand) tasks corresponding to such functions?

In 2000, J. Feldman published a work [97] showing that the cognitive complexity of a Boolean concept is proportional to the number of literals in the minimized (NOT-AND-OR) Boolean expression corre-

sponding to the Boolean concept (minimal description length). The pioneer work of Feldman in the field of Boolean cognitive complexity accelerated research activity in that field.

One of the problems with the minimal description length criterion is how to determine the set of primitive connectives (functions) (like NOT-AND-OR) that are allowed in formulating descriptions. Indeed, by inclusion of just one additional function (connective) - exclusive OR - into the basic set of primitives, one can decrease the length of description and simplify perception of the function by a human, so that the Boolean function is recognized quite easily [96, 122]. An important research question then rises: are there other connectives (functions) that can be successfully used by humans in solving logic problems?

A different way to measure the cognitive complexity is to take into account the ability of learners to recognize specific properties of Boolean concepts. For example, learners easily recognize such property as symmetry, though symmetric Boolean functions have a quite high minimal description length complexity.

There are various types of tasks connected with BCL [60, 93, 145, 178]. These tasks may be very different from the point of human cognitive ability to solve them. They correspond, for example, to analytic or synthetic ways of thinking. More specifically, two types of tasks are of a special interest: recognition and reverse engineering. It is quite natural to assume that different types of tasks have to be studied differently in the context of BCL [178]. One of the important issues in the field of BCL is the evaluation of solutions provided by learners to the above different tasks.

The cognitive complexity of a Boolean concept clearly depends on the number of Boolean variables belonging to the corresponding concept. The majority of works deals with concepts described by a small number of variables (two-three) and actually, the question "How learners recognize Boolean concepts of relatively large number of variables" was never raised. One perspective way to address this question is to use so-called non-exact reasoning in solving BCL tasks. Analyzing non-exact or approximated solutions given by learners was presented in [179]. Obtaining non-exact solutions for BCL tasks does not yet

mean obtaining wrong solutions. Indeed, considering such solutions as wrong may result in evaluation of the corresponding task as cognitively complex, and may further lead to incorrect conclusions about humans abilities in BCL. Moreover, we believe that the non-exactness allows humans to successfully solve Boolean tasks with a great number of variables.

This section addresses the above issues and presents a novel research methodology for studying the above issues. Specifically, we focus on the following three specific research methodologies: recognizing specific properties of Boolean functions when solving BCL tasks; solving different types of BCL tasks (such as recognition and reverse engineering); non-exact reasoning in solving BCL tasks.

3.1.2. Complexity of Boolean Concepts

Cognitive Complexity of Boolean Concepts. Boolean concepts can be defined by a Boolean expression composed of basic logic operations: negation, conjunction, and disjunction. Such types of Boolean concepts have been studied extensively by [39, 42, 224, 274]. These studies were focused on Boolean concepts with small number of variables. Shepard, Hovland, and Jenkins (SHJ) defined 70 possible Boolean concepts that can be categorized into six subcategories.

The results of [274] are highly influential since SHJ proposed two informal hypotheses. The first one is that the number of literals in the minimal expression predicts the level of difficulty. The second hypothesis is that ranking the difficulty among the concepts in each type depends on the number of binary variables in the concept. It is interesting to note that these two relations were originally introduced by Shannon and Lupanov in the context of (asymptotic) *complexity of Boolean functions* [276].

Feldman [97], based on the conclusions from the SHJ study, defined a quantitative relationship between the level of difficulty of learning Boolean concepts and the complexity of Boolean concepts. The complexity measure of a Boolean concept as defined by Feldman is the number of literals in the minimal SOP expression. An alternative ap-

proach for calculating the cognitive-complexity measure of a Boolean concept was proposed in [328] and called a *structural-complexity*.

The majority of works in the field of BCL builds a computational model of complexity that is supposed to approximate a natural, human cognitive complexity of a BCL task. In other words, most researchers intend to translate the cognitive BCL complexity into the quantitative form by proposing various computational models which in turn correspond to computational complexity of the corresponding Boolean functions. Preliminary studies [179, 180] confirm this idea - non-exactness in solving BCL tasks allows to enrich the traditional quantitative approach and to study the cognitive complexity also qualitatively.

Computational Complexity of Boolean Functions. Two types of the computational complexity of Boolean functions are known: *information complexity* and *algorithmic complexity*.

The information complexity is usually associated with the name of Claude Shannon who defined a quantitative approach to measure the complexity. Actually, the information approach says that the complexity is strongly connected with the quantity of information that the concept comprises. The *Shannon complexity* of Boolean functions mostly measures the number of specific components in representations or implementations of Boolean functions. It may be the number of: literals in an expression, gates in a circuit, nodes or paths in the corresponding decision diagram, etc. [97, 107].

For people, studying the computational complexity of Boolean functions, one of the most interesting of Shannon's results can be formulated as follows: Most functions are hard, but we don't have any bad examples [274]. Such a result has to be considered as an optimistic one. It demonstrates perspectives of research in the field of BCL, since despite the fact of high computational complexity of an arbitrary Boolean function, it declares feasibility of studying a lot of really existing Boolean tasks.

The algorithmic approach for measuring the computational complexity of Boolean functions has to be connected with the name of Kolmogorov [164]. According to Kolmogorov, the complexity of a concept

is the complexity of an algorithm producing the concept. The algorithmic complexity is estimated based on regularities existing in the concept. Sometimes these regularities correspond to known properties of functions; but sometimes there may be nontrivial regularities that individuals are able to recognize. For example, symmetric Boolean functions form a well-known example of functions, which, while having high information complexity, are easily recognized by humans owing to their regularity. Obviously, symmetric functions have high information complexity and relatively low algorithmic complexity.

The majority of works in HCL studies the cognitive complexity by applying computational models of the first type, i.e., of the information complexity type. At the same time, the algorithmic complexity of Boolean concepts is not sufficiently studied. The algorithmic complexity may become an especially effective tool to study specific regularities allowing humans to solve concept recognition problems despite the fact of their high information complexity.

3.1.3. The Problem of Studying the Human Concept Learning

In the majority of known studies, the cognitive complexity of Boolean concepts is assessed quantitatively, by using various computational models. Each of the models is associated with a corresponding computational complexity of the Boolean concepts. These methods/models, being mathematically different, are intended to achieve the same goal: to qualitatively reflect human ability to recognize, describe, analyze and synthesize systems corresponding to the Boolean concepts.

The methodology presented in this section aims to go deeper in the above research rational. In our opinion, BCL has to be studied by taking into account a combination of factors, which relate to:

1. non-exactness of human reasoning in recognition of Boolean concepts, as well as in ways of human approximation of Boolean concepts

2. recognizing properties of Boolean concepts by humans

3. study of different types of Boolean tasks to be solved, such as: recognition and reverse engineering.

The factors are presented in the above order to reflect the role of the first factor *non-exactness* as the most important due to its methodological value. According to our approach, both other factors have to be studied by using the methodology of non-exact reasoning in solving BCL tasks.

These new principles are based on considering a BCL task as a non-exact task that requires non-exact reasoning for solving it. A non-exact or approximate reasoning of a learner may very often reflect his/her generalized understanding of a Boolean concept. The corresponding non-exact solution has to be considered as at least better than not understanding the concept at all, but sometimes - even better than the accurate understanding of the concept. By studying the non-exact reasoning, the cognitive complexity of Boolean concepts can be understood deeper than in previous studies.

By using the suggested approach, it would be possible to answer the following important question:

> *How and in which proportion the two kinds of computational complexity (information and algorithmic) should serve as measures of the cognitive complexity?*

For this purpose, the study can be conducted by suggesting two types of tasks to the learners. These two types of tasks are:

1. recognition tasks, and

2. reverse engineering tasks.

Using these two types of tasks allows studying and estimating the cognitive complexity of Boolean concepts deeper than it was done before.

Another important research question that can be answered by using the proposed methodology is the following.

What is the correlation between solutions of the same Boolean problems being defined in two different forms according to the two types of tasks?

3.1.4. New Methodology for Studying Human Learning of Boolean Concepts

Three-dimensional Model. In our approach based on a three-dimensional model, where axes of the model correspond to the mentioned groups of factors:

- the first axis of the model will be formed by non-exactness of human reasoning,
- the second axis will reflect properties of Boolean functions affecting human learning, and
- the third one will correspond to the type of a Boolean task (such as: recognition, reverse engineering) proposed to the learners.

the first axis of the model will be formed by non-exactness of human reasoning, the second axis will reflect properties of Boolean functions affecting human learning; and the third one will correspond to the type of a Boolean task (such as: recognition, reverse engineering) proposed to the learners.

Non-exact Reasoning and Approximation in BCL. To date, the research in the field of Human Concept Learning was concentrated on studying cognitive complexity by using rather simple logic. In the case that a Boolean concept can be presented as a simple logic expression, the majority of learners finds this simple solution and, consequently, the complexity of the task is evaluated as low. In the opposite case, where the Boolean expression is complex (for example, comprises a great number of literals) the majority of the learners fails to find a solution. They either provide a wrong solution, or do not give any solution at all; in the best case the student(s) find(s) a correct solution but with a high value of latency. Such tasks are evaluated as cognitively complex.

It should be noted that in many BCL tasks the situation is much more complex. A BCL task often has a number of different solutions. The best (exact, minimal) solution exists among these solutions and, most probably, it satisfies the criterion of minimal complexity (Feldman). However, alternative solutions very often reflect important properties of human cognition, and they should not be considered as wrong. It may so happen that a cognitively simple solution, which we actually are looking for, can be found among non-minimal solutions. Such *cognitively* simple, though sometimes non-trivial solutions may be those which comprise properties of Boolean concepts that have been recognized by a learner. Such solutions are definitely interesting for scientific research since they not only clarify the term *cognitive complexity*, but also may shed light on some more intimate properties of human cognition (intuition, creativity).

Moreover, among *wrong* answers to complex BCL tasks, simple and elegant solutions can be found (though they are non-exact). It may happen that a learner preferred the simplicity and the elegance of the solution to its exactness. The simple answer may be not so far from the correct one in its Boolean sense: i.e., the two answers may coincide in the major correct points (nodes) of the Boolean space. For example, the learner, being asked to recognize a Boolean function

$$f = x\,y\,z \vee x\,\overline{y}\,z \vee \overline{x}\,y\,z$$

of three variables, can give a wrong but short answer $f = z$: adding only one point (node) $(\overline{x}\,\overline{y}\,z)$ to the ON-set of the initial function. In such an example the student has actually rounded off the answer by approximating it, and that action is close to such an important function of the human intellect as generalization.

Our approach is based on the following basic ideas:

- one should not reject all non-minimal solutions as cognitively complex human's approximation of Boolean concepts
- one should not consider any non-exact (approximate) solution as incorrect
- one should develop new metrics of cognitive complexity, taking

into account properties of a non-minimal and/or an approximate.

One of the main goals of our methodology is to provide the study of principles of the non-exactness in BCL and ways of approximation made by humans. For this aim, it is important to present several data sets to the learners, and ask them to generalize each set according to some rule. For example, the set Bongard Problems [39, 107] can be used as data sets. Each of Bongard Problems comprises two diagrams: BP_{right} and BP_{left}. The set of elements of the BP_{right} diagram has a common characteristic, which does not exist in the BP_{left} diagram. The learners have to find and to formulate a convincing distinction rule between both sides.

There are 100 Bongard problems (BPs), ordered by their difficulty. For each Bongard Problem the learner is told to describe a specific common attribute (derived rule) for the items of BP_{right}, and to do a similar thing for BP_{left}, such that both sides are dissimilar. In order to analyze the thought process, each learner will be asked to describe a way of reaching a final conclusion. The point is to understand not only the distinction defined by the learner, but also to be able to trace what other items in the BP were perceived. In this fashion, we retrace the structures evident to the learner and understand his/her *simplification process* through Boolean analysis. The presence of both BP_{right} and BP_{left} in the Bongard problems is significant and even scientifically more correct than in the standard case based on presenting just positive *what is yes*? patterns, since it allows discussing not only *what is yes*? but also *what is no*? in the patterns characteristics [180]. It also allows evaluating the learners solutions more accurate than in a standard case. However, exactly in such Bongard-like dually formulated tasks, there is great place of non-exactness.

In [179], we demonstrated that usually Boolean functions corresponding to Bongard problems contain considerable redundancy which in turn allows obtaining a number of correct but different solutions of the same task. Study of redundancy contained in Boolean descriptions opens a way for understanding the phenomenon of non-exactness in BCL. Specifically, the redundancy inserts a potential level of subjectivity to any formulation of a unifying or a distinguishing rule. Some of our preliminary results in this direction are published in [179, 180].

Recognizing Properties of Boolean Functions. Several properties of Boolean functions have to be studied in the BCL in order to investigate the human ability to recognize some features of the Boolean functions, and as a result - their ability to reduce the cognitive complexity of the problem. Such properties as monotony, linearity, and symmetry of Boolean concepts have to be studied in the above context.

It is assumed that learning and understanding a certain concept is connected with identification of patterns and regularities in their characteristics. This connection can be measured using an analytical function that applies a series of learner solutions of a Boolean task.

Our methodology is intended to study the learner's ability to solve logic tasks formulated by a set of cards. Each of the cards represents from one to three contours. The cards differ from each other by one or two characteristics from the following three: size, fill, and number of the contours. For our experiments, we choose a set of cards which are in one-to-one correspondence with nodes of a specific algebraic structure: Boolean Cube, Lattice, etc. For example, the Set game can be used for our purpose. The correlation between specific regularities in subsets of cards on the one hand, and students' success in solving logic tasks defined by the subsets on the other hand should be studied. It can be carried out by:

1. measuring the complexity using a specific analytical function expressing regularities

2. measuring the complexity by using known approaches, mostly connected with the number of literals, and

3. an empirical method, by analyzing the corresponding students answers.

Recognition vs. Reverse Engineering in BCL. As it was mentioned above, a number of different types of Boolean problems exist, which types have to be differently and separately studied in the context of Boolean Concepts Learning. These types of problems correspond to different kinds of human thinking activities, such as analytic or synthetic thinking.

Usually the problem of Human Concept Learning is considered as the problem of recognizing relations between visually represented objects. Solving the recognition problem/task may thus be interpreted as recognizing a visually represented Boolean concept, with further formulation of the concept. The recognition problems can be perceived as a parallel process, so the recognition problems are considered as problems of a parallel type. Recognizing patterns is one of the important functions of human consciousness. The concept of cognitive complexity usually relates to complexity of the recognition task. Nevertheless, there is a number of Boolean tasks of different types the cognitive complexity of which also requires estimation. Specifically, such types of problems as recognition, reverse engineering and fault detection are all of a special interest. The process of finding and reconstructing operating mechanisms in a given functional system of a digital electronic device is defined as *Reverse Engineering* (RE) [60]. RE is applied in a wide variety of fields: competition in manufacturing new products, from electronic components to cars, among competing companies without infringing upon the competing company's copyright, replacing system components with refurbished or spare parts [145], solving problems and defects in a system [93]. RE can be referred to as a certain type of problem solving.

In the BCL case the RE problem is a problem of reconstructing a Boolean function implemented within a given black box. Since such a reconstruction is typically performed sequentially, step-by-step, this problem is considered to be of a sequential type. While the recognition task is usually presented as a pattern containing geometric shapes, the RE task is usually presented in a form of a blackbox that can be used to control via it (there-through) a bulb, by using independent switches.

Our methodology is based on studying both the *Recognition* and the *Reverse Engineering* types of problems. It is based on conducting experiments on the Boolean tasks in two scenarios: recognition and reverse engineering and compare the corresponding students solutions. To make it possible, two environments should be developed: a standard one for recognition tasks, and another one in the form of *black box* comprising some switches and a bulb for the reverse engineering tasks. Boolean functions for the recognizing and for the reverse engineering analysis should be presented in two separate groups of

students. The two groups may then exchange the environments.

We hypothesize that the study of BCL on different types of problems allows broadening the field of expertise and bringing an additional dimension to the concept of cognitive complexity. Our preliminary results [274] indicate a correlation between complexities of problems of different types for the same Boolean concept.

3.1.5. Research Methodology

We have presented a new research methodology in the field of Human Concept Learning, more specifically in learning Boolean concepts. The main idea of the methodology can be expressed by using a three-dimensional model for estimating cognitive complexity of learning, where axes of the model correspond to three directions of the research: non-exactness of human reasoning when solving Boolean tasks, properties of Boolean functions affecting the learning of Boolean concepts; and different types of Boolean tasks (recognition and reverse engineering tasks) presented to the learners.

The above three-dimensional research model was never proposed before, though considerable attempts were undertaken to study mechanisms of Human Concept Learning. The aim of the section is to view the problem at a new angle, by adding the qualitative character to the research. We believe that using the non-exactness as a methodological basis to study principles of Boolean concept learning by humans will provide deeper understanding of how the concepts are learned, and will open a way for future research in the field of Human Concept Learning.

3.2. Generalized Complexity of $\mathcal{ALC}$ Subsumption

Arne Meier

Description logics (DL) play an important role in several areas of research, e.g., semantic web, database structuring, or medical ontologies [15, 43, 44, 220]. As a consequence there exists a vast range of different extensions where each of them is highly specialized to its own field of application. Nardi and Brachman describe *subsumption* as the most important inference problem within DL [223]. Given two (w.l.o.g. atomic) concepts C, D and a set of axioms (which are pairs of concept expressions A, B stating A implies B), one asks the question whether C implies D is consistent with respect to each model satisfying all of the given axioms. Although the computational complexity of subsumption in general can be between tractable and EXP [87, 88] depending on which feature[1] is available, there exist nonetheless many DL which provide a tractable (i.e., in P) subsumption reasoning problem, e.g., the DL-Lite and $\mathcal{EL}$ families [10, 11, 13, 14, 57]. One very prominent application example of the subsumption problem is the SNOMED CT clinical database which consists of about 400.000 axioms and is a subset of the DL $\mathcal{EL}^{++}$ [65, 251].

In this section we investigate the subsumption problem with respect to the most general (in sense of available Boolean operators) description logic $\mathcal{ALC}$. It is known that the unrestricted version of this problem is EXP-complete due to reducibility to a specific DL satisfiability problem [15], and is therefore highly intractable. Our aim is to understand where this intractability comes from or to which Boolean operator it may be connected to. Therefore we will make use of the well understood and much used algebraic tool, Post's lattice [239]. At this approach one works with *clones* which are defined as sets of Boolean functions which are closed under arbitrary composition and projection. For a good introduction into this area consider [37]. The

[1] These features, for instance, can be existential or universal restrictions, availability of disjunction, conjunction, or negation. Furthermore, one may extend a description logic with more powerful concepts, e.g., number restrictions, role chains, or epistemic operators [15].

main technique is to investigate fragments of a specific decision problem by means of allowed Boolean functions; in this section this will be the subsumption problem. As Post's lattice considers any possible set of all Boolean functions, a classification by it always yields an exhaustive study. This kind of research has been done previously for several different kinds of logics, e.g., temporal, hybrid, modal, and non-monotonic logics [30, 70, 71, 133, 268, 269].

Main results. The most general class of fragments, i.e., those which have both quantifiers available, perfectly show how powerful the subsumption problem is. Having access to at least one constant (true or false) leads to an intractable fragment. Merely for the fragment where only projections (and none of the constants) are present it is not clear if there can be a polynomial time algorithm for this case and has been left open. If one considers the cases where only one quantifier is present, then the fragments around disjunction (case $\forall$), respectively, the ones around conjunction (case $\exists$) become tractable. Without quantifiers conjunctive and disjunctive fragments are P-complete whereas the fragments which include either the affine functions (exclusive or), or can express $x \vee (y \wedge z)$, or $x \wedge (y \vee z)$, or self-dual functions (i.e., $f(x_1, \ldots, x_n) = \overline{f(\overline{x}_1, \ldots, \overline{x}_n)}$) are intractable. Figure 3.1 depicts how the results directly arrange within Post's lattice. Due to space restrictions the proof of Theorem 3.10 is omitted and can be found in the technical report [194].

3.2.1. Preliminaries

In this section we will make use of standard notions of complexity theory [234]. In particular, we work with the classes NLOGSPACE, P, coNP, EXP, and the class $\oplus$LOGSPACE which corresponds to nondeterministic Turing machines running in logarithmic space whose computations trees have an odd number of accepting paths. Usually all stated reductions are logarithmic space many-one reductions $\leq_{\mathbf{m}}^{\mathbf{log}}$. We write $A \equiv_{\mathbf{m}}^{\mathbf{log}} B$ iff $A \leq_{\mathbf{m}}^{\mathbf{log}} B$ and $B \leq_{\mathbf{m}}^{\mathbf{log}} A$ hold.

Post's Lattice. Let $\top$, $\bot$ denote the truth values true, false. Given a finite set of Boolean functions B, we say the *clone of* B contains

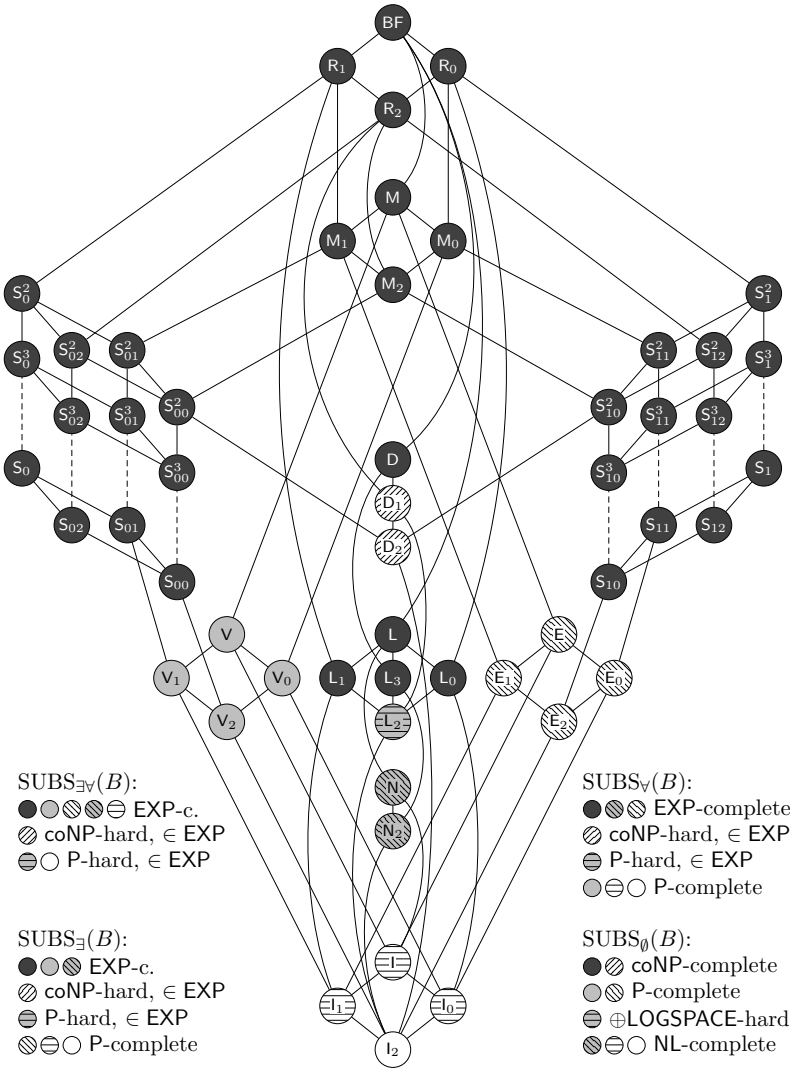


Figure 3.1. Post's lattice showing the complexity of $\mathrm{SUBS}_{\mathcal{Q}}(B)$ for all sets $\emptyset \subseteq \mathcal{Q} \subseteq \{\exists, \forall\}$ and all Boolean clones $[B]$.

Table 3.1. All clones and bases relevant for this classification

	Base		Base		Base
BF	$\{x \wedge y, \overline{x}\}$	S_{00}	$\{x \vee (y \wedge z)\}$	S_{10}	$\{x \wedge (y \vee z)\}$
D_1	$\{\mathrm{maj}\{x, y, \overline{z}\}\}$	D_2	$\{\mathrm{maj}\{x, y, z\}\}$	M_0	$\{x \wedge y, x \vee y, \bot\}$
L	$\{x \oplus y, \top\}$	L_0	$\{x \oplus y\}$	L_1	$\{x \leftrightarrow y\}$
L_2	$\{x \oplus y \oplus z\}$	L_3	$\{x \oplus y \oplus z \oplus \top\}$		
V	$\{x \vee y, \top, \bot\}$	V_0	$\{x \vee y, \bot\}$	V_2	$\{x \vee y\}$
E	$\{x \wedge y, \top, \bot\}$	E_0	$\{x \wedge y, \bot\}$	E_2	$\{x \wedge y\}$
N	$\{\overline{x}, \top\}$	N_2	$\{\overline{x}\}$		
I_0	$\{\mathrm{id}, \bot\}$	I_1	$\{\mathrm{id}, \top\}$	I_2	$\{\mathrm{id}\}$

all compositions of functions in B plus all projections; the smallest such clone is denoted with $[B]$ and the set B is called a *base of* $[B]$. The lattice of all clones has been established in [239] and a much more succinct introduction can be found in [37]. Table 3.1 depicts all clones and their bases which are relevant for this classification. Here maj denotes the majority, and id denotes identity. Let $f\colon \{\top, \bot\}^n \to \{\top, \bot\}$ be a Boolean function. Then the *dual of* f, in symbols $\mathbf{dual}(f)$, is the n-ary function g with $g(x_1, \ldots, x_n) = \overline{f(\overline{x}_1, \ldots, \overline{x}_n)}$. Similarly, if B is a set of Boolean functions, then $\mathbf{dual}(B) := \{\mathbf{dual}(f) \mid f \in B\}$. Further, abusing notation, define $\mathbf{dual}(\exists) := \forall$ and $\mathbf{dual}(\forall) = \exists$; if $\mathcal{Q} \subseteq \{\exists, \forall\}$ then $\mathbf{dual}(\mathcal{Q}) := \{\mathbf{dual}(\ni) \mid \ni \in \mathcal{Q}\}$.

Description Logic. Usually a Boolean function f is defined as mappings $f\colon \{0,1\}^n \to \{0,1\}$ wherefore the appearance within formulas may not be well-defined. Therefore we will use the term of a Boolean *operator* whenever we talk about the to a function corresponding part within a concept description. This approach extends the upper definition of clones to comprise cover operators as well.

We use the standard syntax and semantics of $\mathcal{ALC}$ as in [15]. Additionally we adjusted them to fit the notion of clones. The set of *concept descriptions* (or *concepts*) is defined by

$$C := A \mid \circ_f(C, \ldots, C) \mid \exists R.C \mid \forall R.C \ ,$$

where A is an atomic concept (variable), R is a role (transition relation), and $\circ_f$ is a Boolean operator which corresponds to a Boolean

function $f\colon \{\top,\bot\}^n \to \{\top,\bot\}$. For a given set B of Boolean operators and $\mathcal{Q} \subseteq \{\exists,\forall\}$, we define that a *$B$-$\mathcal{Q}$-concept* uses only operators from B and quantifiers from $\mathcal{Q}$. Hence, if $B = \{\wedge,\vee\}$ then $[B] = \mathsf{BF}$, and the set of B-concept descriptions is equivalent to (full) $\mathcal{ALC}$. Otherwise if $[B] \subsetneq \mathsf{BF}$ for some set B, then we consider proper subsets of $\mathcal{ALC}$ and cannot express any (usually in $\mathcal{ALC}$ available) concept.

An *axiom* is of the form $C \sqsubseteq D$, where C and D are concepts; $C \equiv D$ is the syntactic sugar for $C \sqsubseteq D$ and $D \sqsubseteq C$. A *TBox* is a finite set of axioms and a B-$\mathcal{Q}$-TBox contains only axioms of B-$\mathcal{Q}$-concepts.

An *interpretation* is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a nonempty set and $\cdot^{\mathcal{I}}$ is a mapping from the set of atomic concepts to the power set of $\Delta^{\mathcal{I}}$, and from the set of roles to the power set of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. We extend this mapping to arbitrary concepts as follows:

$$
\begin{aligned}
(\exists R.C)^{\mathcal{I}} &= \left\{x \in \Delta^{\mathcal{I}} \,\middle|\, \{y \in C^{\mathcal{I}} \mid (x,y) \in R^{\mathcal{I}}\} \neq \emptyset\right\} , \\
(\forall R.C)^{\mathcal{I}} &= \left\{x \in \Delta^{\mathcal{I}} \,\middle|\, \{y \in C^{\mathcal{I}} \mid (x,y) \notin R^{\mathcal{I}}\} = \emptyset\right\} , \\
\left(\circ_f (C_1,\ldots,C_n)\right)^{\mathcal{I}} &= \left\{x \in \Delta^{\mathcal{I}} \,\middle|\, f(||x \in C_1^{\mathcal{I}}||,\ldots,||x \in C_n^{\mathcal{I}}||) = \top\right\} ,
\end{aligned}
$$

where $||x \in C^{\mathcal{I}}|| = 1$ if $x \in C^{\mathcal{I}}$ and $||x \in C^{\mathcal{I}}|| = 0$ if $x \notin C^{\mathcal{I}}$. An interpretation $\mathcal{I}$ *satisfies* the axiom $C \sqsubseteq D$, in symbols $\mathcal{I} \models C \sqsubseteq D$, if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Further $\mathcal{I}$ *satisfies a TBox*, in symbols $\mathcal{I} \models \mathcal{T}$, if it satisfies every axiom therein; then $\mathcal{I}$ is called a *model*.

Let $\mathcal{Q} \subseteq \{\exists,\forall\}$ and B be a finite set of Boolean operators. Then for the *TBox-concept satisfiability problem,* $\mathrm{TCSAT}_{\mathcal{Q}}(B)$, given a B-$\mathcal{Q}$-TBox $\mathcal{T}$ and a B-$\mathcal{Q}$-concept C, one asks if there is an $\mathcal{I}$ s.t. $\mathcal{I} \models \mathcal{T}$ and $C^{\mathcal{I}} \neq \emptyset$. This problem has been fully classified w.r.t. Post's lattice in [196]. Further the *Subsumption problem,* $\mathrm{SUBS}_{\mathcal{Q}}(B)$, given a B-$\mathcal{Q}$-TBox and two B-$\mathcal{Q}$-concepts C, D, asks if for every interpretation $\mathcal{I}$ it holds that $\mathcal{I} \models \mathcal{T}$ implies $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

As subsumption is an inference problem within DL some kind of connection in terms of reductions to propositional implication is not devious. In [29] Beyersdorff et al. classify the propositional implication problem IMP with respect to all fragments parameterized by all Boolean clones.

Theorem 3.8 ([29]). *Let B be a finite set of Boolean operators.*

1. *If $C \subseteq [B]$ for $C \in \{\mathsf{S}_{00}, \mathsf{D}_2, \mathsf{S}_{10}\}$, then* $\mathrm{IMP}(B)$ *is* coNP*-complete w.r.t. $\leq_m^{\mathsf{AC}^0}$*[2].

2. *If $\mathsf{L}_2 \subseteq [B] \subseteq \mathsf{L}$, then* $\mathrm{IMP}(B)$ *is* $\oplus$LOGSPACE*-complete w.r.t. $\leq_m^{\mathsf{AC}^0}$.*

3. *If $\mathsf{N}_2 \subseteq [B] \subseteq \mathsf{N}$, then* $\mathrm{IMP}(B)$ *is in* $\mathsf{AC}^0[2]$.

4. *Otherwise* $\mathrm{IMP}(B) \in \mathsf{AC}^0$.

3.2.2. Interreducibilities

The next lemma shows base independence for the subsumption problem. This kind of property enables us to use standard bases for every clone within our proofs. The result can be proven in the same way as in [195, Lemma 4].

Lemma 3.1. *Let B_1, B_2 be two sets of Boolean operators such that $[B_1] \subseteq [B_2]$, and let $\mathcal{Q} \subseteq \{\exists, \forall\}$. Then* $\mathrm{SUBS}_{\mathcal{Q}}(B_1) \leq_{\mathbf{m}}^{\mathbf{log}} \mathrm{SUBS}_{\mathcal{Q}}(B_2)$.

The following two lemmas deal with a duality principle of subsumption. The correctness of contraposition for axioms allows us to state a reduction to the fragment parameterized by the dual operators. Further having access to negation allows us in the same way as in [197] to simulate both constants.

Lemma 3.2. *Let B be a finite set of Boolean operators and $\mathcal{Q} \subseteq \{\forall, \exists\}$. Then* $\mathrm{SUBS}_{\mathcal{Q}}(B) \leq_{\mathbf{m}}^{\mathbf{log}} \mathrm{SUBS}_{\mathbf{dual}(\mathcal{Q})}(\mathbf{dual}(B))$.

Proof. Here we distinguish two cases. Given a concept A define with $A^{\neg}$ the concept $\neg A$ in negation normal form (NNF).

[2] A language A is AC^0 many-one reducible to a language B ($A \leq_m^{\mathsf{AC}^0} B$) if there exists a function f computable by a logtime-uniform AC^0-circuit familiy such that $x \in A$ iff $f(x) \in B$ (for more information, see [330]).

First assume that $\neg \in [B]$. Then $(\mathcal{T}, C, D) \in \mathrm{SUBS}_{\mathcal{Q}}(B)$ if and only if for any interpretation $\mathcal{I}$ s.t. $\mathcal{I} \models \mathcal{T}$ it holds that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ if and only if for any interpretation $\mathcal{I}$ s.t. $\mathcal{I} \models \mathcal{T}' := \{F^{\neg} \sqsubseteq E^{\neg} \mid E \sqsubseteq F \in \mathcal{T}\}$ it holds that $(\neg D)^{\mathcal{I}} \subseteq (\neg C)^{\mathcal{I}}$ if and only if $(\mathcal{T}', D^{\neg}, C^{\neg}) \in \mathrm{SUBS}_{\mathbf{dual}(\mathcal{Q})}(\mathbf{dual}(B))$. The correctness directly follows from $\mathbf{dual}(\neg) = \neg$.

Now assume that $\neg \notin [B]$. Then for a given instance $(\mathcal{T}, C, D)$ it holds that for the contraposition instance $(\{F^{\neg} \sqsubseteq E^{\neg} \mid E \sqsubseteq F \in \mathcal{T}\}, D^{\neg}, C^{\neg})$ before every atomic concept occurs a negation symbol. Denote with $(\{F^{\neg} \sqsubseteq E^{\neg} \mid E \sqsubseteq F \in \mathcal{T}\}, D^{\neg}, C^{\neg})^{\mathrm{pos}}$ the substitution of any such negated atomic concept $\neg A$ by a fresh concept name A'. Then $(\mathcal{T}, C, D) \in \mathrm{SUBS}_{\mathcal{Q}}(B)$ iff $(\{F^{\neg} \sqsubseteq E^{\neg} \mid E \sqsubseteq F \in \mathcal{T}\}, D^{\neg}, C^{\neg})^{\mathrm{pos}} \in \mathrm{SUBS}_{\mathbf{dual}(\mathcal{Q})}(\mathbf{dual}(B))$. □

Lemma 3.3. *Let B be a finite set of Boolean operators such that $\mathsf{N}_2 \subseteq [B]$ and $\mathcal{Q} \subseteq \{\exists, \forall\}$. Then it holds that $\mathrm{SUBS}_{\mathcal{Q}}(B) \equiv^{\mathbf{log}}_{\mathbf{m}} \mathrm{SUBS}_{\mathcal{Q}}(B \cup \{\top, \bot\})$.*

Using Lemma 4.2 in [29] we can easily obtain the ability to express the constant $\top$ whenever we have access to conjunctions, and the constant $\bot$ whenever we are able to use disjunctions.

Lemma 3.4. *Let B be a finite set of Boolean operators and $\mathcal{Q} \subseteq \{\forall, \exists\}$. If $\mathsf{E}_2 \subseteq [B]$, then $\mathrm{SUBS}_{\mathcal{Q}}(B) \equiv^{\mathbf{log}}_{\mathbf{m}} \mathrm{SUBS}_{\mathcal{Q}}(B \cup \{\top\})$. If $\mathsf{V}_2 \subseteq [B]$, then $\mathrm{SUBS}_{\mathcal{Q}}(B) \equiv^{\mathbf{log}}_{\mathbf{m}} \mathrm{SUBS}_{\mathcal{Q}}(B \cup \{\bot\})$.*

The connection of subsumption to the terminology of satisfiability and propositional implication is crucial for stating upper and lower bound results. The next lemma connects subsumption to TCSAT and also to IMP.

Lemma 3.5. *Let B be a finite set of Boolean operators and $\mathcal{Q} \subseteq \{\forall, \exists\}$ be a set of quantifiers. Then the following reductions hold:*

1. $\mathrm{IMP}(B) \leq^{\mathbf{log}}_{\mathbf{m}} \mathrm{SUBS}_{\emptyset}(B)$.
2. $\mathrm{SUBS}_{\mathcal{Q}}(B) \leq^{\mathbf{log}}_{\mathbf{m}} \overline{\mathrm{TCSAT}_{\mathcal{Q}}(B \cup \{\nrightarrow\})}$.
3. $\overline{\mathrm{TCSAT}_{\mathcal{Q}}(B)} \leq^{\mathbf{log}}_{\mathbf{m}} \mathrm{SUBS}_{\mathcal{Q}}(B \cup \{\bot\})$.

Proof.

1. It holds $(\varphi, \psi) \in \mathrm{IMP}(B)$ iff $(C_\varphi, C_\psi, \emptyset) \in \mathrm{SUBS}_\emptyset(B)$, for concept descriptions $C_\varphi = f(\varphi), C_\psi = f(\psi)$ with f mapping propositional formulas to concept descriptions via

$$\begin{aligned} f(\top) &= \top, \text{ and } f(\bot) = \bot, \\ f(x) &= C_x, \text{ for variable } x, \\ f(g(C_1, \ldots, C_n)) &= \circ_g(f(C_1), \ldots, f(C_n)) \end{aligned}$$

   where g is an n-ary Boolean function and $\circ_g$ is the corresponding operator.

2. $(\mathcal{T}, C, D) \in \mathrm{SUBS}_\mathcal{Q}(B)$ iff $(\mathcal{T}, C \mathrel{\not\rightarrow} D) \in \overline{\mathrm{TCSAT}_\mathcal{Q}(B \cup \{\mathrel{\not\rightarrow}\})}$ [15].

3. $(\mathcal{T}, C) \in \overline{\mathrm{TCSAT}_\mathcal{Q}(B)}$ iff $(C, \bot, \mathcal{T}) \in \mathrm{SUBS}_\mathcal{Q}(B \cup \{\bot\})$ [15].

□

3.2.3. Main Results

We will start with the subsumption problem using no quantifiers and will show that the problem either is coNP-, P-, NLOGSPACE-complete, or is $\oplus$LOGSPACE-hard.

Theorem 3.9 (No quantifiers available.)**.** *Let B be a finite set of Boolean operators.*

1. *If $X \subseteq [B]$ for $X \in \{\mathsf{L}_0, \mathsf{L}_1, \mathsf{L}_3, \mathsf{S}_{10}, \mathsf{S}_{00}, \mathsf{D}_2\}$, then $\mathrm{SUBS}_\emptyset(B)$ is* coNP*-complete.*

2. *If $\mathsf{E}_2 \subseteq [B] \subseteq \mathsf{E}$ or $\mathsf{V}_2 \subseteq [B] \subseteq \mathsf{V}$, then $\mathrm{SUBS}_\emptyset(B)$ is* P*-complete.*

3. *If $[B] = \mathsf{L}_2$, then $\mathrm{SUBS}_\emptyset(B)$ is $\oplus$*LOGSPACE*-hard.*

4. *If $\mathsf{I}_2 \subseteq [B] \subseteq \mathsf{N}$, then $\mathrm{SUBS}_\emptyset(B)$ is* NLOGSPACE*-complete.*

All hardness results hold w.r.t. $\leq_\mathbf{m}^\mathbf{log}$ reductions.

Proof.

1. The reduction from the implication problem IMP(B) in Lemma 3.5 (1.) in combination with Theorem 3.8 and Lemma 3.1 proves the coNP lower bounds of $\mathsf{S}_{10}, \mathsf{S}_{00}, \mathsf{D}_2$. The lower bounds for $\mathsf{L}_0 \subseteq [B]$ and $\mathsf{L}_3 \subseteq [B]$ follow from Lemma 3.5 (3.) with $\overline{\mathrm{TCSAT}_\emptyset(B)}$ being coNP-complete which follows from the NP-completeness result of $\mathrm{TCSAT}_\emptyset(B)$ shown in [195, Theorem 27]. Further the lower bound for $\mathsf{L}_1 \subseteq [B]$ follows from the duality of '$\oplus$' and '$\equiv$' and Lemma 3.2 with respect to the case $\mathsf{L}_0 \subseteq [B]$ enables us to state the reduction

$$\mathrm{SUBS}_\emptyset(\mathsf{L}_0) \leq_{\mathbf{m}}^{\mathbf{log}} \mathrm{SUBS}_{\mathbf{dual}(\emptyset)}(\mathbf{dual}(\mathsf{L}_0)) = \mathrm{SUBS}_\emptyset(\mathsf{L}_1) \ .$$

   The upper bound follows from a reduction to $\overline{\mathrm{TCSAT}_\emptyset(\mathsf{BF})}$ by Lemma 3.5 (2.) and the membership of $\mathrm{TCSAT}_\emptyset(\mathsf{BF})$ in NP by [195, Theorem 27].

2. The upper bound follows from the memberships in P for the fragments $\mathrm{SUBS}_\exists(\mathsf{E})$ and $\mathrm{SUBS}_\forall(\mathsf{V})$ in Theorem 3.10.

   The lower bound for $[B] = \mathsf{E}_2$ follows from a reduction from the hypergraph accessibility problem[3] HGAP: set $\mathcal{T} = \{u_1 \sqcap u_2 \sqsubseteq v \mid (u_1, u_2; v) \in E\}$, assume w.l.o.g. the set of source nodes as $S = \{s\}$, then $(G, S, t) \in$ HGAP iff $(\mathcal{T}, s, t) \in \mathrm{SUBS}_\emptyset(\mathsf{E}_2)$. For the lower bound of V_2 apply Lemma 3.2.

3. Follows directly by the reduction from IMP(L_2) due to Theorem 3.8 and Lemma 3.5 (1.).

4. For the lower bound we show a reduction from the graph accessibility problem[4] GAP to $\mathrm{SUBS}_\emptyset(\mathsf{I}_2)$. Let $G = (V, E)$ be a

[3] In a given hypergraph $H = (V, E)$, a hyperedge $e \in E$ is a pair of source nodes $src(e) \in V \times V$ and one destination node $dest(e) \in V$. Instances of HGAP consist of a directed hypergraph $H = (V, E)$, a set $S \subseteq V$ of source nodes, and a target node $t \in V$. Now the question is whether there exists a hyperpath from the set S to the node t, i.e., whether there are hyperedges $e_1, e_2, \ldots, e_k$ such that, for each e_i, there are $e_{i_1}, \ldots, e_{i_\nu}$ with $1 \leq i_1, \ldots, i_\nu < i$ and $\bigcup_{j \in \{i_1, \ldots, i_\nu\}} dest(e_j) \cup src(e_j) \supseteq src(e_i)$, and $src(e_1) = S$ and $dest(e_k) = t$ [283].

[4] Instances of GAP are directed graphs G together with two nodes s, t in G asking whether there is a path from s to t in G.

undirected graph and $s, t \in V$ be the vertices for the input. Then for $\mathcal{T} := \{(A_u \sqsubseteq A_v) \mid (u, v) \in E\}$ it holds that $(G, s, t) \in \text{GAP}$ iff $(\mathcal{T}, A_s, A_t) \in \text{SUBS}_{\emptyset}(\mathsf{I}_2)$.

For the upper bound we follow the idea from [195, Lemma 29]. Given the input instance $(\mathcal{T}, C, D)$ we can similarly assume that for each $E \sqsubseteq F \in \mathcal{T}$ it holds that E, F are atomic concepts, or their negations, or constants. Now $(\mathcal{T}, C, D) \in \text{SUBS}_{\emptyset}(\mathsf{N})$ holds iff for every interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ and $x \in \Delta^{\mathcal{I}}$ it holds that if $x \in C^{\mathcal{I}}$ then $x \in D^{\mathcal{I}}$ holds iff for the implication graph $G_{\mathcal{T}}$ (constructed as in [195, Lemma 29]) there exists a path from v_C to v_D.

Informally if there is no path from v_C to v_D then D is not implied by C, i.e., it is possible to construct an interpretation for which there exists an individual which is a member of $C^{\mathcal{I}}$ but not of $D^{\mathcal{I}}$.

Thus we have provided a coNLOGSPACE-algorithm which first checks accordingly to the algorithm in [195, Lemma 29] if there are not any cycles containing contradictory axioms. Then we verify that there is no path from v_C to v_D implying that C is not subsumed by D.

□

Using some results from the previous theorem we are now able to classify most fragments of the subsumption problem using only either the $\forall$ or $\exists$ quantifier with respect to all possible Boolean clones in the following two theorems.

Theorem 3.10 (Restricted fragments)**.** *Let B be a finite set of Boolean operators, $\supset \in \{\exists, \forall\}$.*

1. *If $C \subseteq [B]$ for $C \in \{\mathsf{N}_2, \mathsf{L}_0, \mathsf{L}_1\}$, then $\text{SUBS}_{\supset}(B)$ is* EXP*-complete.*

2. *If $C \subseteq [B]$ for $C \in \{\mathsf{E}_2, \mathsf{S}_{00}\}$, then $\text{SUBS}_{\forall}(B)$ is* EXP*-complete.*

3. *If* $C \subseteq [B]$ *for* $C \in \{\mathsf{V}_2, \mathsf{S}_{10}\}$, *then* $\mathrm{SUBS}_{\exists}(B)$ *is* EXP*-complete.*

4. *If* $\mathsf{D}_2 \subseteq [B] \subseteq \mathsf{D}_1$, *then* $\mathrm{SUBS}_{\supset}(B)$ *is* coNP*-hard and in* EXP.

5. *If* $[B] = \mathsf{L}_2$, *then* $\mathrm{SUBS}_{\supset}(B)$ *is* P*-hard and in* EXP.

6. *If* $[B] \subseteq \mathsf{V}$, *then* $\mathrm{SUBS}_{\forall}(B)$ *is* P*-complete; if* $[B] \subseteq \mathsf{E}$, *then* $\mathrm{SUBS}_{\exists}(B)$ *is* P*-complete*

All hardness results hold w.r.t. $\leq_{\mathbf{m}}^{\mathbf{log}}$ *reductions.*

Proof. Theorem 3.10 is proven in [194]. □

Finally the classification of the full quantifier fragments naturally emerges from the previous cases to EXP-complete, coNP-, and P-hard cases.

Theorem 3.11 (Both quantifiers available)**.** *Let* B *be a finite set of Boolean operators.*

1. *Let* $X \in \{\mathsf{N}_2, \mathsf{V}_2, \mathsf{E}_2\}$. *If* $X \subseteq [B]$, *then* $\mathrm{SUBS}_{\exists\forall}(B)$ *is* EXP*-complete.*

2. *If* $\mathsf{I}_0 \subseteq [B]$ *or* $\mathsf{I}_1 \subseteq [B]$, *then* $\mathrm{SUBS}_{\exists\forall}(B)$ *is* EXP*-complete.*

3. *If* $\mathsf{D}_2 \subseteq [B] \subseteq \mathsf{D}_1$, *then* $\mathrm{SUBS}_{\exists\forall}(B)$ *is* coNP*-hard and in* EXP.

4. *If* $[B] \in \{\mathsf{I}_2, \mathsf{L}_2\}$, *then* $\mathrm{SUBS}_{\exists\forall}(B)$ *is* P*-hard and in* EXP.

All hardness results hold w.r.t. $\leq_{\mathbf{m}}^{\mathbf{log}}$ *reductions.*

Proof.

1. Follows from the respective lower bounds of $\mathrm{SUBS}_{\exists}(B)$, resp., $\mathrm{SUBS}_{\forall}(B)$ in Theorem 3.10.

2. The needed lower bound follows from Lemma 3.5(3.) and enables a reduction from the EXP-complete problem $\overline{\mathrm{TCSAT}_{\exists\forall}(\mathsf{I}_0)}$

[196, Theorem 2 (1.)]. The case $\mathrm{SUBS}_{\exists\forall}(B)$ with $\mathsf{I}_1 \subseteq [B]$ follows from the contraposition argument in Lemma 3.2.

3.+4. The lower bounds carry over from $\mathrm{SUBS}_{\emptyset}(B)$ for the respective sets B (see Theorem 3.9).

□

3.2.4. Discussion of the Very Difficult Problem

The classification has shown that the subsumption problem with both quantifiers is a very difficult problem. Even a restriction down to only one of the constants leads to an intractable fragment with EXP-completeness. Although we achieved a P lower bound for the case without any constants, i.e., the clone I_2, it is not clear how to state a polynomial time algorithm for this case: We believe that the size of satisfying interpretations always can be polynomial in the size of the given TBox but a deterministic way to construct it is not obvious to us. The overall interaction of enforced concepts with possible roles is not clear (e.g., should a role edge be a loop or not). Further it is much harder to construct such an algorithm for the case L_2 having a ternary exclusive-or operator.

Retrospectively, the subsumption problem is much harder than the usual terminology satisfiability problems visited in [196]. Due to the duality principle expressed by Lemma 3.2 both halves of Post's lattice contain intractable fragments plus it is not clear if there is a tractable fragment at all. For the fragments having access to only one of the quantifiers the clones which are able to express either disjunction (for the universal quantifier) or conjunction (for the existential case) become tractable (plus both constants). Without any quantifier allowed the problem almost behaves as the propositional implication problem with respect to tractability. The only exception of this rule refers to the L-cases that can express negation or at least one constant. They become coNP-complete and therewith intractable.

Finally, a similar systematic study of the subsumption problem for concepts (without respect to a terminology) would be of great interest

because of the close relation to the implication problem of modal formulas. To the best of the author's knowledge, such a study has not been done yet and would enrich the overall picture of the complexity situation in this area of research. Furthermore, it would be interesting to study the effects of several restrictions on terminologies to our classification, e.g., acyclic or cyclic TBoxes.

3.3. Using a Reconfigurable Computer to Compute Algebraic Immunity

M. Eric McCay Jon T. Butler

Pantelimon Stănică

3.3.1. Why do we Need Algebraic Immunity?

The benefits we enjoy from the internet depend critically on our ability to communicate securely. For example, to bank online we depend on the encoding of plaintext messages (our personal identifications numbers, for instance), which, if divulged, could result in significant loss of personal funds. Unfortunately, the theft of credit/debit card numbers is one of the most common crimes in the internet age. Such a theft is especially enticing since the individual committing the theft is likely to be far away from the victim, therefore feeling immune to prosecution.

Encryption is a process of converting a plaintext (cleartext) message into a scrambled message, called *ciphertext*. With the internet as is typically configured now, actual interception of a message is easy; anyone with access to a server and some programming skills can easily acquire the communication. However, in the case of an encrypted message, it is necessary to then decrypt the message, and in so doing extract the original plaintext message. The present benefits of the internet depend critically on the degree of difficulty associated with the discovery of the plaintext message.

For example, one could encrypt a message by exclusive ORing a code (the same code) on each ASCII character in the message. In this case, 'A' is *always* transformed into some other letter, like 'K', 'B' is *always* transformed to say 'T', etc. Breaking such a code is easy, by using statistical analysis, since one can guess that the most common letter represents an 'E', while a rare letter represents a 'Z'. Although the guess may be wrong, with enough computing power and enough of the encrypted message, a dedicated thief would eventually succeed.

Present day encryption algorithms are significantly more sophisticated than this simple example. However, in addition to much better encryption algorithms, there has also been a significant increase in computing power. Indeed, we live in an era of the *supercomputer on the desk*, in which thousands of processors exist within the space that recently housed only one processor. Therefore, there is an enhanced ability to *try out* many guesses and to analyze huge quantities of ciphertext.

An attempt to decipher a message is called a (cryptographic) *attack*, or *cryptanalysis*. An attack is often predicated on a perceived weakness in the encoding algorithm. For example, one of the earliest attacks, called the linear attack, is most successful when the encoding algorithm uses a not too complicated approximation of a linear (or affine) Boolean function. A linear (affine) function is the exclusive OR of linear terms only (or their complement). For example, $f_1 = x_1 \oplus x_3 \oplus x_4$ is a linear function (both f_1 and $f_1 \oplus 1$ are affine), while $f_2 = x_1 x_2 \oplus x_3 x_4$ is not linear (nor affine as it has degree two). In a linear attack, the attacker takes advantage of an affine approximation of the action of the cipher. If the function used in some step of the cipher is linear or even 'close to linear', such an attack is likely to succeed. To mitigate against such attacks, Boolean functions that are highly nonlinear are used in the encryption instead. This led to interest in *bent* functions, Boolean functions whose nonlinearity exceeds that of all others. For a tutorial description of bent functions, see [55].

Since Rothaus' seminal contribution on bent functions in 1976 [250], there has been much work on the cryptographic properties of Boolean functions [73]. Such properties include strict avalanche criterion [106, 332], propagation criteria [241], and correlation immunity [279]. However, within the past 10 years an effective attack that uses Gaussian elimination has emerged [67, 68].

Any stream or block cipher can be described by a system of equations expressing the ciphertext as a function of the plaintext and the key bits. An *algebraic attack* is simply an attempt to solve this system of equations for the plaintext. If the system happens to be overdefined, then the attacker can use linearization techniques to extract a solution. However, in general, this approach is difficult, and not effective, unless the equations happen to be of low degree. That is (somewhat) ensured

if, for instance, the nonlinear Boolean function combiner in an LFSR-based generator (a widely used encryption technique) has low degree or the combiner has a low algebraic immunity (defined below) [67, 68].

Let be $f(x_1, x_2, \ldots, x_n)$ a Boolean function that depends on the bits $\{x_1, x_2, \ldots, x_n\}$ of a linear feedback shift register (LFSR). Let be L another function that defines the LFSR. Specifically,

$$L : \{0,1\}^n \to \{0,1\}^n$$

defines how the values of the LFSR bits at some clock period depend on the bit values in the previous clock period. Suppose the keystream $z_0, z_1, z_2, \ldots$ is computed from some initial secret state (the *key*) given by n bits $a_0, a_1, \ldots, a_{n-1}$ in the following way. Let $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ be the initial state and define the keystream bits by

$$\begin{aligned} z_0 &= f(\mathbf{a}) \\ z_1 &= f(L(\mathbf{a})) \\ z_2 &= f(L^2(\mathbf{a})) \\ &\vdots \qquad\quad \vdots \\ z_t &= f(L^t(\mathbf{a})) \ . \end{aligned}$$

The problem of extracting the plaintext message in this context is equivalent to the problem of finding the initial key $\mathbf{a}$, knowing L and f, and intercepting z_i. Assume that f is expressed in its algebraic normal form (ANF). Specifically,

$$f(x_1, x_2, \ldots, x_n) = c_0 \oplus c_1 x_1 \oplus c_2 x_2 \oplus \ldots c_{2^n-1} x_1 x_2 \ldots x_n \ ,$$

where $c_i \in \{0,1\}$ is uniquely determined by $\mathbf{c}$. Let $\deg(f) = d$ be the number of variables in the term of the ANF with the *largest* number of variables. Since $\deg(f) = d$, every term on the right hand side of any equation in the above set of equations has d or fewer variables. Therefore, there are $M = \sum_{i=0}^{d} \binom{n}{i}$ or fewer of these terms, and we define a variable y_j for each one of them. If a cryptanalyst has access to at least $N \geq M$ keystream bits z_t, then he/she can solve the linear system of N equations for the values of the variables y_j, and thus recover the values of $a_0, a_1, \ldots, a_{n-1}$. If d is not large, then the cryptanalyst may well be able to acquire enough keystream bits so

that the system of linear equations is highly overdefined (that is, N is much larger than M).

If we use Gaussian reduction to solve the linear system, then the amount of computation required is $O\left(\binom{n}{d}^{\omega}\right)$, where ω is the well–known *exponent of Gaussian reduction* ($\omega = 3$ (Gauss-Jordan [303]); $\omega = \log_2 7 = 2.807$ (Strassen [304]); $\omega = 2.376$ (Coppersmith-Winograd [63])). For $n \geq 128$ and $d \sim n$, we are near the upper limits for which this attack is practical for actual systems, since the complexity grows with d.

Courtois and Meier [67] showed that if one can find a function g with small degree d_g such that $fg = 0$ or $(1 \oplus f)g = 0$, then the number of unknowns for an algebraic attack can be reduced from $\binom{n}{d_f}$ to $\binom{n}{d_g}$. We say that g is an *annihilator* of f. That is easy to see, since $f(L^i(\mathbf{a})) = z_i$ becomes $g(L^i(\mathbf{a})) \cdot f(L^i(\mathbf{a})) = 0 = z_i\, g(L^i(\mathbf{a}))$, and so, we get the equations $g(L^i(\mathbf{a})) = 0$, whenever the intercepted $z_i \neq 0$. That gives us a reduction in complexity, from

$$O\left(\binom{n}{d_f}^{\omega}\right) \text{ to } O\left(\binom{n}{d_g}^{\omega}\right) .$$

Therefore, it is necessary to have a fast computation of a low(est) degree annihilator of the combiner f.

3.3.2. Why do we Need a Reconfigurable Computer?

Although the computations needed in an attack or a defense can be done by a conventional computer, a significant speedup can be achieved by using a reconfigurable computer. For example, it has been shown that a reconfigurable computer can compute bent functions for use in encryption at a speed that is 60,000 times faster than by a conventional computer [275]. The speedup is achieved because a reconfigurable computer can implement many copies of basic logic elements, like adders and comparators, which can then execute simultaneously. However, in a conventional computer, there are fast logic elements, like adders, but there are relatively few of them. A conventional computer is limited in what it can efficiently compute because its architecture is limited. As another example, the effectiveness

of a reconfigurable computer has been shown in computing Boolean functions with high correlation immunity, which are more immune to correlation immunity attacks [95]. A speedup of about 190 times was achieved.

In this section, we show that a reconfigurable computer is effective in computing the algebraic immunity of a Boolean function. This study represents a departure from the two studies involving a reconfigurable computer described above. That is, compared to nonlinearity and correlation immunity, the computation of the algebraic immunity of a Boolean function is more complex. Given human inclinations, it would seem that algebraic immunity computations should be relegated to conventional computers rather than reconfigurable computers. However, we show that his intuition is wrong; reconfigurable computers still hold an advantage over conventional computers.

3.3.3. Background and Notation

We start with a combinatorial property of a Boolean function that also plays an important role in the cryptographic world (see [73] for more cryptographic properties of Boolean functions).

Definition 3.6. *The* **degree** d *of a* **term** $x_{i_1}x_{i_2}\ldots x_{i_d}$ *is the number of distinct variables in that term, where* $i_j \in \{1, 2, \ldots, n\}$*, and* n *is the total number of variables.*

Definition 3.7. *The algebraic normal form or* **ANF** *of a Boolean function* $f(x_1, x_2, \ldots, x_n)$ *consists of the exclusive OR of terms; specifically,* $f(x_1, x_2, \ldots, x_n) = c_0 \oplus c_1x_1 \oplus c_2x_2 \oplus \ldots \oplus c_{2^n-1}x_1x_2\ldots x_n$*, where* $c_i \in \{0, 1\}$.

The ANF of a function is often referred to as the **positive polarity Reed-Muller** form.

Definition 3.8. *The* **degree**, $\deg(f)$*, of function* $f(x_1, x_2, \ldots, x_n)$ *is the largest degree among all the terms in the ANF of* f.

Table 3.2. Functions that annihilate the 3-variable majority function f and their degree

$x_1x_2x_3$	f	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8	α_9	α_{10}	α_{11}	α_{12}	α_{13}	α_{14}	α_{15}
000	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
001	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
010	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
011	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
100	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
101	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
110	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
111	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Degree		2	3	3	2	3	2	2	3	3	2	2	3	2	3	3

Example 3.1. *The ANF of the 3-variable majority function is*

$$f(x_1, x_2, x_3) = x_1x_2 \oplus x_1x_3 \oplus x_2x_3 \ .$$

Its degree is 2.

Definition 3.9. *Function $a \neq 0$ is an* **annihilator** *of function f if and only if $a \cdot f = 0$.*

Note that $\bar{f}$ is an annihilator of f. Further, if a is an annihilator of f, so also is α, where $\alpha \leq a$ ('$\leq$' is a partial order on the set of vectors of the same dimension, that is, $(\alpha_i)_i \leq (\beta_i)_i$ if and only if $\alpha_i \leq \beta_i$, for any i).

Definition 3.10. *Function f has* **algebraic immunity** *k where*

$$k = \min\{\deg(a) \mid a \text{ is an annihilator of } f \text{ or } \bar{f}\} \ .$$

Example 3.2. *The annihilators of the 3-variable majority function $f(x_1, x_2, x_3) = x_1x_2 \oplus x_1x_3 \oplus x_2x_3$ include $\bar{f}$ and all g such that $g \leq \bar{f}$, excluding the constant 0 function. In all, there are 15 annihilators of f and 15 annihilators of $\bar{f}$. Among these 30 functions, the minimum degree is 2. Thus, $f(x_1, x_2, x_3) = x_1x_2 \oplus x_1x_3 \oplus x_2x_3$ has algebraic immunity 2. Table 3.2 shows all 15 annihilators (labeled $\alpha_1, \alpha_2, \ldots, \alpha_{15}$) of the 3-variable majority function. Table 3.3 shows all 15 annihilators*

Table 3.3. Functions that annihilate the complement of the 3-variable majority function

$x_1x_2x_3$	$\bar{f}$	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}	β_{11}	β_{12}	β_{13}	β_{14}	β_{15}
000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
001	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
010	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
011	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
100	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
101	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
110	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
111	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Degree		2	3	3	2	3	2	2	3	3	2	2	3	2	3	3

(labeled $\beta_1, \beta_2, \ldots, \beta_{15}$) of the complement of the 3-variable majority function. Across both of these tables, one can verify that the minimum degree among annihilators is 2. It follows that the algebraic immunity of the 3-variable majority function is equal to 2.

Example 3.3. *Let $n = 4$. The function $f = x_1x_2x_3x_4$ has the highest degree, that is equal to 4. Function $a = \bar{x}_1 = x_1 \oplus 1$ annihilates f, since $a \cdot f = \bar{x}_1x_1x_2x_3x_4 = 0$. Since, $\bar{x}_1$ has degree 1 and there exists no annihilator of f of degree 0, the algebraic immunity of f is equal to 1. To reach this conclusion, it is not necessary to check the annihilators of $\bar{f}$, since the only annihilator of $\bar{f}$ is f, which has degree 4.*

We can immediately state the following lemmas.

Lemma 3.6. *The algebraic immunity of a function $f(x_1, x_2, \ldots, x_n)$ is identical to the algebraic immunity of $\bar{f}$.*

Lemma 3.7. *The algebraic immunity of a function $f(x_1, x_2, \ldots, x_n)$ is* $\min\{\deg(\alpha) | \alpha \leq \bar{f}$ *or* $\alpha \leq f\}$.

Proof. The hypothesis follows immediately from the observation that $\{\alpha | \alpha \leq \bar{f}$ or $\alpha \leq f\}$ is the set of all annihilators of f. □

Lemma 3.7 is similar to Definition 3.10. However, there is an important difference. Lemma 3.7 admits an algorithm for determining the algebraic immunity of a function f. Specifically, examine the degree of each function α such that $\alpha \leq \bar{f}$ and determine the minimum degree of the ANF among all α. This requires the examination of $2^{2^n - wt(f)} - 1$ functions, where $wt(f)$ is the number of 1's in the truth table of f, since $\bar{f}$ has $2^n - wt(f)$ 1's in its truth table. In forming an annihilator, each 1 can be retained or set to 0. The '-1' accounts for the case where all 1's are set to 0, which is not an annihilator. The following result is essential to the *efficient* computation of algebraic immunity.

Lemma 3.8. *[67, 198] The algebraic immunity AI(f) of a function* $f(x_1, x_2, \ldots, x_n)$ *is bounded above; specifically, AI(f)* $\leq \lceil \frac{n}{2} \rceil$.

3.3.4. Computation of Algebraic Immunity

Row Echelon Reduction Method. The computation of algebraic immunity is more complex than nonlinearity and correlation immunity, two cryptographic properties that have been previously computed by a reconfigurable computer. There are several methods for computing the algebraic immunity of a Boolean function f. In the brute force method, one checks every function to see if it is an annihilator of the given f, or its complement. In another approach, one can identify directly the annihilators with high degree. Since our attempt is to implement the algebraic immunity computation on a reconfigurable computer, we have not implemented the more recent algorithm of Armchnecht et al. [9], which also deals with *fast* algebraic attack issues. Our approach is based on a simpler version of that linear algebra approach. This enabled us to implement it on the SRC-6 reconfigurable computer and to display the algebraic immunity profiles for all functions on n variables, for $2 \leq n \leq 5$. A similar approach has been used to compute algebraic immunity on a conventional processor [67, 198]. Our implementation is the first known using Verilog on an FPGA.

Here, we create the ANF of a minterm corresponding to each 1 in the truth table of the function. Our approach to solving this system is to

express this in reduced row echelon form using Gaussian elimination. It is based on two elementary row operations: (1) interchange two rows, and (2) add one row to another row. A simple test applied to the reduced row echelon form determines if there is an annihilator of some specified degree.

Our approach is to express the system of linear equations in reduced row echelon form, and, from this, determine if there exists a solution of some specified degree. A matrix is in *row echelon form* if it satisfies the following conditions:

1. the first nonzero element (leading entry) in each row is 1

2. each leading entry is in a column to the right of the leading entry in the previous row

3. rows with all zero elements (if any) are below rows having a nonzero element.

To put the matrix in *reduced* row echelon form, one simply uses elimination on nonzero entries above each pivot.

Example 3.4. *To illustrate, consider solving the algebraic immunity of the majority function* $f(x_1, x_2, x_3) = x_1x_2 \oplus x_1x_3 \oplus x_2x_3$. *The top half of Table 3.4 shows the minterm canonical form of* $\bar{f}$. *Here, the first (leftmost) column represents all binary three tuples on three variables. The second column contains the truth table of the complement function* $\bar{f}$, *which is expressed as* $\bar{x}_1\bar{x}_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3 \vee \bar{x}_1x_2\bar{x}_3 \vee x_1\bar{x}_2\bar{x}_3$, *or more compactly, as the sum of minterms* $m_0 \vee m_1 \vee m_2 \vee m_4$. *This represents the annihilator of* f *with the most* 1*'s.*

The columns are labeled by all possible terms in the ANF of an annihilator. Then, 1's are inserted into the table to represent the ANF of the minterms. For example, since the top minterm, $m_0 = \bar{x}_1\bar{x}_2\bar{x}_3 = (x_1 \oplus 1)(x_2 \oplus 1)(x_3 \oplus 1) = x_1x_2x_3 \oplus x_2x_3 \oplus x_1x_3 \oplus x_1x_2 \oplus x_3 \oplus x_2 \oplus x_1 \oplus 1$, *its ANF has all possible terms, and so, there is a 1 in every column of this row.*

Note that we can obtain the ANF of some combination of minterms as the exclusive OR of various rows in the top half of Table 3.4. This

Table 3.4. Functions that annihilate the 3-variable majority function

ANF Coefficient →			c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0	Minterms
Index	$x_1x_2x_3$	$\bar{f}$	$x_1x_2x_3$	x_2x_3	x_1x_3	x_1x_2	x_3	x_2	x_1	1	
Original											
0	000	1	1	1	1	1	1	1	1	1	m_0
1	001	1	1	1	1	0	1	0	0	0	m_1
2	010	1	1	1	0	1	0	1	0	0	m_2
3	011	0	0	0	0	0	0	0	0	0	
4	100	1	1	0	1	1	0	0	1	0	m_4
5	101	0	0	0	0	0	0	0	0	0	
6	110	0	0	0	0	0	0	0	0	0	
7	111	0	0	0	0	0	0	0	0	0	
Reduced Row Echelon Form											
0	-	-	1	0	0	0	1	1	1	0	$m_1 \oplus m_2 \oplus m_4$
1	-	-	0	1	0	0	1	1	0	1	$m_0 \oplus m_4$
2	-	-	0	0	1	0	1	0	1	1	$m_0 \oplus m_2$
3	-	-	0	0	0	1	0	1	1	1	$m_0 \oplus m_1$
4	-	-	0	0	0	0	0	0	0	0	
5	-	-	0	0	0	0	0	0	0	0	
6	-	-	0	0	0	0	0	0	0	0	
7	-	-	0	0	0	0	0	0	0	0	

follows from the observation that $m_i \vee m_j = m_i \oplus m_j$ *due to the orthogonality of the minterms. For example, one annihilator* a *is*

$$a = \bar{x}_1 x_2 \bar{x}_3 \vee \bar{x}_1 \bar{x}_2 \bar{x}_3 \ ,$$

and so the ANF of a *is generated by simply exclusive ORing the rows associated with these two minterms.*

Elementary Row Operations. Consider a $0-1$ matrix and two row operations:

1. interchange one row with another, and

2. replace one row by the exclusive OR of that row with any other row.

Using elementary row operations, we seek to create columns, starting with the left column with *only one* 1 (called a *pivot*).

Definition 3.11. *A* 0−1 *matrix is in* **row echelon form** *if and only if all nonzero rows (if they exist) are above any rows of all zeroes, and the leading coefficient (pivot) of a nonzero row is always strictly to the right of the leading coefficient of the row above it.*

Definition 3.12. *[342] A* 0−1 *matrix is in* **reduced row echelon form** *if and only if it is in row echelon form and each leading* 1 *(pivot) is the only* 1 *in its column.*

Consider Table 3.4. The top half shows the truth table of $\bar{f}$. For each value 1 (minterm – m_0, m_1, m_2, and m_4) in the $\bar{f}$ column (third column), the ANF of that minterm is expressed across the rows. To form an annihilator of f, we must combine one or more minterms using the exclusive OR operation. The bottom half of Table 3.4 shows the reduced row echelon form of the top half. The row operations we used to derive the bottom half from the top half can be inferred from the rightmost column. For instance, the entry $m_0 \oplus m_1$ in the bottom half of the table indicates that the rows labeled m_0 and m_1 in the top half of the table were combined using the exclusive OR operation.

Note that, like the top half of Table 3.4, the rows in the reduced row echelon form combine to form *any* annihilator of the original function. This follows from the fact that any single minterm can be formed as the exclusive OR of rows in the reduced row echelon form. For example, m_1 is obtained as the exclusive OR of the top three rows of the reduced row echelon form.

The advantage of the reduced row echelon form is that we can simply inspect the rows to determine the annihilators of lowest degree. For example, in the reduced row echelon form, the top row represents an annihilator of degree 3, since there is a value 1 in the column associated with $x_1x_2x_3$. Since the pivot point has the *only* value 1 in this row, the only way to form an annihilator of degree 3 is to include this row.

The other three rows each have a pivot in a column associated with a degree 2 term. And, the only way to have a degree 2 term is to involve at least one of these rows. Since there are no other rows with

a pivot point in a degree 1 or 0 term, we can conclude that there exist *no* annihilators of degree 1 or 0. Thus, the lowest degree of an annihilator of f $(= x_1x_2 \oplus x_1x_3 \oplus x_2x_3)$ is equal to 2.

Steps to Reduce the Computation Time. The matrices for which we seek a reduced row echelon form can be large. For example, each matrix has 2^n rows, of which we manipulate only those with 1's in the function. Potentially, there are also 2^n columns. However, we can reduce the columns we need to examine by a few observations. Recall that no function has an AI greater than $\lceil \frac{n}{2} \rceil$. Thus, we need consider only those columns corresponding to ANF terms where there are $\lceil \frac{n}{2} \rceil$ or fewer variables. However, it is not necessary to consider columns corresponding to terms with exactly $\lceil \frac{n}{2} \rceil$ variables. This is because if no annihilators are found for a function f (or its complement) of degree $\lceil \frac{n}{2} \rceil - 1$ or less, it must have an AI of $\lceil \frac{n}{2} \rceil$.

We can reduce the computation of the AI of a function by another observation. If a degree 1 annihilator for a function is found, there is no need to analyze its complement. Even if the complement has no annihilators of degree 1, the function itself has AI of 1. On the other hand, finding an annihilator of degree $\lceil \frac{n}{2} \rceil$ requires the analysis of its complement for annihilators of smaller degree.

3.3.5. Results and Comments

Approach. A Verilog program was written to implement the row echelon conversion process described above. It runs on an SRC-6 reconfigurable computer from SRC Computers, Inc. and uses the Xilinx Virtex2p (Virtex2 Pro) XC2VP100 FPGA with Package FF1696 and Speed Grade -5. Table 3.5 compares the average time in computing the AI of an n-variable function on this FPGA with that of a typical microprocessor. In this case, we chose the Intel®Core™2 Duo P8400 processor running at 2.26 GHz. This processor runs Windows 7 and has 4 GB of RAM. The code was compiled using Code::Blocks 10.05. The data shown is from a C program that also implements the row echelon conversion process. For ease of presentation, we compute the rate of computation, as measured by the number of functions per second.

Table 3.5. Comparison of the computation times for enumerating the AI of n-variable functions on the SRC-6 reconfigurable computer versus an Intel® Core™2 Duo P8400 microprocessor

SRC-6 Reconfigurable Comp.				
n	Clocks per function	Functions per second	# of samples	Speedup
2	46.3	2,162,162	16,000,000*	0.5
3	70.7	1,414,130	25,600,000*	1.1
4	75.5	880,558	65,536,000*	1.9
5	348.4	287,012	4,294,967,296*	4.9
6	78.0	12,823	25,000,000	0.7
Intel®Processor				
2		4,186,290	16,000,000*	
3		1,317,076	25,600,000*	
4		458,274	65,536,000*	
5		59,029	4,294,967,296*	
6		17,699	500,000,000	

* Exhaustive enumeration of all n-variable functions

Computation Times. Table 3.5 compares the computation times for AI when done on the SRC-6 reconfigurable computer and on an Intel® Core™2 Duo P8400 processor.

The second, third, and fourth columns in the upper part of Table 3.5 show the performance of the SRC-6 and the middle two columns in the lower part of this table show the performance on the Intel® Core™2 Duo P8400 processor. The last column shows the speedup of the SRC-6 over the Intel® Core™2 Duo P8400 processor.

The second column shows the average number of 100 MHz clocks needed by the SRC-6. The third column shows the average number of functions per second. The fourth column shows the number of functions. The term *samples* is used here to indicate that, for smaller n, we had to repeat the computation of the same function to achieve a sufficient number of functions between each measuring point so that

Table 3.6. Comparing the brute force method with the row echelon method on 4-variable functions

	Brute Force	Row Echelon
# Functions	65,536	65,536
Total Time (sec.)	0.807	0.050
Total Clocks	80,748,733	4,946,111
Clocks Per Function	1,232.1	75.5
Functions Per Second	81,160	1,325,000

the time of computation was accurate. In the case of $n \leq 5$, all functions were enumerated, and in the case of $n = 6$, a subset of random functions was enumerated.

The second column in the lower part of Table 3.5 shows the average number of functions per second on the Intel® Core™2 Duo P8400 processor, while the third column of this table shows the number of functions. The last column shows the speedup of the reconfigurable computer over the Intel® Core™2 Duo P8400 processor.

For example, for $n = 5$, the SRC-6 reconfigurable computer is 4.9 times faster than the Intel® processor. For $n = 4$, the SRC-6 is 1.9 times faster. However, for $n = 6$, the processor is actually faster than the reconfigurable computer. In the case of $n = 6$, a sample size of 25,000,000 was used for the SRC-6 and 500,000,000 for the Intel® Core™2 Duo P8400 processor. For all lower values of n, exhaustive enumeration was performed.

Comparing the Row Echelon Method to Brute Force. Table 3.6 compares the row echelon method, which involves the solution of simultaneous equations with the brute force method discussed earlier for the case of $n = 4$.

In both cases, 65,536 functions were considered, all 4-variable functions. The last row shows that the row echelon method is able to process 1,325,000 functions per second verses 81,160 functions per second for the brute force method, resulting in 16.3 times the throughput.

Table 3.7. The number of n-variable functions distributed according to algebraic immunity for $2 \leq n \leq 6$

AI\n	2	3	4	5	6
0	2	2	2	2	(2) 0
1	14	198	10,582	7,666,550	(1,081,682,871,734) *1,114,183,342,052*
2	0	56	54,952	**4,089,535,624**	***1,269,840,659,739,507,264***
3	0	0	0	**197,765,120**	***17,176,902,299,786,702,300***
Total	16	256	65,536	4,294,967,296	18,446,744,073,709,551,616

Bold entries are previously unknown.
Bold and italicized entries are estimates to previously unknown values.

Distribution of Algebraic Immunity to Functions. Table 3.7 shows the number of functions with various algebraic immunities for $2 \leq n \leq 6$. This extends the results of [320] to $n = 5$. In our case, the use of a reconfigurable computer allows this extension.

The entries shown in **bold** in the column for $AI = 5$ are exact values for previously unknown values. The entries shown in ***bold and italics*** for $AI = 6$ are approximate values for previously unknown values. In this case, the approximate values were determined by a Monte Carlo method in which 500,000,000 random 6-variable functions were generated (or $2.7 \times 10^{-9}\%$ of the total number of functions) and their algebraic immunity computed. For $n = 5$ and $n = 6$, the number of functions with algebraic immunity 1 are known. However, Table 3.7 shows the value 0 for the number of functions with algebraic immunity 0 (there are actually 2, the exclusive OR function and its complement). This is because the Monte Carlo method produced no functions with

Table 3.8. Frequency and resources used to realize the AI computation on the SRC-6's Xilinx Virtex2p (Virtex2 Pro) XC2VP100 FPGA

n	Freqency (MHz)	# of LUTs	Total # of slice FFs	# of occupied slices
2	103.1	2,066(2%)	2,977(3%)	2,089(4%)
3	113.0	2,199(2%)	3,011(3%)	2,157(4%)
4	109.4	2,343(2%)	2,760(3%)	2,120(4%)
5	100.9	5,037(5%)	4,110(4%)	3,780(8%)
6	87.5	8,990(10%)	3,235(3%)	5,060(11%)

an AI of 0. The *italicized* value, *1,114,183,342,052*, in Table 3.7 is an estimate of the number of 6-variable functions with algebraic immunity 1. To show the accuracy of the Monte Carlo method, compare this to the previously known exact value 1,081,682,871,734 [320]. The estimated value is 3% greater than the exact value.

Resources Used. Table 3.8 shows the frequency achieved on the SRC-6 and the number of LUTs and flip-flops needed in the realization of the AI computation for various n. The frequency ranges from 113.0 MHz at $n = 3$ to 87.5 MHz for $n = 6$. Since the SRC-6 runs at 100 MHz, the 87.5 MHz value is cause for concern. However, the system works well at this frequency. For all values of n, the number of LUTs, slice flip-flops, and occupied slices were well within FPGA limits. Indeed, among all three parameters and all values of n, the highest percentage was 11%.

Cryptographic Properties. We show that a reconfigurable computer can be programmed to efficiently compute the algebraic immunity of a logic function. Specifically, we show a 4.9 times speedup over the computation time of a conventional processor. This is encouraging given that algebraic immunity is one of the most complex cryptographic properties to compute. This is the third cryptographic property we have concentrated on that has benefited from the highly efficient, parallel nature of the reconfigurable computer. The interested reader may wish to consult two previous papers on nonlinearity [275] and correlation immunity [95].

Digital Circuits

4. Design

4.1. Low-Power Design Techniques for State-of-the-Art CMOS Technologies

Vincent C. Gaudet

4.1.1. Power Dissipation and the Continuing Rule of Moore's Law

Energy consumption has become one of the main issues when designing microelectronic circuits. Circuit designers used to focus their efforts primarily on functionality and on meeting clock speed targets. However, in sub-65-nm complementary metal oxide semiconductor (CMOS) technologies, many more design considerations must be taken into account, including power optimization. To underline the importance of these challenges, a roundtable discussion on how to achieve a 10x reduction in energy consumption in electronic devices was recently held during the plenary session of the 2011 *IEEE International Solid-State Circuits Conference* [243]. Other presentations in previous editions of the conference have also highlighted the need for new approaches [40].

The 2011 edition of the Silicon Industry Association (SIA) International Technology Roadmap for Semiconductors (ITRS) [147] projects that by 2020, lithography will allow the fabrication of devices with dimensions of 10 nm. Next-generation semiconductor technologies will of course have a much greater packing density with better-performing transistors, potentially leading to exciting breakthroughs in computing. However, the transistors that are available in those technologies

will exhibit a very large variation in electric parameters such as device threshold, which will require new design approaches to maximize yield. Furthermore, while power consumption in traditional integrated circuits was dominated by switching power (namely, the power required to switch a wire between different logic values), sub-threshold leakage currents are becoming far more significant than ever before.

Power consumption is a crucial design consideration for many reasons. In portable devices, battery life is determined by the average power consumption, and any reduction in power leads to a commensurate increase in battery life. Power is typically dissipated as heat, and this can lead to several problems.

In compute server farms, the heat generated by computing devices must be removed from server rooms, leading to high air conditioning cost. On a much smaller scale, transistors themselves do not work well at high temperatures, so integrated circuits must be cooled down through expensive heat sinks.

In some cases, e.g., in biomedical applications, there are environmental concerns as well; for instance, biomedical implants can only support a limited power consumption; at a rate of $40mW/cm^2$, the increase in localized body temperature can kill cells [76]; furthermore, biomedical implants do not have access to a convenient source of energy [135]. Of course, there is the largest environmental concern of all: the power used to run integrated circuits must come from somewhere; on a global scale, the International Energy Agency estimates that in 2007, 70% of electricity was generated from fossil fuels [146].

In this section, we begin by reviewing some basics of integrated circuit design and the main sources of power consumption. Then, we outline some of the more common practices in low-power design, starting at a high level of abstraction (software, architectural), and moving towards lower levels of abstraction (circuit, device). We also look at some promising techniques to deal with upcoming issues in power consumption, and present some design examples. Finally, we conclude the section by exploring one of the most fundamental questions in power consumption:

Does there exist a lower bound on power consumption?

4.1.2. Models for Power Consumption

Microelectronic Circuits: the Basics

In order to get to an understanding of power consumption in integrated circuits, we begin with a quick overview of transistor models, notation, and basic logic gate design.

Although there are dozens of reliable textbooks on digital CMOS circuit design, we recommend reading the ones by Weste and Harris [334], and by Rabaey, Chandrakasan, and Nikolic [244]. Also recommended is the widely used textbook on microelectronic circuit design by Sedra and Smith [272].

The metal-oxide-semiconductor field-effect transistor (MOSFET, but usually abbreviated MOS) is the basic building block of CMOS technologies. Designers can inexpensively prototype two types of MOS transistors: n-type enhancement MOS (NMOS) and p-type enhancement MOS (PMOS). Both types of transistors can be modeled as 3-terminal devices with terminals known as the gate (G), drain (D), and source (S). Note that in this context the gate refers to a transistor's terminal and not a logic gate. To differentiate between the two concepts and to prevent confusion, we often speak of a *transistor gate* or *gate terminal*, and a *logic gate*.

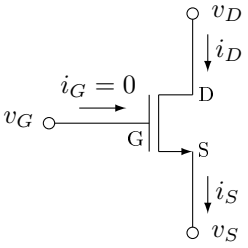


Figure 4.1. NMOS transistor with terminal labels, voltages, and currents.

At each terminal, we wish to know the terminal voltage (with respect to the common node, or *ground*), and the current flowing into (or out of, depending on convention) the terminal (see Figure 4.1). Without

going into complicated device physics (which gets even more complicated with every new generation of CMOS technology!), a high-level explanation of transistor terminal behavior is as follows.

The gate terminal is a high-impedance node, which implies that it draws no current (i.e., $i_G = 0$), no matter what voltage (v_G) is applied to the node; the transistor gate is often seen as a circuit's input (it may even be the input to the logic gate!).

At a high level of abstraction, the gate voltage v_G controls the value of i_D and i_S in a nonlinear way. When $|v_G - v_S|$ falls below a threshold value, known as the transistor threshold V_t, there is an open circuit between the source and drain, and therefore $i_S = i_D = 0$, no matter what the source and drain voltages are; in this case we say that the transistor is OFF.

When v_G exceeds V_t (note that we often use a small v for signal quantities and a capital V for constant values), the transistor acts as a current source between the source and drain terminals; here, we say that the transistor is ON.

Although the current is highly nonlinear, it generally increases for larger values of $|v_D - v_S|$ and $|v_G - v_S|$. Since $i_G = 0$, and due to Kirchhoff's current law, we must have $i_D = i_S$.

The main difference between NMOS and PMOS transistors is that an NMOS requires a voltage $v_G - v_S > V_{tn}$ to turn on the transistor, and $i_D \geq 0$, while a PMOS transistor requires a voltage, $v_S - v_G > |V_{tp}|$ to turn on the transistor, and $i_D \leq 0$. Note the added n and p to differentiate between the NMOS and PMOS threshold voltages (and just to confuse everyone, V_{tp} is usually reported as a negative value, hence the absolute value symbols).

Finally, circuit designers can control a transistor's length L and width W. In general, i_D grows linearly with the ratio W/L. Since W/L really represents the only aspect of a transistor that designers can modify, these values tend to be reported in many circuits papers. However, in practice most digital designs use the smallest value of L available in a CMOS technology, which is usually close to the name of the technology node (e.g., in a 65 nm CMOS technology, minimum

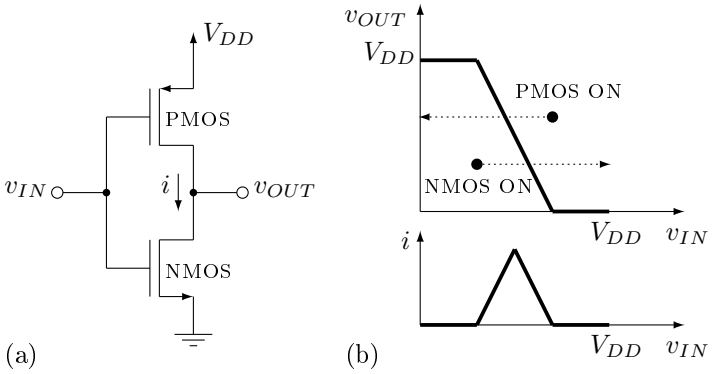


Figure 4.2. CMOS inverter: (a) circuit schematic, and (b) transfer characteristics.

transistor lengths tend to be approximately 65 nm).

Logic Gate Operation

A logic gate is built out of several transistors. For instance, an inverter is constructed using one NMOS transistor, one PMOS transistor, a DC voltage source (usually called V_{DD}), and several wires (see Figure 4.2).

As we sweep the input voltage v_{IN} from 0 V to V_{DD}, each transistor goes through several modes of operation. For $v_{IN} < V_{tn}$, the NMOS transistor is OFF whereas the PMOS transistor is ON. This creates a conducting channel between V_{DD} and v_{OUT} while leaving an open circuit to the ground node; hence $v_{OUT} = V_{DD}$, i.e., the output is at a HIGH voltage, which is used to represent a logic value 1. Similarly, for $v_{IN} > V_{DD} - |V_{tp}|$, the NMOS transistor is ON, the PMOS transistor is OFF, and $v_{OUT} = 0$ V, which represents a logic value 0. For $V_{tn} < v_{IN} < V_{DD} - |V_{tp}|$, both transistors are ON, current flows directly from V_{DD} to the ground node, and v_{OUT} is at an intermediate voltage. In this intermediate region, $\delta v_{OUT}/\delta v_{IN} < -1$, which means that the circuit acts like an *amplifier*. This behavior is known as the *regenerative property* of a logic gate, and is seen by most designers as a necessary condition for a functional gate. When a gate acts as

an amplifier, it can overcome voltage disruptions due to crosstalk and other noise sources. Note the complementarity of the NMOS and PMOS operation: the PMOS produces a logic 1 at the output and the NMOS produces a logic 0.

For brevity, we will not look at the design of more complex gates. Suffice to say, series connections of transistors produce ANDing effects, and parallel connections produce ORing effects.

So far we have only looked at the *static* behavior of a circuit, namely the behavior when no voltages are changing. Let us now look at the *dynamic* behavior, which will lead to the discussion of power consumption. To do this, we introduce a load capacitor C_L between the output of the logic gate and the ground node (see Figure 4.3 (a)). This capacitance may be due to many sources: the input parasitic capacitances of transistors external to the logic gate, the parasitic capacitances of wires connected to the output of the gate (e.g., wires delivering logic values to subsequent gates), and even some parasitic capacitances from the transistors internal to the gate. In practice, C_L grows substantially when there is a large *fan-out*, i.e., when many external gates are being driven, or when the distance between the driving gate and a load gate is large (for many reasons, long wires are bad!).

Consider the following scenario. For $t < 0$, v_{IN} is held at V_{DD}, and so the output v_{OUT} is at 0 V; the NMOS transistor is ON and the PMOS transistor is OFF. At $t = 0$, the input instantaneously drops to 0 V, turning ON the PMOS transistor and turning OFF the NMOS transistor. What happens? Of course, we expect that the output will rise to V_{DD}, but this cannot happen instantaneously since this would require infinite current to flow onto C_L, which is clearly impossible. In reality, recall that a nonlinear resistance now appears between the source and drain of the PMOS transistor, creating an RC circuit. The current flowing through the PMOS transistor eventually charges up the voltage to V_{DD} as $t \to \infty$. Since this is a nonlinear process, derivation of an analytical equation for $v_{OUT}(t)$ is often impossible. Hence, circuit designers usually rely on numerical simulators such as SPICE to verify circuit functionality.

Later on, when v_{IN} returns to V_{DD}, the NMOS turns ON again, the

PMOS turns OFF, the output gradually decreases back to 0 V.

Now we can move on to a discussion on energy consumption.

Power Consumption in CMOS Circuits

There are three main sources of power consumption in microelectronic circuits. The first two: switching power and short-circuit power, depend on the frequency of signal transitions. The last, leakage power, is constant over time.

Switching Power. Recall that a capacitor C with voltage V stores an amount of electrical energy $E = \frac{1}{2}CV^2$. Therefore, as the output goes from logic 0 to logic 1 and back to logic 0, the load capacitance goes from storing $E_{logic0} = 0$ to storing $E_{logic1} = \frac{1}{2}C_L V_{DD}^2$, and back to 0 again, which clearly implies an energy transfer.

It can be demonstrated analytically that as the output switches from 0 to 1, $\frac{1}{2}C_L V_{DD}^2$ of energy is dissipated through the PMOS transistor as heat. As the output switches back to 0, the $\frac{1}{2}C_L V_{DD}^2$ that was stored on C_L is dissipated through the NMOS as heat. Therefore, over the course of one 0-to-1-to-0 cycle, a total of $C_L V_{DD}^2$ of energy is drawn from the power supply and dissipated as heat. Since power consumption represents the *rate* at which energy is consumed (or dissipated), we need to calculate the frequency at which we observe the switching events.

Suppose there is a probability $P_{0\to1}$ that over the course of one clock cycle, a circuit node i switches from logic 0 to 1. For notational simplicity and to avoid confusion with power P, we replace $P_{0\to1}$ with α, a term known as the *switching activity* of a node. For a clock frequency f_{CLK}, the *average* power $P_{switch,i}$ dissipated at the node is:

$$P_{switch,i} = \alpha_i f_{CLK} C_{L,i} V_{DD}^2 \ . \tag{4.1}$$

The subscript i is used for switching activity and load capacitance, to represent the notion that the values may be different at each circuit

node. Clock frequency and supply voltage tend to be held constant for large parts of a circuit, so the subscript is not used in Equation (4.1).

Total switching power is the sum of the switching power at each circuit node i:

$$P_{switch,total} = \sum_i P_{switch,i} \ . \tag{4.2}$$

Switching power is by far the most well-understood form of power consumption, and it is therefore the target of most power-reduction techniques.

Short-circuit Power. As evidenced by Figure 4.2 (b), there is a voltage range $V_{tn} < v_{IN} < V_{DD} - |V_{tp}|$ where both transistors in an inverter are ON. This implies that there is a direct current path from V_{DD} to the ground node, also indicated in Figure 4.2 (a). This extra current, often denoted I_{sc}, i.e., a *short-circuit* current, does not contribute to charging nor discharging the load capacitance. Since this current flows over a voltage drop of V_{DD}, the instantaneous power (that is, the power drawn from V_{DD} as a function of time), is as follows:

$$P_{sc,i}(t) = V_{DD} I_{sc,i}(t) \ . \tag{4.3}$$

As was the case with switching power, total short-circuit power is the sum of short-circuit power at each node:

$$P_{sc,total}(t) = \sum_i P_{sc,i}(t) \ . \tag{4.4}$$

The value of $I_{sc,i}(t)$ is highly dependent on the slope of the input signal v_{IN}, and due to the nonlinear nature of the circuit, analytical equations for $I_{sc,i}(t)$ are rarely available. Instead, numerical simulation using tools such as SPICE are required. Due to the requirement for simulation and the dependence of $I_{sc,i}(t)$ on V_{DD}, short-circuit power is much more difficult to analyze and optimize than switching power. However, we note that $I_{sc}(t)$ is only non-zero when there is

an output signal transition, and hence average short-circuit power is proportional to f_{CLK} and α, even though the terms do not appear in Equations (4.3) and (4.4).

Leakage Power. Earlier, we stated that when $|v_G - v_S| < |V_t|$, then $i_D = i_S = 0$, i.e., a transistor turns OFF and does not draw current. In reality, as the voltage drop between the gate and source approaches the transistor threshold, a *subthreshold* current, which is exponentially dependent on $|v_G - v_S|$, begins to dominate, and therefore i_D is never *exactly* 0.

Consider an inverter with $v_{IN} = 0$ V. The NMOS transistor is clearly OFF, and the PMOS is ON so v_{OUT} approaches V_{DD}. The minute subthreshold current i_D that remains in the NMOS transistor flows over a voltage drop of nearly V_{DD}, and hence power is dissipated through the NMOS transistor.

Historically, this source of power was only a tiny fraction of overall power, so it was not studied in depth. However, in modern CMOS technologies, especially in technology nodes below 90 nm, these subthreshold currents can amount to a significant fraction of total power consumption. As with short-circuit power, leakage power is rather difficult to characterize and to optimize. Subthreshold currents vary exponentially with $|v_G - v_S|$, but they also vary exponentially with $|v_D - v_S|$ (for small values of $|v_D - v_S|$) and exponentially with temperature (just to make matters interesting)!

Following the previous discussion, we can calculate instantaneous leakage power as:

$$P_{lk,i}(t) = V_{DD} I_{lk,i}(t) \ , \tag{4.5}$$

and total instantaneous leakage power as:

$$P_{lk,total}(t) = \sum_i P_{lk,i}(t) \ . \tag{4.6}$$

However, these equations provide little useful insight, so we will move on to another topic, for now.

Table 4.1. Taxonomy of sources of power consumption

Source	Determining factors	Optimization approaches
Switching power	C_L, V_{DD}^2, f_{CLK}, α	architectural and algorithmic optimizations, minimization of circuit activity, minimization of capacitance through placement, lower V_{DD}
Short-circuit power	V_{DD} (nonlinear), f_{CLK}, α, input rise and fall times	difficult to optimize; minimization of circuit activity, minimization of input rise and fall times
Leakage power	V_{DD} (nonlinear), temperature, independent of of f_{CLK}	difficult to optimize; optimizations mostly at the the circuit and device levels

Power Taxonomy. Table 4.1 presents a list of sources of power consumption, their main dependencies, and potential optimization approaches.

Figure 4.3 shows examples of power consumption waveforms for a CMOS inverter whose input switches from 1 to 0 and back to 1. Figure 4.3 (c) shows the instantaneous current drawn from the power supply. For most of the duration, the current is small, albeit nonzero, representing the leakage current. Then, there are two impulses; the larger represents a switching event where the output rises from 0 V to V_{DD}, and where the current has three components: the current required to charge up the capacitor (switching power), the short-circuit current, and the leakage current. The smaller impulse represents a switching event where the output falls back to 0 V, and where the current only has two components: the short-circuit current and the leakage current.

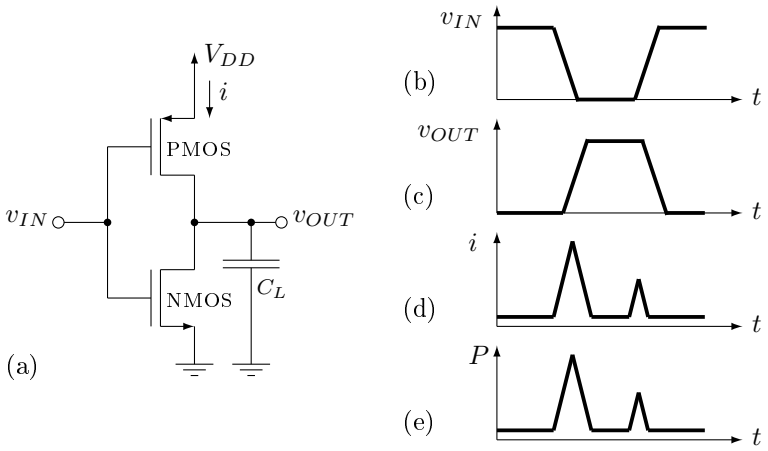


Figure 4.3. Power consumption waveforms: (a) CMOS inverter with load capacitance C_L, (b) input v_{IN}, (c) corresponding output v_{OUT}, (d) current i drawn from power supply, and (e) total instantaneous power.

4.1.3. Power Optimization

High-level Thoughts on Power Optimization

Now that we have some understanding of the sources of the power consumption, we can begin a discussion of power optimization approaches.

It would be enticing to formulate an optimization problem as a search for the argmin over the set of all potential design specifications of total power consumption of a circuit. Unfortunately, this would be unfeasible. Consider the dimensionality of the problem: integrated circuits contain millions of transistors (projected by the ITRS roadmap to reach 10 billion transistors for state-of-the-art microprocessors in 2020 [147]), where each transistor can be independently sized for W and L. Not only that, but the transistor-level netlist for any particular algorithm is not unique: several architectures can solve the same problem.

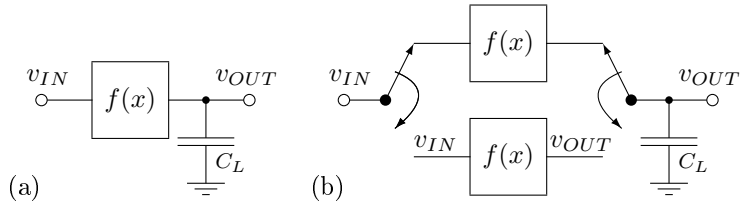


Figure 4.4. Architectural voltage scaling: (a) computational unit $f(x)$, and (b) two parallel units where supply voltage can be decreased.

But that is only the start. Once a particular architecture and circuit topology are selected, integrated circuit designers must choose the *placement* of each transistor in two-dimensional space. The distance between connected transistors is important: the parasitic capacitance of a wire is proportional to its length, so minimizing switching power requires that total wire length be kept to a minimum. There are also additional constraints: minimizing power consumption does not necessarily produce a design that meets *throughput* requirements. Joint optimization is necessary!

For these reasons, global solutions are not usually sought for power optimization problems. Rather, solutions that satisfy design constraints are usually found using heuristic techniques that are often very problem-specific. In the next few subsections, we explore some approaches at the architectural and circuit levels. We caution the reader, though, that low-power design is a huge area of active research, and that only a small sample of techniques are explored in this section.

Leveraging Parallelism to Reduce Switching Power

The second-most-highly cited paper in the history of the *IEEE Journal of Solid-State Circuits* was published by Chandrakasan *et al.* [59] in 1992 and describes a technique known as *architectural voltage scaling*. The technique leverages parallelism to reduce switching power.

Consider a circuit that must perform computation $f(x)$ at a rate of

one computation every T_s seconds, as illustrated in Figure 4.4 (a). Suppose this computation is done using a circuit that runs on a supply voltage of V_{DD} and that it drives a load capacitance C_L, and that its switching activity is α. The switching power of this circuit is given by:

$$P_{switch,cct1} = \alpha C_L V_{DD}^2 / T_s \ . \tag{4.7}$$

Now consider the circuit in Figure 4.4 (b), where two copies of the circuit have been created, and where computations are alternately assigned to each unit. In this case, each unit can take twice the amount of time to complete the calculation. This might seem unwise: more than twice as many transistors are used, so the circuit obviously costs more. Ignoring the incremental cost of the multiplexer/demultiplexer, the switching power is the same:

$$P_{switch,cct2} = \alpha C_L V_{DD}^2 / 2T_s + \alpha C_L V_{DD}^2 / 2T_s = P_{switch,cct1} \ . \tag{4.8}$$

However, consider the following proposition. *Delay* of a logic gate is *inversely proportional to* V_{DD}. In other words, since each computational unit has twice as much time as before to calculate its outputs, the supply voltage can be reduced by a factor of 2. A revised equation is:

$$P_{switch,cct3} = \alpha C_L V_{DD}^2 / 8T_s + \alpha C_L V_{DD}^2 / 8T_s = P_{switch,cct1} / 4 \ . \tag{4.9}$$

In other words, switching power has been reduced by a factor of 4! In reality, this represents a bound on achievable power reduction, since the extra multiplexer/demultiplexer incur a penalty. Also, the noise margins for the circuit will be significantly reduced, i.e., the circuit will not be able to tolerate as much noise and crosstalk.

However, we learn a valuable lesson: exploiting parallelism can be useful in reducing supply voltage, which leads to an overall decrease in switching power. This is one of the main reasons for the push towards multicore and manycore microprocessors [115].

Minimizing Supply Voltage

There is also another underlying lesson: optimizing for power consumption and for throughput may be contradictory goals. To this end, circuit designers have defined several useful metrics beyond average power (P) and delay (D). As its name implies, the *power-delay product* (PDP) is the product of P and D, and represents the average amount of energy E required to perform a computation. However, a circuit with low PDP (i.e., a very energy-efficient one) may be very slow in performing its computation. A metric that is much more preferable is the *energy-delay product* (EDP), which is the product of E and D (or P and D^2)

Recall that the switching energy, and hence the PDP for a 0-to-1-to-0 computation cycle is $C_L V_{DD}^2$. Lowering V_{DD} thus lowers PDP. Delay is inversely proportional to V_{DD} and hence EDP is proportional to V_{DD}; hence, lowering V_{DD} also has a beneficial effect on EDP. A reasonable question to ask, therefore, is: how low can V_{DD} go? The answer is not as easy as it might seem.

Lowering Voltage While Meeting Throughput Requirements. The obvious answer is to choose the minimum value of V_{DD} such that the circuit is fast enough to meet throughput specifications.

As we will see later on, though, in modern CMOS processes there are huge variations in transistor threshold voltages. These variations lead to fluctuations in logic gate delays, so we must build in some margin for error.

On the other hand, there may be an opportunity to do even better. Consider the circuit in Figure 4.5, where two parallel computation paths $f(a)$ and $g(b)$ converge onto a third computation block $h(c, d)$. If we know, for example, that path $f(a)$ takes twice as long to compute than path $g(b)$, we can lower the supply voltage to the circuit that computes $g(b)$ in half, saving energy. Indeed, there is value in reducing the supply voltage to any non-critical computation path.

However, there is a minor issue. A circuit that operates at a certain value of V_{DD} likes its inputs to be at more or less 0 V for a logic 0

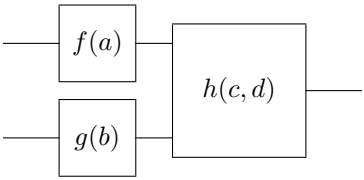


Figure 4.5. Logic circuit with two paths.

and V_{DD} for a logic 1. A higher voltage for logic 1 could break down the transistor gates. A lower voltage could be mistakenly interpreted as a logic 0 (e.g., if a logic gate operating at 1 V supply feeds a logic 1 to another gate that operates on a 3 V supply, and where the transition between logic 0 and logic 1 may be around 1.5 V). The solution is to use extra *voltage-shifting* circuits at any transition point. Of course, there is a penalty for using these circuits: extra cost, delay, and power. Computer-aided design (CAD) tools such as Power Compiler® by Synopsys Inc. [309], are good at finding optimizations.

Subthreshold Logic. For some applications, it is more important to minimize PDP than EDP. Examples include signal processing for implantable biomedical devices, where signals tend to be slow and where energy to supply a circuit is scarce. In this case, voltage scaling may be used quite aggressively, i.e., to a point where the supply voltage is so low that transistors are never really ON. This is called *subthreshold logic* and is currently a very active area of research (see e.g., [247]).

Consider the CMOS inverter of Figure 4.2 (a), and suppose

$$V_{DD} < min(V_{tn}, |V_{tp}|) \ .$$

Classically, we would consider that neither transistor can turn ON, and that the output is therefore left floating (i.e. undefined). Remember, though, that transistors never fully shut OFF. Indeed, for an NMOS transistor operating in the subthreshold region we have:

$$i_D = I_s e^{(|v_G - v_S|)/(n\frac{k_B T}{q})}(1 - e^{-(|v_D - v_S|)/(\frac{k_B T}{q})}) \ . \qquad (4.10)$$

where I_s is a scaling current that is proportional to W/L, k_B is the Boltzmann constant, T is temperature, and n is a unitless constant that depends on transistor geometry and whose value is often close

to 1.5. Several studies (see e.g., [331, 347]) have investigated the minimum supply voltage $V_{DD,min}$ such that certain design constraints, e.g., satisfying the regenerative property, are met.

Assuming balanced transistors (where the I_s values are equal between the NMOS and PMOS devices), and setting the PMOS and NMOS currents to be equal, we find that the minimum value of V_{DD} such that an inverter satisfies the regenerative property is about 2-to-3 times the thermal voltage k_BT/q, which is approximately 26 mV at room temperature. In other words, logic gates should be functional at supply voltages as low as around 70 mV. However, due to component variations in modern CMOS processes, larger supplies, e.g. 200-to-300 mV are typically required.

We note in passing the striking similarity between this result and the one first posited by Landauer in 1961 [173], which argues that the minimum cost of destroying one bit of information is $k_BTln(2)$, but that there is no known lower bound on the energy cost of reversible computation. We will come back to this later.

Encoding Schemes that Minimize Switching

In 1995, Stan and Burleson published a paper [285] on *bus-invert coding*, a technique that uses encoding to minimize switching activity on a bus. A bus typically contains a bundle of wires that transmits n bits per cycle. Since buses tend to run over long distances, their load capacitance C_L tends to be large, and hence there is value in reducing switching power, even if it means a bit of extra (low-cost) computation at the sending and receiving ends.

Consider a bus with n parallel wires, as illustrated in Figure 4.6 (a). Assuming consecutive data are equiprobable and independent, then the switching activity on each wire is 1/4, and we can expect an average of $n/4$ 0-to-1 transitions per clock cycle. Assuming a load capacitance C_L, the switching power of the bus is:

$$P_{switch,bus1} = \frac{n}{4} f_{CLK} C_L V_{DD}^2 \ . \tag{4.11}$$

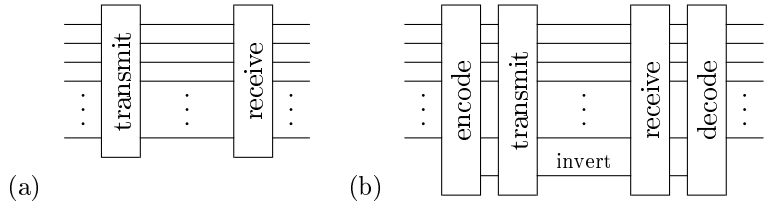


Figure 4.6. Bus-invert coding: (a) conventional bus with n wires, and (b) bus with one additional *invert* wire to indicate whether data has been inverter by the encoder.

Now consider the slightly modified bus shown in Figure 4.6 (b). Here, an extra bus line, called *invert*, is added. A logic circuit at the transmitter side calculates the number of signal transitions. If it is less than $n/2$, then data is sent onto the bus as-is, with $invert = 0$. If there are more than $n/2$ transitions, then each bit is inverted before it is sent to the bus, with $invert = 1$, resulting in *fewer* than $n/2$ transitions. Note that the *invert* line consumes switching power.

Using random data, the authors of [285] claim about an 18% reduction in switching activity for $n = 8$, and hence a reduction in switching power.

Clocking: a Huge Consumer of Switching Power

Consider a clock signal that must be distributed throughout an integrated circuit while minimizing skew (the error in the arrival time of a clock edge with respect to its intended arrival time). Since the clock must be sent to every latch and flip-flop, it is likely a very high-capacitance wire: the wires are long and there are many transistors loading it. Furthermore, every single clock cycle, the clock goes from 0 to 1 and back to 0. Therefore the clock signal has $\alpha = 1$, the highest possible value for switching activity. In other words, the switching power (and yes, the short-circuit power) consumption due to the clock is enormous, and may in fact represent a sizeable fraction of the overall power consumption of a chip, *even though the clock itself performs no actual computation*!

Some design approaches try to relax the constraints on the clock, or to eliminate it altogether. *Asynchronous* processing refers to a broad category of computational styles that eliminate a clock and replace it with local control and handshaking. The main reason that is often cited for the use of asynchronous techniques is that they eliminate synchronization overhead - extra time that is required to properly set up flip flops and to allow signals to propagate through them [334]. Another advantage is that the power consumption of the clock is eliminated, but is replaced by power consumption due to handshaking signals.

A communication processor for error-control coding and decoding that uses asynchronous circuits is reported in [227]. Based on simulation results, the circuit is projected to increase throughput by a factor of approximately 3 and to reduce the energy consumption per processed bit by a factor of 2 (see Table V in [227]). We caution, however, that asynchronous circuits tend to be difficult to design and require a lot of handcrafting by experienced designers.

Designing for the Worst Case: Bad for Power Consumption

In paragraph *Subthreshold Logic* (page 203), we explored the use of low-voltage techniques to lower switching power. Lowering supply voltages even further may be beneficial for power consumption, but can lead to a dramatic increase in the soft failure rate of computations. For good reason, integrated designers have historically avoided doing this and have produced circuits that can handle worst-case scenarios. An active area of research is on the use of fault-tolerant computing techniques to alleviate these soft errors, while still leading to an overall reduction in energy consumption. Example papers include [129] and [339].

4.1.4. Application Example: Baseband Communication Circuits

Based on the previous subsection, we see that power optimizations are available at many levels of hierarchy (algorithm, architecture, circuit,

device), and that these optimizations often tend to be specific to a particular application domain.

In this subsection, we take a look at an application, namely the design of energy-efficient high-throughput forward error control encoders and decoders, and we see how several power optimization techniques can be used.

A Brief Historical Overview

In 1948, Claude Shannon published his famous paper on the capacity limit for communication channels that are impaired by random noise [277]. Shannon's fundamental results have since been used to design communication equipment, whether information is to be communicated in space (wireless, wireline) or in time (memory). On noisy channels, *channel coding* can be applied to protect a data transmission against transmission errors. One form of coding, known as *forward error control* (FEC) introduces controlled redundancy that can be used to recover incorrectly transmitted bits. Over the additive white Gaussian noise (AWGN) channel, a bound exists on the minimum signal-to-noise ratio (SNR) required to achieve a given bit error rate (BER).

Early FEC schemes included Hamming codes [128] and BCH codes [41]. However, these codes operate several dB away from the Shannon bound. In other words, much more energy must be expended at the transmitter that what the Shannon bound postulates. In 1955, Elias introduced convolutional codes [94], a trellis-structured code; Viterbi later published a well-known, efficient maximum-likelihood decoding algorithm [329]. However, even the most powerful convolutional codes operate several dB from the Shannon bound.

Low-Density Parity-Check (LDPC) codes were first proposed by Gallager in the early 1960s [109], and were the first family of error-control codes to approach the Shannon bound. Unfortunately, maximum likelihood decoding techniques are not appropriate to use with LDPC codes and their close cousins, Turbo codes. One of Gallager's proposals was to use an *iterative* decoding algorithm (in FEC-based systems,

decoding is usually a far more complicated process than encoding), but since iterative decoding requires complicated numerical processing, the first practical implementations of LDPC codes did not emerge until the early 2000s [36], well after Turbo codes had been discovered by Berrou *et al.* in 1993 [27], and after several implementations of Turbo decoders had been reported (see e.g., [28]).

After having been long forgotten, LDPC codes were rediscovered by MacKay and Neal in the mid-1990s [184], which sparked a significant research effort on the theoretical and implementation fronts. Since then, LDPC and Turbo codes have been included in several communication standards, including: IEEE 802.3an (10GBASE-T Ethernet) [143], IEEE 802.11n (WiFi) [141], IEEE 802.16m (WiMax) [142], and LTE [1].

Turbo and LDPC decoders use various formulations of the sum-product algorithm [170], an iterative algorithm that processes *a posteriori* probabilities over graph-based models of a code. Due to their numerically intensive nature and seeming randomness of the underlying dataflow diagram, iterative decoders often consume significant power, especially for high-speed implementations. Furthermore, Turbo and LDPC codes are now being considered for ever-higher throughput applications such as 40 and 100 Gb/s Ethernet [144], and beyond. Since switching power and short-circuit power increase with clock frequency, these new speed requirements will make it extremely difficult for circuit designers to produce energy-efficient decoder circuits.

Energy-efficient Decoding

Unlike convolutional codes, it is not computationally feasible to perform maximum-likelihood decoding for LDPC and Turbo codes. The trellis structure that underlies convolutional codes is not available. A maximum-likelihood LDPC decoding algorithm would essentially have to calculate the distance between a received data vector and each possible codeword. The LDPC code used in the 10GBASE-T standard has 2^{1723} codewords, so this approach is clearly unfeasible.

On the other hand, iterative techniques such as the sum-product al-

gorithm (SPA) are near-optimal in terms of BER performance. For brevity, we will not go into the details of the SPA; a good description can be found in [267]. Instead, we focus on those specific aspects of the algorithm that lend themselves well to energy-efficient designs. Of course, the reader is cautioned that only the tip of the iceberg can be examined!

Exploiting Parallelism. There is a very large degree of parallelism in the SPA. For instance, an n-bit code with k parity-check constraints can be implemented using a bank of n *variable node processors* (VNPs) and k *parity-check node processors* (PCNPs), where each processor performs a simple operation on probabilities or log-likelihood ratios. During even clock cycles, the n VNPs have no data inter-dependencies and hence they can perform their operations in parallel. Similarly, during odd clock cycles, the k PCNPs have no data inter-dependencies so they also can perform their operations in parallel. To give an example, the LDPC code in the 10GBASE-T standard has $n = 2048$ and $k = 325$.

The difficulty in using this parallelism is that from one clock cycle to the next, each processor must communicate its outputs to a seemingly random set of other processors. Since the processors are laid out in two-dimensional silicon space, there is a certain likelihood that some messages have to be sent across the entire chip. This is a slow process, but it is also energy-inefficient (remember: long wires are bad!).

One of the first published LDPC decoder circuits exploited parallelism [36], and consumed approximately 1.4 nJ per uncoded bit, for $n = 1024$ and $k = 512$. The decoder operates at 500 Mb/s (uncoded). More recently, other decoders have been reported that significantly improve upon the earlier results [315].

Reducing Switching Activity. Due to the iterative nature of the SPA, messages that are computed by each computational node tend to converge over time. This property can be exploited to produce wires with low switching activity, but a suitable architecture must be chosen.

Consider Figure 4.7 and suppose the SPA messages in (a) are encoded

(a) Sequence: 1, 4, 6, 8, 9, 9, 9, 9

(b) Unidirectional parallel wires: seven switching events

```
0 0 0 1 1 1 1 1
0 1 1 0 0 0 0 0
0 0 1 0 0 0 0 0
1 0 0 0 1 1 1 1
```

(c) Bidirectional parallel wires: expect 30 switching events

```
0 X 0 X 1 X 1 X 0 X 0 X 1 X 1 X
0 X 1 X 0 X 0 X 0 X 1 X 0 X 0 X
0 X 1 X 0 X 0 X 0 X 1 X 0 X 0 X
1 X 0 X 1 X 1 X 1 X 0 X 1 X 1 X
```

(d) Bidirectional parallel wires: 17 switching events

```
00010100011010001001100110011001
```

Figure 4.7. Switching activity dependence on architecture: (a) example sequence, (b) sequence transmitted over four wires, (c) sequence transmitted over four wires that transmit messages in both directions, and where opposite-direction messages are unknown, and (d) sequence transmitted bit-serially over one wire.

as unsigned binary sequences. Figure 4.7 (b) shows an example where 4-bit messages are transmitted over a unidirectional bundle of 4 wires. Since the messages converge, eventually there is no more switching, and a total of seven switching events occur.

Figure 4.7 (c) shows an example where messages are sent from one type of node to the other during odd cycles, and backwards during even cycles. Since consecutive bits are from unrelated messages, we expect to see 30 switching events. This is a more than 4x increase even though only twice as much data is sent, so this architecture does not lead low switching activity.

Finally, Figure 4.7 (d) shows an example where messages are sent bit-serially. Here, consecutive bits are from different bit positions, so once again convergence does not lead to lower switching activity. A total of 17 switching events are observed. Although there are a greater number of switching events, only one wire is used, and many researchers have investigated these types of decoders due to their compactness [46, 75].

Research on reducing the switching activity of LDPC decoders has

been reported in [72, 112, 113].

Controlling Leakage Power. A recent paper by Le Coz *et al.* [176] reported an LDPC decoder that was implemented in two different 65 nm technologies: a conventional low-power bulk CMOS (LP CMOS) process and a low-power partially depleted silicon-on-insulator (LP PD-SOI) process. Not only was the throughput for the LP PD-SOI decoder approximately 20% higher than that of the LP-CMOS decoder, but at comparable operating frequencies the LP PD-SOI decoder had lower switching and short-circuit power, and up to 50% lower leakage power. These results demonstrate convincingly that the choice of implementation (device) technology can also have a significant impact on power consumption.

4.1.5. How Low Can Power Go?

In this section, we have seen that power consumption optimizations are available at the algorithmic, architectural, circuit, and device levels. A valid question to ask is: how low can power consumption go? Are there any known tight lower bounds? The unfortunate answer is that we simply do not know.

Earlier we alluded to the Landauer bound, which states that zero-energy computation can only be done using a reversible process, and that any bit that is destroyed incurs an energy cost of $k_B T ln(2)$. We also reported a strikingly similar result: that the lowest supply voltage at which a logic gate can operate while still acting as an amplifier is only a few times larger than $k_B T/q$. These two results should not be confused. The latter result does not imply any minimum level of energy dissipated as heat; in fact the dissipated heat could conceivably be very large if the load capacitance C_L driven by the gate is big.

Furthermore, at room temperature, $k_B T ln(2)$ is about $2.9 \times 10^{-21} J$, whereas $C_L V_{DD}^2$ at $V_{DD} = 70mV$ and using $C_L = 1fF$ is still about $5 \times 10^{-18} J$, more or less three orders of magnitude larger. We conclude that there is a lot of room for improvement. We must also conclude that processes that rely on switching voltages on capacitances, i.e., exactly what CMOS circuits do and will continue to do even in 10

nm technologies, will remain far from the Landauer limit unless load capacitances can be significantly reduced through scaling. Even then, noise variations in device performance may require supply voltages that are significantly higher than the minimum.

Clearly, more research is required to achieve a breakthrough in power consumption. There is possibly a lesson to be learned from the history of advances in error control coding. More than 40 years separated Claude Shannon's 1948 paper on channel capacity and Claude Berrou's 1993 discovery of capacity-approaching Turbo codes. In the meantime, several families of codes were proposed. None, other than Gallager's rapidly forgotten LDPC codes, approached the limit; for the longest time coding theorists believed that there might be some other unknown bound on communication. The reality was that Shannon's bound was in fact quite close to what was practically achievable, and the problem just required some thought by curious and creative people.

Perhaps the same will happen with low-power design. However, it will require some fundamental research in the fields of mathematics, physics, and engineering. We also point to the exciting field of reversible computing, explored later in this book.

4.2. Permuting Variables to Improve Iterative Re-Synthesis

Petr Fišer Jan Schmidt

4.2.1. Iterative Logic Synthesis

Basic principles of logic synthesis of Boolean networks have been established already in 1960's. The synthesis consists of two subsequent steps: the technology independent optimization and technology mapping.

The technology independent optimization process starts from the initial circuit description (sum of products, truth table, multi-level network) and tries to generate a minimum multi-level circuit description, such as a factored form [126], And-Inverter-Graph (AIG) [151, 213] or a network of BDDs [7, 50]. Then the technology mapping follows [214, 215, 217, 222].

The synthesis process, where the forms of description of its input and output are the same (Boolean networks, AIGs, mapped design, layout), is called *re-synthesis* [216]. Thus, by re-synthesis we understand a process modifying the combinatorial circuit in some way, while keeping the format of its description.

The academic state-of-the-art logic synthesis tool is ABC [23, 47] from Berkeley, a successor of SIS [273] and MVSIS [110]. Individual re-synthesis processes in SIS and ABC are represented by commands. Since the number of available re-synthesis processes is large (e.g., don't care-based node simplification [266], re-writing [213], refactoring, re-substitution [212, 216], it is difficult to determine a universal sequence of these commands leading to optimum results. Thus, different synthesis scripts were proposed (e.g., `script.rugged` and `script.algebraic` in SIS, `resyn` scripts, `choice`, and `dch` in ABC). These scripts are supposed to produce satisfactory results.

The re-synthesis process may be iterated, to further improve the results. Iteration of re-synthesis was proposed in ABC [23, 47] too. The authors of ABC suggest repeating the sequence of technology independent optimization (e.g., the `choice` script) followed by technology mapping several times. Also the synthesis process of SIS can be efficiently iterated. Iterating the complete synthesis process (i.e., the technology independent optimization and technology mapping) will be denoted as *high-level iteration*, in contrast to low-level iteration used, e.g., as a part of individual synthesis steps.

The necessary condition for using high-level iteration is that the network structure must not be completely destroyed in the process, e.g., by collapsing it into a two-level (SOP) form or turning it into a global BDD [7], [50]. Then all the effort made in previous iteration would be in vain. Fortunately, this is not the case of the synthesis scripts mentioned above.

In a typical iterative re-synthesis algorithm, the result quality (area, delay) gradually improves in time, until it converges to a stable solution. In an ideal case, it reaches the global optimum. However, the process usually quickly converges to a local optimum, which is sometimes far from the global one. Thus, introducing some kind of *diversification*, as known in other iterative optimization processes [117, 118, 159], could be beneficial. One way of "painlessly" introducing diversity into the iterative process is the topic of this section.

4.2.2. Randomness in Logic Synthesis

Most of synthesis processes in ABC are greedy and not systematic. They use some heuristic functions to guide the search for the solution. Even though the heuristic is usually deterministic, there are often more equally valued choices. In such situations, the first occurrence is taken. Note that these choices are equally valued just at the point of decision and they will most likely influence the subsequent decisions. Therefore, different choices can produce different results. A typical example of such a behavior is the topological traversal of AIG nodes in algorithms employed in ABC, where there are usually many nodes at the same topological level. Then the node with the lowest ID (the first

one encountered) is usually taken [212, 213]. Since different orderings of input and output variables will involve different AIGs or differently arranged networks (in sense of their internal representation), the result can be significantly influenced by the ordering [101].

Because of this observation, different runs of one process with different variable orderings produce different results. We can take advantage of this, in order to diversify the search for the solution.

A method of using random permutations of input and output variables is described in this section. The order of variables is randomly changed at the beginning of each iteration. Thus, randomness is painlessly introduced into the process; the very synthesis and optimization algorithms need not be modified.

A similar approach, where randomness was introduced *from outside*, was published in [102, 103]. Here randomly extracted large parts of the circuit are synthesized separately, in an iterative way, too. The method presented here is a special case of this, therefore it is inferior to [102, 103], in terms of the result quality. However, extraction of the parts involves some computational overhead. Since the random permutations are made in time linear with the number of variables, no noticeable time overhead is involved. Therefore, the main message of this section is to document that using random permutations just pays off.

4.2.3. The Influence of the Source File Structure

As it was stated above, we can observe that many synthesis processes are not immune to the structure of the source file, like the ordering of variables [101, 104] and ordering and syntax of HDL statements [242]. Therefore, different runs of one process with differently structured source file produce different results. Possible reasons for it will be discussed in this subsection and some quantitative results will be given.

Typically, variables in the synthesis algorithms are processed in a lexicographical order, which is defined a priori, usually by their order

in the source file. Then, different orderings of variables can make heuristic algorithms run differently, possibly producing different (but definitely still correct) results. A typical and well known example of such a behavior are BDDs [7, 50]. Here the ordering of variables is essential; the BDD size may explode exponentially with the number of variables under a "bad" ordering [7].

Computing the optimum ordering of variables is NP-hard itself [38], thus infeasible in practice. Even though there are efficient heuristics for determining a possibly good variable ordering [252], they consume some time, whereas do not guarantee any success, and thus they are usually not employed in practice. Typically, the default variable ordering in the BDD manipulation package CUDD [282] (which is used in SIS and ABC too) is *just equal* to the ordering of variables in the source file—no reordering technique is employed.

Another, and more important example, is the topological traversal of nodes in algorithms implemented in ABC and SIS. Even different orderings of input and output variables will involve different AIGs or differently arranged networks (in sense of their internal representation), see Subsection 4.2.2.

Also the well-known two-level Boolean minimizer ESPRESSO [48] (which is used both in SIS and ABC too) is sensitive to the ordering of variables. There are many essential parts of the overall algorithm, where decisions are made in a lexicographical way. Some decisions do not influence the result quality; they just can influence the run-time (e.g., in the tautology checking process), some do influence the result as well (e.g., the irredundant phase) [48].

Therefore, even changing the variable ordering in the input file header (be it PLA [48] for ESPRESSO or BLIF [22] for ABC) can significantly affect the algorithms runs and induce different results. It will be documented here, how serious differences there are in practice.

Also, some commercial synthesis tools are sensitive even to the order of nodes (which are coordinate RTL statements). This issue will be documented here as well, without any attempt for explanation. For another experimental study of the influence of small modifications of the RTL code on the result, see [242].

Experimental evaluation of several basic optimization and technology mapping commands in ABC [23], technology independent optimization scripts (which comprise of the basic synthesis commands), and complete synthesis scripts, targeted to standard cells (the `strash; dch; map` script) and look-up tables (4-LUTs), the `strash; dch; if; mfs` script, will be presented here. Finally, results of ESPRESSO [48] and even ESPRESSO-EXACT are shown. The dependency on both the input and output variables ordering will be studied. No influence of the PLA terms ordering or nodes ordering in BLIF was observed in ESPRESSO or any of the studied ABC processes.

The ABC experiments were conducted as follows: 228 benchmarks from the IWLS and LGSynth benchmarks sets [191, 344] were processed. Given a benchmark, its inputs and/or outputs were randomly permuted in the source BLIF file [22] (or PLA for ESPRESSO), the synthesis command was executed, and the number of AIG nodes, gates, LUTs or literals, respectively, was measured. This was repeated 1,000-times for each circuit. In order to compactly represent all the results, the maximum and average percentages of size differences (minimum vs. maximum) were computed, over all the 228 circuits. The results are shown in Tables 4.2, 4.3, and 4.4.

We can observe striking size differences (up to more than 95%), especially for the complete synthesis processes. Even the numbers of literals obtained by ESPRESSO-EXACT differ, since ESPRESSO EXACT guarantees minimality of the number of terms only, nothing is guaranteed for literals.

Distributions of frequencies of occurrence of solutions of a given size are shown in Figure 4.8 for the `apex2` and `cordic` circuits [344]. The complete 4-LUT synthesis script (`strash; dch; if; mfs`) was executed, for 100,000 different orderings of variables. The result obtained using the original ordering is indicated by the bold vertical line.

We can see a Gaussian-like distribution for the `apex2` circuit, or actually, a superposition of two Gaussian distributions. Even the original ordering of variables falls to the "better" part of the chart.

For the `cordic` circuit, we can observe two completely isolated regions. There are apparently two or more classes of similar implementations

Table 4.2. The influence of permutation of variables – permuted inputs

	Process	Unit	Max.	Avg.
Technology	`balance`	AIG	7.69%	1.04%
independent	`rewrite`	AIG	15.38%	0.68%
optimization:	`refactor`	AIG	12.07%	0.36%
commands	`resub`	AIG	2.50%	0.06%
Technology	`resyn2`	AIG	44.53%	4.60%
independent	`resyn3`	AIG	13.56%	1.57%
optimization:	`choice`	AIG	34.40%	7.17%
scripts	`dch`	AIG	60.53%	10.42%
Technology	`map`	gates	17.09%	1.35%
mapping	`fpga`	LUTs	0.00%	0.00%
	`if`	LUTs	0.00%	0.00%
Complete	`strash; dch; map`	gates	74.38%	8.67%
synthesis	`strash; dch; if; mfs`	LUTs	92.14%	11.50%
Two-level	`ESPRESSO`	literals	34.90%	1.51%
optimization	`ESPRESSO-EXACT`	literals	0.63%	0.02%

(similar in size, probably similar in structure too), which synthesis produce depending on the ordering of variables. This phenomenon is still under examination, reasons for it are disputable. Note that the `apex2` case also shows hints of two structurally different classes of solutions.

Dependency of the result quality on the ordering of variables was observed in commercial tools too. Two tools were studied and both were found to be very sensitive to the structure of the HDL statements. Surprisingly enough, the tools were also sensitive to a mere reordering of the gates instantiation, i.e., coordinate statements in the HDL code, which was not the case of any examined process in ABC.

The experiment started with BLIF [22] descriptions and after permuting the variables (and nodes in the BLIF file), each benchmark was converted to VHDL and processed by commercial LUT mapping synthesis. The numbers of 4-LUTs in the results were measured. Sum-

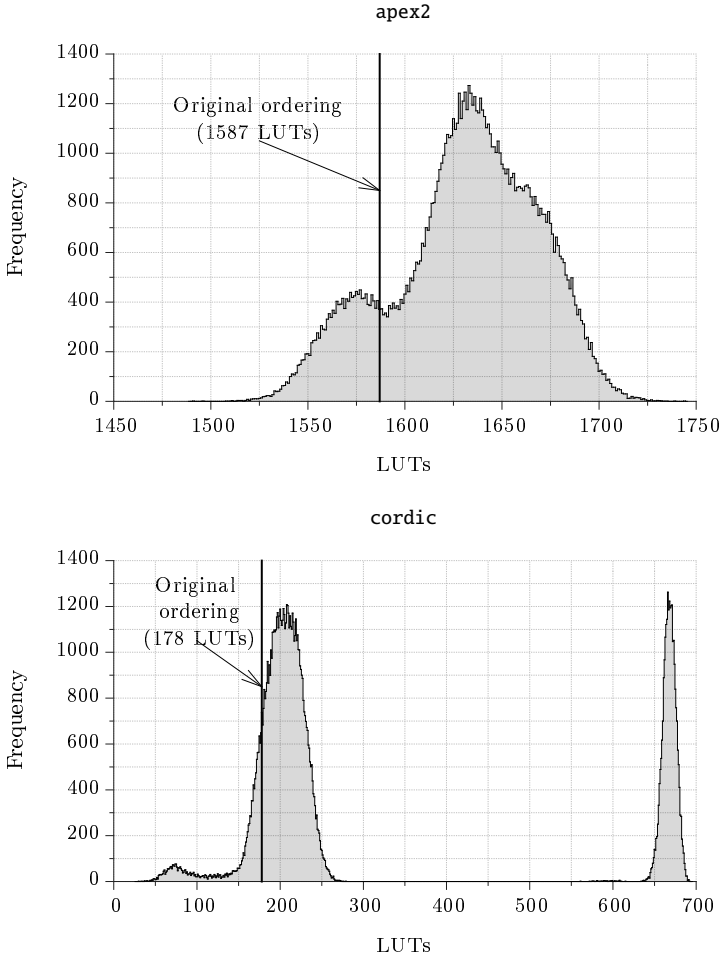


Figure 4.8. Distribution of solutions for the circuits `apex2` and `cordic`.

Table 4.3. The influence of permutation of variables – permuted outputs

	Process	Unit	Max.	Avg.
Technology	`balance`	AIG	11.48%	1.60%
independent	`rewrite`	AIG	19.30%	2.41%
optimization:	`refactor`	AIG	29.73%	2.49%
commands	`resub`	AIG	20.83%	1.70%
Technology	`resyn2`	AIG	52.75%	5.58%
independent	`resyn3`	AIG	22.50%	2.74%
optimization:	`choice`	AIG	38.14%	7.14%
scripts	`dch`	AIG	40.39%	9.33%
Technology	`map`	gates	12.28%	1.93%
mapping	`fpga`	LUTs	5.26%	0.29%
	`if`	LUTs	2.88%	0.24%
Complete	`strash; dch; map`	gates	70.47%	10.52%
synthesis	`strash; dch; if; mfs`	LUTs	85.42%	12.60%
Two-level	`ESPRESSO`	literals	11.82%	1.04%
optimization	`ESPRESSO-EXACT`	literals	6.06%	0.23%

mary results of the 228 benchmarks [191, 344] are shown in Table 4.5. Again, maximum and average differences of the obtained LUT counts are shown.

4.2.4. The Proposed Method

Keeping all the above observations in mind, all the studied algorithms that claim to be deterministic are not deterministic at all, actually. The initial variable ordering shall be considered as random as any other random ordering. But anyway, the algorithms should be designed to succeed under any ordering. Therefore, introducing random ordering to the synthesis process should not make the process perform worse. From the search space point of view, the global optimum is approached from different sides.

Table 4.4. The influence of permutation of variables – permuted inputs & outputs

	Process	Unit	Max.	Avg.
Technology independent optimization: commands	`balance`	AIG	12.50%	2.27%
	`rewrite`	AIG	19.13%	2.78%
	`refactor`	AIG	29.73%	2.79%
	`resub`	AIG	20.83%	1.71%
Technology independent optimization: scripts	`resyn2`	AIG	52.69%	7.38%
	`resyn3`	AIG	22.66%	3.72%
	`choice`	AIG	36.17%	10.13%
	`dch`	AIG	60.50%	13.50%
Technology mapping	`map`	gates	17.09%	2.84%
	`fpga`	LUTs	5.26%	0.29%
	`if`	LUTs	2.88%	0.24%
Complete synthesis	`strash; dch; map`	gates	86.27%	13.40%
	`strash; dch; if; mfs`	LUTs	**95.07%**	14.81%
Two-level optimization	`ESPRESSO`	literals	42.95%	2.11%
	`ESPRESSO-EXACT`	literals	6.06%	0.24%

Next we may also think about exploiting these facts to systematically improve logic synthesis. In particular, in *iterative re-synthesis*.

The state-of-the-art high-level iterative process, as it can be used, e.g., in ABC, can be described as follows: first, an internal representation (SOP, AIG, network of gates, network of BDDs, etc.) for the technology independent optimization is generated from the initial description or a mapped network (e.g., from a BLIF file [22]). Then a technology independent optimization, followed by technology mapping is performed. The process is repeated (iterated), until the stopping condition (number of iterations, result quality, timeout, etc.) is satisfied, see Figure 4.9.

The general aim of the process is to transform the initial circuit description into the target technology (ASIC library gates, FPGA LUTs), while trying to optimize the quality (size, delay, power con-

Table 4.5. The influence of permutation of variables and nodes – commercial tools

Tool		#1	#2
Permuted inputs	Max.	0.00%	43.62%
	Avg.	0.00%	4.71%
Permuted outputs	Max.	0.00%	52.19%
	Avg.	0.00%	5.57%
Permuted nodes	Max.	15.76%	38.81%
	Avg.	0.21%	3.40%
Permuted all	Max.	17.26%	66.62%
	Avg.	0.22%	9.23%

sumption) of the solution.

Assuming that each iteration does not deteriorate the solution, the solution quality improves in time. This needs not be true in practice, however. For such cases, several options are possible:

1. to hope that the overall process will "recover" from a small deterioration,

2. to accept only improving (non-deteriorating) iterations,

3. to record the best solution ever obtained and return it as the final result,

4. combination of 1. and 3.

The first and the last options are usually used in practice.

Usually it happens that the iterative process quickly converges to a stable solution, which does not improve any more in time. In an ideal case, it is the best possible solution (global optimum). However, this is usually not the case in practice; such an iterative process tends to get stuck in a local optimum [102, 104]. Just a slight modification of the algorithm from Figure 4.9 might help to escape local optima and thus improve the iterative power of the re-synthesis, see the algorithm

```
do
     generate_internal_representation
     technology_independent_optimization
     technology_mapping
 while (!stop)
```

Figure 4.9. The iterative re-synthesis.

```
do
     randomly_permute_variables
     generate_internal_representation
     technology_independent_optimization
     technology_mapping
 while (!stop)
```

Figure 4.10. The iterative re-synthesis with random permutations.

in Figure 4.10.

Here only the `randomly_permute_variables` step was added, where random reordering of variables (input, output, or both) is performed. This step can be executed in a time linear with the number of variables, hence it does not bring any significant time overhead.

Note that, unlike in the previous subsection, the reordering of variables is performed in each iteration, not only at the beginning of the iterative synthesis process. Therefore, the permutations effects may accumulate.

4.2.5. Experimental Results

Very exhaustive experiments were performed in order to justify the benefit of using random permutation of variables in the high-level iteration process. There were processed 490 benchmark circuits, coming from academic IWLS [344] and LGSynth93 [191] benchmark suites,

as well as from large industrial designs from OpenCores [228] (having up to 100,000 LUTs after synthesis). The 4-LUTs mapping process was chosen for testing purposes. However, the same behavior can be expected for any target technology.

The most recent LUT-mapping synthesis script suggested by the authors of ABC was used: `strash; dch; if; mfs; print_stats -b` as a reference. Then, the ABC command `permute` randomly permuting both inputs and outputs was employed, yielding the script `permute; strash; dch; if; mfs; print_stats -b`. Both scripts were executed 20-, 100-, 1,000-, and 5,000-times for each circuit, while the best result ever obtained was recorded and returned as the solution (this is accomplished by the `print_stats -b` command). The numbers of resulting 4-LUTs and the delay (in terms of the length of the longest path—the circuit levels) were measured.

Results for all the 490 circuits are shown in Figure 4.11 and Figure 4.12, for area (4-LUTs) and delay (levels), respectively. The scatter-graphs visualize the relative improvements w.r.t. no permutations used. Positive values indicate an improvement, the negative ones deterioration. The size of the original mapped circuit, in terms of 4-LUTs, is indicated on the x-axis. Two border cases, 20 and 5,000 iterations are shown here only. Results of 100 and 1,000 iterations lay in-between.

We see that a significant improvement can be reached even when the process is run for 20 iterations only. However, also more deteriorating cases are observed. When iterated more, the results become more positive, especially for larger circuits. This is quite obvious, since these circuits usually converge slower (see Subsection 4.2.6).

Summary statistics are shown in Tables 4.6 and 4.7, for the number of LUTs and levels, respectively. Only 290 circuits, whose resulting implementation exceeded 100 LUTs, were accounted in these statistics, to make the practical impact more credible. The minimum, maximum and average percentage improvements for both area and delay are given. Also the percentages of cases, where the improvement is positive (*Better in*) and negative (*Worse in*), are shown. The complement to 100% of the sum of these two values represents cases where solutions of equal quality (LUTs, levels) were obtained.

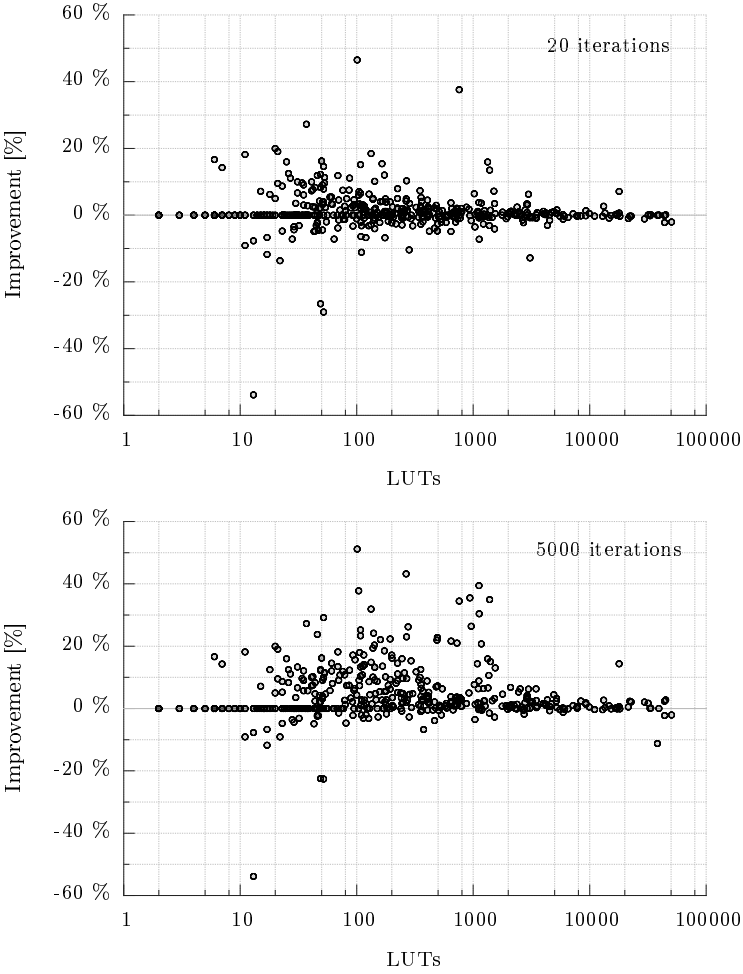


Figure 4.11. Area improvements w.r.t the standard iterative process.

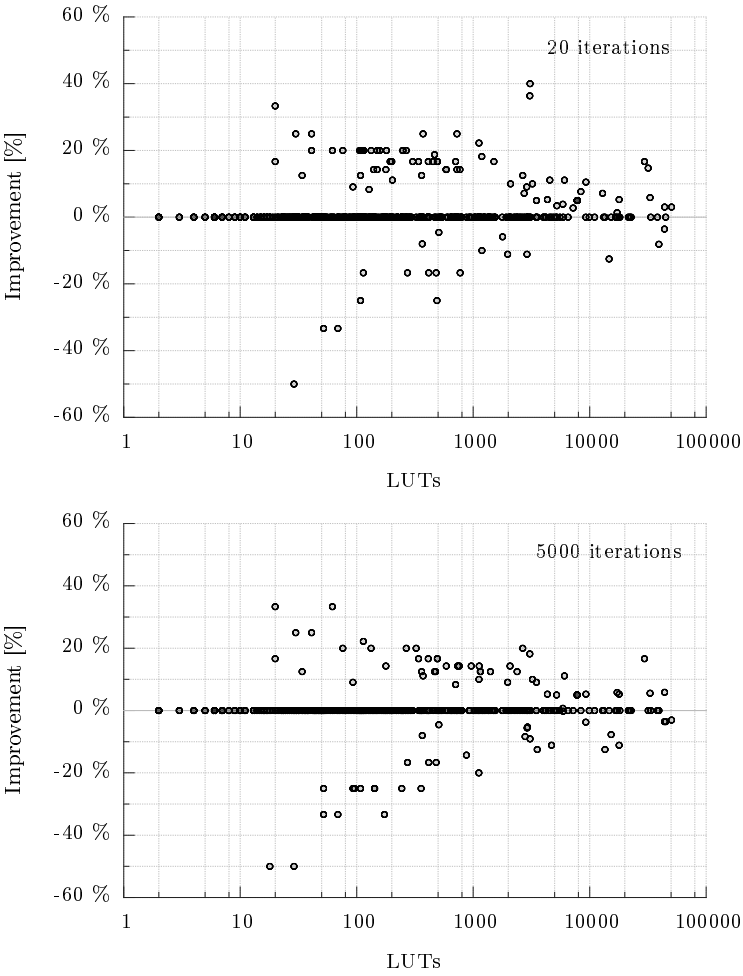


Figure 4.12. Delay improvements w.r.t the standard iterative process.

Table 4.6. Summary statistics – LUTs

Iterations	20	100	1,000	5,000
Minimum	-12.80%	-8.20%	-5.40%	-6.70%
Maximum	46.50%	51.20%	74.60%	75.20%
Average	1.00%	2.10%	4.90%	6.10%
Better in	52.20%	64.90%	81.00%	82.60%
Worse in	39.80%	28.80%	15.20%	13.90

Table 4.7. Summary statistics – levels

Iterations	20	100	1,000	5,000
Minimum	-33.30%	-33.30%	-25.00%	-25.00%
Maximum	22.20%	27.30%	40.00%	40.00%
Average	0.60%	0.60%	1.60%	2.50%
Better in	16.30%	13.80%	19.70%	23.90%
Worse in	9.30%	5.50%	6.20%	5.50%

We see that with an increasing number of iterations, the results become more stable and tend to improve, both in area and delay. There is a positive average improvement obtained even for the 20 iterations run. Moreover, the percentage of cases where improvement was obtained is less than the percentage of deteriorating cases.

For the 5,000 iterations case the average improvement reaches 6.1% in area and 2.5% in delay. Also cases, where deterioration was obtained, are becoming even more rare (13.9% and 5.5% for area and delay, respectively).

Let us make a theoretical reasoning about the observed facts now. Assume the worst case, where the number of deteriorating solutions (w.r.t. the process with no permutations used) of one iteration of re-synthesis is 50% (equal chance for both the improvement and deterioration). Then, also chances for improvement of the overall process would be 50%.

However, in Tables 4.6 and 4.7 we see that all the minimum improvements (maximum deteriorations) are much less than 50%, even for 20

iterations. From these figures we can conclude that using permutation in synthesis always pays off.

4.2.6. Convergence Analysis

Illustrative examples of convergence curves for the iterative synthesis with and without using random permutations for two of the LGSynth benchmark circuits [191] `alu4` and `apex2` are shown in Figure 4.13. The progress of the size reduction during 1,000 iterations was traced.

Here we see the experimental justification of the presented theory. In general, it is not possible to say what method converges faster. Theoretically, both should converge equally fast. This can be seen, e.g., in the `alu4` case, where the standard synthesis converges faster at the beginning, but then the convergence slows down. When the re-synthesis without using permutations converges to a local minimum, the permutations will help to escape it (see the `apex2` curves—here the local minimum was reached around the 300th iteration, whereas the solution quality still improves after 1,000 iterations when permutations are used). Similar behavior can be observed for most of the tested circuits. This confirms the theory—the permutations do increase the iterative power and help to keep the convergence longer.

4.2.7. Advantages of Re-Synthesis with Permutations

Experiments presented in this section have shown that the property of synthesis algorithms documented in Subsection 4.2.3—dependency on the ordering of variables in the initial description—can be advantageously exploited to increase the iterative power of re-synthesis.

A positive average improvement in quality (both in area and delay) was obtained. Since introducing the permutations into the iterative process takes almost no time, it can be concluded that employing random permutations definitely pays off—random permutations help avoiding local optima. Cases, where worse results are obtained, are relatively rare.

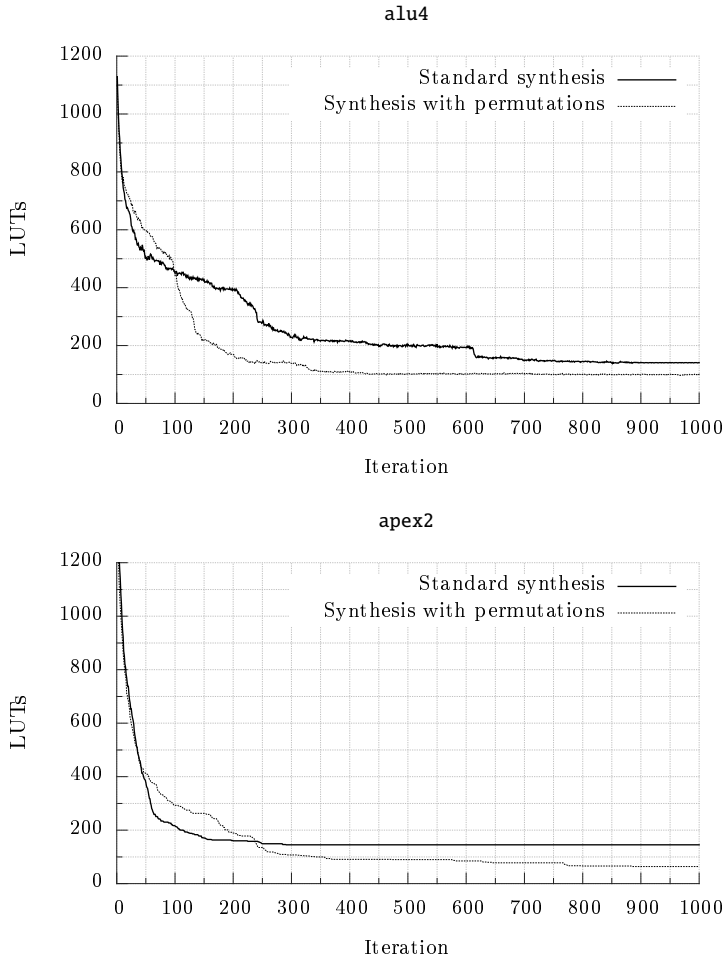


Figure 4.13. Convergence curves for the circuits `alu4` and `apex2`.

Permutation also offers a possibility of obtaining many different solutions, possibly having the same quality (under any quality measure). This feature can be exploited in subsequent synthesis, e.g., a secondary quality criterion may be applied. More details can be found in [101] and [104].

4.3. Beads and Shapes of Decision Diagrams

Stanislav Stanković Jaakko Astola

Radomir S. Stanković

4.3.1. Three Concepts

In this section, we discuss the relationships between three concepts:

1. the decision diagram (DD) as a representation of a discrete function,

2. the intuitive concept of the shape of the decision diagram, and

3. a special type of character (binary) sequences, the so-called beads.

In [160], Donald Knuth used beads to describe binary decision diagrams (BDDs) and to express the assignment of functions to BDDs in terms of beads instead of referring to the decomposition rules. In this section, beads are used to express the shape of decision diagrams in terms of sets of beads. Relating the set of beads with the shape of decision diagrams provides a formalism for the application of the concept of shape as a characteristic of decision diagrams [287], in a manner different from that when decision diagrams are viewed as particular data structures. This opens ways to discuss and use the concept of shape of decision diagrams in certain applications such as, for instance, implementation of functions by mapping decision diagrams to hardware [204], [346], [351], and classification of switching functions [289].

Beads have been used with binary sequences, but the concept generalizes naturally to sequences over arbitrary alphabets. We discuss the use of beads for various word-level decision diagrams including,

for example, multi-terminal binary decision diagrams (MTBDDs) and Walsh decision diagrams (WDDs) [263] that will be discussed in this section.

As it is a customary practice in dealing with decision diagrams, except when discussing their optimization by permutation of variables, we assume that the order of variables in functions to be represented is fixed. This is especially understandable since in this section we are discussing applications of the decision diagrams to the classification of switching functions. Thus, we consider the set of all switching functions for a given number of variables, and the permutation of variables is out of interest since it converts a function into another function in the same set. For the considered applications, we assume that we are working with reduced decision diagrams.

4.3.2. Basic Concepts: Beads, Switching Functions, and Decision Diagrams

Switching Functions

A switching function of n variables is defined as a mapping:

$$f : \{0,1\}^n \rightarrow \{0,1\} \ ,$$

and can be specified by the truth-vector listing all the values that f takes for different inputs usually arranged in the lexicographic order. Thus, the switching function f is specified by the truth-vector:

$$\mathbf{F} = [f(0,0,\ldots,0), f(0,0,\ldots,1), \ldots, f(1,1,\ldots,1)]^T \ .$$

The truth-vector of a function of n variables has the form:

$$\mathbf{F} = [\mathbf{F}_0, \mathbf{F}_1]^T \ .$$

where $\mathbf{F}_0$ and $\mathbf{F}_1$ are the truth-vectors of the functions:

$$\begin{aligned} f_0 &= f(0, x_2, \ldots, x_n) \ , \text{ and} \\ f_1 &= f(1, x_2, \ldots, x_n) \ . \end{aligned}$$

These functions are called the subfunctions of f. In general, a function of n variables has 2^k subfunctions of order $(n-k)$ for $k \in \{0,1,\ldots,n\}$, corresponding to 2^k possible values for the first k variables $(x_1, x_2, \ldots, x_k)$. Many of these subfunctions can be identical or constant subfunctions. It should be noted that a reduced decision diagram gives a decomposition of the function into distinct subfunctions.

Beads

When a switching function f of n-variables is viewed as a binary sequence of length 2^n, then there are direct links between so called beads of the sequence, subfunctions of f, and the corresponding subdiagrams in the decision diagram for f.

Definition 4.13. *(Bead [160]) A binary sequence of length 2^n is called a bead of order n if it cannot be written in the form $\alpha\alpha$, where α is a binary sequence of length 2^{n-1}, otherwise the sequence is called a square of order n.*

In a bead, the left and right subsequences of length 2^{n-1} are beads or squares of order $(n-1)$. Each sub-bead can be split into sub-beads or sub-squares of smaller orders. The beads of the smallest order 0 are singleton sequences (0) and (1). The set of all beads derived from a switching function f is denoted by $B(f)$.

Example 4.5. *Consider the switching function:*

$$f(x_1, x_2, x_3) = x_3 \oplus \overline{x}_1 x_2 \oplus x_1 \overline{x}_2 x_3$$

whose truth-vector is $\mathbf{F} = [01101101]^T$. *The corresponding set of beads of f is*

$$B(f) = \{(01101101), (0110), (1101), (01), (10), (0), (1)\} .$$

The first part of the second bead of order 2 is a square (11) and therefore does not appear in $B(f)$.

A square of order k consists of at least one bead of order $l \leq k-1$.

Example 4.6. *The square* (01010101) *of order 3 consists of the beads* (01) *of order* 1.

In general there are 2^{2^n} switching functions of n variables, and $2^{2^{n-1}}$ of them are squares. Therefore, there are $2^{2^n} - 2^{2^{n-1}}$ beads of order n. The beads of a switching function are subfunctions that happen to be beads [160]. In other words, a bead of order k is a subfunction of k variables whose truth-vector is not a square, i.e., $\mathbf{F}_0 \neq \mathbf{F}_1$.

There can be subfunctions that are squares, and a particular example are the constant subfunctions, i.e., the subfunctions taking always either the value 0 or 1 for any assignment of inputs.

Decision Diagrams

Binary decision diagrams (BDDs) can be rigorously defined as function graphs [51]. Here we use the intuitive concept where the decision diagram of a function is defined as the graph obtained by applying reduction rules to the decision tree that corresponds to the truth table of the function. The following example illustrates some features of BDDs that will be used later.

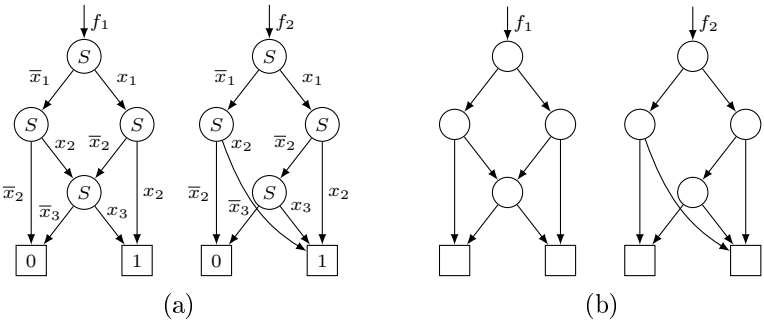


Figure 4.14. BDDs for functions f_1 and f_2 in Example 4.7.

Example 4.7. *Consider BDDs in Figure 4.14 (a) for two functions* f_1 *and* f_2. *The BDD*(f_1) *has three paths from the root node to the constant node* 1 *(*1*-paths) and by multiplying labels at the edges along*

these paths, we see that it represents the function:

$$f_1 = \overline{x}_1 x_2 x_3 \oplus x_1 \overline{x}_2 x_3 \oplus x_1 x_2 \ ,$$

whose truth-vector is $\mathbf{F}_1 = [0,0,0,1,0,1,1,1]^T$. *The* $BDD(f_2)$ *has also three* 1*-paths, however, two paths are of length* 2 *compared with a single path of the same length in the* $BDD(f_1)$*, and represents the function:*

$$f_2 = \overline{x}_1 x_2 \oplus x_1 \overline{x}_2 x_3 \oplus x_1 x_2 \ ,$$

whose truth-vector is $\mathbf{F}_1 = [0,0,1,1,0,1,1,1]^T$. *Figure 4.14 (b) illustrates the intuitive interpretation of the shape of BDDs for these functions.*

4.3.3. Beads and Decision Diagrams

It is shown in [160] that there is a direct correspondence between the set $B(f)$ for a switching function f and nodes in its BDD. This consideration can be further elaborated as follows.

The number of elements in $B(f)$ is equal to the size (the number of nodes) of the $BDD(f)$. Each element is represented by a node in the BDD. The order of a bead determines the level at which the node corresponding to the bead is positioned.

The structure of the bead (its left and right half) determines the outgoing edges of this node. A bead of the form $\alpha = \beta\gamma$ of order k consists of two parts β and γ of orders $(k-1)$. If both β and γ are beads, the left and the right outgoing edges of the node representing α point to the nodes at the level $(k-1)$ representing β and γ, respectively. If β or γ is a square, the corresponding edge points to the node at the level $(k-l)$ representing the bead of order l contained in the square. If β or γ are constant sequences (all their elements are equal to either 0 or 1), which means they are a particular square of order $(k-l)$, the corresponding path ends in the constant node. These rules for establishing edges between nodes have to be fulfilled in a BDD.

Being elements of a set, the beads in $B(f)$ are distinct. Each element in $B(f)$ corresponds to a node which is the root node of a subdiagram

Table 4.8. Sets of beads for functions in Example 4.8

Level	$B(f_1)$	$B(f_2)$	Order of the bead
1	(00010111)	(00110111)	3
2	(0001), (0111)	(0011), (0111)	2
3	(01)	(01)	1
4	(0), (1)	(0), (1)	0

in the $BDD(f)$ and represents a distinct subfunction of f. Identical subfunctions are represented by isomorphic subdiagrams, and a single copy of them is retained in the diagram. In f, there can be subfunctions that are represented by squares, however, as noticed above, each square contains at least a single bead. This bead is an element of $B(f)$, and the repeated copies of it in the square are removed and a path of length longer than 1 appears. Since the $BDD(f)$ is obtained by the recursive decomposition of f into distinct subfunctions (equivalently sub-beads), the outgoing edges of each node point to the sub-beads of the bead represented by the considered node. Therefore, the set of beads uniquely determines the function and, thus, also the shape of the BDD for a given function f.

Example 4.8. *Table 4.8 shows the sets $B(f_1)$ and $B(f_2)$ of beads of functions f_1 and f_2 discussed in Example 4.7. Beads are ordered by their orders, i.e., by the levels in the BDDs. In $B(f_1)$, there is a single bead of order* 1, (01)*, which means that the node representing it is shared between nodes representing beads of order* 2 *at the level* 2*. This determines that an edge of these nodes points to the node representing the bead* (01)*. Since the first half of the bead* (0001) *is a square of order* 1, (00) *and does not produce another bead of order* 1*, the other edge of this node points to the node for the bead* (0)*, i.e., the constant node* 0*. It is similar for the right half of the bead* (0111)*, since it is the square* (11) *of order* 1 *and the other edge of this node points to the constant node* 1*.*

In $B(f_2)$, there are also two beads of order 2*, and a bead of order* 1*. The first bead of order* 2 *consists of two squares of order* 1, (00) *and* (11) *and, therefore, the bead* (01) *cannot be shared and the outgoing edges of this node point to the constant nodes. The right half of the bead* (0111) *is a square* (11) *and it follows that the corresponding edge*

Table 4.9. Sets of beads for functions in Example 4.9

Level	$B(f_{AND})$	$B(f_{OR})$	$B(f_{EXOR})$	$B(f_e)$
1	(000000001)	(01111111)	(01101001)	(00010010)
2	(0001)	(0111)	(0110), (1001)	(0001), (0010)
3	(01)	(01)	(01), (10)	(01), (10)
4	(0), (1)	(0), (1)	(0), (1)	(0), (1)

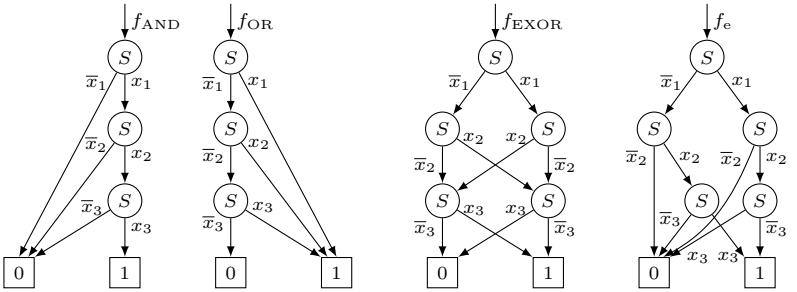


Figure 4.15. BDDs for functions f_{AND}, f_{OR}, f_{EXOR} and f_{e} in Example 4.9.

points to the constant node 1. *The other edge points to the node representing the bead* (01).

The following example presents bead sets for several characteristic BDDs.

Example 4.9. *Figure 4.15 shows BDDs of the functions*

$$\begin{aligned} f_{AND} &= x_1x_2x_3 \ , \\ f_{OR} &= x_1 \vee x_2 \vee x_3 \ , \\ f_{EXOR} &= x_1 \oplus x_2 \oplus x_3 \ , \\ f_e &= \overline{x}_1x_2x_3 \oplus x_1x_2\overline{x}_3 \ . \end{aligned}$$

Table 4.9 shows the bead sets of these functions.

Example 4.10. *Consider three variable switching functions:*

$$\begin{aligned} f_1 &= x_1\overline{x}_2 \oplus x_1x_2x_3 \oplus \overline{x}_1\overline{x}_2x_3 \oplus \overline{x}_1x_2\overline{x}_3 \ , \\ f_2 &= \overline{x}_1\overline{x}_2 \oplus \overline{x}_1x_2x_3 \oplus x_1\overline{x}_2x_3 \oplus x_1x_2\overline{x}_3 \ , \text{ and} \\ f_3 &= x_1\overline{x}_2 \oplus x_1x_2x_3 \oplus \overline{x}_1x_2x_3 \oplus \overline{x}_1\overline{x}_2\overline{x}_3 \ , \end{aligned}$$

Table 4.10. Sets of beads for functions in Example 4.10

Level	$B(f_1)$	$B(f_2)$	$B(f_3)$
1	(01101101)	(11010110)	(10011101)
2	(0110), (1101)	(1101), (0110)	(1001), (1101)
3	(01), (10)	(01), (10)	(10), (01)
4	(0), (1)	(0), (1)	(0), (1)

specified also by the truth-vectors:

$$\mathbf{F}_1 = [01101101]^T \ ,$$
$$\mathbf{F}_2 = [11010110]^T \ , \text{ and}$$
$$\mathbf{F}_3 = [10011101]^T \ .$$

As evident from Figure 4.16, the $BDD(f_1)$, $BDD(f_2)$, and $BDD(f_3)$ have the same shape, while $B(f_1)$, $B(f_2)$, and $B(f_3)$ are different as shown in Table 4.10. The sets $B(f_1)$ and $B(f_2)$ differ just in the first elements, while there is a larger difference with respect to $B(f_3)$, although these different functions are, however, mutually related. Their BDDs differ in the labels at the edges, which is consistent with the intuitive understanding of the concept of the shape of a decision diagram. In $BDD(f_2)$, permuted are labels at the edges of the node for x_1, which corresponds to the complementing of x_1 in f_1. In $BDD(f_3)$, permuted are labels at the edges corresponding to the left node for x_2. This corresponds to the complementing of x_2 in the right half of the truth-vector of f_1, but not in the left half. The example illustrates that we can perform certain global (over the entire truth-vector) and local (over sub-vectors) transformations in a given function to produce new functions that will have decision diagrams of the same shape. It should be noticed that these transformations are different from transformations used in other approaches to the classification of switching functions and related applications in the implementation of switching functions [77, 84, 319, 351].

Intuitively we understand that two decision diagrams have the same shape if their underlying directed graphs are isomorphic and the nodes that correspond to each other under this isomorphism are on the same levels. This means that as function graphs [51] the diagrams are isomorphic without regard to the order of variables and assignment of

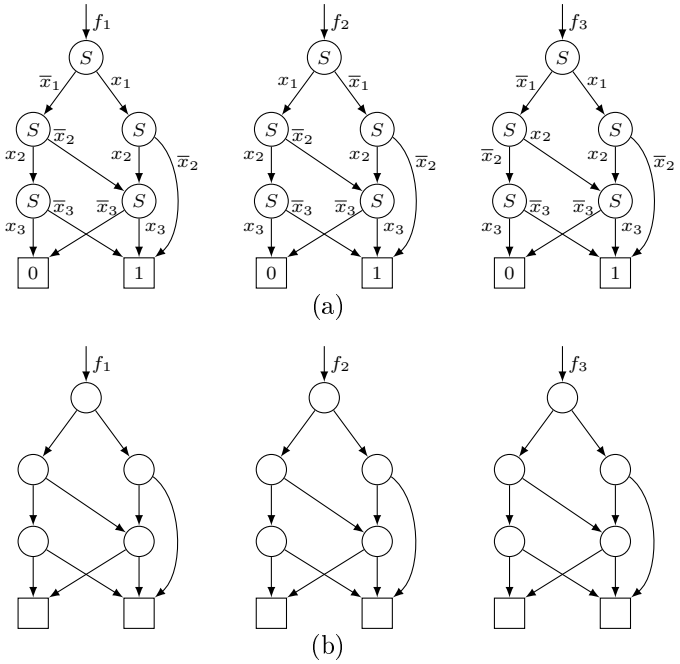


Figure 4.16. BDDs for functions f_1, f_2, and f_3 in Example 4.10.

variable values to edge labels.

The truth table uniquely determines the decision diagram of a function and thus also the sub-beads of each order. The edges of the BDD can be also read directly from the set of beads of order $0, 1, \ldots, n$ of the truth table. This means that if we change the truth table so that the bead set with its interconnection structure does not change, then the shape of the decision diagram also remains the same. In the following, we use this to define the shape of a decision diagram by defining operations on bead sets such that maintain the bead structure. The different shapes are thus defined as the invariance classes under these operations.

Definition 4.14. *Let $\alpha = \beta\gamma$ be a bead of order k, where β and γ are beads or squares of order $(k-1)$. Then, $SW(\alpha) = \gamma\beta$ is the bead obtained by the switch operation SW from α.*

Definition 4.15. *Let B be a bead set and $\alpha = \beta\gamma$ a bead of order k. Then, $SW_k(\alpha, B)$ is the bead set obtained from B by applying the switch operation to α, to $SW(\alpha)$, and also to all sub-beads equal to α and $SW(\alpha)$ in beads of order $k+1, \ldots, n$.*

Definition 4.16. *Two functions f_1 and f_2 have decision diagrams of the same shape if $B(f_2)$ can be obtained form $B(f_1)$ by applying a sequence of operations $SW_{l_1}, SW_{l_2}, \ldots$, where $l_i \leq l_j$ if $i \leq j$.*

Example 4.11. *In Example 4.10 the sets of beads $B(f_1)$ and $B(f_2)$ are equal up to the switch of beads at the level 2, which also requires the switch of the corresponding sub-beads of beads at the level 1. It is similar for the set of beads $B(f_3)$.*

4.3.4. Generalizations to Word-level Decision Diagrams

The notion of beads can be further generalized by replacing the binary sequences with sequences of integers. This allows us to extend the previous discussion to multi-terminal binary decision diagrams (MTBDDs) [263]. Recall that MTBDDs are a generalization of BDDs by allowing integers as values of constant nodes instead logic values 0 and 1 [62]. Therefore, the consideration of beads for integer sequences has the same justification as the generalization of BDDs into MTBDDs and, since generalization is done in the same way as for the values of constant nodes in BDDs and MTBDDs, all conclusions about beads and BDDs can be extended to beads for integer sequences and MTBDDs. This generalization is not trivial since allows representation by decision diagrams of multi-output functions and further introduction of various other word-level decision diagrams [263].

The Definition 4.13 directly applies to integer sequences with a single difference that entries of beads are integers instead of logic values 0 and 1. The analysis analogous to that in Example 4.8 can be performed in the case of integer valued functions represented by MTBDDs, as illustrated in Example 4.12.

Algorithm 4.1 can be used to check if two functions $f_1(x_1, x_2, \ldots, x_n)$

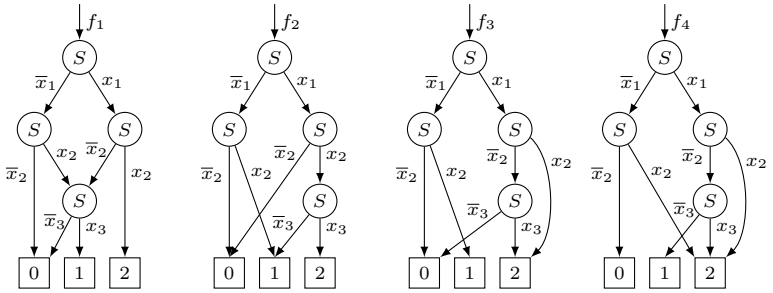


Figure 4.17. MTBDDs for functions f_1, f_2, f_3, and f_4 in Example 4.12.

Table 4.11. Sets of integer beads for functions in Example 4.12

Level	$B(f_1)$	$B(f_2)$	$B(f_3)$	$B(f_4)$
1	(00010122)	(00110012)	(00110222)	(00221222)
2	(0001), (0122)	(0011), (0012)	(0011), (0222)	(0022), (1222)
3	(01)	(12)	(02)	(12)
4	(0), (1), (2)	(0), (1), (2)	(0), (1), (2)	(0), (1), (2)

and $f_2(x_1, x_2, \ldots, x_n)$ have decision diagrams of the same shape.

Example 4.12. *Consider the MTBDDs in Figure 4.17 representing the integer-valued functions*

$$f_1 = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + 2x_1 x_2 \ ,$$
$$f_2 = \overline{x}_1 x_2 + x_1 x_2 \overline{x}_3 + 2x_1 x_2 x_3 \ ,$$
$$f_3 = \overline{x}_1 x_2 + 2x_1 \overline{x}_2 x_3 + 2x_1 x_2 \ ,$$
$$f_4 = 2\overline{x}_1 x_2 + x_1 \overline{x}_2 \overline{x}_3 + 2x_1 x_2 + 2x_1 \overline{x}_2 x_3 \ .$$

Table 4.11 shows the sets of integer beads for these functions. We see that these functions have different shapes and different sets of integer beads. The considerations about sharing nodes and interconnections analogous to that in Example 4.8 obviously hold also in the case of integer beads. The sets of beads cannot be reduced to each other by the switch operation over integer beads.

Algorithm 4.1 sets mathematical foundations for the comparison of shapes of decision diagrams, since expresses in terms of the switch operation transformations over decision diagrams that preserve the

Algorithm 4.1 Equal shape

Require: functions f_1 and f_2
Ensure: Are the shapes of f_1 and f_2 are equal to each other?

1: The distinct values of functions f_1 and f_2 can be taken as encoded by elements of the same set of distinct symbols, as for instance a, b, c, etc.
2: Beads in $B(f_1)$ and $B(f_2)$ are checked from the level $(n+1)$ for constant nodes up to the root node at the level 1. Constant nodes must agree or decision diagrams are immediately of different shape.
3: If at a level beads do not agree, perform the operation switch as specified in Definition 4.15 to resolve the disagreement. The switch is to be done simultaneously everywhere at the upper levels. If the switch operation cannot resolve the disagreement, conclude that BDDs have different shapes.
4: The comparison of beads per levels is done until the root node is reached.

shape. From the implementation point of view, Algorithm 4.1 it is useful to notice the following. Beads are actually distinct subfunctions in a decision diagram which are represented by subdiagrams. In other words, from a decision diagram, we read beads of the required order by simply traversing the subdiagrams rooted at the corresponding levels. Therefore, checking of equality of shapes is performed over decision diagrams by traversing them, and no extra memory is required. Furthermore, by referring to usual programming implementations of decision diagrams, checking the equivalence of shapes of two BDDs reduces to the comparison of the sets of identificators ignoring their order in the linked lists of records representing the nodes in the decision diagrams [77], [89], [125], [199].

4.3.5. Beads and the Classification in Terms of Walsh decision diagrams

The application of integer beads to problems of classification in terms of shapes of Walsh decision diagrams (WDD) [263], is illustrated by the following example.

Table 4.12. LP-representative functions for $n = 3$

LP-representative
$\mathbf{F}_1 = [0,0,0,0,0,0,0,0]^T$
$\mathbf{F}_2 = [0,0,0,0,0,0,0,1]^T$
$\mathbf{F}_3 = [0,0,0,0,0,1,1,0]^T$
$\mathbf{F}_4 = [0,0,0,1,0,1,1,0]^T$
$\mathbf{F}_5 = [0,0,0,1,1,0,0,0]^T$
$\mathbf{F}_6 = [0,1,1,0,1,0,1,1]^T$

Table 4.13. Walsh spectra of LP-representative functions for $n = 3$

Walsh spectrum	Encoding
$\mathbf{S}_{f_1} = [8,0,0,0,0,0,0,0]^T$	$\mathbf{S}_{f_1} = [a,b,b,b,b,b,b,b]^T$
$\mathbf{S}_{f_2} = [6,2,2,-2,2,-2,-2,2]^T$	$\mathbf{S}_{f_2} = [a,b,b,c,b,c,c,b]^T$
$\mathbf{S}_{f_3} = [4,0,0,4,4,0,0,-4]^T$	$\mathbf{S}_{f_3} = [c,b,b,c,c,b,b,a]^T$
$\mathbf{S}_{f_4} = [2,2,2,2,2,2,2,-6]^T$	$\mathbf{S}_{f_4} = [b,b,b,b,b,b,b,a]^T$
$\mathbf{S}_{f_5} = [4,0,0,-4,0,4,4,0]^T$	$\mathbf{S}_{f_5} = [c,b,b,a,b,c,c,b]^T$
$\mathbf{S}_{f_6} = [-2,-2,2,2,2,2,-2,6]^T$	$\mathbf{S}_{f_6} = [c,c,b,b,b,b,c,a]^T$

Example 4.13. *In [162], [161], it is shown that for $n = 3$ there are six LP-representative functions. These representatives can be represented by WDDs of three different shapes [289]. Table 4.12 and Table 4.13 show the LP-representative functions for $n = 3$ and their Walsh spectra. To show that functions f_1 and f_4 can be represented by WDDs of the same shape, we first do encoding $(8,0) \rightarrow (-6,2)$ and then permute the labels at the outgoing edges of all the nodes in the WDD for f_1 which reverses the order of elements in the spectrum.*

The functions f_2, f_3, and f_5 can also be represented by the WDD of the same shape. In [289], this is demonstrated using linear combinations of variables. The same can be shown in terms of integer beads by showing that the set of beads for the Walsh spectra of the functions f_2, f_3, and f_5 can be converted each to other by the switch operation over the integer beads.

Table 4.14 shows the sets of beads for the Walsh spectra of f_2, f_3, and f_5 under encodings $(6,2,-2) \rightarrow (a,b,c)$ for f_2, and $(-4,0,4) \rightarrow (a,b,c)$, f_3 and f_5.

Table 4.14. Sets of integer beads for Wash spectra of functions f_2, f_3, and f_5 in Example 4.13

Level	$B(S_{f_2})$	$B(S_{f_3})$	$B(S_{f_5})$
1	$(abbcbccb)$	$(cbbccbba)$	$(cbbabccb)$
2	$(abbc), (bccb)$	$(cbbc), (cbba)$	$(cbba), (bccb)$
3	$(ab), (bc), (cb)$	$(cb), (bc), (ba)$	$(cb), (bc), (ba)$
4	$(a), (b), (c)$	$(a), (b), (c)$	$(a), (b), (c)$

We compare $B(S_{f_5})$ and $B(S_{f_3})$ to show that f_3 and f_5 can be represented by the WDDs of the same shape. At levels 4 and 3, there are the same subsets of beads of order 0 and 1, respectively. At the level 3, we perform switch$(bccb) = (cbbc)$, which requires to do the same in the right sub-bead at the level 1. Thus, at the level 2, there are now beads $(cbba)$ and $(cbbc)$, and at the level 1, the bead $(cbbacbbc)$. Thus, we perform switch$(cbbacbbc) = (cbbccbba)$, which shows that after these transformations $B(S_{f_5})$ is converted into $B(S_{f_3})$, and it follows that f_3 and f_5 can be represented by the WDDs of the same shape.

To show that f_2 and f_3 can be represented by the WDDs of the same shape, we should show that $B(S_{f_2})$ and $B(S_{f_3})$ can be converted to each other by the operation switch over beads as follows. For clarity of the presentation, we describe the procedure step by step, although operation switch requires that some transformations have to be done simultaneously.

At the level 4, there are equal beads, but at the level 3 there is (ab) in $B(S_{f_2})$ and (ba) in $B(S_{f_3})$. Thus, we perform switch$(ba) = (ab)$ at this level and also in the beads at upper levels where (ba) appears as a sub-bead. This produces

$$B(S_{f_3})_1 = \left\{ \begin{array}{c} (cbbccbab) \\ (cbbc), (cbab) \\ (cb), (bc), (ab) \\ (a), (b), (c) \end{array} \right\} .$$

At the level 3, we now perform switch$(cb) = (bc)$, and do the same consistently at all the levels where (cb) appears as a sub-bead. We underline the newly produced sub-beads (bc) to note the difference with

the existing sub-beads of the same form. Thus, switch$(cb) = (bc)$ *produces*

$$B(S_{f_3})_2 = \left\{ \begin{array}{c} (\underline{bc}bc\underline{bc}ab) \\ (\underline{bc}bc), (\underline{bc}ab) \\ (\underline{bc}), (bc), (ab) \\ (a), (b), (c) \end{array} \right\} .$$

Now, we perform switch$(bc) = (cb)$ *at all the levels, whenever* (bc) *appears, except the newly produced sub-beads, since as required in the definition of the operation switch switch*$(bc) = (cb)$ *should be performed simultaneously with switch*$(cb) = (bc)$ *and cannot be preformed over sub-beads produced in the same step. The produced set of beads is*

$$B(S_{f_3})_3 = \left\{ \begin{array}{c} (bccbbcab) \\ (bccb), (bcab) \\ (bc), (cb), (ab) \\ (a), (b), (c) \end{array} \right\} .$$

Now, we process the level 2 *by performing switch*$(bcab) = (abbc)$ *and the corresponding operation over sub-beads at the level* 1*. This produces*

$$B(S_{f_3})_4 = \left\{ \begin{array}{c} (bccbabbc) \\ (bccb), (abbc) \\ (bc), (cb), (ab) \\ (a), (b), (c) \end{array} \right\} .$$

If we now process beads at the level 1 *by performing switch*$(bccbabbc) = (abbcbccb)$*, if follows that the produced set of beads is equal to that for* $B(S_{f_2})$ *and, therefore,* f_2 *and* f_3 *can be represented by the WDDs of the same shape.*

This example illustrated that the Definition 4.16 imply the following observation.

Two functions f_1 *and* f_2 *are WDD-equivalent if the set* $B(f_2)$ *of integer beads of the Walsh spectrum for* f_2 *can be obtained from the set* $B(f_1)$ *of integer beads of the Walsh spectrum for* f_1 *by applying a sequence of operations* $SW_{l_1}, SW_{l_2}, \ldots$*, where* $l_i \leq l_j$ *if* $i \leq j$*.*

4.3.6. Approaches for Classification

When switching functions are viewed as binary sequences of length 2^n, where n is the number of variables, their subfunctions can be viewed as sub-beads. In decision diagrams, each non-terminal node represents a subfunction in the represented function and there cannot be isomorphic subdiagrams due to the reduction rules. From this, it follows that the set of all beads for a given function f directly describes the shape of the decision diagram for f. We extend the notion of beads to integer-valued sequences in order to characterize the shape of word-level decision diagrams. This observation about relationships between the sets of beads and the shape of decision diagrams is useful in classification of switching functions by shapes of decision diagrams. It is also useful in implementation of switching functions by mapping decision diagrams into various technological platforms. The same considerations can be extended to other classes of discrete functions and the corresponding decision diagrams.

4.4. Polynomial Expansion of Symmetric Boolean Functions

Danila A. Gorodecky Valery P. Suprun

4.4.1. Polynomials of Boolean Functions

Polynomial expansion of Boolean functions is among the most complex problems of discrete mathematics [238]. The polynomial expansion of Boolean functions is used in the design of devices based upon on field-programmable gate array (FPGA) [259, 261] and in cryptography [245].

Reed-Muller polynomials are important forms of polynomial representations of Boolean functions $F = F(x_1, x_2, \ldots, x_n)$ of the n variables, where each variable appears either complemented or not complemented, but never in the both forms.

Positive polarity Reed-Muller polynomial (all variables are not complemented) is called as the Zhegalkin polynomial and is referred as $P(F)$. Negative polarity Reed-Muller polynomial (all variables are complemented) is referred as $Q(F)$.

There are many techniques for generation of the polynomial expansions for Boolean functions represented in a truth table [238, 255, 305] and in a disjunctive normal form (DNF) [307].

The most known methods generate polynomial from based on the truth table of the function $F = F(\mathbf{x})$ with $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, hence, the complexity of the methods is $O(2^n)$.

Due to high computational complexity to generate of the polynomial $P(F)$ for an arbitrary Boolean function $F = F(\mathbf{x})$ the universal methods of the polynomial expansion are not effective. Hence, development of the methods for restricted classes of Boolean functions is more effective along with universal methods. One of these classes consist of

symmetric Boolean functions (SBF).

The complexity to generate the Zhegalkin polynomial expansion using the method from [308] for SBF of n variables is equal to $O(n^2)$.

The polynomial expansions of an arbitrary SBF and an elementary SBF (ESBF) will be discussed in this section.

4.4.2. Main Definitions

An arbitrary Boolean function $F = F(\mathbf{x})$ of the n variables, where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, with unchanged value after swapping any pair of variables x_i and x_j, where $i \neq j$ and $i, j = 1, 2, \ldots, n$, is called symmetric Boolean function (SBF).

SBF F of the n variables is characterized by the set of valued numbers $A(F) = \{a_1, a_2, \ldots, a_r\}$. The function F is equal 1 if and only if a set of variables $x_1, x_2, \ldots, x_n$ has exactly a_i values 1, where:

$$0 \leq a_i \leq n, \quad 1 \leq i \leq r, \quad \text{and} \quad 1 \leq r \leq n+1 \ .$$

These SBFs F are referred as $F = F_n^{a_1, a_2, \ldots, a_r}(\mathbf{x})$. If $r = 1$, then a function $F = F_n^a(\mathbf{x})$ is called elementary SBF (ESBF). If $F \equiv 0$, then $A(F) = \emptyset$.

There is a one-to-one correspondence between the SBF F and the $(n+1)-$bits binary code $\pi(F) = (\pi_0, \pi_1, \ldots, \pi_n)$ the (carrier vector [56] or the reduced truth vector [262]), where the *i-th* entry π_i is a value of the function F with the i values 1 where $0 \leq i \leq n$. In other words, $\pi_i = 1$ if and only if the i is the valued number for the SBF F.

The following formula is true for an arbitrary SBF F:

$$F(\mathbf{x}) = \bigvee_{i=0}^{n} \pi_i F_n^i(\mathbf{x}) = \bigoplus_{i=0}^{n} \pi_i F_n^i(\mathbf{x}) \ . \tag{4.12}$$

SBF $F = F(\mathbf{x})$ of the n variables is called the positive polarity polynomial-unate SBF (homogeneous SBF [45]), if the Zhegalkin polynomial form $P(F)$ contains $\binom{n}{i}$ $i-$rank products with the i positive

literals, where $0 \leq i \leq n$. This function is referred as $F = E_n^i(\mathbf{x})$. Hence, it follows:

$$\begin{aligned} E_n^0(\mathbf{x}) &= 1 \ , \\ E_n^1(\mathbf{x}) &= x_1 \oplus x_2 \oplus \ldots \oplus x_n \ , \\ E_n^2(\mathbf{x}) &= x_1x_2 \oplus \ldots \oplus x_1x_n \oplus \ldots \oplus x_{n-1}x_n \ , \\ &\vdots \\ E_n^n(\mathbf{x}) &= x_1x_2 \ldots x_n \ . \end{aligned}$$

SBF $F = F(\mathbf{x})$ of the n variables is called as the negative polarity polynomial-unate function, if the polynomial form $Q(F)$ contains $\binom{n}{i}$ $i-$rank products with the i negative literals, where $0 \leq i \leq n$. This function is referred as $F = G_n^i(\mathbf{x})$. Hence, it follows:

$$\begin{aligned} G_n^0(\mathbf{x}) &= 1 \ , \\ G_n^1(\mathbf{x}) &= \overline{x}_1 \oplus \overline{x}_2 \oplus \ldots \oplus \overline{x}_n \ , \\ G_n^2(\mathbf{x}) &= \overline{x}_1\overline{x}_2 \oplus \ldots \oplus \overline{x}_1\overline{x}_n \oplus \ldots \oplus \overline{x}_{n-1}\overline{x}_n \ , \\ &\vdots \\ G_n^n(\mathbf{x}) &= \overline{x}_1\overline{x}_2 \ldots \overline{x}_n \ . \end{aligned}$$

As shown in [308] the polynomial forms $P(F)$ and $Q(F)$ of SBF F can be represented as:

$$\begin{aligned} P(F) = &\gamma_0 \oplus \gamma_1(x_1 \oplus x_2 \oplus \ldots \oplus x_n) \oplus \\ &\gamma_2(x_1x_2 \oplus \ldots \oplus x_1x_n \oplus \ldots \oplus x_{n-1}x_n) \oplus \ldots \oplus \\ &\gamma_n x_1x_2 \ldots x_n \ , \end{aligned} \tag{4.13}$$

and

$$\begin{aligned} Q(F) = &\mu_0 \oplus \mu_1(\overline{x}_1 \oplus \overline{x}_2 \oplus \ldots \oplus \overline{x}_n) \oplus \\ &\mu_2(\overline{x}_1\overline{x}_2 \oplus \ldots \oplus \overline{x}_1\overline{x}_n \oplus \ldots \oplus \overline{x}_{n-1}\overline{x}_n) \oplus \ldots \oplus \\ &\mu_n\overline{x}_1\overline{x}_2 \ldots \overline{x}_n \ , \end{aligned} \tag{4.14}$$

where $\gamma(F) = (\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n)$ (reduced Zhegalkin (Reed-Muller) spectrum of SBF) and $\mu(F) = (\mu_0, \mu_1, \mu_2, \ldots, \mu_n)$ (the negative reduced Reed-Muller spectrum of SBF) are binary vectors of the coefficients for $P(F)$ and $Q(F)$ polynomials, respectively.

In accordance with the definitions of the functions $F = E_n^i(\mathbf{x})$ and $F = G_n^i(\mathbf{x})$, the formulas (4.13) and (4.14) may be represented as follows:

$$P(F) = \bigoplus_{i=0}^{n} \gamma_i E_n^i(\mathbf{x}) \ , \tag{4.15}$$

and

$$Q(F) = \bigoplus_{i=0}^{n} \mu_i G_n^i(\mathbf{x}) \ . \tag{4.16}$$

According to the equations (4.15) and (4.16) the length (number of products) of polynomials $P(F)$ and $Q(F)$ is calculated as follows:

$$d(P(F)) = \sum_{i=0}^{n} \gamma_i \binom{n}{i} \ , \tag{4.17}$$

and

$$d(Q(F)) = \sum_{i=0}^{n} \mu_i \binom{n}{i} \ . \tag{4.18}$$

The polynomial expansion method of the SBF $F = F(\mathbf{x})$ is proposed in [308]. The essence of the method is to transform the reduced truth vector $\pi(F)$ to the reduced Zhegalkin spectrum $\gamma(F)$ and to the negative reduced Reed-Muller spectrum $\mu(F)$ of polynomials $P(F)$ and $Q(F)$, respectively. The method may be used to solve the reverse task, i.e., the transformation of the reduced Zhegalkin spectrum $\gamma(F)$ and the negative reduced Reed-Muller spectrum $\mu(F)$ to the reduced truth vector $\pi(F)$. The method is called the transeunt triangle method.

4.4.3. Transeunt Triangle Method

The transeunt triangle method is the method to generate binary vectors $\gamma(F)$ and $\mu(F)$ of polynomials $P(F)$ and $Q(F)$ in the equations (4.15) and (4.16). The method, firstly proposed by one of the authors of this article (V.P. Suprun) in 1985, was focused on solving the task for totally positive $P(F)$ or totally negative $Q(F)$ polarity of polynomials of SBF $F = F(\mathbf{x})$ [308].

The transeunt triangle method was generalized for arbitrary Boolean functions of the n variables in [305], and for the Reed-Muller polynomial expansion of SBF with an arbitrary polarization [306].

As this section focuses on the polynomial expansion of SBFs, so here only the description of the transeunt triangle method for monotone polarized polynomials of SBFs will be considered.

Note that the transeunt triangle method is often and incorrectly referred to as "Pascal method" or "Steinhaus method". But scientists with these famous names have no relation to the development of this method.

Therefore we present the transeunt triangle method in its original setting and its rationale, which has been known only in Russian publications by now [308].

The transformation method of the reduced truth vector $\pi(F)$ to the reduced Zhegalkin spectrum $\gamma(F)$ and to the negative reduced Reed-Muller spectrum $\mu(F)$ is based on the generation of the binary vectors:

$$\begin{aligned} z^0 &= (z_0^0, z_1^0, \ldots, z_n^0) \ , \\ z^1 &= (z_0^1, z_1^1, \ldots, z_{n-1}^1) \ , \\ z^2 &= (z_0^2, z_1^2, \ldots, z_{n-2}^2) \ , \\ &\vdots \\ z^n &= (z_0^n) \ . \end{aligned}$$

In this case $\pi(F) = (z_0^0, z_1^0, \ldots, z_n^0)$, and the coefficients z_i^j are calculated as follows:

$$z_i^j = z_i^{j-1} \oplus z_{i+1}^{j-1} \ , \tag{4.19}$$

where $j = 1, 2, \ldots, n$ and $i = 0, 1, \ldots, n-j$.

The binary vectors $z^0, z^1, \ldots, z^n$ generate the triangular binary matrix. This matrix is referred as $T_n(\pi(F)) = [z^r]$, where $r = 0, 1, \ldots, n$. Obviously, $T_n(\pi(F))$ has the form of the regular triangle. Each side of the triangle is a $(n+1)$-bit binary vector.

The method of the generation $\gamma(F)$ and $\mu(F)$ proposed in [308] is

based upon use of $T_n(\pi(F))$. Hence, it follows the method is called the transeunt triangle method.

The upper base of the triangle $T_n(\pi(F))$ corresponds to the reduced truth vector $\pi(F)$. Let's assume the left side of $T_n(\pi(F))$ is the result of τ_1-transformation of $\pi(F)$ and the right side of $T_n(\pi(F))$ is the result of τ_2-transformation of $\pi(F)$. Hence, it follows $\tau_1(\pi(F)) = (z_0^0, z_0^1, \ldots, z_0^n)$ and $\tau_2(\pi(F)) = (z_n^0, z_{n-1}^1, \ldots, z_0^n)$.

The main results of [308] consist in the formulation, the proof, and the implementation the following theorem.

Theorem 4.12. *For an arbitrary SBF $F = F(\mathbf{x})$ of the n variables with the reduced truth vector $\pi(F)$ the formulas are true*

$$\tau_1(\pi(F)) = \gamma(F) \ , \tag{4.20}$$

and

$$\tau_2(\pi(F)) = \mu(F) \ . \tag{4.21}$$

Proof. Let be $F^0(x_1, x_2, \ldots, x_n) = F(x_1, x_2, \ldots, x_n)$ and

$$\begin{aligned} F^k(x_1, x_2, \ldots, x_{n-k}) &= \frac{\partial F^{k-1}(x_1, x_2, \ldots, x_{n-k}, x_{n-k+1})}{\partial x_{n-k+1}} \\ &= F^{k-1}(x_1, x_2, \ldots, x_{n-k}, 0) \oplus F^{k-1}(x_1, x_2, \ldots, x_{n-k}, 1) \ , \end{aligned} \tag{4.22}$$

where $k = 1, 2, \ldots, n$.

Hence, it follows based on the definition of the k-fold derivative [240, 299–301]:

$$F^k(x_1, x_2, \ldots, x_{n-k}) = \frac{\partial^k F(x_1, x_2, \ldots, x_n)}{\partial x_n \, \partial x_{n-1} \ldots \partial x_{n-k+1}} \ .$$

Considering that $F = F(\mathbf{x})$ is a SBF, it follows that the functions $F = F^k(x_1, x_2, \ldots, x_{n-k})$ are also SBFs.

There is a dependency between the reduced truth vectors $\pi(F^k)$ and $\pi(F^{k-1})$, where $k = 1, 2, \ldots, n$.

The function $F^{k-1}(x_1, x_2, \ldots, x_{n-k+1})$ can be represented based on the Shannon decomposition as follows:

$$\begin{aligned} F^{k-1}(x_1, x_2, \ldots, x_{n-k+1}) = &\overline{x}_{n-k+1} F^{k-1}(x_1, x_2, \ldots, x_{n-k}, 0) \vee \\ &x_{n-k+1} F^{k-1}(x_1, x_2, \ldots, x_{n-k}, 1) \\ = &\overline{x}_{n-k+1} F_0^{k-1}(x_1, x_2, \ldots, x_{n-k}) \vee \\ &x_{n-k+1} F_1^{k-1}(x_1, x_2, \ldots, x_{n-k}) \ . \end{aligned}$$

It is obvious that the Boolean functions $F = F_0^{k-1}(x_1, x_2, \ldots, x_{n-k})$ and $F = F_1^{k-1}(x_1, x_2, \ldots, x_{n-k})$ are SBFs, and that the equations:

$$\pi_j(F_0^{k-1}) = \pi_j(F^{k-1}) \ , \tag{4.23}$$

$$\pi_j(F_1^{k-1}) = \pi_{j+1}(F^{k-1}) \ , \tag{4.24}$$

are true, where $j = 0, 1, \ldots, n-k$ and $k = 1, 2, \ldots, n$.

A short notation of (4.22) is $F^k = F_0^{k-1} \oplus F_1^{k-1}$ so that:

$$\begin{aligned} \pi(F^k) &= \pi(F_0^{k-1}) \oplus \pi(F_1^{k-1}) \ , \text{ and} \\ \pi_j(F^k) &= \pi_j(F_0^{k-1}) \oplus \pi_j(F_1^{k-1}) \ . \end{aligned} \tag{4.25}$$

The substitution of (4.23) and (4.24) into (4.25) results in:

$$\pi_j(F^k) = \pi_j(F^{k-1}) \oplus \pi_{j+1}(F^{k-1}) \ , \tag{4.26}$$

where $j = 0, 1, \ldots, n-k$ and $k = 1, 2, \ldots, n$.

Using (4.13), (4.14), and (4.22), it follows $\gamma_i(F) = F^i(0, 0, \ldots, 0)$ and $\mu_i(F) = F^i(1, 1, \ldots, 1)$, where $i = 0, 1, \ldots, n$. Therefore $\gamma_i(F) = \pi_0(F^i)$ and $\mu_i(F) = \pi_{n-i}(F^i)$.

The main property of the binary matrix $T_n(\pi(F))$ is represented by the formula (4.19) which is equivalent to (4.26), so that we have Theorem 4.12. □

Remark 4.1. *Let's generate the binary matrices*

$$T_n(\gamma(F)) \quad \textit{and} \quad T_n(\mu(F)) \ .$$

$$T_{10}\left(\pi\left(F\right)\right) = \begin{array}{ccccccccccccccccccccc}
0 & & 0 & & 0 & & 1 & & 1 & & 1 & & 0 & & 0 & & 0 & & 0 \\
& 0 & & 0 & & 1 & & 0 & & 0 & & 0 & & 1 & & 0 & & 0 & & 0 \\
& & 0 & & 1 & & 1 & & 0 & & 0 & & 1 & & 1 & & 0 & & 0 & \\
& & & 1 & & 0 & & 1 & & 0 & & 1 & & 0 & & 1 & & 0 & & \\
& & & & 1 & & 1 & & 1 & & 1 & & 1 & & 1 & & 1 & & & \\
& & & & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & & & \\
& & & & & & 0 & & 0 & & 0 & & 0 & & 0 & & & & & \\
& & & & & & & 0 & & 0 & & 0 & & 0 & & & & & & \\
& & & & & & & & 0 & & 0 & & 0 & & & & & & & \\
& & & & & & & & & 0 & & 0 & & & & & & & & \\
& & & & & & & & & & 0 & & & & & & & & &
\end{array}$$

Figure 4.18. Binary matrix $T_{10}(\pi(F))$ with $\pi(F) = (00011110000)$.

It can be easily detected that

$$\begin{aligned}
\tau_1(\gamma(F)) &= \pi(F)\ , \qquad & \tau_1(\mu(F)) &= \widetilde{\pi}(F)\ , \\
\tau_2(\gamma(F)) &= \widetilde{\mu}(F)\ , \qquad & \tau_2(\mu(F)) &= \widetilde{\gamma}(F)\ ,
\end{aligned}$$

where

$$\begin{aligned}
\widetilde{\pi}(F) &= (\pi_n, \pi_{n-1}, \ldots, \pi_0)\ , \\
\widetilde{\gamma}(F) &= (\gamma_n, \gamma_{n-1}, \ldots, \gamma_0)\ , \ \textit{and} \\
\widetilde{\mu}(F) &= (\mu_n, \mu_{n-1}, \ldots, \mu_0)\ .
\end{aligned}$$

Let's consider the example of the application of Theorem 4.12.

Example 4.14. *Let's assume SBF* $F = F_{10}^{3,4,5,6}(x_1, x_2, \ldots, x_{10})$ *and it is necessary to generate polynomials* $P(F)$ *and* $Q(F)$.

From this condition it follows $\pi(F) = (00011110000)$. *Figure 4.18 shows the associated binary matrix.*

Considering the formulas (4.20), (4.21) and the matrix $T_{10}(\pi(F))$ *it follows* $\gamma(F) = (00011000000)$ *and* $\mu(F) = (00001000000)$, *i.e.,* $\gamma_3 = \gamma_4 = 1$ *and* $\mu_4 = 1$. *These values 1 are used in the formulas* (4.15) *and* (4.16) *with the following results:*

$$\begin{aligned}
P(F) &= E_{10}^3(\mathbf{x}) \oplus E_{10}^4(\mathbf{x}) \\
&= x_1x_2x_3 \oplus x_1x_2x_4 \oplus \ldots \oplus x_8x_9x_{10} \oplus \\
&\quad x_1x_2x_3x_4 \oplus x_1x_2x_3x_5 \oplus \ldots \oplus x_7x_8x_9x_{10}\ ,
\end{aligned}$$

and

$$\begin{aligned} Q(F) &= G_{10}^{4}(\mathbf{x}) \\ &= \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4 \oplus \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_5 \oplus \ldots \oplus \overline{x}_7\overline{x}_8\overline{x}_9\overline{x}_{10} \ . \end{aligned}$$

The number products of the polynomials $P(F)$ *and* $Q(F)$ *are:*

$$d(P(F)) = 330 \quad \textit{and} \quad d(Q(F)) = 210$$

due to Formulas (4.17) *and* (4.18).

4.4.4. Matrix Method to Generate $\gamma(F)$ and $\mu(F)$

The matrix method of the polynomial expansion of ESBFs $F = F_n^i(\mathbf{x})$ and an arbitrary SBFs $F = F_n^{a_1,a_2,\ldots,a_r}(\mathbf{x})$, where n is the number of variables, will be shown below.

Elementary Symmetric Boolean Function

The matrix method is applied for the polynomial expansion of ESBF $F = F_n^i(\mathbf{x})$ to monotone polarized Reed-Muller polynomials $P(F)$ and $Q(F)$, where $i = 0, 1, \ldots, n$.

The basic principle of the method is to generate the binary matrix H_n of the size $(n+1) \times (n+1)$, where n is the number of variables of ESBF $F = F_n^i(\mathbf{x})$.

Firstly, the binary matrix D_m with the 2^m rows and the 2^m columns is defined as the following recurrence form:

$$D_0 = 1 \quad \text{and} \quad D_j = \begin{bmatrix} D_{j-1} & D_{j-1} \\ 0 & D_{j-1} \end{bmatrix} \ ,$$

where $j = 1, 2, \ldots, m$. Figure 4.19 shows as example the matrices D_2 and D_4.

Let's assume that ESBF $F = F_n^i(\mathbf{x})$ depends on the n variables and $2^{m-1} < n + 1 \leq 2^m$.

$$D_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D_4 = \begin{bmatrix} & & & & & & & & & & & & & & & & \\ & & & H_3 & & & & & & & H_{10} & & & & & H_{15} & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \end{bmatrix}$$

Figure 4.19. Matrices D_2 and D_4

The binary matrix H_n consists of the first $n+1$ rows and the first $n+1$ columns of the matrix D_m. In particular, the matrices H_3, H_{10}, and H_{15} are indicated by the dotted lines in the matrix D_4 of Figure 4.19. Note that $D_2 = H_3$.

The main properties of the matrix H_n are:

- the row h_i of the matrix H_n corresponds to the reduced Zhegalkin spectrum of the ESBF $F = F_n^i(\mathbf{x})$ for $P(F)$; hence,

$$H_n = \left[\gamma(F_n^i)\right] \ , \tag{4.27}$$

- the row h_i of the matrix H_n corresponds to the negative reduced Reed-Muller spectrum of the ESBF $F = F_n^{n-i}(\mathbf{x})$ for $Q(F)$;

hence,

$$H_n = \left[\mu(F_n^{n-i})\right] \ . \tag{4.28}$$

From (4.27) and (4.28) follows:

$$h_i = \gamma(F_n^i) = \mu(F_n^{n-i}) \ , \tag{4.29}$$

where $i = 0, 1, \ldots, n$.

Example 4.15. *Let's assume it is necessary to generate polynomials* $P(F)$ *and* $Q(F)$*, where* $F = F_{10}^6(\mathbf{x})$.

If $n = 10$*, then from the two-side inequality* $2^{m-1} < n+1 \leq 2^m$ *follows* $m = 4$*. Considering the matrix* D_4 *and the procedure to generate the matrix* H_{10}*; below the matrix* H_{10} *is given.*

$$H_{10} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \\ h_{10} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The binary matrix H_{10} *contains the reduced Zhegalkin spectra for polynomial* $P(F)$ *and the negative reduced Reed-Muller spectrum for polynomial* $Q(F)$ *for all ESBFs, which depend on 10 variables, i.e.,* $H_{10} = \left[\gamma(F_{10}^i)\right]$ *and* $H_{10} = \left[\mu(F_{10}^{10-i})\right]$, $i = 0, 1, \ldots, 10$.

To generate the Zhegalkin polynomial form $P(F_{10}^6)$ *the row* h_6 *of the matrix* H_{10} *should be taken, i.e.,* $\gamma(F) = (00000011000)$.

Considering $h_i = \mu(F_{10}^{10-i})$ *it follows to generate the negative Reed-Muller polynomial form* $Q(F_{10}^6)$ *the row* h_4 *of the matrix* H_{10} *must be taken, i.e.,* $\mu(F) = (00001111000)$.

Therefore $\gamma_6 = \gamma_7 = 1$ *and* $\mu_4 = \mu_5 = \mu_6 = \mu_7 = 1$, *i.e.,*

$$P(F) = E_{10}^6(\mathbf{x}) \oplus E_{10}^7(\mathbf{x}) \ ,$$

and

$$Q(F) = G_{10}^4(\mathbf{x}) \oplus G_{10}^5(\mathbf{x}) \oplus G_{10}^6(\mathbf{x}) \oplus G_{10}^7(\mathbf{x}) \ .$$

Therefore according (4.17) *and* (4.18)*)the length* d *of the polynomial forms* $P(F)$ *and* $Q(F)$ *are* $d(P(F)) = 330$ *and* $d(Q(F)) = 792$.

It follows from the definitions of the functions $F = F_n^i(\mathbf{x})$, $F = E_n^i(\mathbf{x})$, the reduced Zhegalkin spectrum $\pi(F)$, and the negative reduced Reed-Muller spectrum $\gamma(F)$ that $\gamma(F_n^i) = \pi(E_n^i)$, where $i = 0, 1, \ldots, n$.

Considering $H_n = \left[\gamma(F_n^i)\right]$, $\gamma(F_n^i) = \pi(E_n^i)$ and formula (4.29), i.e., $\gamma(F_n^i) = \mu(F_n^{n-i})$ we get the following identity:

$$H_n = \left[\gamma(F_n^i)\right] = \left[\mu(F_n^{n-i})\right] = \left[\pi(E_n^i)\right] \ , \tag{4.30}$$

where $i = 0, 1, \ldots, n$.

It follows from the formula (4.30) that the matrix method to generate the binary vectors $\gamma(F)$ and $\mu(F)$ can be applied to generate the reduced truth vector $\pi(F)$, where $F = E_n^i(\mathbf{x})$ or $F = G_n^i(\mathbf{x})$.

Example 4.16. *Let's assume that it is necessary to generate the reduced truth vectors* $\pi(F_1)$ *and* $\pi(F_2)$ *for the Boolean functions* $F_1 = E_{10}^2(\mathbf{x})$ *and* $F_2 = G_{10}^7(\mathbf{x})$. *The matrix* H_{10} *provides the reduced truth vectors for all ESBF* $F = E_{10}^i(\mathbf{x})$ *of* 10 *variables, i.e.,* $H_{10} = \left[\pi(E_{10}^i)\right]$, *where* $i = 0, 1, \ldots, n$.

To generate the reduced truth vector $\pi(F_1)$ *of* $F_1 = E_{10}^2(\mathbf{x})$ *the row* h_2 *of the matrix* H_{10} *must be taken:*

$$\pi(F_1) = (00110011001) \ .$$

Thus the set of valued numbers of the function $F_1 = E_{10}^2(\mathbf{x})$ *is specified by* $A(F_1) = \{2, 3, 6, 7, 10\}$, *and Formula* (4.12) *is derived to the form:*

$$\begin{aligned} F_1 &= F_{10}^2(\mathbf{x}) \vee F_{10}^3(\mathbf{x}) \vee F_{10}^6(\mathbf{x}) \vee F_{10}^7(\mathbf{x}) \vee F_{10}^{10}(\mathbf{x}) \\ &= F_{10}^2(\mathbf{x}) \oplus F_{10}^3(\mathbf{x}) \oplus F_{10}^6(\mathbf{x}) \oplus F_{10}^7(\mathbf{x}) \oplus F_{10}^{10}(\mathbf{x}) \ . \end{aligned}$$

Let's consider the Boolean function $F_2 = G_{10}^7(\mathbf{x})$. *For negative reduced Reed-Muller spectrum* $\mu(F_2) = (00000001000)$, *where* $\mu_7 = 1$, *and considering (4.30), the row* h_3 *of the matrix* H_{10} *must be taken:*

$$\pi(F_2) = (00010001000) \ .$$

Since $\pi_3 = \pi_7 = 1$ *it follows* $A(F_2) = \{3, 7\}$ *and*

$$F_2 = F_{10}^3(\mathbf{x}) \vee F_{10}^7(\mathbf{x}) = F_{10}^3(\mathbf{x}) \oplus F_{10}^7(\mathbf{x}) \ .$$

Thus the positive polarity polynomial-unate SBF $F_1 = E_{10}^2(\mathbf{x})$ *is equal to a unity on* 496 *products among all positive polarity products of function of* 10 *variables. The function* $F_2 = G_{10}^7(\mathbf{x})$ *is equal to a unity over* 240 *products among all negative polarity products of function of* 10 *variables.*

Arbitrary Symmetric Boolean Function.

The polynomial $P(F)$ and the polynomial $Q(F)$ for SBF $F(\mathbf{x}) = F_n^{a_1,a_2,\ldots,a_r}(\mathbf{x})$ depend on the n variables must be generated with the binary matrix H_n.

From SBF $F(\mathbf{x}) = F_n^{a_1,a_2,\ldots,a_r}(\mathbf{x})$ Formula (4.12) assumes the form

$$F(\mathbf{x}) = F_n^{a_1}(\mathbf{x}) \oplus F_n^{a_2}(\mathbf{x}) \oplus \ldots \oplus F_n^{a_r}(\mathbf{x}) \ ,$$

$$P(F) = P(F_n^{a_1}) \oplus P(F_n^{a_2}) \oplus \ldots \oplus P(F_n^{a_r}) \ ,$$

and

$$\gamma(F) = \gamma(F_n^{a_1}) \oplus \gamma(F_n^{a_2}) \oplus \ldots \oplus \gamma(F_n^{a_r}) \ . \tag{4.31}$$

Considering:

$$\gamma(F_n^{a_1}) = \mu(F_n^{n-a_1}),\ \gamma(F_n^{a_2}) = \mu(F_n^{n-a_2}),\ \ldots, \gamma(F_n^{a_r}) = \mu(F_n^{n-a_r}) \ ,$$

and Formula (4.31), it follows:

$$\mu(F) = \mu(F_n^{n-a_1}) \oplus \mu(F_n^{n-a_2}) \oplus \ldots \oplus \mu(F_n^{n-a_r}) \ . \tag{4.32}$$

According to the property (4.29) of the rows of the binary matrix H_n, (4.31) and, (4.32) assume the forms

$$\gamma(F) = h_{a_1} \oplus h_{a_2} \oplus \ldots \oplus h_{a_r} \ , \tag{4.33}$$

and

$$\mu(F) = h_{n-a_1} \oplus h_{n-a_2} \oplus \ldots \oplus h_{n-a_r} \ . \tag{4.34}$$

Example 4.17. *Let's assume* $n = 10$ *and* $A(F) = \{2, 3, 8\}$. *It is necessary to generate the reduced Zhegalkin spectrum for polynomial* $P(F)$ *and the negative reduced Reed-Muller spectrum for polynomial* $Q(F)$ *of the SBF.*

Obviously, Formulas (4.33) *and* (4.34)*) assume the forms:*

$$\gamma(F) = h_2 \oplus h_3 \oplus h_8 \quad \textit{and} \quad \mu(F) = h_2 \oplus h_7 \oplus h_8 \ ,$$

where h_2, h_3, h_7, h_8 *are the rows of the binary matrix* H_{10} *shown above. It follows:*

$$\begin{aligned} \gamma(F) &= (00110011001) \oplus (00010001000) \oplus (00000000111) \\ &= (00100010110) \ , \\ \mu(F) &= (00110011001) \oplus (00000001000) \oplus (00000000111) \\ &= (00110010110) \ . \end{aligned}$$

Since $\gamma_2 = \gamma_6 = \gamma_8 = \gamma_9 = 1$ *and* $\mu_2 = \mu_3 = \mu_6 = \mu_8 = \mu_9 = 1$, *hence, we derive the polynomial* $P(F)$ *that contains all* $2, 6, 8$, *and* 9 *rank products, and the polynomial* $Q(F)$ *that contains all* $2, 3, 6, 8$, *and* 9 *rank products. In accordance to* (4.17) *and* (4.18) *the lengths of* $P(F)$ *and* $Q(F)$ *are* $d(P(F)) = 310$ *and* $d(Q(F)) = 430$.

According to the transeunt triangle method [308], the task of generating of the polynomials $P(F)$ and $Q(F)$ can be solved with the transeunt triangle $T_{10}(\pi(F))$ shown in Figure 4.20.

The left side of the triangle $T_{10}(\pi(F))$ corresponds to the reduced Zhegalkin spectrum $\gamma(F) = (00100010110)$ and the right side corresponds to the negative reduced Reed-Muller spectrum $\mu(F) = (00110010110)$.

Furthermore the proposed matrix method may by applied to generate the reduced truth vector $\pi(F)$ for an arbitrary SBF $F = F(\mathbf{x})$, when the function represented by the reduced Zhegalkin spectrum $\gamma(F)$.

$$T_{10}\left(\pi\left(F\right)\right) = \begin{array}{ccccccccccccccccccc}
0 & & 0 & & 1 & & 1 & & 0 & & 0 & & 0 & & 0 & & 1 & & 0 & & 0 \\
& 0 & & 1 & & 0 & & 1 & & 0 & & 0 & & 0 & & 1 & & 1 & & 0 & \\
& & 1 & & 1 & & 1 & & 1 & & 0 & & 0 & & 1 & & 0 & & 1 & & \\
& & & 0 & & 0 & & 0 & & 1 & & 0 & & 1 & & 1 & & 1 & & & \\
& & & & 0 & & 0 & & 1 & & 1 & & 1 & & 0 & & 0 & & & & \\
& & & & & 0 & & 1 & & 0 & & 0 & & 1 & & 0 & & & & & \\
& & & & & & 1 & & 1 & & 0 & & 1 & & 1 & & & & & & \\
& & & & & & & 0 & & 1 & & 1 & & 0 & & & & & & & \\
& & & & & & & & 1 & & 0 & & 1 & & & & & & & & \\
& & & & & & & & & 1 & & 1 & & & & & & & & & \\
& & & & & & & & & & 0 & & & & & & & & & &
\end{array}$$

Figure 4.20. Binary matrix $T_{10}(\pi(F))$ with $\pi(F) = (00110000100)$.

Example 4.18. *Let's assume* $\gamma(F) = (00100001100)$. *It is necessary to generate the reduced truth vector* $\pi(F)$ *for SBF*

$$F = F(x_1, x_2, \ldots, x_{10}) \ .$$

Because of $n = 10$, *the matrix* H_{10} *must be applied. The condition implies* $\gamma_2 = \gamma_7 = \gamma_8 = 1$, *then* $\pi(F) = h_2 \oplus h_7 \oplus h_8$. *Hence, it follows*

$$\begin{aligned}\pi(F) &= (00110011001) \oplus (00000001000) \oplus (00000000111) = \\ &= (00110010110) \ .\end{aligned}$$

In accordance with the reduced truth vector $\pi(F) = (00110010110)$, *the set of the valued numbers is* $A(F) = \{2, 3, 6, 8, 9\}$ *and the polynomial form* $P(F)$ *of SBF* F *can be represented as:*

$$\begin{aligned}F(\mathbf{x}) &= E_{10}^{2}(\mathbf{x}) \oplus E_{10}^{7}(\mathbf{x}) \oplus E_{10}^{8}(\mathbf{x}) \\ &= F_{10}^{2}(\mathbf{x}) \oplus F_{10}^{3}(\mathbf{x}) \oplus F_{10}^{6}(\mathbf{x}) \oplus F_{10}^{8}(\mathbf{x}) \oplus F_{10}^{9}(\mathbf{x}) \\ &= F_{10}^{2,3,6,8,9}(\mathbf{x}) \ .\end{aligned}$$

Thus the SBF $F = F(x_1, x_2, \ldots, x_{10})$ *is equal to a unity for* 430 *products among all positive polarity products of function on* 10 *variables.*

It should be noted the matrix method is effective to generate polynomials $P(F)$ and $Q(F)$ for an arbitrary SBF $F = F_n^{a_1, a_2, \ldots, a_r}(\mathbf{x})$ in the case of few valued numbers r (relatively n).

4.4.5. Efficiency of the Matrix Method

The matrix method of the polynomial expansion of SBFs $F = F(\mathbf{x})$ of n variables is proposed. The method focuses on the generation of the Zhegalkin polynomial $P(F)$ and on the negative polarity Reed-Muller $Q(F)$.

The discussed method is more efficient along with the known methods to polynomial factoring of ESBF $F = F_n^i(\mathbf{x})$. For example, as shown in [308], to obtain the polynomial forms of the SBF $P(F)$ and $Q(F)$ transeunt triangle must be generated which requires the calculation of $\frac{n^2+n}{2}$ EXOR-operations.

The method is based upon the generation of a binary matrix H_n of the size $(n+1) \times (n+1)$. The main property is that of the matrix H_n contains the binary vectors of the coefficients $P(F)$ and $Q(F)$ of the Boolean functions $F = E_n^i(\mathbf{x})$ and $F = G_n^i(\mathbf{x})$, where $i = 0, 1, \ldots, n$.

4.5. Weighted Don't Cares in Logic Synthesis

Anna Bernasconi Valentina Ciriani
Petr Fišer Gabriella Trucco

4.5.1. Don't Care Conditions in Logic Synthesis

The synthesis of digital circuits often exploits the flexibility given by don't care conditions [260]. A *don't care* of an incompletely specified Boolean function $f : \{0,1\}^n \to \{0,1,-\}$, is a vertex v of the Boolean space $\{0,1\}^n$ (minterm) where the value of the function is not specified, i.e., $f(v) = -$, with the meaning that it could be either 0 or 1. The final value, 0 or 1, assigned to these minterms is usually decided by the minimization algorithm used to synthesize the function. For instance, if we want to represent f as a minimal sum of products, we will set to 1 all don't cares that allow to enlarge the dimension of the implicants [48] and to get a more compact algebraic form. On the contrary, we will set to 0 all don't cares that would require the insertion of new unnecessary products in the final form.

These don't cares are called external because they are intrinsic to the functionality of the circuit and to its working environment, whatever is the chosen structure of the final circuit.

Besides external don't cares, there exist internal (or structural) don't cares due to the specific structure of the implementation, for example, satisfiability don't cares (SDC) and observability don't cares (ODC) [132]. The former arise when a node has limited controllability, because some input combinations local to the node are ruled out by the network structure. The latter arise when a node has limited observability and local changes cannot be observed at the primary outputs.

In this section we show how to enrich the notion of internal and external don't cares by assigning them a *weight*. Thus, we define and study the new concept of *weighted don't cares*. These weights might be used

to guide and refine the choices operated by the minimization algorithms in handling the don't care conditions. Our idea comes from the observation that, in some synthesis scenarios, possibly different from the classical sum-of-products (SOP) minimization, some don't care minterms might help to reduce the area of the final circuit more than others. In other words, instead of treating all don't cares equally, we propose to enforce the minimization algorithms giving them some criteria to choose which don't cares should be preferentially covered (assigned to 1). Weighted don't cares could also be applied in scenarios where there are don't cares that, for some reason, should be chosen before others. Note that the weights can be decided before the synthesis phase, or can be assigned dynamically by the minimization algorithms during logic synthesis.

In particular, we analyze an application of the concept of weighted don't cares in a synthesis framework of decomposition of Boolean functions onto overlapping subspaces [25, 26, 74, 92, 253, 260]. We then deal with another important issue, which is the development of the first synthesis tool for functions with weighted don't cares. We have considered the two-level Boolean minimizer BOOM [99, 100, 134], and we have derived a new version, called wBOOM, that handles weighted don't cares and uses the weights for choosing the don't cares that will be covered in the final circuit implementation of the function. We have experimentally evaluated this new tool, with interesting results.

This section is organized as follows. In Subsection 4.5.2 we discuss the concept of weighted don't cares and in Subsection 4.5.3 we explain some possible applications. In Subsection 4.5.4 we describe a minimization tool, wBOOM, that is sensitive to the presence of weighted don't cares. Experiments are reported in Subsection 4.5.5, while in Subsection 4.5.6 we discuss the efficiency of wBOOM theoretically. Subsection 4.5.7 concludes this section.

4.5.2. Weighted Don't Cares

A completely specified Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be simply represented by a subset of $\{0,1\}^n$ containing the vertices (minterms) v such that $f(v) = 1$, i.e., the so-called *ON-set* of f. The

set of all other vertices, i.e., the vertices v such that $f(v) = 0$, is called the *OFF-set* of f. Hereafter, we will often denote the ON-set of the function f with the function f itself and vice versa.

Let us now consider an incompletely specified Boolean function $f : \{0,1\}^n \rightarrow \{0,1,-\}$. A vertex v of the Boolean space $\{0,1\}^n$ such that $f(v) = -$ is called *don't care*. Thus, the *don't care set* (or DC-set) of f contains all the don't cares for f. When we synthesize an incompletely specified function f, the algebraic form for f must fulfill these requirements:

- it *must* cover (assign to 1) all ON-set minterms;
- it *must not* cover any OFF-set minterm;
- it *might* cover some minterms from the DC-set, in order to ease the minimization and to get a more compact resulting form.

In other words, the ON-set is the set of minterms that must be covered, the OFF-set represents the minterms that must NOT be covered, while we do not have precise requirements for the minterms in the DC-set, and we can choose whether covering or not covering them.

In this section we propose to enforce the notion of don't cares, by assigning them a *weight* and by using these weights to treat the don't care minterms differently. Indeed, as already pointed out earlier, some don't cares might help in reducing the area of the final circuit more than others. So, instead of treating all don't cares equally, a minimization algorithm should choose, according to some given criteria, which don't cares should be preferentially covered. To this aim, we introduce the following definition.

Definition 4.17. *Let f be an incompletely specified function, and let min, max ($min < max$) be two positive integers. The* weight *of a don't care minterm v of f is an integer $weight(v)$, $min \leq weight(v) \leq max$, that reflects the* degree of preference *of the don't care v, i.e., the convenience of covering v in the final circuit.*

Thus, don't cares with a bigger weight should be preferred to the other don't cares, and should be preferentially covered by the algebraic form for f.

4.5.3. Application

Weighted don't cares can be applied to different scenarios. For instance, one could decide to minimize a given function in steps. That is, instead of minimizing the whole function, that could be too big to handle, one could decide to minimize subsets of its ON-set, and then take the sum (i.e., the OR) of the resulting algebraic forms (see for instance [25, 92, 260]).

Observe that these subsets are not necessarily disjoint, as it happens for instance when the function to be minimized is given in a PLA form [345], and we decide to minimize subsets of its products in cascade. This *cascade minimization* of the function immediately suggests the use of weighted don't cares [26]. Indeed, minterms already covered during the first minimization steps do not need to be covered, if encountered again in the next minimization phases. Thus, these minterms naturally become don't cares during the minimization process.

It is also evident that, in order to avoid redundancies in the final circuit that would compromise its testability, the minterms of a function should not be covered by too many products. Therefore, we could assign a weight to the don't cares generated during the minimization steps, reflecting the number of times a product has already covered that minterm: minterms covered only a few times should get a higher weight, while minterms already covered by many products should be assigned a low degree of preference, in order to condition the choices of the minimization algorithm.

We now describe a concrete application of the concept of weighted don't cares. This application arises in a very natural way from the framework of decomposition of Boolean functions onto overlapping subspaces [25, 26, 74, 92, 253, 260]. Suppose that we want to minimize a completely specified Boolean function f depending on n binary variables $x_1, x_2, \ldots, x_n$. Consider a variable x_i and the two subspaces of $\{0,1\}^n$ where $x_i = 1$ and where $x_i = 0$, whose characteristic functions are x_i and $\overline{x}_i$, respectively. If we project the given function f onto the two subspaces, we obtain two Shannon cofactors $f|_1$ and $f|_0$, that represent the projections of f onto the spaces x_i and $\overline{x}_i$. Consider

now the intersection of the two sets, i.e., consider the set $I = f|_0 \cap f|_1$. Observe that I, $f|_1$ and $f|_0$ do not depend on x_i. Thus, the original function f can be represented by the following algebraic form:

$$f = x_i f|_1 + \overline{x}_i f|_0 + I \ .$$

Consider for example the Boolean function f in Figure 4.21 (a) depending on four variables $x_1, \ldots, x_4$, whose ON-set is given by:

$$f = \{0000, 0010, 0100, 0101, 0110, 1000, 1010, 1101, 1110, 1111\} \ ,$$

and let be $x_i = x_1$. We have:

$$\begin{aligned} f|_0 &= \{000, 010, 100, 101, 110\} && \text{(Figure 4.21 (b))}, \\ f|_1 &= \{000, 010, 101, 110, 111\} && \text{(Figure 4.21 (c))}, \text{ and} \\ I &= \{000, 010, 101, 110\} && \text{(Figure 4.21 (d))}. \end{aligned}$$

Note that the minterms in I are minterms that are also in the ON-sets of $f|_0$ and $f|_1$. For this reason we set such minterms as don't cares in $f|_0$ and $f|_1$, in fact they can be used for minimization of $f|_0$ and $f|_1$, but are not necessary since are already covered by I. Therefore, we have in our example as shown in parts (e), (f), and (d) of Figure 4.21:

$$\begin{aligned} \text{ON-set } (f|_0) &= \{100\} \qquad \text{DC-set } (f|_0) = \{000, 010, 101, 110\} \\ \text{ON-set } (f|_1) &= \{111\} \qquad \text{DC-set } (f|_1) = \{000, 010, 101, 110\} \\ \text{ON-set } (I) &= \{000, 010, 101, 110\} \ . \end{aligned}$$

Now, we note that each minterm of $f|_1$ corresponds to one minterm of f where $x_i = 1$, each minterm of $f|_0$ corresponds to one minterm of f where $x_i = 0$, while each minterm in I corresponds to two minterms in f (one with $x_i = 0$ and the other with $x_i = 1$). If we minimize $f|_1$ and $f|_0$ as SOP forms, obtaining for instance SOP_1 and SOP_0, respectively, we can note that if a minterm P of I is covered by both SOP_1 and SOP_0, then P can be set as a don't care in I. Thus, we propose to minimize the functions $f|_0$, $f|_1$ and I in this order: $f|_0$, $f|_1$, and finally I.

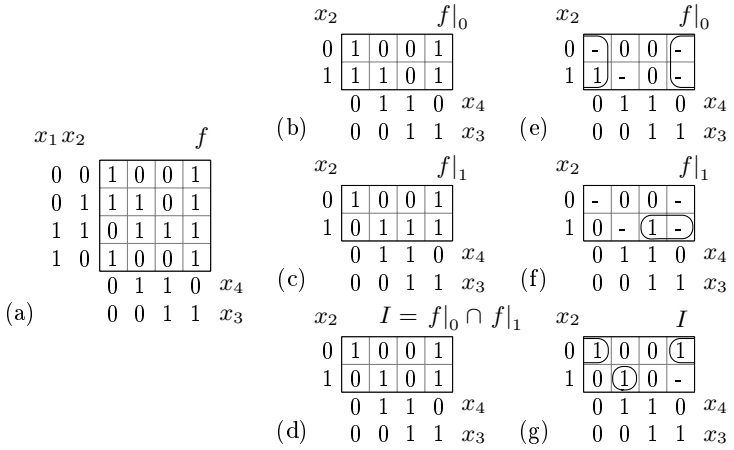


Figure 4.21. The Boolean function f of the running example.

Consider again our running example, and suppose to minimize $f|_0$ in SOP form. We obtain $SOP|_0 = \overline{x}_4$ (Figure 4.21 (e)); thus we have covered the only minterm, 100, in the ON-set, and the three minterms $000, 010, 110$ from the DC-set.

Let us now minimize $f|_1$. Our main observation is that we can set now a weight in the DC-set of $f|_1$, introducing two subsets of don't cares: *DC-set with high preference* $= \{000, 010, 110\}$ and *DC-set with low preference* $= \{101\}$. Indeed, if SOP_1 covers one of the minterms in $\{000, 010, 110\}$ (already covered by SOP_0) this minterm will be then set as a don't care for the intersection set I.

According to our idea, a *weighted minimizer* would prefer the cover given by the product x_2x_3 to the cover given by x_2x_4. Therefore, we have $SOP_1 = x_2x_3$.

We can now compute the don't care set for I as $I \cap (SOP_0 \cap SOP_1)$ (Figure 4.21 (g)):

$$\text{ON-set } (I) = \{000, 010, 101\} \qquad \text{DC-set } (I) = \{110\} \ .$$

Thus we derive the SOP form $SOP_I = \overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4$, instead of $SOP_I = \overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4 + x_3\overline{x}_4$, that we would have computed from

the alternative choice $SOP_1 = x_2x_4$. Finally, with the right choice of don't cares for $f|_1$, we obtain:

$$\begin{aligned} f &= \overline{x}_1 SOP_0 + x_1 SOP_1 + SOP_I \\ &= \overline{x}_1(\overline{x}_4) + x_1(x_2x_3) + (\overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4) \end{aligned}$$

containing 10 literals, instead of:

$$f = \overline{x}_1(\overline{x}_4) + x_1(x_2x_4) + (\overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4 + x_3\overline{x}_4)$$

containing 12 literals.

Observe that in the two particular decomposition techniques we have discussed, the weights of the don't cares are assigned dynamically during the synthesis phase.

4.5.4. Weighted BOOM: a Synthesis Tool for Functions with Weighted Don't Cares

A heuristic multi-output two-level (SOP) minimizer BOOM (acronym of BOOlean Minimizer) was proposed in [99, 134]. Later it was extended to handle multi-output functions more efficiently [100]. Basically, the algorithm consists of two steps: generation of implicants and covering problem solution. The major contribution of BOOM lies in the implicant generation phase. First of all, it is randomized. Thus, different results may be obtained from different runs on the same source data. This is exploited in the iterative minimization process, where the implicant generation phase is repeatedly executed. Different implicants are collected in the *implicant pool*. The covering problem is solved at the end, using all the implicants, to obtain the final solution. This offers a possibility of trade-off between the result quality and runtime—better results may be obtained at expense of runtime. This is visualized by Figure 4.22.

A randomly generated function of twenty input variables, five output variables and 200 care product terms of average dimension 2 were minimized here. A random function was chosen for this experiment, in order to maximally suppress the influence of any possible singular

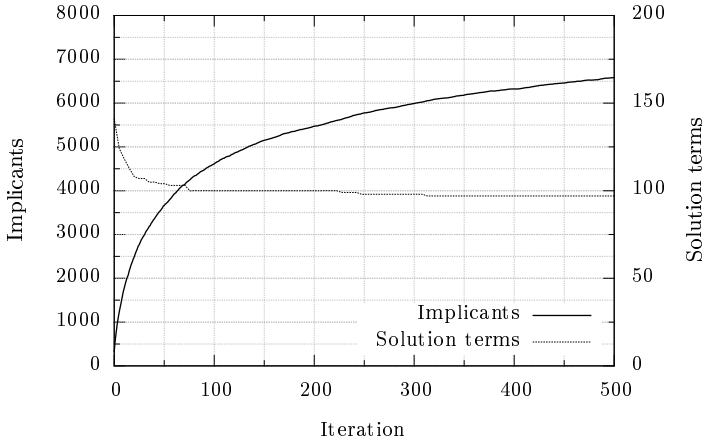


Figure 4.22. The implicant generation progress in BOOM.

behaviors of standard benchmark circuits. The total number of implicants in the pool is depicted by the solid line (and the left y-axis) and the solution quality, as the number of the SOP terms is depicted by the dotted line (and the right y-axis). Notice the different scales for these two curves. We can observe that the number of implicants follows the saturation curve, while the solution improves in the progress.

Algorithm 4.2 describes the steps of the BOOM algorithm. Before executing the algorithm, the ON-set (F) and OFF-set (R) of the source function must be provided. In the case that one set missing, it is computed as a complement of the two other sets (e.g., the OFF-set is computed from the ON-set and DC-set). The CD-Search (Coverage-Directed Search) function is the vital phase of BOOM. It produces the initial cover (set of implicants) of the source ON-set. This is also the phase where randomness is mostly used. The generated implicants are stored in the implicant pool.

The implicants are further expanded to prime implicants, while the original (non-prime) implicants are not discarded. Then all the implicants are reduced to obtain group implicants, i.e., implicants which can be used for several output functions. Again, no implicants are discarded; only the new ones are added to the pool. At the end

Algorithm 4.2 BOOM

Require: F: ON-set of the source function
Require: R: OFF-set of the source function
Ensure: *Solution*: Boolean expression as set of products (SOP) minimizing the number of literals

```
1:  Pool ← ∅
2:  repeat
3:      Cover ← CD-Search(F, R)
4:      Pool ← Pool ∪ Cover
5:      Pool ← Pool ∪ Cover.Expand(R)
6:      Pool ← Pool ∪ Cover.Reduce(F, R)
7:      Pool.Purge()
8:  until stop()
9:  Solution ← UCP_Solve(F, Pool)
10: Solution.Sparse(F, R)
11: return Solution
```

of each iteration the pool is "purged" by resolving clear dominance relations. Thereby, apparently redundant terms are removed. This process is repeated, until the stopping criterion is met. This is usually a user-specified maximum number of iterations, timeout, or the desired solution quality.

The solution is then formed by solving the Unate Covering Problem (UCP) [66] using all the implicants in the pool—an irredundant subset of implicants covering the ON-sets of all functions is formed. Finally, the solution is tried for the final refinement to keep only necessary group implicants, which are then further expanded. Note that the UCP solution and Sparse can be executed inside the main iteration loop too. This would be necessary, e.g., if the stopping condition is to be determined based on the solution quality.

The unate covering problem (UCP) may be either solved exactly or approximately, using some heuristics to select implicants into the solution. As for the exact method, AURA-II [119] was implemented. This algorithm allows to choose any implicant cost function and it generates optimum solutions minimizing this cost. The implicant cost in the original BOOM is set to the number of literals. Note that there can also be several optimal solutions. In this case, AURA-II returns

the first one found.

The approximate heuristic employed in BOOM is purely greedy; it constructs the solution by gradually adding implicants to it. The heuristic has several decision stages, where the candidate implicants are gradually filtered out:

- Select implicants covering most of yet uncovered ON-set terms
- From these, select implicants covering ON-set terms that are difficult to be covered (they are covered by the minimum of implicants)
- From these, select the ones with the least cost (the number of literals)
- If there are still more possibilities, choose one randomly.

The Weighted BOOM (wBOOM) benefits from the excess of implicants entering the UCP phase. Only few terms from the implicant pool form the solution, see Figure 4.22. Here, e.g., the solution consists of less than 100 terms out of 7,000 generated ones after 5,000 iterations. It is also very likely that many different solutions of equal quality exist. The number of produced implicants may be further increased by introducing mutations into the CD-Search phase. Implicants that are valid, but normally unlikely to be selected into the solution, are produced in this way. For details see [99].

We can easily favor weighted DCs by influencing the UCP cost function. We have decided for keeping the number of solution literals as the primary criterion for our purpose. But next, if there are more equally valued solutions, the solution covering most of the wDCs will be preferred. This can be achieved by a very simple modification of the cost function: instead of being the number of literals, it will be defined as follows:

$$cost(term) = literals(term) \cdot big_{number} - covered.wDCs(term) \ ,$$

where big_{number} is any number higher than the number of possibly covered wDCs. This ensures that the number of literals will be minimized preferably, while the number of covered DCs will be the secondary criterion. But definitely, the cost function may be modified

in other ways, depending on the actual designer's demands. Note that this approach can easily be generalized to support multiple DC weights, too. We can just compute the summary weight of covered DCs, instead of counting the number of covered wDCs. This is the way actually implemented in wBOOM.

4.5.5. Experimental Results

The synthesis tool that takes the weighted don't cares into account, wBOOM, has been applied to the Espresso benchmark suite [345], running on a Pentium 1.6 GHz processor with 1 GB RAM. We used the weights 0 and 1 for don't cares derived by the overlapping synthesis problem described in [26] and briefly recalled in Section 4.5.2. Weight 1 means that the don't care has a higher priority than don't cares with the weight 0. We have tested the practical performance of wBOOM, by comparing its results to the classical BOOM minimizer [100]. Both tools have been run with the settings of 200 iterations and 20% of CD-Search mutations [99].

wBOOM optimizes first the number of products and literals, and then it tries to maximize the covered don't care weights, that is, in the considered scenario, wBOOM maximizes the number of covered don't cares with weight equal to 1 (shortly denoted as 1-wDCs). Therefore, the discussion is based on the comparison between the numbers of 1-wDCs covered by wBOOM compared to the ones covered by BOOM.

We report in Table 4.15 a significant subset of the results. The first column reports the name of the instance considered. The following three columns refer to the experiments with wBOOM and report the number of products, the number of covered 1-wDCs, and the computational time (in seconds). The next three columns report the number of products, the number of covered 1-wDCs, and the computational time (in seconds) obtained with the BOOM minimizer. The last two columns compare the results showing the absolute gain (i.e., the difference between the number of 1-wDCs covered by wBOOM and BOOM) and the Gain rate (i.e., the difference between the number of 1-wDCs covered by wBOOM and BOOM, divided by the number of 1-wDCs covered by wBOOM).

Table 4.15. Comparison between wBOOM and BOOM

	wBOOM			**BOOM**			**Comparison**	
	Prod	1-wDCs	Time	Prod	1-wDCs	Time	Abs gain	Gain rate
add6	93	30	16.94	93	11	17.16	19	0.63
alu2	14	28	0.66	14	6	0.66	22	0.79
amd	1	0	0.30	1	0	0.28	0	0.00
b12	11	7	0.22	11	5	0.20	2	0.29
b9	12	0	3.19	12	0	3.17	0	0.00
bench	3	3	0.05	3	1	0.03	2	0.67
co14	13	117	1.06	13	0	1.06	117	1.00
dc2	18	162	0.30	18	0	0.34	162	1.00
ex1010	52	49	72.61	50	29	92.25	20	0.41
exep	50	148	13.77	50	1	13.78	147	0.99
f51m	14	78	0.22	14	6	0.22	72	0.92
ibm	12	0	3.73	12	0	3.77	0	0.00
in0	34	126	4.95	34	3	4.94	123	0.98
in5	25	161	9.45	26	7	9.47	154	0.96
life	35	315	2.61	35	0	2.59	315	1.00
m1	12	12	0.09	12	12	0.09	0	0.00
m2	44	148	1.06	44	1	1.08	147	0.99
max1024	86	312	5.55	87	6	5.53	306	0.98
max512	40	122	1.47	40	15	1.45	107	0.88
newcwp	5	21	0.02	5	3	0.01	18	0.86
newtpla2	3	19	0.09	3	0	0.11	19	1.00
p3	10	7	0.25	10	5	0.25	2	0.29
prom2	225	751	57.63	227	49	62.89	702	0.93
rckl	31	279	13.53	31	0	13.52	279	1.00
rd73	32	240	1.55	32	6	1.55	234	0.98
shift	50	18	0.92	50	18	0.84	0	0.00
sqn	12	49	0.25	12	4	0.16	45	0.92
t1	2	18	0.17	2	0	0.16	18	1.00
test1	36	51	2.52	36	6	2.63	45	0.88
tial	170	202	180.09	171	89	163.03	113	0.56
x1dn	15	15	36.09	15	10	36.36	5	0.33
z4	14	126	0.33	14	0	0.25	126	1.00
AVG							**104.38**	**0.66**
STD DEV							**189.77**	**0.39**

Comparing the number of covered 1-wDCs, we can notice that even if the main synthesis objective is the minimization of the number of products, wBOOM succeeds in maximizing the number of covered weighted don't cares without losing in minimization time. Indeed, the wBOOM covers, on average, 66% more weighted don't cares than BOOM.

We have also tested wBOOM in the particular decomposition problem described in Subsection 4.5.3 wBOOM improved the final form in about 10% of the considered benchmarks, but the gain was quite low (about 1% less products), probably because the SOP_Is were already very near to the optimum.

4.5.6. Solutions Count Analysis

Assuming the cost function from Subsection 4.5.4, the necessary condition for success of wBOOM is the existence of different solutions with the same number of literals. Then, wBOOM will return the one maximizing the number of covered DCs. One may wonder if this happens in practice—do there exist more solutions of equal size for practical examples? We may also ask how many different optimum solutions exist.

We have performed the following experiment to answer these questions: we have run BOOM (not wBOOM this time) in the same configuration as in Subsection 4.5.4 (200 iterations, 20% CD-Search mutations) 100-times and recorded all different solutions ever obtained (even in the course of the iteration). Note that all the results were prime and irredundant covers [48]. Results for some of the benchmarks coming from the decomposition process (see Subsection 4.5.3) are shown in Table 4.16.

The total number of different solutions is shown in the second column, after the benchmark name. Then the number of obtained different "best" solutions is given. Then, numbers and percentages of solutions, whose quality is less than 5% (10%, 20%, respectively) worse than the best solution are shown. We can observe that for most of the circuits only one "best" solution was obtained, which was probably the optimum one. Of course, symmetric functions, like `max512`, `sym10`, `Z9sym` [345] have adequate numbers of different P-equivalent solutions [130]. However, a plentiful of different near-optimum solutions can be observed. This is illustrated by Figure 4.23 for the `ex1010` [345] circuit. Such a behavior can be observed for most of the tested circuits.

We can conclude that wBOOM becomes efficient especially for sym-

Table 4.16. Numbers of solutions

benchmark	sol.	best sol.	$\leq 5\%$	$\leq 10\%$	$\leq 20\%$
add6	2598	1	19 (1%)	218 (8%)	2580 (99%)
alu2	114	1	38 (33%)	73 (64%)	105 (92%)
alu3	1444	1	204 (14%)	495 (34%)	1133 (78%)
amd	2	1	1 (50%)	1 (50%)	1 (50%)
b12	212	1	26 (12%)	88 (42%)	212 (100%)
b9	929	1	52 (6%)	154 (17%)	319 (34%)
bench	4283	50	2599 (61%)	3803 (89%)	4213 (98%)
co14	1	1	1 (100%)	1 (100%)	1 (100%)
dc1	1	1	1 (100%)	1 (100%)	1 (100%)
dc2	16	1	1 (6%)	7 (44%)	16 (100%)
ex1010	17020	1	293 (2%)	6144 (36%)	16172 (95%)
ex7	909	2	47 (5%)	143 (16%)	315 (35%)
exep	2343	2	2139 (91%)	2334 (100%)	2343 (100%)
f51m	315	1	19 (6%)	70 (22%)	296 (94%)
ibm	1242	1	45 (4%)	169 (14%)	700 (56%)
in7	43	4	4 (9%)	18 (42%)	34 (79%)
inc	24	2	8 (33%)	11 (46%)	24 (100%)
jbp	1166	1	97 (8%)	429 (37%)	829 (71%)
life	1	1	1 (100%)	1 (100%)	1 (100%)
log8mod	37	2	19 (51%)	30 (81%)	37 (100%)
luc	4	2	4 (100%)	4 (100%)	4 (100%)
m1	9	1	2 (22%)	2 (22%)	9 (100%)
m2	141	1	87 (62%)	135 (96%)	141 (100%)
max512	1131	122	682 (60%)	947 (84%)	1129 (100%)
misj	15	1	1 (7%)	1 (7%)	1 (7%)
mlp4	338	2	25 (7%)	138 (41%)	328 (97%)
newcwp	2	1	1 (50%)	1 (50%)	2 (100%)
newtpla2	25	1	4 (16%)	7 (28%)	9 (36%)
p3	20	2	2 (10%)	2 (10%)	18 (90%)
p82	12	1	6 (50%)	9 (75%)	12 (100%)
radd	344	1	3 (1%)	11 (3%)	80 (23%)
rckl	214	1	214 (100%)	214 (100%)	214 (100%)
rd73	1	1	1 (100%)	1 (100%)	1 (100%)
risc	1	1	1 (100%)	1 (100%)	1 (100%)
root	1163	3	249 (21%)	902 (78%)	1163 (100%)
ryy6	8499	1	566 (7%)	2649 (31%)	7406 (87%)
shift	414	1	230 (56%)	313 (76%)	414 (100%)
soar	1043	1	9 (1%)	59 (6%)	506 (49%)
sqn	113	1	23 (20%)	93 (82%)	113 (100%)
sym10	875	100	147 (17%)	242 (28%)	483 (55%)
t1	8	1	1 (13%)	1 (13%)	7 (88%)
test1	2937	1	1463 (50%)	2449 (83%)	2864 (98%)
test4	9702	4	58 (1%)	1116 (12%)	9666 (100%)
vg2	2	1	1 (50%)	1 (50%)	2 (100%)
vtx1	1149	4	929 (81%)	1009 (88%)	1103 (96%)
x1dn	8	1	1 (13%)	1 (13%)	1 (13%)
x2dn	5	1	1 (20%)	4 (80%)	5 (100%)
x9dn	741	1	634 (86%)	675 (91%)	731 (99%)
z4	395	1	37 (9%)	202 (51%)	381 (96%)
Z5xp1	153	1	61 (40%)	137 (90%)	153 (100%)
Z9sym	3178	99	751 (24%)	1329 (42%)	2541 (80%)

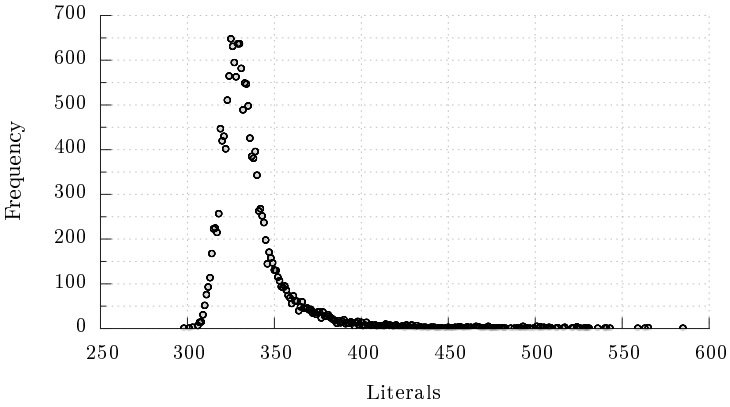


Figure 4.23. Distribution of different implicants.

metric functions, provided that don't cares are not symmetrical as well. Next, if the optimal solution is not required, many solution choices are available, thus don't cares can be exploited very efficiently, too.

4.5.7. Future Applications

We have introduced a new concept of weighted don't cares, and proposed a minimizer wBOOM, for a two-level SOP synthesis of Boolean functions with weighted don't cares.

Future possibilities include a more complete exploration of different synthesis scenarios that could benefit from the notion of weighted don't cares. Moreover, as the present version of wBOOM optimizes the number of products before trying to maximize the covered DC weights, we would like to design a new version of wBOOM even more sensitive to the weights of the don't care minterms, by considering DC weights in the generation phase. It would be also interesting to find other applications of the concept of weighted don't cares showing more interesting gains in the size of the final circuit implementations.

4.6. Determining Assignments of Incompletely Specified Boolean Functions

Suzana Stojković Milena Stanković
Radomir S. Stanković

4.6.1. Incompletely Specified Boolean Functions

Incompletely specified logic functions (binary and multiple-valued) are often met in theory of computing and related practice. Selecting a good assignment of unspecified values is very important in many applications and this problem is a permanent subject of study, with a variety of approaches to solve it. Compact representations of incompletely specified Boolean functions have also been considered in the context of decision diagram representations.

The compactness is mainly expressed in terms of the number of nonterminal nodes (the size of the diagram), although some other characteristics of the diagrams such as the number of paths, the average path length, the number of 1-paths, the width, etc., can be also considered in certain applications.

A variety of different methods to find compact representations of incompletely specified functions have been proposed. Selected references which illustrate different aspects of the problem and also different approaches to solve it are [169, 190, 226, 231, 237, 264, 270, 278, 286]. See also [154, 346] and references therein.

In this section we discuss the following problem. We assume that an incompletely specified function is given. It is our aim to find an assignment of unspecified values of this incompletely specified function such that the resulting completely specified function f_s can be represented by a compact binary decision diagram.

4.6.2. Decision Diagrams for Incompletely Specified Functions

An incompletely specified function of n variables can be viewed as a binary-input ternary-output function $f_i : \{0,1\}^n \rightarrow \{0,1,*\}$ and represented by a decision diagram consisting of non-terminal nodes with two outgoing edges and three constant nodes showing the logic values 0 and 1 and the unspecified value (don't care) $*$. We call these diagrams as *BDD**, where $*$ refers to the unspecified values, not to be confused with $*$BDD representing the edge-valued version of binary moment diagrams.

BDD* are used as the input into the proposed method. The unspecified values in subfunctions represented by subdiagrams in the *BDD** for a given incompletely specified function f_i are specified such that these subdiagrams are converted into subdiagrams for a completely specified subfunction f_s with minimum number of non-terminal nodes. The correspondingly determined completely specified function f_s is represented by the BDD which is the output of the algorithm implementing the proposed method.

Example 4.19. *Figure 4.24 (a) shows the BDD** *representing the incompletely specified function of four variables* $f_1(x_1, x_2, x_3, x_4)$ *defined by the function vector:*

$$\mathbf{F}_1 = [1, 0, *, 0, *, *, 0, 1, *, *, *, *, *, *, *, *]^T \ .$$

If all unspecified values are assigned to 0*, the resulting function* f_2 *is specified by the truth-vector:*

$$\mathbf{F}_2 = [1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]^T \ ,$$

and represented by the BDD of Figure 4.24 (b) requiring six non-terminal nodes comparing to seven non-terminal nodes in the BDD**. If all unspecified values are replaced by* 1*, the function* f_1 *is converted into the function* f_3 *with the truth-vector:*

$$\mathbf{F}_3 = [1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]^T \ ,$$

and the corresponding BDD Figure 4.24 (c) requires 5 *non-terminal nodes. The even more reduced BDD in Figure 4.24 (d) is produced by the method proposed as will be explained below.*

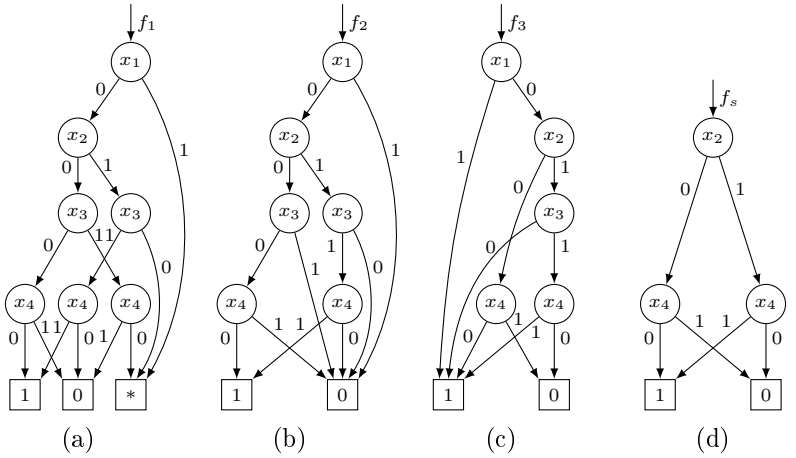


Figure 4.24. *BDD** for the incompletely specified function f_1 and completely specified functions f_2 and f_3 and the function f_s produced by the proposed method.

This example illustrates that by a suitable assignment of unspecified values isomorphic subtrees in the *BDD** can be produced resulting in a compact BDD for the corresponding completely specified function f_s assigned to a given incompletely specified function f_i. The example implies the following definition of compatible decision diagrams.

Definition 4.18. *Two BDD* which represent incompletely specified functions f_{i_1} and f_{i_2} are compatible if there exists an assignment for the unspecified values in f_{i_1} and f_{i_2} that converts them into isomorphic BDDs for the resulting completely specified functions f_{s_1} and f_{s_2}.*

Compatible subdiagrams in the *BDD**(f_i) are candidates to be converted into isomorphic subdiagrams in the *BDD*(f_s) for the completely specified function f_s assigned to f_i.

Example 4.20. *Table 4.17 shows truth-vectors* $\mathbf{F}_1$, $\mathbf{F}_2$, *and* $\mathbf{F}_3$ *of the subfunctions f_{i_1}, f_{i_2}, and f_{i_3} represented by the subdiagrams in Figure 4.25. The function values of all three functions for the assignment of* $(x_1, x_2) = (0, 0)$ *are equal to* 1. *Hence, we get:*

$$f_{compatible}(x_1 = 0, x_2 = 0) = 1 \ .$$

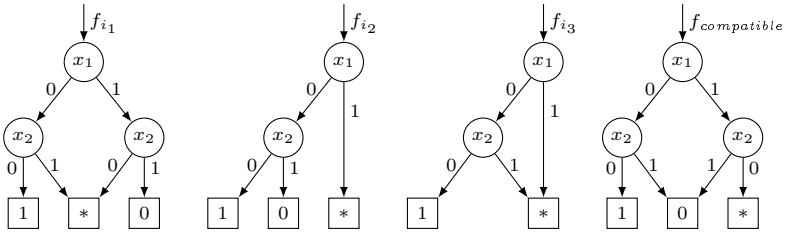


Figure 4.25. Examples of compatible subdiagrams.

Table 4.17. Subfunctions for subdiagrams in Example 4.20

(x_1, x_2)	$\mathbf{F}_1$	$\mathbf{F}_2$	$\mathbf{F}_3$	$\mathbf{F}_{compatible}$
$(0,0)$	1	1	1	1
$(0,1)$	$*$	0	$*$	0
$(1,0)$	$*$	$*$	$*$	$*$
$(1,1)$	0	$*$	$*$	0

The function value of f_{i_2} *is equal to* 0 *for the assignment of* $(x_1, x_2) = (0,1)$*, while the function values of the other two functions are unspecified for the same assignment. Thus, we assign:*

$$f_{compatible}(x_1 = 0, x_2 = 1) = 0 \ .$$

Similarly, the function value 0 *of* f_{i_1} *for* $(x_1, x_2) = (1,1)$ *determines that:*

$$f_{compatible}(x_1 = 1, x_2 = 1) = 0 \ .$$

The function values of all three incompletely specified functions f_i *for the assignment* $(x_1, x_2) = (1,0)$ *are not determined so that:*

$$f_{compatible}(x_1 = 1, x_2 = 0) = * \ .$$

In this way f_{i_1}*,* f_{i_2}*, and* f_{i_3} *can be merged into the identical subvector* $\mathbf{F}_{compatible} = [1, 0, *, 0]^T$ *and therefore their subdiagrams are compatible. Note that the terminal nodes in the subdiagram for* $f_{compatible}$ *are permuted in comparison to the subdiagram of* f_{i_1}*.*

4.6.3. A Method for Assignment of Unspecified Values

The method for determining the assignments of unspecified values through BDD^* consists in the following.

The incompletely specified function $f_i(x_1, x_2, \ldots, x_n)$ is represented by an ordered $BDD^*(f_i)$. Nodes belonging to the same level are linked and stored as linked lists which enables processing of nodes per levels. The procedure starts from the root node and it is performed top-down over the levels. Compared to the bottom-up methods, this ensures that larger compatible subfunctions might be captured. The method is, however, greedy in the sense that it always replaces compatible subdiagrams for unspecified subfunctions by subdiagrams for the corresponding completely specified subfunctions with the smallest number of non-terminal nodes as illustrated in Example 4.20.

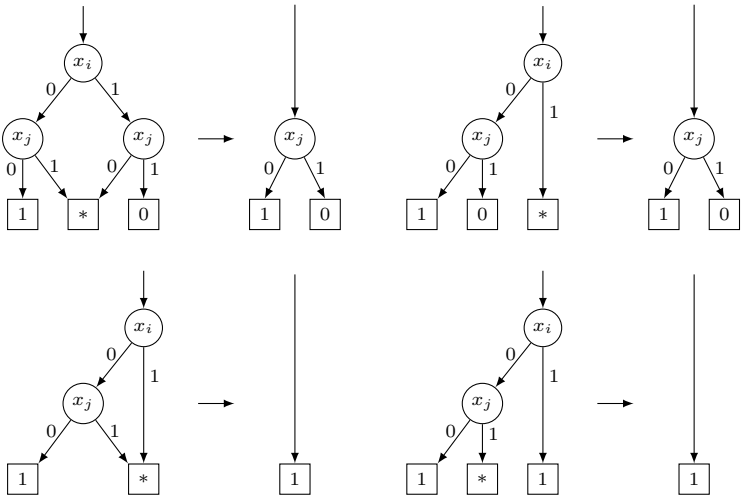


Figure 4.26. Examples of conversions of subdiagrams for incompletely specified subfunctions into subdiagrams for completely specified functions.

Figure 4.26 shows some characteristic cases of conversion of subdiagrams for incompletely specified subfunctions into subdiagrams for specified functions.

The method can be described as follows. For each level, we check the compatibility of subfunctions represented by nodes at the same level, and if two nodes C_1 and C_2 represent compatible subfunctions, i.e., subfunctions that can be converted to each other by a suitable assignment of unspecified values, we say that these nodes are compatible. The compatible unspecified nodes are compared with the corresponding completely specified nodes, i.e., nodes that are roots of subdiagrams representing the corresponding completely specified subfunctions. The replacement is organized so that we first try to replace a node with the simplest possible compatible node. The procedure can be described by the following steps.

The outer of two nested loops is executed for each level k in the $BDD^*(f)$, with $k \in [1, n]$. The inner loop calculates for each node C at the level k the following steps:

1. If the node C is compatible with the constant 0 or 1, replace it by the corresponding constant, otherwise go to the step 2.

2. If the nodes C_1 and C_2 pointed by the left and the right outgoing edge of C are compatible, replace them with a single instance of a simpler node Q compatible to them both, delete C and point its incoming edges to Q. Otherwise, go to the step 3.

3. For each node C_i at the level k that appears in the linked list after the processed node C, replace C and C_i by the simpler node compatible to both C_i and C.

Note that the same replacement of nodes by simpler compatible nodes as in Step 2 is used in the method in [190] which is realized as the Algorithm 3.1 in [190]. By allowing two more options for the replacement as in Step 1 and the Step 3 above, as well as by starting with the replacement by the simplest possible compatible nodes increases the possibility to find an assignment leading to more compact representations. The following example illustrates the proposed method.

Example 4.21. *For the incompletely specified function f_i in Example 4.19, the truth-vector of the subfunction f_{i_0} represented by the left subtree of the root node for x_1 is:*

$$\mathbf{F}_{i_0} = [1, 0, *, 0, *, *, 0, 1]^T \ .$$

The right subtree consists of a path pointing to constant node for the unspecified value $*$. Therefore, it represents the subfunction:

$$\mathbf{F}_{i_1} = [*, *, *, *, *, *, *, *]^T \ .$$

Thus, obviously we assign these unspecified values as $\mathbf{F}_{i_1} = \mathbf{F}_{i_0}$. Since this produces compatible subdiagrams, the node for x_1 can be deleted, a single instance of the subdiagrams is retained, and the node for x_2 is declared as the root node.

The next step is to check the compatibility of subdiagrams rooted in the nodes for the decision variable x_3. The truth-vectors of the subfunctions represented by the left and the right subdiagrams are:

$$\mathbf{F}_{i_0} = [1, 0, *, 0]^T \ , \ and$$
$$\mathbf{F}_{i_1} = [*, *, 0, 1]^T \ .$$

Since the specified values in these truth-vectors at the elements $(1, 1)$ are different, the subdiagrams are incompatible. Therefore, we now process the left and the right subdiagrams rooted in the nodes for x_3.

For the left node, the subdiagrams represent subfunctions

$$\mathbf{F}_{i_0} = [1, 0]^T \ , \ and$$
$$\mathbf{F}_{i_1} = [*, 0]^T \ .$$

These subdiagrams are compatible, and we assign the unspecified value to 1. Therefore, both diagrams represent the same subfunction:

$$\mathbf{F}_s = [1, 0]^T \ .$$

For the right node for x_4, of x_3, the subfunctions are

$$\mathbf{F}_{i_0} = [*, *]^T \ , \ and$$
$$\mathbf{F}_{i_1} = [0, 1]^T \ .$$

The subdiagrams are compatible, and we assign the unspecified values as:

$$\mathbf{F}_{i_0} = \mathbf{F}_{i_1} = [0, 1]^T \ .$$

The outgoing edges of both non-terminal nodes for x_3 point to the isomorphic subdiagrams and, therefore, these nodes can be deleted.

In this way, the BDD(f_s) for the completely specified function f_s is obtained. The truth-vector of f_s is:

$$\mathbf{F}_s = [1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1]^T \ .$$

Figure 4.24 (d) shows the BDD for f_s.

4.6.4. Implementation and Experimental Results

The proposed method is implemented in Algorithm 4.3 that uses the following functions. The function `CreateBDD()` is designed to create the $BDD^*(f_{iu})$ for the given incompletely specified function f_i. The function `CheckCompatible()` checks the compatibility of two nodes. The function `GetEqual()` returns the node equivalent to the given two compatible nodes.

Algorithm 4.3 CreateBDD

Require: f_i: incompletely specified function
Require: n: number of variable of f_i
Ensure: $BDD(f)$: compact BDD of f_i

```
 1: BDD*(f_i) ← CreateBDD(f_i)
 2: for k ← 1 to n do    ▷ k enumerates the levels of the BDD nodes
 3:     for all Node at level k do
 4:         if CheckCompatible(Node, 0) then              ▷ constant 0
 5:             Node ← 0
 6:         else if CheckCompatible(Node, 1) then         ▷ constant 1
 7:             Node ← 1
 8:         else if CheckCompatible(Node.left, Node.right) then
 9:             Node ← GetEqual(Node.left, Node.right)
10:         else
11:             for all Node_1 ∈ NodesOfLevel(k) do
12:                 if CheckCompatible(Node, Node_1) then
13:                     Node ← GetEqual(Node, Node_1)
14:                 end if
15:             end for
16:         end if
17:     end for
18: end for
```

Table 4.18. Code converters

Code converter	n_0	n	r
8421 $\rightarrow$ 2421	16	12	25.00
8421 $\rightarrow$ Gray	14	10	28.57
8421 $\rightarrow$ Huffman	21	11	47.62
8421 $\rightarrow$ Exscess 3	16	8	50.00
2421 $\rightarrow$ 8421	17	8	52.94
Gray $\rightarrow$ 8421	15	14	6.67
Huffman $\rightarrow$ 8421	44	4	90.91
Excess 3 $\rightarrow$ 8421	17	8	52.94
Average			44.33

The proposed method is applied to the conversion of *BDD**s into BDDs for the correspondingly determined completely specified functions for three groups of incompletely specified functions, code converters, benchmark functions, and randomly generated incompletely specified switching functions of ten variables.

Table 4.18 shows the experimental results for code converters. The assignments are performed on the shared *BDD**s. The column n_0 shows the number of non-terminal nodes in the BDDs when all unspecified values are assigned to 0. The column n shows the number of non-terminal nodes in the BDDs produced by the proposed algorithm. The last column r shows the ratio of the number of nodes in these BDDs.

Table 4.19. Randomly generated functions

% of $*$	n_0	n	r
10	233.9	215.1	8.04
20	230.2	194.7	15.42
30	222.9	176.5	20.82
40	209.7	157.2	25.04
50	199.0	136.9	31.21
60	185.9	118.8	36.09
70	165.7	94.1	43.21
80	134.9	65.5	51.45
90	99.8	38.4	61.52
Average			32.53

Table 4.20. Benchmark functions

f	In	Out	n_0	n	r
Alu2	10	8	180	108	40.00
Alu3	10	8	143	125	12.59
Apla	10	12	221	94	57.47
Bw	5	28	114	98	14.04
Dk17	10	11	145	63	56.55
Dk27	9	9	62	27	56.45
Dk48	15	17	190	61	67.89
Exp	8	18	223	192	13.90
Inc	7	9	89	73	17.98
Mark1	20	31	243	85	65.02
Misex3c	14	14	847	639	24.56
Pdc	16	40	705	313	55.60
Spla	16	46	681	618	9.25
T2	17	16	153	135	11.76
T4	12	8	116	52	55.17
Average					37.36

Tables 4.20 and 4.19 show the results of the assignments for the *BDD**s for benchmarks and randomly generated functions of ten variables. For randomly generated functions, the experiments are performed over groups of ten functions with different percentages of unspecified elements in truth-vectors. All functions take values 0 and 1 with the equal probability. Columns n_0 and n show the average sizes of the BDDs for all the functions in the group. It can be seen that the reduction ratio increases when the number of unspecified values in the truth vector increases.

4.7. On State Machine Decomposition of Petri Nets

Remigiusz Wiśniewski Andrei Karatkevich

4.7.1. Petri Nets as a Model of Concurrent Digital Controllers

The logical control algorithms, which can be implemented by digital circuits, are often specified by the classical Finite State Machines (FSM) [18, 20]. However, if such algorithms are concurrent, and especially if they have a complex structure of branching of the parallel threads, a concurrent model is needed.

Petri nets [116, 221, 232] can be used as such models. They provide a semantics allowing to describe discrete concurrent processes in a simple and convenient way. That's the reason why the Petri nets are used in some cases of digital design, especially in the design of parallel logical control devices and systems [3, 350]. Some programming languages for logical controllers, such as SFC [181] and PRALU [350], are based on the binary Petri nets.

One of the approaches to design parallel discrete systems specified by Petri nets is based on the decomposition of such nets into sequential components which are implemented as the separate modules (with proper synchronization between them) [32, 33, 171]. Such an approach requires to obtain covers of Petri nets by state machines. This problem has several applications and is also of theoretical interest [221]. It is worth noting that the Petri nets in their original form, as they were presented by C.A. Petri in [233], can always be covered by state machines, however, that is not true for the more general model developed later [221].

There exist several methods of obtaining a cover of a Petri net by sequential components. However, most of such methods have exponential computational complexity. On the other hand, it is not clear

for many cases, how to check whether a given net can be covered by the state machine components.

4.7.2. Petri Nets: Main Definitions

Definition 4.19. *A* Petri net *[221] is a tuple* $\Sigma = (P, T, F, M_0)$ *where* P *is a finite nonempty set of* places, T *is a finite nonempty set of* transitions *(*$P \cap T = \emptyset$*),* F *is a set of* arcs *such that:*

$$F \subseteq (P \times T) \cup (T \times P) \ ,$$

and M_0 *is an* initial marking.

A Petri net can be considered as a bipartite oriented graph.

A state of a Petri net, called a *marking*, is defined as a function:

$$M : P \to \mathbb{N} \ .$$

It can be considered as a number of tokens situated in the net places. A place containing at least one token is called a *marked place*. Sets of input and output places of a transition are defined respectively as follows:

$$\bullet t = \{p \in P : (p, t) \in F\} \ ,$$
$$t\bullet = \{p \in P : (t, p) \in F\} \ .$$

A transition t is *enabled* and can *fire* (be executed), if:

$$\forall p \in \bullet t : (M(p) > 0) \ .$$

Transition firing removes one token from each input place and adds one token to each output place. A marking can be changed only by a transition firing which is denoted as $M_1 t M_2$. A Petri net is *alive*, if for each transition in each reachable marking M there is a possibility to execute one of the markings reachable from M. A Petri net is *safe*, if in every reachable marking each place contains not more than 1 token. Any marking of a safe net can be expressed by a Boolean vector. A place p of a Petri net is *unsafe*, if a reachable marking M exists such that $M(p) > 1$. A Petri net is *conservative*, if the number of tokens in all its reachable markings is the same.

Definition 4.20. *A* State Machine *(*SM-net*) is a Petri net for which each transition has exactly one input place and exactly one output place:*

$$\forall t \in T : |\bullet t| = |t \bullet| = 1 \ .$$

Definition 4.21. *An* Extended Free-Choice net *(*EFC-net*) is a Petri net for which every two transitions having a common input place, have the equal sets of input places:*

$$\forall p \in P : (t_1 \in T, t_2 \in T, p \in \bullet t_1, p \in \bullet t_2) => (\bullet t_1 = \bullet t_2) \ .$$

Definition 4.22. *A Petri net* $\Sigma' = (P', T', F', M_0')$ *is a* subnet *of a Petri net* $\Sigma = (P, T, F, M_0)$*, if and only if:*

$$\begin{aligned} 1: &\quad P' \subseteq P \ , \\ 2: &\quad T' \subseteq T \ , \\ 3: &\quad F' = F \cap ((P' \times T') \cup (T' \times P')) \ , \\ 4: &\quad \forall p \in P' : M_0'(p) = M_0(p) \ . \end{aligned}$$

Definition 4.23. *An* SM-component *of a Petri net* $\Sigma = (P, T, F, M_0)$ *is its subnet* $\Sigma' = (P', T', F', M_0')$ *such that:*

$$\begin{aligned} 1: &\quad \Sigma' \text{ is an SM-net} \ , \\ 2: &\quad T' = \{x | x \in T, \exists p \in P' : ((p, x) \in F \vee (x, p) \in F)\} \ , \\ 3: &\quad \Sigma' \text{ contains exactly one token in the initial marking} \ . \end{aligned}$$

Sometimes one more condition is added, according to which Σ' should be strongly connected.

Definition 4.24. *A State Machine cover (*SM-cover*) of a Petri net* Σ *is a set* $\Sigma_1 ... \Sigma_k$ *of its SM-components such that the union of all sets of places of the SM-components contains all places of* Σ*.*

Definition 4.25. *A* Concurrency graph *of a Petri net is such a graph the nodes of which correspond to the places of the net, and there is an arc between two nodes if and only if there exists a reachable marking in which both places corresponding to them are marked.*

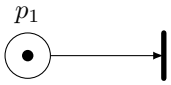


Figure 4.27. A very simple Petri net.

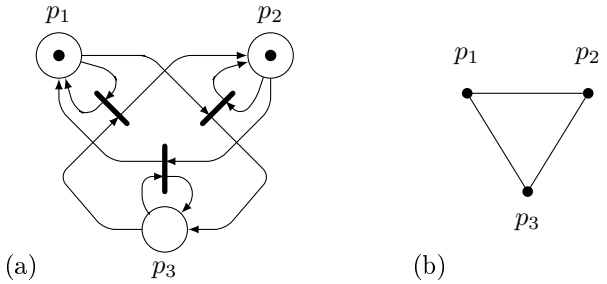


Figure 4.28. A Petri net (a) and its concurrency graph (b).

4.7.3. Conditions of SM-coverability of Petri Nets

Discussion of Known Conditions

Not every Petri net can be covered by SM-components (for example, the trivial net shown in Figure 4.27 does not contain any SM-component). Some necessary and sufficient conditions of such coverabililty can be formulated, but the authors are not aware of any general and easy-to-check condition which is both necessary and sufficient. At the same time, in some publications the statements about conditions of the coverability can be found which are not justified strictly enough.

Liveness and safeness were mentioned (for example, in [17, 33]) as the necessary conditions of coverability by strongly connected SM-components. They are surely not the sufficient conditions. Figure 4.28 demonstrates the simplest alive and safe Petri net which is not SM-coverable.

Theorem 4.13. *The safeness of a Petri net is a necessary condition of its SM-coverability.*

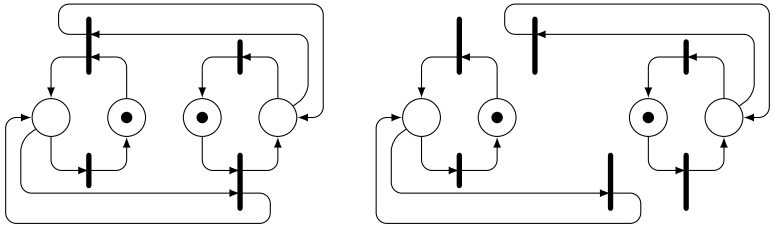


Figure 4.29. A Petri net (a) and SM-components covering it (b).

Proof. Suppose that an unsafe Petri net $\Sigma = (P, T, F, M_0)$ is SM-coverable, and let the SM-component $\Sigma' = (P', T', F', M_0)$ contain an unsafe place. Then there exists a transition $t \in T$ such that $M_1 t M_2$, both M_1 and M_2 are reachable in Σ, in M_1 P' contains one token, and in M_2 it contains more than one token (M_1 exists because at M_0 Σ' contains one token by definition). Then there are two possibilities: $t \in T'$ or $t \notin T'$. In the first case t removes one token from P' and puts one token back to P', so the number of tokens in P' does not change, and there is a contradiction. The second case contradicts item 2 of Definition 4.23 of the SM-component. □

Liveness, however, is neither a sufficient nor a necessary condition of the SM-coverability, because two or more State Machines may block each other. Figure 4.29 shows a Petri net, which is not alive (the presented marking is a deadlock), but it can be covered by two SM-components. Of course, question of SM-coverability of the nets which are not alive is rather of theoretical interest, without an evident application to logical design.

In [17] a statement can be found, according to which conservativeness is a sufficient condition of SM-coverability of a Petri net. The nets from Fig. 4.28 and Fig. 4.30 are counter-examples for this statement. Both Petri nets contain exactly two tokens in every reachable marking, and for both of them an attempt to construct an SM-component leads to constructing a subnet containing both of those tokens.

An important condition of SM-coverability is based on the correspon-

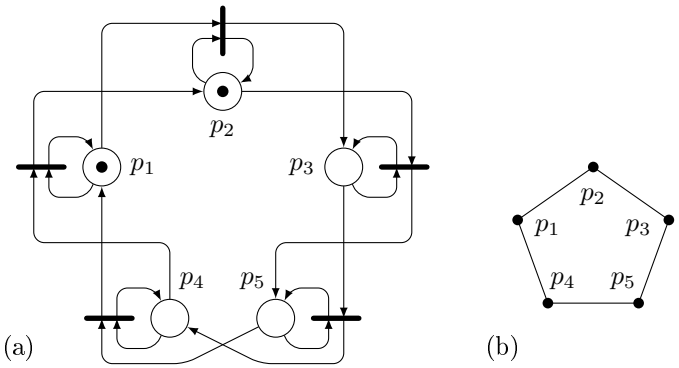


Figure 4.30. A Petri net and its concurrency graph.

dence between reachable markings of a net and the maximal cliques of its concurrency graph. Such a condition is given in [349, 350].

Theorem 4.14. *If a Petri net is SM-coverable, then for every reachable marking of the net exists a maximal clique of its concurrency graph corresponding to this marking.*

This condition is necessary, but not sufficient. For the net from Figure 4.30 the condition holds, but no SM-cover exists.

It is worth noting that the relation between the reachable markings and the maximal cliques of concurrency graph in an SM-coverable net does not have to be a bijection. In the concurrency graph of an SM-coverable Petri net the cliques may exist which do not correspond to any reachable marking. Such an example demonstrates Figure 4.31. Here $\{p_2, p_4, p_5\}$ is a maximal clique in the concurrency graph, and there is no reachable marking in which all these places are marked simultaneously. The net nevertheless can be covered by the SM-components, defined by the sets of places $\{p_1, p_5\}$, $\{p_2, p_6\}$, and $\{p_3, p_4\}$.

In [200, 201] there is a statement according to which a necessary (not sufficient) condition of SM-coverability of a Petri net is that its concurrency graph is a *perfect graph*, i.e., a graph in which the chromatic

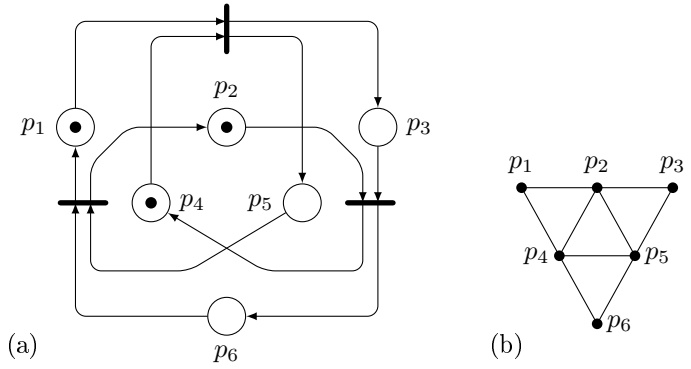


Figure 4.31. A Petri (a) net and its concurrency graph (b).

number of every induced subgraph equals the clique number of that subgraph. This statement is purely empirical. For many SM-coverable nets, their concurrency graphs belong to this class; indeed, but generally the statement does not hold. A counter-example is shown in Figure 4.32. This net can be covered by three SM-components, and its concurrency graph is not perfect (according to the strong perfect graph theorem [61], a perfect graph cannot contain a chordless cycle of length at least five, and it is easy to see, that the concurrency graph in Figure 4.32 (b) contains such a cycle).

In [166] a notion of *structural concurrency relation* $||^A$ on the set of places of a Petri net is introduced. The relation $||^A$ can be calculated in polynomial time for any Petri net (a quick algorithm calculating this relation is presented in [167]).

Theorem 4.15. *[166]. Irreflexibility of the structural concurrency relation of a Petri net is a necessary condition of its SM-coverability.*

The following Theorem, proved by Hack, is probably the most general known sufficient condition of SM-coverability.

Theorem 4.16. *[127, 221, 349]. Every alive and safe EFC-net is coverable by strongly connected SM-components.*

This condition is not necessary. In [3, 17, 317], there are examples of

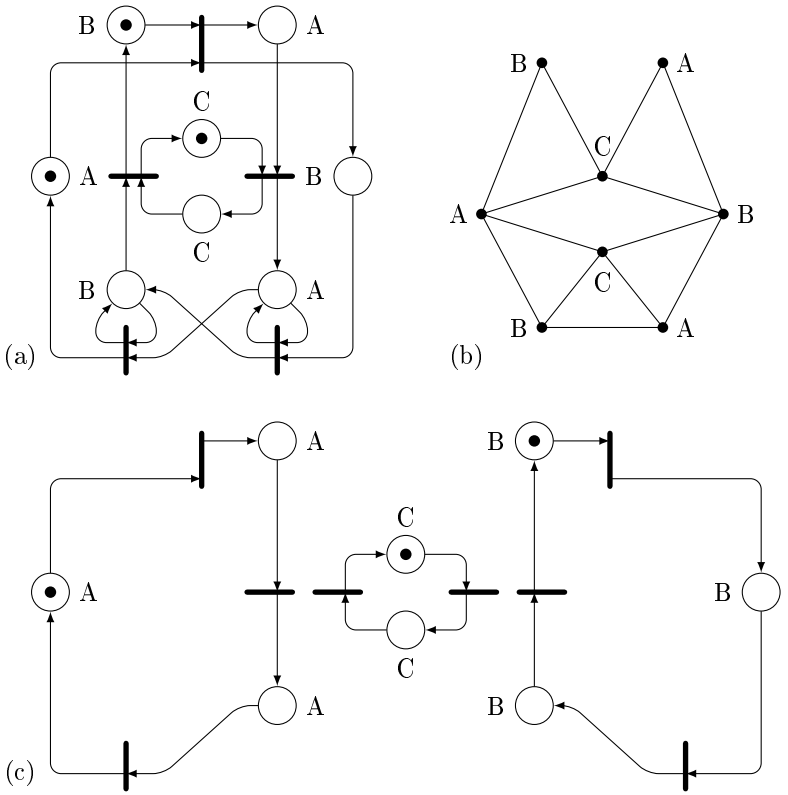


Figure 4.32. A Petri net (a), its concurrency graph (b) and SM-components covering the net (c). Letters A, B and C denote places of 3 SM-components.

SM-coverable nets not belonging to the Extended Free Choice class. The net shown in Figure 4.32 also presents such an example.

New results on coverability

Below two new conditions of existing of SM-cover for given Petri net are presented.

Theorem 4.17. *Let Σ be a conservative Petri net such that every reachable marking of it corresponds to a maximal clique of its concurrency graph. Σ is SM-coverable, if the chromatic number of the concurrency graph is equal to its clique number.*

Proof. If the chromatic number of the graph is equal to its clique number (let it be n), then there exists its coloring using n colors, and for every maximal clique all n colors are used. As far as there is a maximal clique for every reachable marking, in every reachable marking there is exactly one place of each color (and each place has a color). Hence, such a coloring specifies an SM-cover of the net. □

This condition is sufficient; it is not clear yet, whether it is necessary (for the conservative nets in which all reachable markings correspond to the maximal cliques). The answer on this question depends on the possibility that there is a net which is conservative and SM-coverable, but the minimal number of SM-components covering it is bigger than the number of tokens in its reachable markings.

Theorem 4.18. *Let be $\Sigma = (P, T, F, M_0)$ an alive and safe Petri net such that:*

$$\forall t_1 \in T, t_2 \in T : (\bullet t_1 \cap \bullet t_2 \neq \emptyset) \Rightarrow (\bullet t_1 \subseteq \bullet t_2) \vee (\bullet t_1 \supseteq \bullet t_2) \ ,$$
$$\forall t_1 \in T, t_2 \in T : \bullet t_1 \subset \bullet t_2 \Rightarrow |\bullet t_1| = 1 \ .$$

If every place $p \in P$ such that:

$$\exists t_1 \in T, t_2 \in T : \bullet t_1 \subset \bullet t_2, \bullet t_1 = \{p\}$$

is a cutvertex of the net graph, then Σ is coverable by strongly connected SM-components.

Proof. If the described condition holds, then every connected component of the net graph, obtained by removing the cutvertices, is, together with the cutvertices incident to the nodes of the component, an EFC-net by construction. In such a way a cover of Σ by the EFC-nets can be obtained. Σ is alive and safe, hence, every covering EFC-net is alive and safe (with M_0 as the initial marking or with one of the cutvertex places marked initially). Every such net can be

covered by strongly connected SM-components, according to Theorem 4.16. From such covers a cover for Σ consisting of strongly connected SM-components can be obtained in an evident way. □

4.7.4. Calculation of SM-decompositions of Petri Nets

Methods of SM-decomposition

The decomposition of a Petri net can be achieved in several ways. Different decomposition methods are based, among others, on the transformation of Boolean terms and obtaining prime implicants of Boolean functions [230], applying BDD [111], covering of a Boolean matrix presenting reachable markings of the Petri net [340, 349], or a special kind of Petri net coloring [168]. Most of the mentioned methods reduce the problem to the set cover problem. Such an approach can be interpreted as constructing a minimal cover of the set of places of the Petri net by subsets consisting of the places of the SM-components. The algorithms differ in the way of calculating of the SM-components. Some of them use the methods of symbolic logic [12, 317]. The main flaw of this approach is the high computational complexity. The set cover problem is known to be NP-complete, however, there are good approximate algorithms for this task [64]. The number of SM-components of a Petri nets depends exponentially (in the worst case) on the net size. Most of the mentioned methods generate all possible SM-components. In the method from [317] together with SM-components the traps not being SM-components can be generated, which should be dropped before the cover can be constructed.

More popular, the classical approach is based on coloring of the concurrency graph of a Petri net [17]. The whole algorithm may be divided into two main parts:

1. *Formation of a reachability set*: In this step, the concurrency relation between all places in the Petri net is computed. The reachability set contains all reachable markings of the net. The detailed and well described algorithm can be found in [53]. The

computational complexity of this stage is exponential [53, 166, 341].

2. *Computation of the SM-components*: This step is based on the coloring of the concurrency graph, where the traditional coloring methods can be applied. According to the results presented in [341], approximate coloring algorithms (like LF or SL) may be very efficient.

Finally, the Petri net is decomposed, using the SM-components which are achieved during the coloring of its concurrency graph. The main bottleneck of the presented decomposition method is the first stage, which causes the computational complexity of the whole method being exponential. Moreover, application of an exact coloring algorithm makes the second step exponential as well. Therefore, such a method practically cannot be applied in case of bigger nets (such like one that models the problem of 8 dining philosophers - the benchmark *Philosophers8* mentioned in the next subsection).

We propose a new method based on the computation of a concurrency relation and further depth-first search (DFS-search):

1. *Computation of the concurrency graph*: The *structural* concurrency relation $||^A$ between all places in the Petri net is computed. This relation is calculated by the algorithm presented in [166]. This algorithm does not require the achieving of all reachable markings of the net, but computes pairwise concurrency relations between all places in the net, which is equal to the *usual* concurrency relation $||$ for the alive and safe EFC-nets and $|| \subseteq ||^A$ in the general case. The relation $||^A$ can be used for the construction of the SM-decomposition of the Petri net. According to [166], the calculation of the structural concurrency relation for the places of a Petri net has polynomial computational complexity. The result of this stage is the graph of the structural concurrency the vertices of which correspond to the places of the Petri net, while its edges refer to the concurrency relation.

2. *Computation of the SM-components*: This step is performed via DFS-search, starting from a place that has not been covered and

ignoring all the places which are in the structural concurrency relation with any of the places already selected for currently constructed SM-components. The process is repeated until the obtained SM-components cover all the places of the net. Since the computational complexity of the DFS-search is linear (in the number of places of the net) [64], the whole stage can be executed in quadratic time (the maximum number of DFS-search repetitions is equal to the number of places).

The main advantage of the proposed method is its polynomial computational complexity. Moreover, the second step can be executed in quadratic time. On the other hand, the achieved results may not be optimal, i.e., the number of achieved SM-components may be larger than in the case of the application of the traditional solution.

Results of Experiments

The presented algorithm has been verified experimentally. It was compared with the traditional way of the decomposition, described in the previous subsection. Three coloring methods were applied for the experiments. Two of them are classified as greedy algorithms: *Largest-first method (LF)* and *Smallest-last method (SL)* [6, 21], while the remaining one (*backtracking algorithm*) is exact.

The library of used test examples contains benchmarks that describe models of real-life systems and devices and also some nets of theoretical significance. They are taken from [17, 35, 149, 174, 221, 232, 310, 341].

Table 4.21 presents the results of experiments obtained for the representative benchmarks. The subsequent columns contain execution time (in milliseconds [ms]) of corresponding algorithms. The number of SM-components is presented in brackets. Average results were calculated for all nets in the library.

From Table 4.21, we can see that the proposed method almost always finds a solution that is not optimal. On the other hand, it is much faster than the traditional methods, even based on the greedy coloring.

The gain is especially high in case of bigger systems, like *philosophers8* where the traditional methods failed to obtain a solution in one hour. In such a case the proposed decomposition method obtains the result in less than one second.

Let us point out that further reduction of achieved SM-covers can also be done in certain cases. This step can be executed in polynomial time, according to the algorithms presented in [341].

4.7.5. Evaluation of the Results

No *good* (necessary *and* sufficient, easy-to check) condition of SM-coverability of Petri nets is known by the authors. Some of the known conditions cannot be checked in polynomial time. To verify conditions described in Theorems 4.14 and 4.17, all reachable markings should be constructed, for checking the condition of Theorem 4.17 additionally the chromatic number of a graph must be computed which is an NP-hard problem. The conditions described in Theorems 4.15, 4.16, and 4.18 seem to be the most practically important, because checking of them is feasible. It has to be underlined that only in some restricted cases we can answer the question whether a given Petri net is SM-decomposable. Deep research allowing to obtain more general conditions of SM-coverability is important both theoretically and practically.

Besides of some new conditions of SM-coverability, a novel method of constructing a minimized SM-cover of alive and safe Petri nets is presented in this section. There are several methods which allow to obtain a minimal SM-cover, but they have at least exponential computational complexity. The proposed method makes an attempt to complement existing approaches; the method is approximate and quick.

Table 4.21. The results of experiments

Benchmark	Number of places	Number of transitions	LF coloring [1]	SL coloring [1]	Backtracking coloring [1]	Proposed method
cncrr001	7	4	(3) 0.02	(3) 0.03	(3) 0.11	(4) 0.10
cn_crr8	64	17	(32) 84840.20	(32) 84869.10	(–) $> 1h$	(48) 288.18
IEC	15	12	(3) 0.43	(3) 0.49	(3) 0.74	(4) 0.64
mixer	19	15	(4) 1.50	(4) 1.67	(4) 20.98	(5) 1.40
multi_robot	9	6	(3) 0.03	(3) 0.04	(3) 0.14	(5) 0.32
pcnccp	6	4	(3) 0.02	(3) 0.03	(3) 0.06	(3) 0.09
philosophers2	14	10	(6) 0.41	(6) 0.50	(6) 7532.12	(6) 1.02
philosophers5	35	25	(15) 7870.30	(15) 7873.02	(–) $> 1h$	(15) 37.85
philosophers8	56	40	(–) $> 1h$	(–) $> 1h$	(–) $> 1h$	(–) 227.45
reactor_small	9	8	(2) 0.09	(2) 0.11	(2) 0.09	(3) 0.16
speedway	9	7	(3) 0.12	(3) 0.14	(3) 0.16	(3) 0.22
speedway_macronet	5	3	(3) 0.02	(3) 0.02	(3) 0.04	(3) 0.05
average	18.99	10.97	(7.53) 17.66	(7.53) 19.71	(–) (>1h)[1]	(8.98) 3.49

[1] The tests were stopped, if the result was not obtained in one hour.

5. Test

5.1. Boolean Fault Diagnosis with Structurally Synthesized BDDs

Raimund Ubar

5.1.1. From Functional BDDs to Structural BDDs

A special class of BDDs is presented called Structurally Synthesized BDD (SSBDD). The one-to-one mapping between the nodes of an SSBDD and signal paths in the circuits provides new possibilities for structural fault diagnosis. Two problems are discussed using SSBDDs: fault diagnosis and diagnostic test pattern generation in combinational circuits for the general case when arbitrary multiple faults are present. For diagnostic modeling of test experiments, a Boolean differential equation is introduced, and SSBDDs are used for solving it. To find test patterns with good diagnostic properties, a new concept of test groups is proposed. All faults of any multiplicity are assumed to be present in the circuit and we don't need to enumerate them. Unlike the known approaches, we do not target faults as test objectives. The goal is to verify the correctness of parts of the circuit. A special attention is given to showing the possibility of simple and straightforward detection and removing of fault masking effects in testing multiple faults by reasoning the topology of SSBDD.

Within the last two decades BDDs have become the state-of-the-art data structure in VLSI CAD for representation and manipulation of Boolean functions. BDDs were first introduced for logic simulation in 1959 [177], and for logic level diagnostic modeling in [8, 324]. In

1986, Bryant proposed a new data structure called reduced ordered BDDs (ROBDDs) [50]. He showed the simplicity of the graph manipulation and proved the model canonicity that made BDDs one of the most popular representations of Boolean functions [89, 209, 265]. Different types of BDDs have been proposed and investigated during decades such as shared or multi-rooted BDDs [211], ternary decision diagrams (TDD), or in more general, multi-valued decision diagrams (MDD) [284], edge-valued BDDs (EVBDD) [172], functional decision diagrams (FDD) [156], zero-suppressed BDDs (ZBDD) [210], algebraic decision diagrams (ADD) [16], Kronecker FDDs [91, 256], binary moment diagrams (BMD) [52], free BDDs [24], multiterminal BDDs (MTBDD) and hybrid BDDs [62], Fibonacci decision diagrams [288] etc. Overviews about different types of BDDs can be found, for example, in [89, 154, 265].

Traditional use of BDDs has been functional, i.e., the target has been to represent and manipulate the Boolean functions by BDDs as efficiently as possible. Less attention has been devoted to represent with BDDs the structural properties of the circuits in the form of mapping between the BDD nodes and the gates, subcircuits or signal paths of the related circuit implementations. The structural aspect of logic circuits was first introduced into BDDs in [324, 325], where one-to-one mapping between the nodes of BDDs and signal paths in the related gate-level circuit was introduced. These BDDs were called initially alternative graphs [324], and later structurally synthesized BDDs (SSBDD) [325] to stress the way how the BDDs were synthesized – directly from the gate-level network structure of logic circuits.

The direct mapping between SSBDDs and circuits allows us to model different test relations and properties of gate level networks like signal paths, faults in gates, delays on paths, the properties of faults like masking, equivalence, dominance, redundancy, etc. These issues cannot be simulated explicitly with "classical" BDDs.

Today's nanoscale systems are characterized by an increasing complexity of testing. Due to the high density of circuits, a manufacturing defect may result in a fault involving more than one line, and the ability of the classical Single Stuck-at-Fault (SSAF) model to represent the real physical defects decreases considerably. On the other hand, targeting the multiple faults as the test generation objective (MSAF

model) becomes even more complex since a n-line circuit may have $3^n - 1$ faulty situations compared to $2n$ faulty situations under the SSAF model.

The problem of multiple fault detection as a research issue dates back to the seventies. The phenomenon of masking between faults [86] limits a test set derived for SSAF from detecting MSAF in complex circuits which may be redundant and may contain a large number of internal re-convergent fan-outs [4, 280]. A complete test for SSAF, in general, is often incomplete for MSAF due to fault masking among MSAFs [137]. Moreover, an undetected SSAF may mask the presence of an otherwise detectable fault. Therefore, multiple fault detection has become a necessity, and the test generation for MSAF is of great interest for achieving high reliability of digital circuits. The problems of ATPG-based grading of self-checking properties and strong fault-secureness under the conditions of possible multiple faults are discussed in [139, 140].

Most approaches to multiple fault test have been tried to reduce the complexity of handling MSAFs [2, 4, 58, 150, 158, 183, 235, 348]. A totally different idea which is based directly only on the SSAF model involves two-pattern approach [5, 34, 69, 148]. Test pairs were proposed to identify fault-free lines. In [163], the insufficiency of *test pairs* was shown to guarantee the detection of multiple faults. To overcome the deficiency of test pairs, a new method of *test groups* was proposed in [321]. The results of [34, 163, 321] were published in Russian and have been later never referenced in the Western literature. Recently in [323, 326], the problem of multiple fault detection by test groups was again put on the table as one efficient application of SSBDDs.

We present in this section a new idea for multiple fault diagnosis in combinational circuits, which combines the concept of multiple fault testing by test groups and solving Boolean differential equations by manipulation of Binary Decision Diagrams (BDDs). A discussion is presented how this approach can be used as a basis for hierarchical fault diagnosis to cope with the complexity of the problem. All multiple faults of any multiplicity are allowed to be present in the circuit and we don't need to enumerate them. Differently from known approaches, we don't target the faults themselves as the objectives of

testing. Instead of that, the goal of testing will be to verify the correctness of a selected part of the circuit. In this way we will be able to reduce the exponential complexity of the problem characterized by $3^n - 1$ different possible combinations of faults to be linear with the size of the circuit under diagnosis.

The rest of this section is organized as follows. First, a description of the model of SSBDDs for representing digital circuits is presented. Thereafter we formulate the diagnosis problem as a problem of solving a system of Boolean differential equations. We consider the general case of presence of any combination of multiple faults, and show how the system of equations can be solved by manipulations of SSBDDs. Then we discuss the problem of fault masking which makes fault diagnosis extremely difficult in case of multiple faults. A topological view on the activated paths' interaction in SSBDDs is introduced, which makes it easier to understand the phenomenon of multiple fault masking and allows to create straightforward algorithms for generating special types of test groups for multiple fault detection. Finally, it will be shown how the fault diagnosis with test groups can be supported by solving Boolean differential equations using SSBDDs to facilitate hierarchical fault diagnosis. The last subsection of this section presents some experimental data to demonstrate the feasibility of using test groups for multiple fault diagnosis.

5.1.2. Structurally Synthesized Binary Decision Diagrams

Mapping a digital circuit into SSBDD

Let us have a gate level combinational circuit with fan-outs only at inputs. Consider the maximum *fan-out free region* (FFR) of the circuit with inputs at the fan-out branches and fan-out free inputs. Let be n the number of the inputs of FFR. For such a *tree-like sub-circuit* with n inputs we can create an SSBDD with n nodes by sequential *superposition* of BDDs of gates in the circuit [325].

Example 5.22. *In Figure 5.1 we have a circuit with an FFR-module*

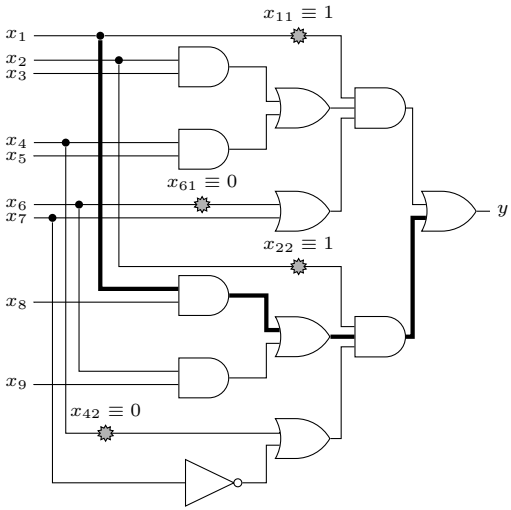


Figure 5.1. Combinational circuit.

which can be represented by the Boolean expression:

$$y = x_{11}(x_{21}x_3 \vee x_{41}x_5)(x_{61} \vee x_{71}) \vee \\ x_{22}(x_{12}x_8 \vee x_{62}x_9)(x_{42} \vee \overline{x}_{72}) \ , \tag{5.1}$$

and as the SSBDD of Figure 5.2. The literals with two indices in the formula and in the SSBDD denote the branches of fan-out stems, and represent ***signal paths*** *in the circuit. In this example, there are only two branches for each fan-out in the circuit; the second index 1 is for the upper branch, and the second index 2 is for the lower branch. For instance, the bold signal path in Figure 5.1 is represented by the literal x_{12} in the formula and by the bold node x_{12} of the SSBDD in Figure 5.2.*

The variables in the nodes of SSBDD, in general, may be inverted. They are inverted when the number of inverters on the corresponding signal path in the circuit is odd. The two terminal nodes of the SSBDD are labeled by Boolean constants #1 (truth) and #0 (false).

Every combinational circuit can be regarded as a network of modules,

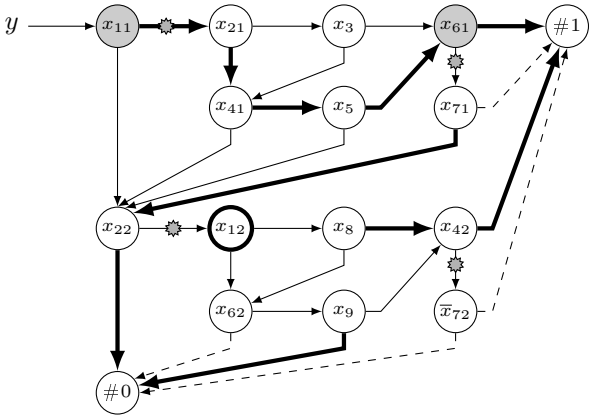


Figure 5.2. Structurally Synthesized BDD for the circuit in Figure 5.1.

where each module represents an FFR of maximum size. This way of modeling of the circuit by SSBDDs allows to keep the complexity of the model (the total number of nodes in all graphs) linear to the number of gates in the circuit.

SSBDD model. The SSBDD model for a given circuit is a set of SSBDDs, where each SSBDD represents an FFR, plus a set of single node SSBDDs, where each SSBDD represents a primary fan-out input.

As a side effect of the synthesis of SSBDDs, we build up a strict *one-to-one relationship* between the nodes in SSBDDs and the signal paths in the modules (FFRs) of the circuit.

Since all the *stuck-at faults* (SAF) at the inputs of an FFR form a collapsed fault set of the FFR, and since all these faults are represented by the faults at the nodes of the corresponding SSBDD, it follows that the synthesis of an SSBDD, described in [325] is equivalent to the fault collapsing procedure similar to fault folding [327].

The direct relation of nodes to signal paths allows to handle easily such problems like fault modeling, fault collapsing, and fault masking by SSBDDs .

Logic simulation with SSBDDs. The tracing of paths on an SSBDD can be interpreted as a procedure of calculating the value of the output variable y for the given input pattern. The procedure is carried out by traversing the nodes in the SSBDD, depending on the values of the node variables for the given input pattern.

By convention, the value 1 of the node variable means the direction to the right from the node, and the value 0 of the node variable means the direction down.

The simulation of the input pattern begins in the root node, and the procedure will terminate in one of the terminal nodes #1 or #0. The value of y will be determined by the constant in the terminal node where the procedure stops.

Example 5.23. *Consider again the circuit in Figure 5.1 and its SSBDD in Figure 5.2. For the input pattern 100111010 (123456789), a path* $(x_{11}, x_{21}, x_{41}, x_5, x_{61}, \#1)$ *in the SSBDD is traced (shown by bold edges in Figure 5.2), which produces the output value* $y = 1$ *for the given pattern.*

Topological diagnostic modeling with SSBDD

Let be given an FFR-module of a circuit which implements a function $y = f(\mathbf{x})$ where $\mathbf{x}$ is the set of input variables of the module, and is represented by an SSBDD with a set of nodes M. Let $x(m) \in \mathbf{x}$ be the variable at the node $m \in M$, and let m^0 and m^1 be the neighbors of the node m for the assignments $x(m) = 0$ and $x(m) = 1$, respectively.

Activation of SSBDD paths. Let be T_t a pattern applied at the moment t on the inputs $\mathbf{x}$ of the module. The edge (m, m^e) in the SSBDD with $e \in \{0, 1\}$, is called *activated* by T_t if $x(m) = e$. A path (m, n) is called activated by T_t if all the edges which form the path are activated.

To activate a path (m, n) means to assign by T_t proper values to the node variables along this path. Path activation can be interpreted as a reverse task to SSBDD simulation.

Test generation. A test pattern T_t will detect a *single stuck-at-fault* (SSAF) $x(m) \equiv e$, $e \in \{0,1\}$, if it activates in the SSBDD three paths (see Figure 5.3): a path (m_0, m) from the root node m_0 to the node m under test, two paths $(m^0, \#0)$ and $(m^1, \#1)$ for fault-free and faulty cases, and satisfies the fault activation condition $x(m) = e \oplus 1$.

Assume $e = 1$. To simulate the test experiment for T_t, generated for the fault $x(m) \equiv 1$, first, the path (m_0, m) will be traced up to the node m which will "serve as a switch". If the fault is missing, the path $(m^0, \#0)$ will be traced, otherwise if the fault is present, the path $(m^1, \#1)$ will be traced.

Note, that a test pattern T_t for a node fault $x(m) \equiv e$ detects single SAFs on all the lines of the signal path in the circuit, which is represented by the node m in the SSBDD.

Example 5.24. *Consider the fault $x_{11} \equiv 1$ in the circuit of Figure 5.1 represented by the fault $x(m) = x_{11} \equiv 1$ in the SSBDD in Figure 5.2. Since the node m under test and the root node are the same, $m = m_0$, the first path (m_0, m) collapsed, and needs no activation. Hence, to generate a test pattern T_t for $x_{11} \equiv 1$, we have to activate only two paths instead of three: $(m^0, \#0)$ and $(m^1, \#1)$, as an example, the paths $(x_{22} = 0, \#0)$ and $(x_{21} = 0, x_{41} = 1, x_5 = 1, x_{61} = 1, \#1)$, respectively. For the node under test we take $x_{11} = 0$ which means that the expected value will be $y = 0$. Since the fault $x_{11} \equiv 1$ is present, the path $(x_{11}, x_{21}, x_{41}, x_5, x_{61}, \#1)$ will be traced when simulating the test experiment, and the value $\#1$ in the terminal node will indicate the presence of the fault.*

Fault simulation of a test pattern T_t on the SSBDD is carried out by the following procedure:

1. The path $(m_0, \#e)$ where $e \in \{0,1\}$, activated by the test pattern T_t , will be determined.

2. For each node $m \in (m_0, \#e)$, its successor $m* \notin (m_0, \#e)$ is found, and the path $(m^*, \#e^*)$ from m^* up to a terminal node $\#e^*$ will be simulated; if $e^* \neq e$ then the fault of the node m is detectable by T_t , otherwise not.

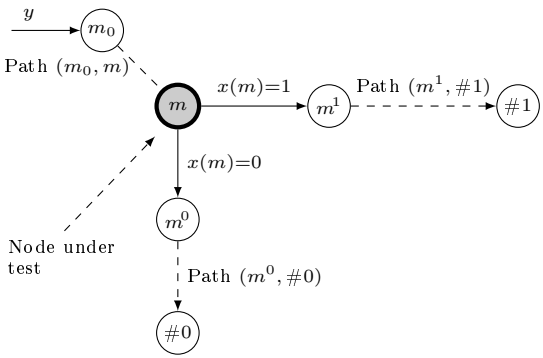


Figure 5.3. Topological view on testing of nodes on the SSBDD.

Example 5.25. *Consider the SSBDD in Figure 5.2. For the input pattern* $T_t = 100111010(123456789)$*, a path* $(x_{11}, x_{21}, x_{41}, x_5, x_{61}, \#1)$ *in the SSBDD is activated, which produces* $e = 1$*. According to Step 2 we find that the nodes* x_{11}*,* x_{41}*, and* x_5 *have all the same successor* x_{22}*, and by simulating the path* $(x_{22}, \#e^*)$ *we find that* $e^* = 0$*, which means that the test pattern is able to detect the faults* $x_{11} \equiv 0$*,* $x_{41} \equiv 0$*, and* $x_5 \equiv 0$*, since* $e \neq e^*$*. The fault* $x_{61} \equiv 0$ *is as well detectable, since the activated path* $(x_7, x_{22}, \#e^*)$ *gives* $e^* = 0$*. It is easy to see that the fault* $x_{21} \equiv 1$ *is not detectable since the activated path* $(x_3, x_{41}, x_5, x_{61}, \#e^*)$ *produces the same result* $e^* = 1$ *as in the case when the node* x_{21} *is correct.*

Fault diagnosis. Let be a test pattern T_t carried out during the diagnosis experiment. First, we relate to T_t the set of faults:

$$R(T_t) = \{x_{11} \equiv 0, x_{41} \equiv 0, x_5 \equiv 0, x_{61} \equiv 0\}$$

detectable by T_t. This set of faults was calculated by fault simulation. We have now two possibilities:

1. If the test pattern T_t fails, we will suspect all the faults of $R(T_t)$ as faulty. To have a better diagnostic resolution we have to carry out additional test patterns to prune the set of candidate faults as much as possible.

2. If the test pattern T_t passes, it would be logical to conclude from

this result that the faults of $R(T_t)$ are not present. However, it is correct only in the case when it is assumed that the circuit may have always only a single fault.

Example 5.26. *Consider the circuit in Figure 5.1 and its SSBDD in Figure 5.2. Assume that the circuit contains four faults:*

$$R = \{x_{11} \equiv 1, x_{22} \equiv 1, x_{42} \equiv 0, x_{61} \equiv 0\} \ .$$

Let us apply again to this faulty circuit the test pattern $T_t = 100111010$ (123456789). *Since* $(x_{61} \equiv 0) \in R(T_t)$, *we should expect that the test pattern will fail. However, the test will pass because the detectable fault* $(x_{61} \equiv 0) \in R(T_t) \cap R$, *is masked by the fault* $(x_{22} \equiv 1) \in R$.

To avoid fault masking during the test experiments, more advanced methods for test pattern generation and fault diagnosis must be used.

5.1.3. Fault Diagnosis in the General Case of Multiple Faults

Boolean Differential Equations and diagnosis

Consider a single output combinational circuit with n inputs as a Boolean function:

$$y = F(x_1, x_2, \ldots, x_n) \ . \tag{5.2}$$

A test experiment with the circuit can be modeled as a (total) Boolean differential [316, 322]:

$$\begin{aligned} \mathrm{d}y = {} & F(x_1, x_2, \ldots, x_n) \oplus \\ & F((x_1 \oplus \mathrm{d}x_1), (x_2 \oplus \mathrm{d}x_2), \ldots, (x_n \oplus \mathrm{d}x_n)) \ . \end{aligned} \tag{5.3}$$

When applying a test pattern T_t, the diagnosis on the basis of this experiment can be represented as a logic assertion:

$$\mathrm{d}y^t = F^t(\mathrm{d}x_1^t, \mathrm{d}x_2^t, \ldots, \mathrm{d}x_n^t) \ , \tag{5.4}$$

where

$$\mathrm{d}x_j^t \in \{\mathrm{d}x_j, \overline{\mathrm{d}x}_j\} \ . \tag{5.5}$$

$\mathrm{d}y^t \in \{0,1\}$ denotes the test result: $\mathrm{d}y^t = 0$, if the test pattern T_t has passed, and $\mathrm{d}y^t = 1$, if the test pattern T_t has failed, $\mathrm{d}x_j$ means that there is a suspected fault related to x_j , and $\overline{\mathrm{d}x_j}$ means that no fault at x_j is suspected. The fault type under question is defined by the value of x_j at the given pattern.

To create the possibility of manipulations with faults of different types, let us introduce for each variable x_j a set of suspected diagnostic states:

$$DV(x_j) = \{\mathrm{d}x_j^0, \overline{\mathrm{d}x_j}^0, \mathrm{d}x_j^1, \overline{\mathrm{d}x_j}^1, \overline{\mathrm{d}x_j}\} \ , \tag{5.6}$$

where the assertions $\mathrm{d}x_j^0, \overline{\mathrm{d}x_j}^0, \mathrm{d}x_j^1, \overline{\mathrm{d}x_j}^1$, and $\overline{\mathrm{d}x_j}$, which may be true or false, have the following meanings: the fault $x_j \equiv 1$ is suspected (the upper index at $\mathrm{d}x_j$ means the value of x_j at the given test pattern), the fault $x_j \equiv 1$ is not suspected, the fault $x_j \equiv 0$ is suspected, the fault $x_j \equiv 0$ is not suspected, and no faults are suspected at the variable x_j, respectively.

Let us introduce on the basis of (5.3) the following diagnostic equation as a true assertion:

$$D(T_t) = \overline{\mathrm{d}y^t} \oplus (y^t \oplus F^t) = 1 \ . \tag{5.7}$$

Assume, we have carried out a test experiment $T = (T_1, T_2)$ with two test patterns, and we have got the following test results $(\mathrm{d}y^1, \mathrm{d}y^2)$, respectively. The statement about fault diagnosis $D(T)$ based on the test T can be calculated from the logic multiplication of two assertions:

$$D(T) = D(T_1) \wedge D(T_2) = 1 \ . \tag{5.8}$$

For processing the diagnosis equations (5.8), we can use the 5-valued algebra depicted in Table 5.1, to find out the inconsistencies of two assertions and to carry out all the possible simplifications in (5.8).

If the final reduced assertion $D(T)$ will consist of a single DNF term, the diagnosis statement is unambiguous. More than one terms will mean ambiguity. The more test patterns we will use in the test experiment, the less ambiguous the diagnosis will become.

Example 5.27. *Consider the circuit of Figure 5.4, and the diagnostic equations of test experiments after each test pattern in Table 5.2.*

Table 5.1. 5-valued algebra for calculating Boolean differentials

	$\mathrm{d}x^0$	$\overline{\mathrm{d}x}^0$	$\mathrm{d}x^1$	$\overline{\mathrm{d}x}^1$	$\overline{\mathrm{d}x}$
$\mathrm{d}x^0$	$\mathrm{d}x^0$	$\emptyset$	$\emptyset$	$\mathrm{d}x^0$	$\emptyset$
$\overline{\mathrm{d}x}^0$	$\emptyset$	$\overline{\mathrm{d}x}^0$	$\mathrm{d}x^1$	$\overline{\mathrm{d}x}$	$\overline{\mathrm{d}x}$
$\mathrm{d}x^1$	$\emptyset$	$\mathrm{d}x^1$	$\mathrm{d}x^1$	$\mathrm{d}x^0$	$\emptyset$
$\overline{\mathrm{d}x}^1$	$\mathrm{d}x^0$	$\overline{\mathrm{d}x}$	$\emptyset$	$\overline{\mathrm{d}x}^1$	$\overline{\mathrm{d}x}$
$\overline{\mathrm{d}x}$	$\emptyset$	$\overline{\mathrm{d}x}$	$\emptyset$	$\overline{\mathrm{d}x}$	$\overline{\mathrm{d}x}$

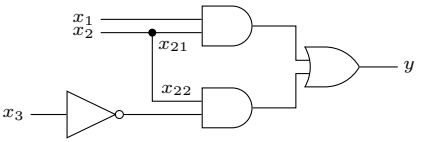


Figure 5.4. Combinational circuit.

Table 5.2 illustrates the course of the diagnostic process if all test patterns pass:

- *After applying the first two test patterns, we can state unambiguously that the signal path from the input x_1 up to the output y is working correctly, and the fault $x_{21} \equiv 0$ is missing.*
- *After the third test, we know that the fault $x_{21} \equiv 1$ is as well missing. After the 5^{th} test we can state that the circuit is functioning correctly.*

General Diagnostic Equation

Formula (5.3) represented in a vector form:

$$\mathrm{d}y = F(\mathbf{x}) \oplus F(\mathbf{x} \oplus \mathrm{d}\mathbf{x}) \tag{5.9}$$

can be regarded as a General Diagnostic Equation, since it models simultaneously three main problems of testing: test pattern genera-

Table 5.2. Diagnostic process with 5 passed test patterns

T_t	x_1	x_2	x_3	y	Diagnostic assertions
T_1	0	1	1	0	$(\overline{\mathrm{d}x_1}^0 \vee \mathrm{d}x_{21}^1)(\mathrm{d}x_{22}^1 \vee \overline{\mathrm{d}x_3}^1) = 1$
T_2	1	1	1	1	$\overline{\mathrm{d}x_1}^1\overline{\mathrm{d}x_{21}}^1 \vee \overline{\mathrm{d}x_{22}}^1\mathrm{d}x_3^1 = 1$
$D_2 = D(T_1, T_2)$					$\overline{\mathrm{d}x_1}\overline{\mathrm{d}x_{21}}^1(\mathrm{d}x_{22}^1 \vee \overline{\mathrm{d}x_3}^1) = 1$
T_3	1	0	1	0	$(\mathrm{d}x_1^1 \vee \overline{\mathrm{d}x_{21}}^0)(\overline{\mathrm{d}x_{22}}^0 \vee \overline{\mathrm{d}x_3}^1) = 1$
$D_3 = D(T_1, T_2, T_3)$					$\overline{\mathrm{d}x_1}\overline{\mathrm{d}x}_{21}(\overline{\mathrm{d}x_3}^1 \vee \overline{\mathrm{d}x_{22}}^0\mathrm{d}x_{22}^1) = 1$
T_4	0	1	0	1	$\mathrm{d}x_1^0\overline{\mathrm{d}x_{21}}^1 \vee \overline{\mathrm{d}x_{22}}^1\overline{\mathrm{d}x_3}^0 = 1$
$D_4 = D(T_1, T_2, T_3, T_4)$					$\overline{\mathrm{d}x}_1\overline{\mathrm{d}x}_{21}\overline{\mathrm{d}x}_3\overline{\mathrm{d}x_{22}}^1 = 1$
T_5	0	0	0	0	$(\overline{\mathrm{d}x_1}^0 \vee \overline{\mathrm{d}x_{21}}^0)(\overline{\mathrm{d}x_{22}}^0 \vee \mathrm{d}x_3^0) = 1$
$D_5 = D(T_1, T_2, T_3, T_4, T_5)$					$\overline{\mathrm{d}x}_1\overline{\mathrm{d}x}_{21}\overline{\mathrm{d}x}_{22}\overline{\mathrm{d}x}_3 = 1$

tion, fault simulation, and fault diagnosis, depending what is given and what is to find by solving the equation.

Let us give the following interpretations to the variables in the equation (5.9):

- $\mathrm{d}y$ – is the binary result of the test experiment,
- $\mathbf{x}$ – is the test vector applied to the circuit inputs during the test experiment, and
- $\mathrm{d}\mathbf{x}$ – is the fault vector for the inputs in the 5-valued alphabet.

The three problems modeled by Equation (5.9) are as follows:

1. Test generation: $\mathrm{d}\mathbf{x}$ is given and $\mathbf{x}$ is to be found whereas it is assumed that $\mathrm{d}y = 1$. By $\mathrm{d}\mathbf{x}$ it is possible to define any combination of multiple faults. Traditionally, test patterns are generated for single faults, in this case one $\mathrm{d}x_i = 1$ is given, and for all other $\mathrm{d}x_j \in \mathrm{d}\mathbf{x}, j \neq i$, we have $\mathrm{d}x_j = 0$.

2. Fault diagnosis: $\mathbf{x}$ is given and $\mathrm{d}\mathbf{x}$ is to be found whereas $\mathrm{d}y = 0$ if the test pattern passes, and $\mathrm{d}y = 1$ if the test pattern fails.

3. Fault simulation as a special case of fault diagnosis, where it is assumed that $\mathrm{d}y = 1$, as for test generation.

Test generation is an easy task in this sense, because only a single solution of the equation from all possible solutions is sufficient (since only a single test pattern is needed to detect a particular fault). In case of fault diagnosis, a single solution is preferred as well, which would mean an exact (multiple or single) fault diagnosis statement. In most cases, the solution will be, however, a disjunction of possible candidate faulty cases. The more terms we have in the solution, the less is the diagnostic resolution.

Solving Boolean Differential Equations with SSBDD

BDDs have been proven to be an efficient data structure for manipulations with Boolean functions, [50] and several efficient tools for their calculation were developed. Since the diagnostic equations (5.7) and (5.8) represent Boolean expressions, we can easily find the final diagnostic assertions by manipulations of BDDs, which allows us to avoid the explosion of the expressions (5.8) when the parentheses are opened.

Algorithm 5.1 Fault diagnosis

Require: $T = \{T_1, T_2, \ldots, T_n\}$: set of n test patterns
Ensure: $\forall k = 1, \ldots n$: D_k: diagnosis after the experiment with the sequence of test patterns $T_1, \ldots, T_k$

1: $D(T_1) \leftarrow \mathrm{CreateSSBDD}(T_1)$
2: $D_1 \leftarrow D(T_1)$ ▷ diagnosis after the first test experiment
3: **for** $k \leftarrow 2, \ldots, n$ **do**
4: $D(T_k) \leftarrow \mathrm{CreateSSBDD}(T_k)$
5: $D_k \leftarrow \mathrm{MergeSSBDD}(D_{k-1}, D(T_k))$ ▷ $D_k = D_{k-1} \wedge D(T_k)$
6: $D_k \leftarrow \mathrm{SimplifySSBDD}(D_k)$ ▷ based on Table 5.1
7: **end for**

Example 5.28. *Consider the two SSBDDs of Figure 5.5 represent-*

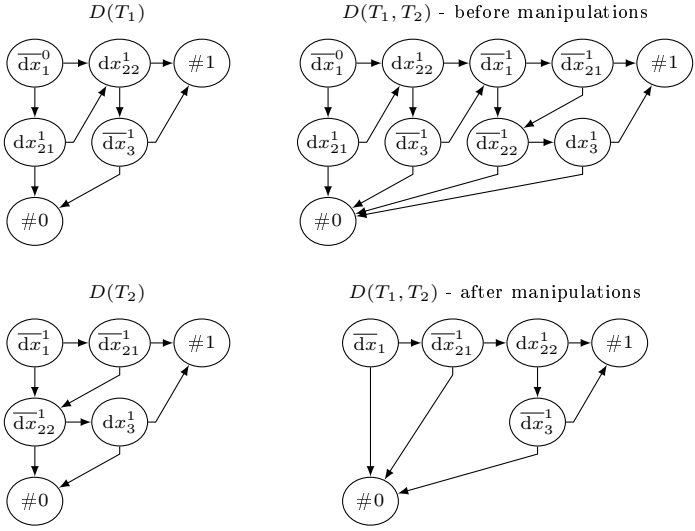


Figure 5.5. SSBDDs for the diagnostic experiment $D(T_1, T_2)$.

ing $D(T_1)$ and $D(T_2)$ in Table 5.2 (in the first two rows). The labels on the edges of the SSBDDs are omitted, the right-hand edge from a node corresponds to the value 1, and the down-hand edge corresponds to the value 0 of the node variable. The graph for $D(T_1)$ contains four 1-paths, the graph for $D(T_2)$ two, and the result of the function MergeSSBDD$(D_1, D(T_2))$ *is a SSBDD that contains eight 1-paths. By processing the paths the function* SimplifySSBDD(D_2) *can exclude 6 inconsistent paths, and create a SSBDD with two 1-paths, which represents a statement about two possible diagnostic cases:*

$$\overline{\mathrm{d}x_1}\,\overline{\mathrm{d}x}^1_{21}\mathrm{d}x^1_{22} = 1 \quad \textit{and} \quad \overline{\mathrm{d}x_1}\,\overline{\mathrm{d}x}^1_{21}\overline{\mathrm{d}x}^1_{3} = 1\ .$$

5.1.4. Fault Masking in Digital Circuits

Consider again the combinational circuit of Figure 5.1 which contains four stuck-at faults: $x_{11} \equiv 1$, $x_{22} \equiv 1$, $x_{42} \equiv 0$, and $x_{61} \equiv 0$. All the faults are depicted also in SSBDD of Figure 5.2.

Table 5.3. Test patterns for selected faults in Figure 5.1

t	Test patterns T_t									Target	Masking
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	faults	faults
1	0	0	-	1	1	1	0	1	0	$x_{11} \equiv 1$	$x_{61} \equiv 0$
2	1	0	-	1	1	1	0	0	1	$x_{61} \equiv 0$	$x_{22} \equiv 1$
3	0	0	1	1	0	1	1	-	1	$x_{22} \equiv 1$	$x_{42} \equiv 0$
4	0	1	0	1	1	1	1	-	1	$x_{42} \equiv 0$	$x_{11} \equiv 1$
5	1	0	-	1	1	0	0	1	0	$x_{22} \equiv 1$	$\emptyset$

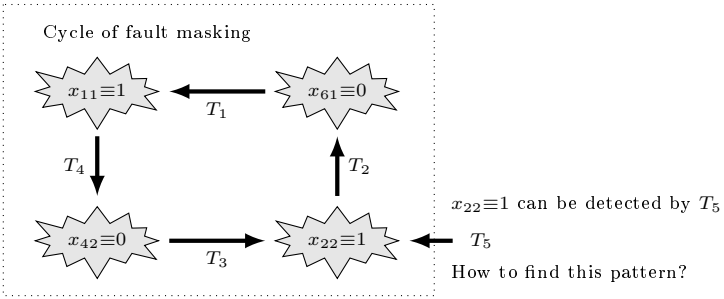


Figure 5.6. Four faults masking each other in a cycle.

Example 5.29. *Table 5.3 represents in the first four rows four test patterns targeting these faults ("target faults" in column 11) as single faults. All the four test patterns will pass and not detect the target faults because of circular masking of each other (column 12). The cycle of masking is shown in Figure 5.6.*

However, there exists another test pattern T_5 (in Table 5.3 and Figure 5.6) which would be able to "break the masking cycle" by detecting the fault $x_{22} \equiv 1$, one of the targeted four faults. The problem is how to find this pattern, or in general, how to find a test pattern for a given fault, which would be immune against masking by any possible combination of multiple faults.

As already mentioned above, a method has been proposed to avoid fault masking by using two pattern test pairs where the first pattern has the task to test the target fault, and the second pattern has the

Table 5.4. Test pairs for testing signal paths in the circuit in Figure 5.1

t	Test pairs $TP_t = \{T_t, T_{t+1}\}$									Target faults	Target wires	Masking faults
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9			
1	0	0	-	1	1	1	0	1	0	$x_{11} \equiv 1$	x_{11}	$x_{61} \equiv 0$
2	1	0	-	1	1	1	0	1	0	$x_{61} \equiv 0$		$x_{22} \equiv 1$
3	1	0	-	1	1	1	0	0	1	$x_{61} \equiv 0$	x_{61}	$x_{22} \equiv 1$
4	1	0	-	1	1	0	0	0	1	$x_{22} \equiv 1$		$x_{42} \equiv 0$
5	0	0	1	1	0	1	1	-	1	$x_{22} \equiv 1$	x_{22}	$x_{42} \equiv 0$
6	0	1	1	1	0	1	1	-	1	$x_{42} \equiv 0$		$x_{11} \equiv 1$
7	0	1	0	1	1	1	1	-	1	$x_{42} \equiv 0$	x_{42}	$x_{11} \equiv 1$
8	0	1	0	0	1	1	1	-	1	$x_{11} \equiv 1$		$x_{61} \equiv 0$
9	1	0	-	1	1	1	0	1	0	$x_{61} \equiv 0$	x_{61}	$x_{22} \equiv 1$
10	1	0	-	1	1	0	0	1	0	$x_{22} \equiv 1$		$\emptyset$

role of testing the possible masking faults [34, 69]. The main idea of this concept is to conclude from the passed test pair the correctness of the wire x_i under test, i.e., the absence of the two faults $x_i \equiv 0$ and $x_i \equiv 1$.

Unfortunately, not always the test pairs are working as expected [163, 321].

Example 5.30. *The first 8 rows in Table 5.4 contain four test pairs targeting the same four faults as shown in Figure 5.1 ("target faults" in column 11) by testing the corresponding wires x_{11}, x_{22}, x_{42}, and x_{61} for both faults SAF-1 and SAF-0. None of the test pairs will detect any of the four faults (see Figure 5.7), all 8 patterns will pass returning the message that all four wires are working correctly, however, this is not the case.*

The first test pair $TP_1(T_1, T_2)$ consisting of test patterns T_1 and T_2 is not able to prove the correctness of the wire x_{11}: the first pattern T_1 targeting the fault $x_{11} \equiv 1$ will pass because of the masking fault $x_{61} \equiv 0$ whereas the second pattern T_2 which targets the masking fault $x_{61} \equiv 0$ will pass because of another masking fault $x_{22} \equiv 1$. The test

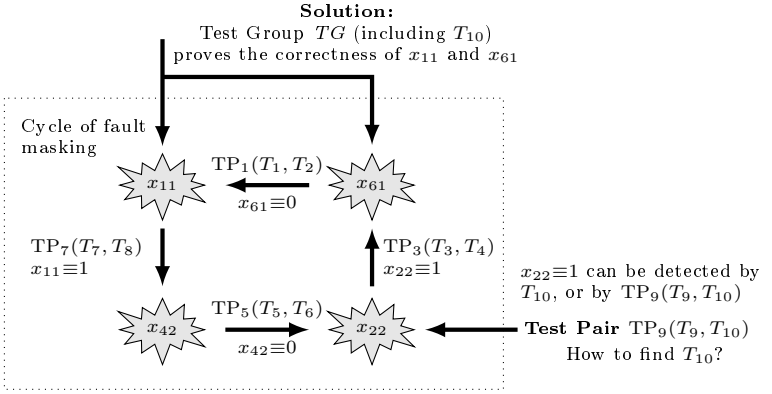


Figure 5.7. Breaking the fault masking cycle.

pair fails to prove the correctness of the wire under test.

In a similar way the test pair $TP_3(T_3, T_4)$ will fail at testing the wire x_{61}, the test pair $TP_5(T_5, T_6)$ will fail at testing the wire x_{22}, and the test pair $TP_7(T_7, T_8)$ will fail at testing the wire x_{42}. The cycle of masking closes.

There is however a test pair $TP_9(T_9, T_10)$ in Table 5.4 and Figure 5.7 which would be able to "break the masking cycle" by testing the wire x_{61}, and detecting the fault $x_{22} \equiv 1$, one of the four faults of Figure 5.1. The problem is how to find the test pair TP_9 involving the pattern T_{10}, or in general, how to find a test pair for a given wire, which would be immune against masking by any possible combination of multiple faults.

The answer lays in a solution based on constructing test groups instead of test pairs [323, 326]. A possible solution for this example is presented in Table 5.5 as a set of three test patterns which are targeted to test the wires x_5 and x_{61}, being immune to the masking fault $x_{22} \equiv 1$. The first pattern T_0 will pass and not detect the fault $x_{61} \equiv 0$, because of the masking fault $x_{22} \equiv 1$, the second pattern T_1 will fail as well when trying to detect the fault $x_{22} \equiv 1$ because of x_6 is changing its value, however, the third pattern T_2 will detect $x_{22} \equiv 1$ and break in this way the cycle of masking.

Table 5.5. Partial test group which detects all the four faults in Figure 5.1

t	Test group $TG = \{T_t, T_{t+1}, T_{t+2}\}$									Target faults	Target wires	Masking faults
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9			
0	1	0	0	1	1	1	0	1	0	$(x_5 \equiv 0)$, $x_{61} \equiv 0$		$x_{61} \equiv 0$ masked by $x_{22} \equiv 1$
1	1	0	0	1	1	0	0	1	0	$x_{22} \equiv 1$, $(x_{61} \equiv 1)$	x_{61}, x_5	$x_{22} \equiv 1$ not detected
2	1	0	0	1	0	1	0	1	0	$x_{22} \equiv 1$, $(x_5 \equiv 1)$		$x_{22} \equiv 1$ detected

Let us describe shortly the main idea of constructing the test groups [323, 326], i.e., how to find the third test pattern in this example which allows to break the masking cycle.

5.1.5. Topological View on Fault Masking

The concept of Test Groups

Definition 5.26. *Let us introduce the terms:*

1. ***full test group*** $TG = \{T_0, T_1, \ldots, T_k\}$,
2. ***main pattern*** $T_0 \in TG$ *of the test group, and*
3. *subset of* ***co-patterns*** $TG^* = \{T_1, \ldots, T_k\} \subset TG$ *of the test group.*

The main pattern T_0 *activates a* ***main path*** $L_0 = (m_0, \#e)$ *in a SSBDD from the root node* m_0 *to one of the terminal nodes* $\#e, e \in \{0, 1\}$*, and each co-pattern* T_i *activates a* ***co-path*** $L_i = (m_0, \#(\neg e))$ *through the node* $m_i \in L_0$*, so that all* T_i *will differ from* T_0 *only in the value of* $x(m_i)$*.*

The test group TG has the target to test a subset of nodes:

$$M_{TG} = \{m_1, ..., m_k\} \subseteq L_0 \ ,$$

where at T_0, for all $i = 1, \ldots, k : x(m_i) = e$. T_0 has the target to test all the faults $x(m_i) \equiv \neg e$, $m_i \in M_{TG}$, and each co-pattern T_i has the target to test the fault $x(m_i) \equiv e$. The main condition of the test group TG is that all the variables which do not belong to the nodes $M_{TG} = \{m_1, \ldots, m_k\} \subseteq L_0$ should keep the same value for all the patterns in TG.

Example 5.31. *Table 5.6 depicts the test group*

$$TG = (T_0, T_1, T_2, T_3, T_4) \ .$$

Let D0-DDD010 (123456789) be a symbolic representation of the test group where $D = 1$ in T_0, and for other T_t, only one of the D-s is equal to 0.

The main pattern T_0 activates the main path

$$L_0 = (x_{11}, x_{21}, x_{41}, x_5, x_{61}, \#1)$$

shown by bold edges in Figure 5.2. TG has the target to test the subset of nodes:

$$M_{TG} = \{x_{11}, x_{41}, x_5, x_{61}\}$$

in the SSBDD, particularly, $x_{11} \equiv 0, x_{41} \equiv 0, x_5 \equiv 0, x_{61} \equiv 0$ by T_0, $x_{11} \equiv 1$ by T_1, $x_{41} \equiv 1$ by T_2, $x_5 \equiv 1$ by T_3, and $x_{61} \equiv 1$ by T_4. Consequently, according to the definition of SSBDD, this test group tests all the SSAF on the signal paths starting on the inputs of the FFR $x_{11}, x_{41}, x_5, x_{61}$ up to the output y of the circuit.

Note, the values of the other variables x_2, x_7, x_8, x_9, not belonging to the main path L_0, remain unchanged for TG. The values of the variables which are not essential for creating the test group, e.g., x_3, may remain ***don't care****, however if assigned they have to be stable for all of $T_t \in TG$.*

Theorem 5.19. *A full test group TG for a subset of nodes $M_{TG} \subseteq L_0$ is robust with respect to any MSAF in the corresponding combinational circuit.*

Proof. The proof is given in [323]. □

Table 5.6. Full test group for testing an SSBDD path in Figure 5.2

t	Test group TG for testing M_{TG}									Tested faults	Masking faults
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9		
T	**D**	0	-	**D**	**D**	**D**	0	1	0		
0	1	0	-	1	1	1	0	1	0	all $\equiv 0$	$x_{22} \equiv 1$
1	0	0	-	1	1	1	0	1	0	$x_{11} \equiv 1$	$x_{61} \equiv 0$
2	1	0	-	0	1	1	0	1	0	$x_{41} \equiv 1$	$\emptyset$
3	1	0	-	1	0	1	0	1	0	$x_5 \equiv 1$	$\emptyset$
4	1	0	-	1	1	0	0	1	0	$x_{61} \equiv 1$	$\emptyset$

Theorem 5.19 gives sufficient conditions for generating test patterns immune to fault masking at any combination of MSAF. However, it is not always necessary to create the full test groups for detecting multiple faults.

Definition 5.27. *Consider a full test group $TG = \{T_0, TG^*\}$ with main pattern T_0, and a set of co-patterns $TG^* = \{T_1, \ldots, T_k\}$. Let us call any subset $TG_p = \{T0, TG_p^*\}$ where $TG_p^* \subset TG^*$, a **partial test group**.*

Definition 5.28. *Let us introduce a term **activated masking path**. Note that the role of each co-pattern $T_i \in TG^*$ of the partial test group TG^* is to keep the masking paths, which may corrupt the result of the main pattern T_0, activated. Activation of the masking path is the necessary and sufficient condition for detecting the faults targeted by the test group.*

Consider a skeleton of the SSBDD in Figure 5.8 (a) with the root node m_0, two terminal nodes #0, #1, and two faulty nodes $a \equiv 0$, $c \equiv 1$. The dotted lines represent activated paths during a test pair $TP = \{T_0, T_1\}$ which has the goal to test the correctness of the node a. T_0 is for activating the correct path $L_1 = (m_0, a, \#1)$ for detecting $a \equiv 0$ with expected teh test result #1. If the fault is present, then instead of L_1, a "faulty" path $L_0 = (a \equiv 0, c, \#0)$ should be activated with the faulty result #0.

In case of a masking fault $c \equiv 1$ on L_0, a masking path $L_M = (a, c, \#1)$

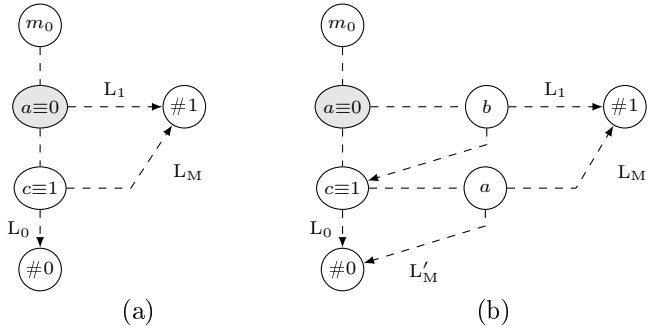


Figure 5.8. Topological view: (a) test pair, (b) test group.

will be activated, and $a \equiv 0$ will not be detected by T_0. However, at T_1 the masking path L_M remains activated because of the masking fault $c \equiv 1$, and the wrong test result #1 will indicate the presence of a masking fault in the circuit.

Both patterns of $TP = \{T_0, T_1\}$ will pass and not detect any faults if the masking path L_M will contain a node labeled by the same variable as the tested node. For example, in Figure 5.8 (b), both L_1 and L_M contain a node with the same variable a, which is the reason why the test pair is not sufficient for detecting the multiple fault $\{a \equiv 0, c \equiv 1\}$. In this case the co-pattern T_1 of the test pair TP is not able to keep the masking path activated.

To overcome the problem, it would be necessary and also sufficient to include into the set of nodes to be tested by a partial test group at least one node which is labeled by a variable not labeling any node on L_M. For example, in Figure 5.8 (b), it would be sufficient for detecting the multiple fault $\{a \equiv 0, c \equiv 1\}$ to generate the partial test group for testing the nodes $\{a, b\}$.

Multiple Fault Testing by Partial Test Groups

Theorem 5.20. *A partial test group $TG' \subseteq TG$ for a subset of nodes $M'_{TG} \subseteq M_{TG}$ is robust with respect to any MSAF if for each possible*

masking path L_M, there exists a node $m \in M'_{TG}$, so that no node on L_M will have the same variable $x(m)$.

Proof. Indeed, suppose, the main test pattern $T_0 \in TG'$ does not detect a fault A because of another fault B which activates a masking path L_M. According to Definition 5.26 of test groups, each co-pattern $T_i \in TG'$ differs from T_0 in a value of a single variable over the variables of M'_{TG}. Suppose T_i is testing the node m labeled by the variable $x(m)$. Since L_M does not contain any node labeled by the variable $x(m)$, it remains activated during T_i and, hence, provides the same result for T_i as it was for T_0. This means that the pattern T_i has detected the masking fault B . The same considerations hold for every possible masking path. □

Corollary 5.2. *The test pair is a special case of the partial test group where $|M'_{TG}| = 1$.*

Algorithm 5.2 Generation of a robust test group

Require: $TG' \subseteq TG = \{T_0, TG^*\}$: partial test group
Ensure: $TG_r \subseteq TG$: robust test group
1: **for all** $T_k \in TG' \setminus T_0$ **do** ▷ all masking paths
2: $L_k \leftarrow \text{CoPath}(T_k)$
3: **if** L_k does not satisfy the conditions of Theorem 5.20 **then**
4: **if** TG' can be extended according Theorem 5.20 **then**
5: $TG' \leftarrow \text{Extend}(TG', T_i)$ ▷ according Theorem 5.20
6: **else**
7: $L_k \leftarrow \emptyset$ ▷ masking path L_k is redundant [323]
8: **return** $\emptyset$ ▷ there is no robust test group of TG'
9: **end if**
10: **end if**
11: **end for**
12: $TG_r \leftarrow TG'$
13: **return** TG_r

Example 5.32. *Figure 5.9 presents a topological view based on a skeleton of an SSBDD on different possibilities of fault masking. L_0 represents a main path of a possible test group as a basis for a partial test group under construction. The partial test group TG' will target the nodes $M'_{TG} = \{a, b, c\}$. $T_0 \in TG'$ will not detect the fault $a \equiv 0$*

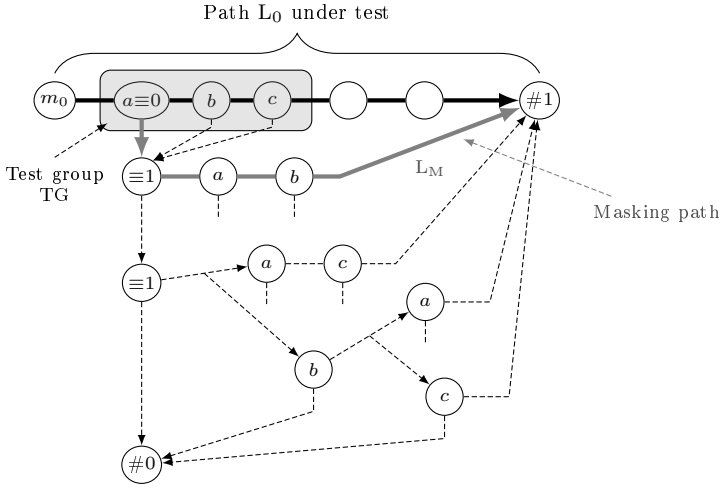


Figure 5.9. Topological view on the fault masking mechanism.

because of another SAF-1 fault. There may be arbitrary combinations of masking faults in the circuit denoted by $\equiv 1$. *Possible masking paths are depicted by dotted lines. As we see, for each such a path* L_M, *there exists always a node* $m \in M'_{TG}$ *with a variable* $x(m)$ *which is missing on the particular* L_M.

Example 5.33. *An example of a full test group is depicted in Table 5.6, which tests the nodes* $M_{TG} = \{x_{11}, x_{41}, x_5, x_{61}\}$ *of Figure 5.2. Another example of a partial test group is depicted in Table 5.5, which tests the nodes* $M'_{TG} = \{x_5, x_{61}\} \subset M_{TG}$. *Both test groups are immune to fault masking, however, it would be always easier to generate smaller partial test groups because of eventual inconsistencies during test group generation.*

The concept of the test groups was discussed up to now for single SSBDD models. In case of a system of SSBDDs, we have to either reduce the set of SSBDDs by superposition [325] to the single SSBDD case, or to use a hierarchical approach. The test generation complexity problem inherent in the concept of test groups can be overcome by splitting the full test groups into partial test groups as suggested by Theorem 5.20.

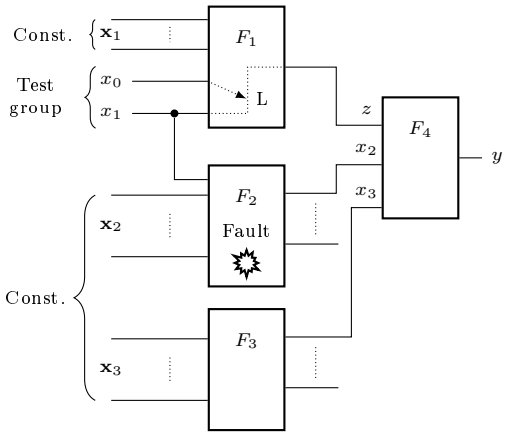


Figure 5.10. Hierarchical fault diagnosis.

The main importance of the idea of test groups is to identify or prove the correctness of a subset of nodes in SSBDD. The knowledge of the nodes identified already as correct allows to generate smaller partial test groups to ease test generation, and the known correct nodes can be dropped during the analysis. The method allows us to create a sequential procedure of fault diagnosis by extending step by step the faultfree core in the circuit at any present multiple fault.

5.1.6. Test Groups and Hierarchical Fault Diagnosis

In [323, 326], a conception of test groups was introduced, and the necessary and sufficient conditions for detecting MSAF in combinational circuits were introduced. The goal of a test group or a partial test group is to verify the correctness of a selected part of the circuit. In case of passing of all the test groups, the circuit is proven fault-free. If not all test groups will pass, a fault diagnosis is needed, which can be carried out by solving diagnostic equations (5.7) and (5.8) locally, i.e., in the selected region of the circuit targeted by the test group.

Consider a circuit of Figure 5.10 as a higher level network with blocks F_1, F_2, F_3, and F_4. Let us start the fault diagnosis first, at the lower

Table 5.7. Diagnostic processes for the circuit in Figure 5.10

T_t	x_0	x_1	x_2	x_3	y	$\mathrm{d}y$	Assertions
T_0	1	1	0	0	1	0	$\overline{\mathrm{d}x_0^1}\,\overline{\mathrm{d}x_1^1} \vee \mathrm{d}x_2^0 \vee \mathrm{d}x_3^0$
T_1	0	1	0	0	0	0	$(\overline{\mathrm{d}x_0^0} \vee \mathrm{d}x_1^1)\,\overline{\mathrm{d}x_0^2}\,\overline{\mathrm{d}x_0^3}$
$D(T_0, T_1)$							$\overline{\mathrm{d}x_0}\,\overline{\mathrm{d}x_1^1}\,\overline{\mathrm{d}x_2^0}\,\overline{\mathrm{d}x_3^0}$
T_2	1	0	0	0	0	0	$(\mathrm{d}x_0^1 \vee \overline{\mathrm{d}x_1^0})\,\overline{\mathrm{d}x_0^2}\,\overline{\mathrm{d}x_0^3}$
$D(T_0, T_1, T_2)$							$\overline{\mathrm{d}x_0}\,\overline{\mathrm{d}x_1}\,\overline{\mathrm{d}x_2^0}\,\overline{\mathrm{d}x_3^0}$
T_2'	1	0	0	0	0	1	$\overline{\mathrm{d}x_0^1}\,\mathrm{d}x_1^0 \vee \mathrm{d}x_2^{0*} \vee \mathrm{d}x_3^0$
$D(T_0, T_1, T_2')$							$\overline{\mathrm{d}x_0}\,\overline{\mathrm{d}x_1^1}\,\overline{\mathrm{d}x_3^0}\,(\mathrm{d}x_1^0 \vee \mathrm{d}x_2^{0*}\,\overline{\mathrm{d}x_2^0})$

level, by selecting a subcircuit with inputs x_0 and x_1 as a target in the block F_1. The subcircuit will be tested along an activated test path L through the wire z up to the output y of the circuit.

Assume, we have a test group $TG = \{T_0, T_1, T_2\}$ applying to the inputs x_0 and x_1 a set of changing patterns whereas the values on all other primary inputs of the circuit are kept constant, according to the partial test group conception. The role of the test group is to concentrate on testing by all the patterns T_0, T_1, T_2 a joint signal path L. If the test group will pass, we conclude that the path L through the wire z is working correctly. Each additional passed test group will add information about other correct signal paths. This information will help to locate the faults if some test groups will fail.

If we have proven the correctness of a subset S of signal paths then we can use these paths for sending correct signals to the needed connections in the higher network level which makes easier to test other blocks at the lower level. For example, if we can force correct signals on the wires z and x_3 in Figure 5.10, to activate a test path from x_2 up to the output y, we may carry out the fault reasoning in the block F_2 only locally to reduce the complexity of the diagnosis problem.

Example 5.34. *Consider again the circuit in Figure 5.10, and apply the test group $TG = \{T_0, T_1, T_2\}$ to the block F_1, so that the inputs*

x_0 and x_1 will have the local patterns $\{11, 01, 10\}$ and the values on all other primary inputs of the circuit are kept constant. Assume that the activated test path by TG can be represented by a function:

$$y = F(x_0, x_1, x_2, x_3) \ . \tag{5.10}$$

Assume also that all the signals activating the test path of the test group and originating at the inputs $\mathbf{x}_1$ have been proved as correct. This allow us to reduce the diagnosis problem raised by the test group TG to processing of the simplified function, for example as:

$$y = x_0 x_1 \vee x_2 \vee x_3 \ . \tag{5.11}$$

The diagnostic process with three passed test patterns of the test group is depicted in Table 5.7. The final assertion $D(T_0, T_1, T_2)$ states that no faults are present along the signal paths from the inputs x_0 and x_1 up to the output y, and the faults $x_2 \equiv 1$ and $x_3 \equiv 1$ are missing on the inputs of the block F_4. The knowledge about the correctness of the wire z and the missing fault $x_3 \equiv 1$ allows to carry out the fault reasoning in the block F_2 only locally.

Suppose the pattern $T_2' \in TG$ will fail. The diagnostic statement $D(T_0, T_1, T_2')$ refers to either the fault candidate $x_1 \equiv 1$ or to the unstable behavior of the wire x_2 during the execution of the test group. The reason of the instability of x_2 may be a fault in the block F_2 which because of the changing value of x_1 may sometimes influence on x_2, and sometimes not. In this case the test group has not fulfilled its role to prove the correctness of the wire z.

As shown in Example 5.34, the role of the Boolean differential lays on specifying the fault candidates in the case when the test group will fail. The method of solving differential equations will help also in this case when some of the test groups cannot be synthesized and it would not be possible to prove the correctness of some parts of the circuit.

5.1.7. Experimental Data

Table 5.8 summarizes experimental data regarding the test group synthesis for the ISCAS'85 benchmark circuits [49]. The columns 3 and

Table 5.8. Experimental data of generating test groups for diagnosis

Circuit	Gates #	SSAF test #	MSAF test #	Group cover %
c432	275	53	314	82,91
c499	683	86	482	67,2
c880	429	84	546	99,8
c1355	579	86	514	65,6
c1908	776	123	621	96,3
c2670	1192	103	820	76,3
c3540	1514	148	995	80,3
c5315	2240	104	1523	91,8
c6288	2480	22	465	98,1
c7552	3163	202	1863	87,8

4 show the number of patterns in the SSAF test and in the MSAF test, respectively.

The 3-pattern test groups were built for the gates of the circuits whereas many test groups were possible to merge. Repeated patterns were removed from the test set. The number of synthesized test groups in the test sets is several times larger compared to the traditional SSAF tests. The fair comparison between the SSAF and MSAF test lengths, however, cannot be done in the present research, since the test groups for different outputs were not merged. Merging of test groups will provide a significant reduction of the MSAF test length.

The group coverage means the percentage of the test groups that were built successfully. The test group coverage characterizes the feasibility of the concept to prove the correctness of subcircuits instead of targeting the faults to be tested. For diagnosing the subcircuits not covered by test groups, Boolean differentials can be utilized.

5.1.8. General Comments About Proposed Methods

In this section we investigated the two sides of the fault diagnosis problem: how to develop efficient diagnostic test sequences and how

to locate the faults if some test patterns will not pass. The test group concept affords to concentrate on the diagnosis of small parts of circuits to cope with the complexity problem by hierarchical fault reasoning. On the other hand, using test groups allows us to prove the correctness of selected parts in a given circuit.

We presented a new idea for multiple fault diagnosis, which combines the concept of multiple fault testing by test groups with solving Boolean differential equations by manipulation of SSBDDs. The role of the test groups is to prove step by step the correctness of the given circuit. We showed how to generate partial test groups with SSBDDs to cope with the complexity of the problem. If the whole circuit is covered by test groups, and all the test groups will pass, fault reasoning is not needed. If some parts of the circuit will be detected as faulty by test groups, the more exact fault reasoning by solving Boolean differential equations is needed.

To avoid the memory explosion when solving the Boolean differential equations, the manipulation of SSBDDs based on the proposed 5-valued algebra for fault reasoning will serve as an efficient tool.

5.2. Blind Testing of Polynomials by Linear Checks

Ariel Burg Osnat Keren

Ilya Levin

5.2.1. Functional Testing by Linear Checks

Two alternative types of system testing are known: on-line and off-line testing. The on-line testing (usually called concurrent checking) requires introducing an additional circuitry for detecting faults during the normal operation of the system. This kind of testing protects the system from both permanent and transient faults that may occur during its operation. In contrast, the off-line testing is a procedure allowing to check the system before use. This kind of test protects the system from two types of faults: fabrication faults and faults that occurred before the test has been applied. The off-line testing is based on applying a predefined set of test vectors. Two types of off-line testing are used:

- testing by using external equipment, and
- self-testing running on a *built-in* circuit.

This section belongs to the area of built-in-self-test.

There are two conceptual levels for testing which in turn define different testing methods: gate level and functional level testing. In gate level testing, the test vectors suit a specific implementation, while on the functional level, the testing is independent from the specific implementation and tests the correctness of the operation.

Up to now, design of functional testing has been carried out only if the functionality of the system was known to the test designer. In this section we address the question whether it is possible to design functional testing for systems whose functionality is unknown.

In the following subsections it is shown that spectral approaches can be applied to solve the above problem. The spectral approach to testing was studied in [136, 205], and in the papers collected in the compendium "*Spectral Techniques and Fault Detection*" [153].

There are two main courses in spectral testing:

- verification of the correctness of the corresponding Walsh coefficients, and
- *testing by linear checks* (also called test vectors).

Testing by verification of Walsh coefficients can be viewed as data compression of test responses. This approach eliminates the problem of check set (test set) generation and storage, but it requires exhaustive application of all 2^n possible input patterns. The second approach, testing by a linear check set, is considered to be more efficient. In most cases, it does not require exhaustive application of all 2^n possible n-bit input patterns and at the same time it eliminates the problem of check set generation [152–155].

Linear checks are used in the context of detecting permanent stack-at faults. Linear checks allow us to define the check set analytically. For functions whose Walsh spectra contains sufficiently many zeros the check set forms a relatively small subgroup, and thus the implementation cost of the testing mechanism becomes negligible in respect to the cost of the overall system. Polynomials of low order belong to this class of functions.

Denote by $K^{n_m,\ldots,n_1}_{s_m,\ldots,s_1}[\mathbf{x}_m,\ldots,\mathbf{x}_1]$ the class of polynomials

$$f(\mathbf{x}_m,\ldots,\mathbf{x}_1)=\sum_{i_m=0}^{s_m}\cdots\sum_{i_1=0}^{s_1}a(i_m,\ldots,i_1)\mathbf{x}_m^{i_m}\cdots\mathbf{x}_1^{i_1}$$

of m integer variables $\mathbf{x}_m,\ldots,\mathbf{x}_1$ with known precision $n_m,\ldots,n_1$. Methods for constructing linear (equality) checks for *a given polynomial* in $K^{n_m,\ldots,n_1}_{s_m,\ldots,s_1}[\mathbf{x}_m,\ldots,\mathbf{x}_1]$ satisfying $s_t<n_t$ where s_t is the maximal degree of t'th variable, are presented in [152, 154]. This chapter addresses the problem of construction of linear checks for cases where *no information is provided* on the system except the fact that it has

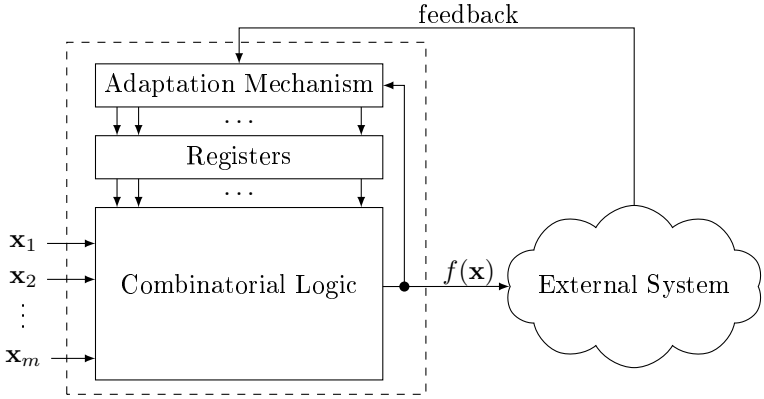


Figure 5.11. The architecture of a WbAH system.

an acceptable representation as a polynomial of order M. In particular, a method to construct linear checks for *Walsh-based Adaptive Hardware* (WbAH) is provided.

A WbAH is based on representing the system in the Walsh frequency domain (see Figure 5.11) [157]. An (n, M) WbAH is an n-input bit circuit that can acquire the spectral coefficients of any polynomial of order M (and hence acquire the functionality of the system). As reported in [157], a WbAH provides better performance than conventionally *Multiply and Accumulate*(MAC) based architectures in terms of its acquisition time and the average residual error (which reflects the difference between the target functionality of the system and the functionality that the hardware has converged to). However, its main advantage over a MAC-based architecture is that it acquires the functionality of the system even if neither

- the coefficients of the polynomial, nor
- the number of real (or complex) variables, nor
- the maximal degree of each variable, nor
- the precision of each variable, nor
- the order of the variables,

are known to the system designer.

This section presents linear checks for spectral testing of WbAH. As shown later in this chapter, the proposed check set is optimal - it is the smallest set that allows testing the WbAH without identifying the polynomial the system has converged to. The efficiency of the suggested approach in terms of implementation cost and in comparison to structural testing is demonstrated in [54].

5.2.2. Walsh Spectrum of Polynomials

Consider a system that has m real (or complex) inputs, and let the functionality of the system be represented by the function $f(\mathbf{x}_m, \ldots, \mathbf{x}_1)$. Without loss of generality, assume that the domain and the range of the function are $[-1,1)^m$ and $[-1,1)$, respectively. The inputs and the output of the system are quantized; each variable $\mathbf{x}_w$, $w = 1, \ldots, m$, is represented by a binary vector of n_w bits, and the value of the function is represented by a binary vector of length k. The mapping between the binary vectors and the numbers in interval $[-1,1)$ may be according to the 2's complement representation or any other weighted number system [165].

To simplify the presentation, bold letters are used for real variables, and italic letters for Boolean variables. For example, the real variable $\mathbf{x}_w$ is represented by a binary vector $(x^{(w)}_{n_w-1}, \ldots, x^{(w)}_1, x^{(w)}_0)$. Similarly, a vector $\mathbf{x} = (\mathbf{x}_m, \ldots, \mathbf{x}_1)$ of m real variables, where each variable $\mathbf{x}_w$ is represented by n_w bits, can be referred to as a binary vector $x = (x_{n-1}, \ldots, x_0)$ of length $n = \sum_{w=1}^{m} n_w$.

Definition 5.29 (Order of polynomial)**.** *Define,*

$$\mathbf{x} = (\mathbf{x}_m, \ldots, \mathbf{x}_1) \in \mathcal{C}^m \quad \textit{and} \quad D = (d_m, \ldots, d_1), d_i \in \{0\} \cup \mathcal{Z}^+ \ ,$$

and denote by $\mathbf{x}^D$ *the monomial (product)* $\prod_{i=1}^{m} \mathbf{x}_i^{d_i}$*. Let*

$$f(\mathbf{x}) = \sum_j a_j \mathbf{x}^{D_j}$$

be a polynomial of m *variables,* $a_i \in \mathcal{C}$*. The order* M *of the polynomial*

is:

$$M = \max_j L_1(D_j) \ ,$$

where L_1 *is the* 1*-norm,* $L_1(D) = \sum_{i=1}^{m} d_i$.

The class of polynomials $f(\mathbf{x})$ of order $\leq M$ is denoted by K_M. The class of polynomials $f(\mathbf{x}) = f(\mathbf{x}_m, \ldots, \mathbf{x}_1)$ of order $\leq M$ in $m \leq n$ quantized real variables is denoted by K_M^n. Indeed,

$$K_M^n \subset \bigcup_{m \leq n,\ \sum n_t \leq n,\ s_t \leq M} K_{s_m,\ldots,s_1}^{n_m,\ldots,n_1}[\mathbf{x}_m, \ldots, \mathbf{x}_1] \ . \tag{5.12}$$

A polynomial $f \in K_M^n$ can be referred to as a set of k switching functions of n binary variables $\{x_i\}_{i=0}^{n-1}$, or equivalently as a single multi-output function $f(x) = f(x_{n-1}, \ldots, x_0)$. The properties of a single multi-output function (or a set of switching functions) can be analyzed via the Walsh spectrum.

Definition 5.30 (Walsh functions)**.** *Let be* $x = (x_{n-1}, \ldots, x_0)$ *and* $i = (i_{n-1}, \ldots, i_0)$ *two binary vectors of length* n*. The Walsh function* $W_i(x)$ *is defined as* $W_i(x) = (-1)^{<x,i>} = exp(j\pi \sum_{m=0}^{n-1} x_m i_m)$.

Denote by $\mathcal{C}_2^n$ the group of all binary n-vectors with respect to the operation $\oplus$ of component-wise addition mod 2.

Definition 5.31 (Walsh spectrum)**.** *The coefficients* s_i *of the Walsh spectrum are elements of the vector* $S = (s_{2^n-1}, \ldots, s_0)$*, where,*

$$s_i = \sum_{x \in \mathcal{C}_2^n} W_i(x) f(x), \ \textit{and} \ f(x) = 2^{-n} \sum_{i \in \mathcal{C}_2^n} W_i(x) s_i \ .$$

The Walsh spectrum of $f \in K_M^n$ has the following property:

Theorem 5.21. *[157] Let* $f \in K_M^n$ *be a switching function in* n *binary variables that corresponds to a polynomial of order* $M < n$*. Then, the spectral coefficient* s_i*,* $i = 0, \ldots, 2^n - 1$*, equals zero if the Hamming weight of* i *is greater than* M*.*

The correctness of Theorem 5.21 follows from the linearity of the Walsh transform and from the fact that the polynomial can be represented as a sum of products of up to M Boolean variables.

Theorem 5.21 provides an upper bound on the number of non-zero spectral coefficients of any polynomial in K_M^n. The bound does not depend on the number of real (or complex) inputs nor on their precision. In this sense, a WbAH based on this bound is more robust than a conventional MAC implementation of a system that has an acceptable representation as a low order polynomial, since it acquires its target functionality even in cases where almost no information about the system is available.

5.2.3. Spectral Testing of a Given Polynomial by Linear checks

Linear checks are a method for an off-line self-testing [154]. The method is based on the fact that for any given multi-valued function f in n Boolean variables there exists a subgroup T of $\mathcal{C}_2^n$ and a constant d such that $\sum_{\boldsymbol{\tau}\in T} f(\mathbf{x}\oplus\boldsymbol{\tau}) = d$. Construction of optimal linear checks for a given f involves finding a minimal check set (the subgroup T) and computation of d.

Denote by $V(n_t, s_t+1)$ a maximal linear code $[n_t, k_t, s_t+1]$ in $\mathcal{C}_2^{n_t}$ of length n_t, dimension k_t and Hamming distance s_t+1. The dual code of $V(n_t, s_t+1)$ is the linear subgroup :

$$V^{\perp}(n_t, s_t+1) = \{\boldsymbol{\tau}_t | \bigoplus_{j=0}^{n_t-1} \tau_{t,j} y_{t,j} = 0, \forall \mathbf{y}_t \in V(n_t, s_t+1)\}$$

of dimension $n_t - k_t$, where $\boldsymbol{\tau}_t = (\tau_{t,n_t-1}, \ldots, \tau_{t,0}), \mathbf{y}_t = (y_{t,n_t-1}, \ldots, y_{t,0})$. Defined by $V^{\perp}$, a linear code of length n that is the Cartesian product of m codes,

$$V^{\perp} = \prod_{t=1}^{m} V^{\perp}(n_t, s_t+1) = \{\boldsymbol{\tau} = (\boldsymbol{\tau}_m, \ldots, \boldsymbol{\tau}_1) | \boldsymbol{\tau}_t \in V^{\perp}(n_t, s_t+1)\} \ .$$

The following theorem presents linear equality checks for testing *a given polynomial*:

Theorem 5.22 ([154]). *Let* $f \in K_{s_m,\ldots,s_1}^{n_m,\ldots,n_1}[\mathbf{x}_m,\ldots,\mathbf{x}_1]$ *and* $s_t < n_t$ *for all* $1 \leq t \leq m$. *Then, the code* $V^{\perp}$ *is the check set for* f, *that is*

$$\sum_{\boldsymbol{\tau}\in V^{\perp}} f(\mathbf{x}\oplus\boldsymbol{\tau}) = \sum_{\boldsymbol{\tau}=(\boldsymbol{\tau}_m,\ldots,\boldsymbol{\tau}_1)\in V^{\perp}} f(\mathbf{x}_m\oplus\boldsymbol{\tau}_m,\ldots,\mathbf{x}_1\oplus\boldsymbol{\tau}_1) = d \quad (5.13)$$

where

$$\begin{aligned} d &= \prod_{t=1}^{m} |V(n_t, s_t+1)|^{-1} \sum_{\mathbf{x}_1,\ldots,\mathbf{x}_m} f(\mathbf{x}_m,\ldots,\mathbf{x}_1) \\ &= \prod_{t=1}^{m} |V(n_t, s_t+1)|^{-1} s_0 \ . \end{aligned}$$

Notice that for constructing $V(n_t, s_t+1), t = 1,\ldots,m$ one has to know the number of variables (m) and their precision (n_t). Furthermore, in Theorem 5.22, s_t must be smaller than n_t, so it is impossible to use this method to construct a check set other than the trivial check set ($\mathcal{C}_2^{n_t}$) in cases where $s_t \geq n_t$. The following example illustrates the difficulty in using Theorem 5.22 when the number of variables and their precision are unknown.

Example 5.35. *Consider three polynomials of order* $M = 3$,

$$\begin{aligned} f_1 &\in K_3^{62}[\mathbf{x}_1], \\ f_2 &\in K_{3,3}^{31,31}[\mathbf{x}_2,\mathbf{x}_1], \\ f_3 &\in K_{1,\ldots1,3,3}^{1,\ldots,1,15,15}[\mathbf{x}_{34},\ldots,\mathbf{x}_3,\mathbf{x}_2,\mathbf{x}_1] \ . \end{aligned}$$

According to Theorem 5.22, the linear checks for each polynomial can be constructed by defining a specific code for each one of its variables. For the cases where $s_t = 3 < n_t$ *the linear checks can be obtained by shortening the extended Hamming code [185]. The parameters of the shortened code are* $[n = n_t, k = n_t - \lceil log_2(n_t)\rceil - 1.4]$. *For* $s_t = n_t$ *the trivial check set which contains all the binary vectors of length* n_t *must be used. That is,*

1. *For the polynomial f_1 let us choose the code $V_1(62,4)$. The code is of dimension $62-(6+1)$. The dual code $V^{\perp}$ is a code of dimension 7. In other words, $V^{\perp}$ is a check set of size 2^7 for f_1.*

2. *For the polynomial f_2 let us choose two identical codes $V(31,4)$. The codes are of dimension $31-(5+1)$. The dual code $V^{\perp}$ is a Cartesian product of the two codes $V^{\perp}(31,4)$, it is a code of length 62 and of dimension $2\cdot(5+1)=12$. Namely, the dual code $V^{\perp}$ is a check set of size 2^{12} for f_2.*

3. *The polynomial f_3 does not fulfill the requirements of Theorem 5.22 since $s_i = n_i$ for $i > 2$. Nevertheless, it is possible to construct linear checks for f_3 by using the trivial checks for the variables $\mathbf{x}_3, \mathbf{x}_4, \ldots, \mathbf{x}_{34}$, and two identical codes $V(15,4)$ of length 15 and dimension $15-(4+1)$ for the variables $\mathbf{x}_1$ and $\mathbf{x}_2$. The dual code $V^{\perp}$ is a Cartesian product of all the $32+2$ codes, it is a linear code of length 62 and dimension $32+2\cdot(4+1)$. The dual code is a check set of size 2^{42} for f_3. The check set is defined by a generator matrix*

$$G = \begin{pmatrix} I_{32\times 32} & 0 & 0 \\ 0 & G_{5\times 15} & 0 \\ 0 & 0 & G_{5\times 15} \end{pmatrix}$$

where I is the identity matrix and $G_{5\times 15}$ is the (5×15) generator matrix of the dual code $V^{\perp}(15,4)$. It follows from Theorem 5.22 that the structure of G depends on the order of the variables - if the order of the variables is changed to $(\mathbf{x}_2, \mathbf{x}_1, \mathbf{x}_{34}, \ldots, \mathbf{x}_4, \mathbf{x}_3)$ then G must be changed accordingly.

Notice that each one of the three polynomials is associated with a different check set. The size of the check set and its structure depend on the number of variables, their precision and their order. Moreover, the size of the check set grows as the number of variables increases. Yet, all the three polynomials can be represented as functions in 62 Boolean variables in K_3^{62}. As such, a Walsh-based adaptive hardware that has 62 inputs may converge to each of them. Indeed, the worst-case scenario (from the point of view of the test designer) happens, for example, when there are 31 variables each represented by two bits. In this case the size of the test is 2^{62}.

5.2.4. Universal Linear Checks

Since no information about the polynomial (except the fact that it is in K_M^n) is available, we are interested in a check set that will be suitable for any polynomial in K_M^n. There are two options:

- prepare in advance a check set for each polynomial in K_M^n and select the proper one in real time, and
- construct a fixed check set that can be applied without identifying the polynomial.

The latter case is called *blind testing.* In blind testing, the check set is universal, it is applicable to any polynomial in K_M^n.

First, notice that it is impossible for an n_t-bit word to have Hamming weight of $n_t + 1$ and therefore Lemma 5.9 holds.

Lemma 5.9. *If* $s_t \geq n_t$ *then* $|V^{\perp}(n_t, s_t + 1)| = 2^{n_t}$.

Consequently, the worst-case scenario (in terms of test duration) happens when a WbAH system has converged to a polynomial for which $s_t \geq n_t$ for all t. For these polynomials, the size of the check set is 2^n.

The following lemma shows that the best-case scenario happens when $m = 1$ (and thus $n_1 = n$).

Lemma 5.10. *Denote by* $V_{(w)}^{\perp}$ *the check set for a polynomial of order* M *in* w *quantized variables in* K_M^n. *Then, for* $1 \leq m \leq n$, *we have* $|V_{(1)}^{\perp}| \leq |V_{(m)}^{\perp}|$.

Proof. $V_{(m)}^{\perp}$ is a Cartesian product of m linear codes, each has a dimension r_i. Without loss of generality, assume that the variables are indexed such that $n_1 \leq n_2 \leq \cdots \leq n_m \leq n$. The dimension of $V_{(m)}^{\perp}$ is then

$$r = r_1 + r_2 + \cdots + r_m \geq 1 + 1 + \cdots + 1 + r_m = (m-1) + r_m \ .$$

Since $n \geq n_m$, the number of codewords in the largest $V(n, M+1)$ code is greater or equal to the number of codewords in the largest

$V(n_m, M+1)$ code. Therefore,

$$\begin{aligned} |V^{\perp}_{(1)}| &= |V^{\perp}(n, M+1)| \leq |V^{\perp}(n_m, M+1)| \\ &\leq 2^{m-1}|V^{\perp}(n_m, M+1)| \leq 2^r = |V^{\perp}_{(m)}| \ . \end{aligned}$$

□

From Lemmas 5.9 and 5.10, the linear checks are spanned by n (linearly independent) vectors in the worst-case scenario, and in the best case scenario the linear checks are spanned by $log_2(|V^{\perp}_{(1)}|)$ vectors of length n. In all other cases the linear checks are determined by a Cartesian product of m codes of different lengths and dimensions.

The question is whether it is possible to aggregate the proper codes and construct the linear checks without knowing the function in advance. In other words, is it possible to identify the function from its spectrum and choose the check set accordingly?

The following example shows that it is impossible.

Example 5.36. *Let* $f, g \in K_2^{20}$ *and let*

$$\begin{aligned} f(\mathbf{x}_1, \mathbf{x}_2) &= a_1\mathbf{x}_1^2 + a_2\mathbf{x}_2^2 + a_3\mathbf{x}_1\mathbf{x}_2 + a_4\mathbf{x}_1 + a_5\mathbf{x}_2 \\ &\quad + a_6 \in K_{2,2}^{10,10}, \\ g(\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3) &= b_1\mathbf{y}_1^2 + b_2\mathbf{y}_2^2 + b_3\mathbf{y}_3^2 + b_4\mathbf{y}_1\mathbf{y}_2 + b_5\mathbf{y}_1\mathbf{y}_3 \\ &\quad + b_6\mathbf{y}_2\mathbf{y}_3 + b_7\mathbf{y}_1 + b_8\mathbf{y}_2 + b_9\mathbf{y}_3 + b_{10} \in K_{2,2,2}^{10,5,5} \end{aligned}$$

For simplicity, assume that the variables are positive numbers represented in base 2. Define $\mathbf{z}_1 = \mathbf{y}_1$ *and* $\mathbf{z}_2 = 2^5\mathbf{y}_2 + \mathbf{y}_3$. *Then for any set of a's there exists a set of b's such that* f *and* g *have the same spectral coefficients. Namely, for:*

$$\begin{array}{lllll} b_1 = a_1, & b_2 = 2^{10}a_2, & b_3 = a_2, & b_4 = 2^5a_3, & b_5 = a_3, \\ b_6 = 2^6a_2, & b_7 = a_4, & b_8 = 2^5a_5, & b_9 = a_5, & b_{10} = a_6 \end{array}$$

we have $g = f$ *and therefore* f *and* g *have the same spectral coefficients.*

It is clear from Example 5.36 that it is impossible to extract information about the type of a polynomial from its spectral coefficients.

The question is then, how to construct a non-trivial check set that can diagnose the health of the system without knowing the function it has converged to. The following theorem answers this question,

Theorem 5.23. *Let be V a subgroup of $\mathcal{C}_2^n$, and*

$$V^{\perp} = \{\boldsymbol{\tau} | \boldsymbol{\tau} \in \mathcal{C}_2^n, \sum_{s=0}^{n-1} \boldsymbol{\tau}_s i_s = 0, \forall i \in V\} \ .$$

Then for any $f(\mathbf{x})$ defined on $\mathcal{C}_2^n$:

$$\sum_{\boldsymbol{\tau} \in V^{\perp}} f(\mathbf{x} \oplus \boldsymbol{\tau}) = \frac{1}{|V|} \sum_{i \in V} W_i(\mathbf{x}) s_i \ . \tag{5.14}$$

Proof. Using Definition 5.31 for the Walsh spectrum:

$$\begin{aligned} \sum_{\boldsymbol{\tau} \in V^{\perp}} f(\mathbf{x} \oplus \boldsymbol{\tau}) &= \sum_{\boldsymbol{\tau} \in V^{\perp}} 2^{-n} \sum_{i \in \{0,1\}^n} W_i(\mathbf{x} \oplus \boldsymbol{\tau}) s_i \\ &= \sum_{\boldsymbol{\tau} \in V^{\perp}} 2^{-n} \sum_{i \in \{0,1\}^n} W_i(\mathbf{x}) W_i(\boldsymbol{\tau}) s_i \\ &= 2^{-n} \sum_{i \in \{0,1\}^n} W_i(\mathbf{x}) s_i \sum_{\boldsymbol{\tau} \in V^{\perp}} W_i(\boldsymbol{\tau}) \ . \end{aligned}$$

Therefore,

$$\sum_{\boldsymbol{\tau} \in V^{\perp}} f(\mathbf{x} \oplus \boldsymbol{\tau}) = 2^{-n} |V^{\perp}| \sum_{i \in V} W_i(\mathbf{x}) s_i = \frac{1}{|V|} \sum_{i \in V} W_i(\mathbf{x}) s_i \ .$$

□

Notice that the subgroup $V^{\perp}$ is defined in advance. Nevertheless, the actual set of test vectors to be applied, $\{\mathbf{x} \oplus \boldsymbol{\tau}\}_{\boldsymbol{\tau} \in V^{\perp}}$, depends on the value of the inputs at the time the test is activated. This allows testing different propagation paths in the hardware. Next, we address the question of how to choose the subgroup $V^{\perp}$.

Corollary 5.3. *Let be $f \in K_M^n$ and let $V = V(n, \delta)$ a maximal linear code of length n and minimum distance δ, and let be*

$$V^{\perp} = V^{\perp}(n, \delta) = \{\boldsymbol{\tau} | \boldsymbol{\tau} \in \mathcal{C}_2^n, \sum_{s=0}^{n-1} \boldsymbol{\tau}_s i_s = 0, \forall i \in V\} \ .$$

Then,

$$\sum_{\boldsymbol{\tau} \in V^{\perp}} f(\mathbf{x} \oplus \boldsymbol{\tau}) = \frac{1}{|V|} \left(s_0 + \sum_{i \in V, \delta \leq wt(i) \leq M} W_i(\mathbf{x}) s_i \right) \ . \qquad (5.15)$$

For the special case where $\delta = M + 1$ we have Corollary 5.4.

Corollary 5.4. *Universal linear checks constructed with $\delta = M + 1$ are optimal, i.e., it forms the smallest check set and covers all the scenarios including the worst-case scenario defined in Lemma 5.9.*

Proof. $\delta = M + 1$ provides linear checks which are analogous to those in (5.13), where $V^{\perp} = V_{(1)}^{\perp}$.

From Lemma 5.10, $V_{(1)}^{\perp}$ is considered as the *best-case-scenario*, i.e., it is the smallest check set.

From Lemma 5.9, the largest check set is formed when using Theorem 5.22 to construct linear checks for a given polynomial whose variables satisfy $s_t \geq n_t$ for all t.

Since the universal linear checks cover all the scenarios including the worst-case scenario, and the size of its check set is equal to the size of the check set of the best-case scenario, they are optimal. □

5.2.5. Computation Complexity of Universal Linear Checks

The computation complexity $N(\delta)$ of the linear checks as derived from a code V of Hamming distance δ can be measured as the number of

additions required for the computation of the two sums in (5.15). That is,

$$N(\delta) = |V^{\perp}(n,\delta)| + \sum_{i=\delta}^{M} \binom{n}{i} \ . \tag{5.16}$$

The following theorem says that for even values of M, $\delta = M+1$ minimizes the computation complexity $N(\delta)$.

Theorem 5.24 (M even)**.** *Let M be an even integer. Define $p = \lceil log(n) \rceil$. Then $N(M+1) \leq N(M)$ for $M \leq 2^{\frac{p}{2}-1}$.*

Proof. The complexity of optimal linear checks based on a code of distance $M+1 \leq 2^{p/2}$ is smaller than the complexity of linear checks based on a binary BCH code of the same minimal distance [185] :

$$N(M+1) = |V^{\perp}(n,M+1)| \leq 2^{r_{BCH}} \leq 2^{\frac{Mp}{2}} \ .$$

The complexity of linear checks based on a code of distance M is

$$N(M) = |V^{\perp}(n,M)| + \binom{n}{M} \geq \binom{n}{M} = \prod_{i=0}^{M-1} \frac{n-i}{M-i} \geq \left(\frac{n}{M}\right)^{M} \ .$$

Since

$$\frac{n}{M} \geq \frac{2^{p-1}}{2^{\frac{p}{2}-1}} = 2^{\frac{p}{2}} \ ,$$

we have $N(M+1) \leq N(M)$. □

Theorem 5.24 can be generalized as follows.

Theorem 5.25 (M even)**.** *Let M be an even integer. Define $p = \lceil log(n) \rceil$. Then, $N(M+1) \leq N(\delta)$ for $\delta \leq M$ and $M \leq 2^{\frac{p}{2}-1}$.*

Proof. Based on the proof of Theorem 5.24, we have

$$N(\delta) = |V^{\perp}(n,\delta)| + \sum_{i=\delta}^{M} \binom{n}{i} \geq \binom{n}{M} \geq \left(\frac{n}{M}\right)^{M} \geq 2^{\frac{Mp}{2}} \geq N(M+1) \ .$$

□

Table 5.9. Complexity of BCH based linear checks for $M = 6, n = 63$.

δ	V code	$N(\delta)$
7	$[63, 45, 7]$	$262,144 = 2^{18}$
5	$[63, 51, 5]$	$74,978,464 > 2^{26}$
3	$[63, 57, 3]$	$75,609,808 > 2^{26}$
1	$[63, 63, 1]$	$75,611,761 > 2^{26}$
trivial set	-	2^{63}

Example 5.37. *Let $M = 6$ and $n = 63$. The complexity of BCH-code based linear checks $N(\delta)$ for $\delta \leq M + 1$ is given in Table 5.9. It is clear from Table 5.9 that the most efficient way to perform the testing is by using a linear code of minimum distance 7. The testing complexity is 2^{18} which is much smaller than testing based on allying all the $2^n = 2^{63}$ possible input vectors.*

Theorem 5.26 (M odd)**.** *Let M be an odd integer. Define $p = \lceil log(n) \rceil$. Then $N(M+1) \leq N(M)$ for $M \leq 2^{\frac{p}{2}-2}$ and $M > \frac{p}{2}$.*

Proof. The proof is similar to the proof of Theorem 5.24, with a slight change, since $r_{BCH} \leq \frac{(M+1)p}{2}$ for odd values of M.

$$N(M+1) = |V^{\perp}(n, M+1)| \leq 2^{r_{BCH}} \leq 2^{\frac{(M+1)p}{2}}.$$

The complexity of linear checks based on a code of distance M is:

$$N(M) = |V^{\perp}(n, M)| + \binom{n}{M} \geq \binom{n}{M} = \prod_{i=0}^{M-1} \frac{n-i}{M-i} \geq \left(\frac{n}{M}\right)^M .$$

Since

$$\frac{n}{M} \geq \frac{2^{p-1}}{2^{\frac{p}{2}-2}} = 2^{\frac{p}{2}+1} ,$$

we have

$$\left(\frac{n}{M}\right)^M \geq 2^{M(\frac{p}{2}+1)} = 2^{\frac{Mp}{2}+M} \geq 2^{\frac{Mp}{2}+\frac{p}{2}} = 2^{\frac{(M+1)p}{2}} ,$$

and therefore $N(M+1) \leq N(M)$. □

Theorem 5.27 (M odd)**.** *Let M be an odd integer. Define $p = \lceil log(n) \rceil$. Then, $N(M+1) \leq N(\delta)$ for $\delta \leq M$, $M \leq 2^{\frac{p}{2}-2}$ and $M > \frac{p}{2}$.*

Proof. Based on the proof of Theorem 5.26, we have:

$$\begin{aligned} N(\delta) &= |V^{\perp}(n,\delta)| + \sum_{i=\delta}^{M} \binom{n}{i} \geq \binom{n}{M} \geq \left(\frac{n}{M}\right)^{M} \\ &\geq 2^{\frac{(M+1)p}{2}} \geq N(M+1) \ . \end{aligned}$$

□

To summarize, this section deals with functional testing of Boolean systems whose functionality is unknown. The functional testing is performed off-line and is based on applying linear checks. The suggested linear checks are optimal for testing systems that have an acceptable representation as low order polynomials. In contrast to existing methods which require some information about the functionality of the system for constructing the tests, the universal linear checks allow to construct a check set, which does not depend on:

- the actual functionality of the system, and
- the number of input variables and their precision.

Towards Future Technologies

6. Reversible and Quantum Circuits

6.1. The Computational Power of the Square Root of NOT

Steven Vandenbrande Raphaël Van Laer

Alexis De Vos

6.1.1. Reversible Computing Versus Quantum Computing

Reversible computing [78, 337] has become a well-established branch of computer science. One of the problems tackled is the synthesis of an arbitrary (classical) reversible circuit, making use of a particular set of building blocks (a.k.a. gates). Such 'simple' units are e.g., the NOT gate, the controlled NOT gate (a.k.a. the Feynman gate), the multiply-controlled NOT gate (a.k.a. the Toffoli gate), the Fredkin gate, the Peres gate, etc. These building bricks are classical reversible circuits themselves.

Only recently, researchers proposed to add a non-classical, i.e., a quantum, brick to the tool-box for building classical circuits [202, 206, 208, 246, 257, 258, 338, 343]. The prototype quantum gate applied for this purpose is the 'square root of NOT' gate [19, 82, 85, 108]. The big advantage of this gate is the fact that it allows the synthesis of a classical reversible computer with 2-bit building blocks. E.g., the 3-bit Toffoli gate can be replaced by a cascade of five 2-qubit gates (see [337] p. 17 or [78] p. 147). This powerful property is a consequence of a more general 'simulation' theorem by Barenco et al. [19].

A tool-box consisting of a small number of classical gates plus the square root of NOT can not only build classical reversible circuits, but may also be applied to construct quantum computers. The question arises whether an arbitrary quantum circuit can be synthesized (or, at least, be approximated) by means of such a tool-kit.

6.1.2. One-qubit Circuits

We consider the building block, called the 'square root of NOT' and denoted $\sqrt{\mathtt{N}}$, with the symbol:

.

Its matrix representation is the unitary 2×2 matrix:

$$\frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix} .$$

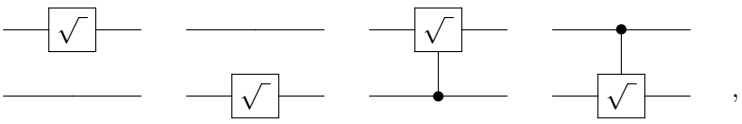
With this single building block, we can only construct four different circuits: the follower, the 'square root of NOT', the NOT, and the 'other square root of NOT'. They form a finite matrix group:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix},$$
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \text{ and } \frac{1}{2}\begin{pmatrix} 1-i & 1+i \\ 1+i & 1-i \end{pmatrix} .$$

The group is isomorphic to the cyclic group $\mathbf{Z}_4$. Two circuits are classical; two are quantum.

6.1.3. Two-qubits Circuits

We consider the following four 2-qubit building blocks:



,

denoted $\sqrt{\mathtt{N}_1}$, $\sqrt{\mathtt{N}_2}$, $\sqrt{\mathtt{C}_1}$, and $\sqrt{\mathtt{C}_2}$, respectively. The former two are 'square roots of NOT'; the latter two are 'controlled square roots of NOT' (or 'square roots of controlled NOT' or 'square roots of Feynman gate'). They are represented by the four 4×4 unitary matrices:

$$\frac{1}{2}\begin{pmatrix} 1+i & 0 & 1-i & 0 \\ 0 & 1+i & 0 & 1-i \\ 1-i & 0 & 1+i & 0 \\ 0 & 1-i & 0 & 1+i \end{pmatrix},$$

$$\frac{1}{2}\begin{pmatrix} 1+i & 1-i & 0 & 0 \\ 1-i & 1+i & 0 & 0 \\ 0 & 0 & 1+i & 1-i \\ 0 & 0 & 1-i & 1+i \end{pmatrix},$$

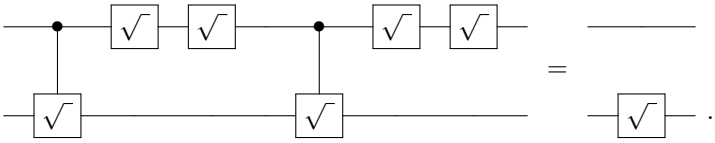
$$\frac{1}{2}\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 1+i & 0 & 1-i \\ 0 & 0 & 2 & 0 \\ 0 & 1-i & 0 & 1+i \end{pmatrix}, \text{ and } \frac{1}{2}\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1+i & 1-i \\ 0 & 0 & 1-i & 1+i \end{pmatrix},$$

respectively.

If we apply only a single building block, we obtain a group of order 4, isomorphic to the group in the previous section, irrespective of the choice of the block. If we apply two different building blocks, the result depends on the choice of the set. With one of the sets $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{N}_2}\}$, $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{C}_1}\}$, or $\{\sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_2}\}$, we generate a finite group of order 16, isomorphic to the direct product $\mathbf{Z}_4 \times \mathbf{Z}_4$. Four circuits are classical; twelve are quantum. All circuits are represented by 4×4 unitary matrices, where all 16 entries are a multiple of $\frac{1}{2}$ plus i times a multiple of $\frac{1}{2}$. With one of the sets $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{C}_2}\}$ or $\{\sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_1}\}$, we generate a group[1] of 384 different circuits. Eight circuits are classical; 376 are quantum. All circuits are represented by 4×4 unitary matrices, where all entries are a multiple of $\frac{1}{4}$ plus i times a multiple of $\frac{1}{4}$. Finally, with the generator set $\{\sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$, we generate a group of infinite order. Six circuits are classical; a countable infinity of circuits is quantum. All circuits are represented by 4×4 unitary matrices, where all entries are a multiple of $\frac{1}{2^k}$ plus i times a multiple of $\frac{1}{2^k}$, where k can be any non-negative integer. Choosing a set of three

[1] The group has GAP identity `[384, 5642]` and structure description: $\mathbf{Z}_4 \times$ **(SL**(2,3)**:Z**$_4$**)** .

building blocks leads to a group of order either 384 or $\aleph_0$. Although the group generated by $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$ has the same order as the group generated by $\{\sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$, both being of order $\aleph_0$, the former is nevertheless 'bigger' than the latter, in the sense that the former is a supergroup of the latter. E.g., it contains all 24 classical reversible 2-bit circuits, instead of only six. Adding the fourth generator $\sqrt{\mathtt{N}_2}$ does not give rise to any 'bigger' infinite group, as $\sqrt{\mathtt{N}_2}$ can be fabricated by $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{C}_2}\}$. Indeed, we have the identity:



We partition each of the generated groups into classes of different levels. A matrix whose entries all are Gaussian rationals (i.e., a rational number plus i times a rational number) can be written as $1/l$ times a matrix whose entries all are Gaussian integers (i.e., an integer number plus i times an integer number), with l a positive integer. If at least one of the integer numerators is not divisible by the denominator l, then the matrix cannot be 'simplified' and l is called the level of the matrix. Table 6.1 shows the number of matrices of different levels in each of the discussed groups.

The last two columns of Table 6.1 consist of a list of never-ending ever-increasing numbers $f(l)$. The last but one column obeys (for $l > 1$) the equation $f(l) = \frac{27}{2}\, l^2$. A proof of this property is provided in [83]. The last column seems to obey (for $l > 4$) the equation:

$$f(l) = \frac{1125}{4}\, l^4 \ .$$

No proof is available (yet).

Each of the four generators are matrices of level 2. The product of a matrix of level $l_1 = 2^{p_1}$ and a matrix of level $l_2 = 2^{p_2}$ is a matrix of level 2^p with $p \leq p_1 + p_2$. Therefore, synthesizing a matrix of level 2^p needs a cascade of at least p building blocks. We call $g_S(n)$ the number of different circuits that can be built with n or less blocks from the given set S. Figure 6.1 shows $g_{\{\sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}}(n)$

Table 6.1. Number of circuits built from different generator sets, with different levels l

	$\{\sqrt{\mathtt{N}_1}\}$ $\{\sqrt{\mathtt{N}_2}\}$ $\{\sqrt{\mathtt{C}_1}\}$ $\{\sqrt{\mathtt{C}_2}\}$	$\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{N}_2}\}$ $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{C}_1}\}$ $\{\sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_2}\}$	$\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{C}_2}\}$ $\{\sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_1}\}$ $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_1}\}$ $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_2}\}$
$l = 1$	2	4	8
$l = 2$	2	12	184
$l = 4$	0	0	192
$l = 8$	0	0	0
...			
total	4	16	384

	$\{\sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$	$\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$ $\{\sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$ $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$
$l = 1$	6	24
$l = 2$	54	2,472
$l = 4$	216	73,920
$l = 8$	864	1,152,000
...		
total	$\aleph_0$	$\aleph_0$

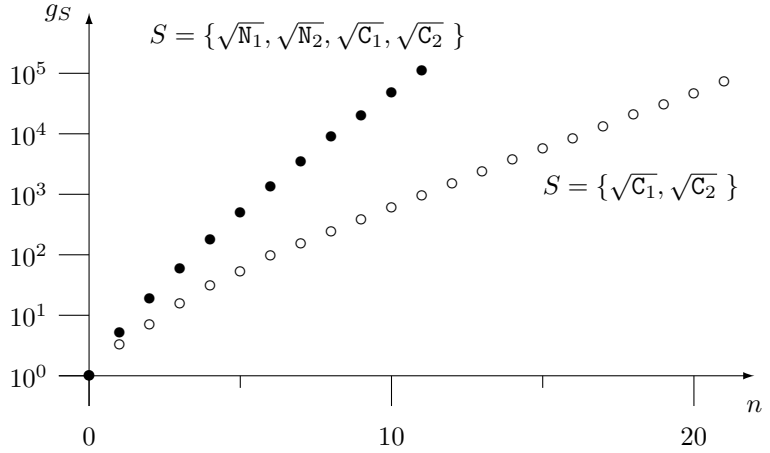


Figure 6.1. Number $g_S(n)$ of different circuits, built by a cascade of n or less building blocks from a given set S.

and $g_{\{\sqrt{\mathtt{N}_1},\sqrt{\mathtt{N}_2},\sqrt{\mathtt{C}_1},\sqrt{\mathtt{C}_2}\}}(n)$. In general, a curve $g_S(n)$ can grow either polynomially or exponentially [336] or even can have intermediate growth [335]. We see how, in both cases here, the growth rate is exponential. We indeed have:

$$g_S(n) \approx a\, b^n \ ,$$

with both a and b positive constants. For $S = \{\sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$, we have $a \approx 8$ and $b \approx 1.5$; for $S = \{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$, we have $a \approx 12$ and $b \approx 2.2$. As expected, the numbers b are smaller than the cardinality of the corresponding sets S (i.e., 2 and 4, respectively).

We now call $g_S^p(n)$ the number of circuits of level 2^p that can be built with n or less blocks from the set S. We thus have:

$$g_S(n) = \sum_{p=0}^{\infty} g_S^p(n) = \sum_{p=0}^{n} g_S^p(n) \ .$$

As an example, Figure 6.2 shows $g^p_{\{\sqrt{\mathtt{C}_1},\sqrt{\mathtt{C}_2}\}}(n)$, for p ranging from 0 to 3, i.e., for levels 1, 2, 4, and 8. We see how $g_S^p(n)$ first is zero, then grows monotonically with increasing n, and finally saturates at

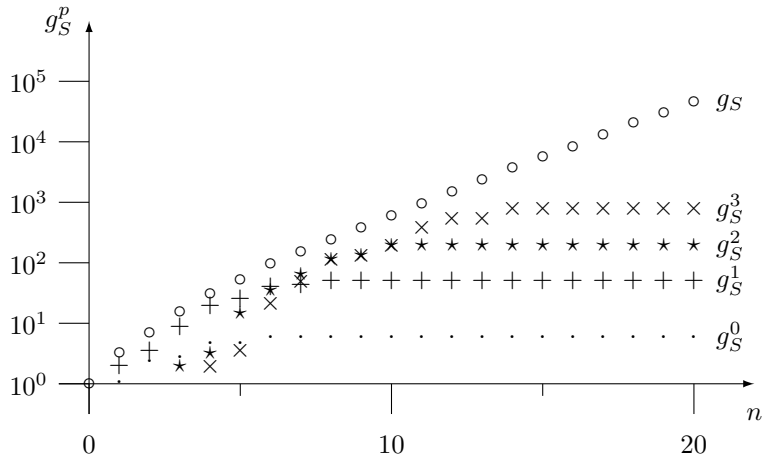


Figure 6.2. Number $g_S^p(n)$ of different circuits, built by a cascade of n or less building blocks from the set $S = \{\sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$, with different levels $l = 2^p$.

a value equal to the corresponding number in the last but one column of Table 6.1.

The reader easily verifies that each of the four members of the matrix set $\{\sqrt{\mathtt{N}_1}, \sqrt{\mathtt{N}_2}, \sqrt{\mathtt{C}_1}, \sqrt{\mathtt{C}_2}\}$ fulfills the following three properties:

P1 the matrix is unitary;

P2 each matrix entry is a dyadic Gaussian; and

P3 each line sum equals to 1.

Here, a dyadic Gaussian is a Gaussian rational with denominator of the form 2^p and a line sum is either a row sum or a column sum. The reader may also easily verify that the product of two matrices obeying **P1** also obeys **P1**, and similar for **P2**, and similar for **P3**. Thus, we have easily proven the following theorem: all circuits generated by the above set will have the three properties. The difficult part is to prove the inverse theorem: each 4×4 matrix fulfilling the three conditions **P1**, **P2**, and **P3**, can be generated by the four generators

$\sqrt{\mathtt{N}_1}$, $\sqrt{\mathtt{N}_2}$, $\sqrt{\mathtt{C}_1}$, and $\sqrt{\mathtt{C}_2}$ (and thus by the three generators $\sqrt{\mathtt{N}_1}$, $\sqrt{\mathtt{C}_1}$, and $\sqrt{\mathtt{C}_2}$). Proof is provided by Reference [83]. The matrices obeying the three properties form a group (which we will denote **G**), i.e., a discrete subgroup of the 16-dimensional Lie group U(4), consisting of all unitary 4×4 matrices.

The uncountably infinite group U(4) represents all 2-qubit quantum computers. The countably infinite group **G** represents a subset of these computers. The question arises whether or not the points representing the members of **G** are densely distributed on the surface of the 16-dimensional manifold representing U(4). That would imply that any member of U(4) can be approximated by a member of **G**, with unlimited accuracy, just like any real number can be approximated infinitely well by a dyadic rational, and just like any point on the s-dimensional hypersphere ($s > 4$) of radius 1 can be approximated infinitely well by a point with s dyadic rational coordinates (a consequence of Pommerenke's theorem [236]). The answer to the question is definitely 'no', and that for two reasons:

- Condition **P3** restricts the points representing **G** to the nine-dimensional subspace V of U(4), representing the 4×4 unitary matrices with constant line sum equal to 1. In Reference [81] is demonstrated that the subgroup V of U(4) is isomorphic to U(3).

- Conditions **P1** and **P2** together lead to the conclusion that all members of **G** have a determinant from the set $\{ 1, i, -1, -i \}$. Therefore, condition **P2** restricts the points representing **G** to a 15-dimensional subspace of U(4), isomorphic to a semi-direct product SU(4)$:\mathbf{Z}_4$ of the special unitary group SU(4) and the cyclic group $\mathbf{Z}_4$.

The intersection of the above two subspaces of U(4) is an eight-dimensional non-connected Lie group with four components. See Figure 6.3 in which the stars ($\star$) represent the $4! = 24$ members of the symmetric group $\mathbf{S}_4$ (i.e., the classical reversible computers); the dots ($\bullet$) represent the $\aleph_0$ members of the group **G**. The factor x is a constant matrix, chosen from U(4) such that $\det(x) = i$.

The open question remains whether or not the points representing

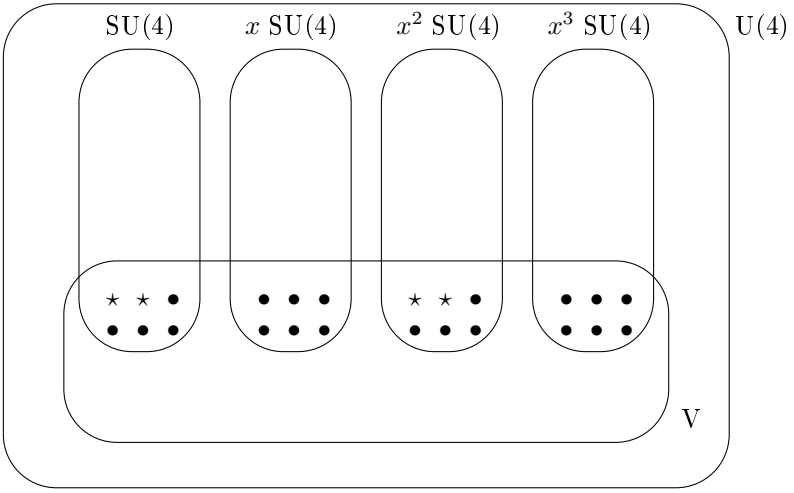


Figure 6.3. The Lie group U(4) (i.e., the quantum computers).

the members of **G** (i.e., the dots and stars in Figure 6.3) are densely distributed on this surface.

6.1.4. Many-qubits Circuits

Quantum circuits acting on w qubits are represented by unitary matrices of size $2^w \times 2^w$. Applying the short-hand notation m for 2^w, we can say they form the Lie group U(m). The latter is represented by a connected m^2-dimensional manifold. The classical reversible circuits acting on w bits form a finite subgroup, isomorphic to the symmetric group $\mathbf{S}_m$ of order $m!$. Again, the square roots of NOT, the singly-controlled square roots of NOT, and the multiply-controlled square roots of NOT obey the properties **P1**, **P2**, and **P3**. Therefore they generate an infinite but discrete group, i.e., $\aleph_0$ points located on the m^2-dimensional Lie group U(m). If $w > 2$, we have to distinguish three cases:

- If we allow controlled square roots with $w-1$ controlling qubits as a building block, then matrices can be generated with a de-

terminant equal to any member of the set $\{ 1, i, -1, -i \}$.

- If we do not allow controlled square roots with $w-1$ controlling qubits, but allow controlled square roots with $w-2$ controlling qubits, then only matrices can be generated with a determinant from the set $\{ 1, -1 \}$.

- If we only allow controlled square roots with less than $w-2$ controlling qubits, then only matrices with determinant 1 can be generated.

This leads to:

- non-connected groups with four components,

- non-connected groups with two components, and

- connected groups, respectively.

Like in Subsection 6.1.3, the condition **P3** restricts the points to a Lie subspace V of U(m). This subspace [79] represents the $m \times m$ unitary matrices with constant line sum equal to 1. In [80] is demonstrated that the subgroup V of U(m) is isomorphic to U($m-1$) and thus forms a $(m-1)^2$-dimensional Lie group.

6.1.5. Increased Computational Power

We conclude that the introduction of (controlled) square roots of `NOT` as circuit building blocks leads to rich mathematics. The resulting computers have much more computational power than the classical computers, but much less computational possibilities than the quantum computers.

6.2. Toffoli Gates with Multiple Mixed Control Signals and no Ancillary Lines

Claudio Moraga

6.2.1. On the Evolution of Toffoli Gates Until Present Days

Toffoli gates [318] are basic for the realization of reversible and, especially, quantum computing circuits. Much work has been done to extend the original proposal, comprising two control signals which are active when they take the value 1, to realize Toffoli gates with $n > 2$ control signals. The present section discusses efficient realizations of Toffoli gates with two, three, or more control signals, which may be active with value 1, or value 0, without requiring additional inverters or ancillary lines. The realizations have a quantum cost of $2^{n+1} - 3$ and are efficient in the sense that they extend proposals for Toffoli gates with n control inputs, all of them being active when having the value 1, to Toffoli gates with n mixed control signals.

In a binary Boolean context, a reversible gate is an elementary circuit component that realizes a bijection. As a consequence, it has the same number of inputs and outputs. A circuit is reversible if it has a feed forward realization with fan-out free reversible gates. Quantum Mechanics based gates and circuits are reversible. A Toffoli gate represents a reversible covering of the binary `AND` gate, (which is not reversible). An efficient abstract realization as a quantum gate was introduced in the seminal paper referred to as Barenco et al. [19], showing that a Toffoli gate may be realized with a quantum cost of 5. Figure 6.4 shows the unitary matrix representation of a Toffoli gate, the accepted symbol and the quantum realization disclosed in [19], where new unitary matrices V and V^+ are introduced, such that $V^2 =$ `NOT` and V^+ represents the adjoint of V.

In recent years, some design efforts have been focused in the generalization of Toffoli gates to accept both 1-valued and 0-valued control

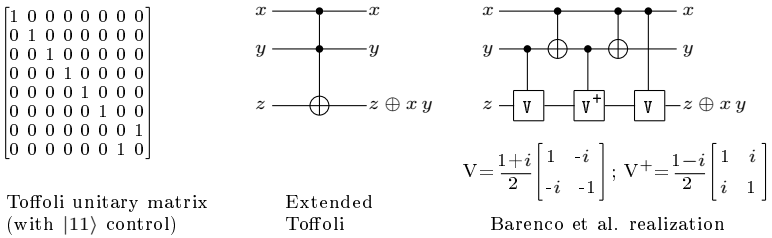
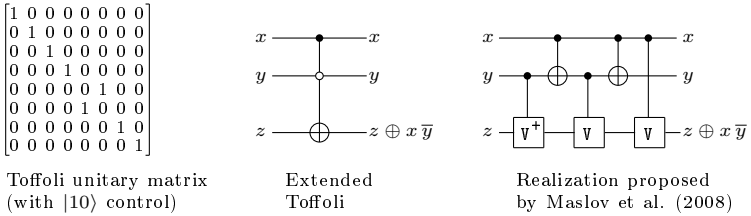


Figure 6.4. Unitary matrix, symbol, and quantum realization of the Toffoli gate.

signals (see e.g., [182], [188], [219], [218], [90]). White dots have been used to identify control signals that are active when their value is equal to 0. A "Barenco-type" realization of a Toffoli gate with two control inputs, where one of them is negated, was introduced in [189] and extended in [219] to the case where the other control input is negated, (without swapping the inputs), and when both control inputs are negated (without requiring additional inverters). For the sake of completeness these realizations are reproduced below in Figures 6.5 and 6.6.



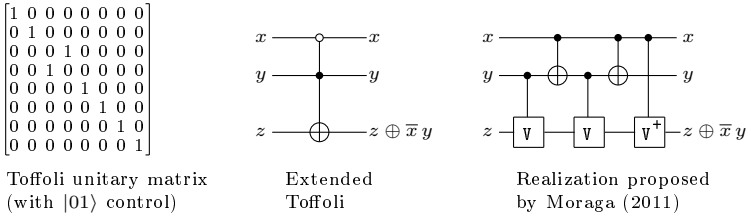


Figure 6.5. Toffoli gates with one negated control input.

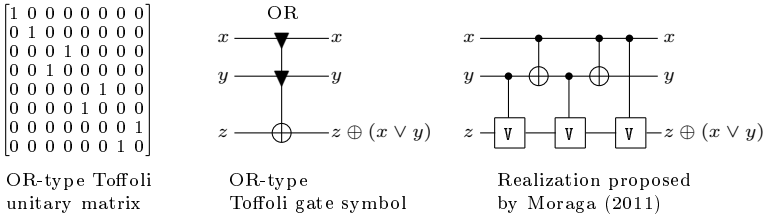


Figure 6.6. OR-type Toffoli gate.

In the Barenco-type realizations of Figure 6.5 it may be observed that the 0-valued control signals act by inhibiting a V or V^+ elementary gate. Accordingly, if both control signals are 0-valued, all elementary gates would be inhibited and no action would follow. The desired output $z \oplus \overline{x}\ \overline{y}$ would not be reached.

However, the De Morgan Laws allow expressing the desired output as $z \oplus \overline{x \vee y}$. The question can therefore be restated, whether an `OR/NOR`-type of Toffoli gate is possible and whether an efficient quantum realization for it may be obtained. Figure 6.6 shows an `OR`-type Toffoli gate [219] and its Barenco-type realization. If the target line is free, i.e., z may be chosen to be 0 or 1, then with $z = 1$ a `NOR` gate is obtained, and since $\overline{x \vee y} = \overline{x}\ \overline{y}$ then a Toffoli gate with both control inputs 0-valued is possible.

6.2.2. Toffoli Gates with Three Mixed Control Signals

Toffoli gates with three control inputs are important for two reasons. On the one hand, they provide a simple testbed for decomposition strategies leading to circuits (mainly cascades) of Toffoli gates with two inputs, which are recursively scalable to Toffoli gates with a larger number of control inputs. In this respect, see e.g., [188] and the references provided there. On the other hand, Barenco et al. [19] also provided an efficient quantum realization for this kind of Toffoli gates, which however until only recently [258] did not receive much attention in the reversible/quantum community. The realization makes use of

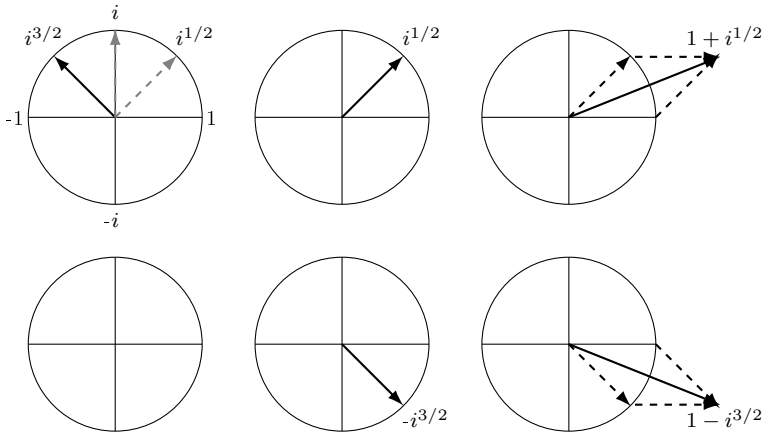


Figure 6.7. Analysis of the first element of W and W^{-1}.

elementary matrices W and W^{-1}, where $W^4 = \texttt{NOT}$:

$$W = \frac{1}{2}\begin{bmatrix} 1+i^{1/2} & 1-i^{1/2} \\ 1-i^{1/2} & 1+i^{1/2} \end{bmatrix}, \text{and}$$

$$W^{-1} = \frac{1}{2}\begin{bmatrix} 1-i^{3/2} & 1+i^{3/2} \\ 1+i^{3/2} & 1-i^{3/2} \end{bmatrix}.$$

As it may be seen in Figure 6.7, the complex conjugate of $1+i^{1/2}$ equals $1-i^{3/2}$, and similarly, the complex conjugate of $1-i^{1/2}$ equals $1+i^{3/2}$. Therefore, $W^{-1} = W^* = W^+$ meaning that W is unitary. In what follows, we will show that the former circuits may be extended to provide efficient quantum realization of Toffoli gates with all possible combinations of three mixed (1-valued and 0-valued) control inputs. Figure 6.8 shows an "abstract" version of the circuit in Section 7 of [19], where $U_1, U_2, \ldots, U_7$ are arbitrary unitary matrices. The local controlling expressions are shown associated to the corresponding unitary matrices. When a local controlling expression takes the value 1, the controlled U-gate is activated, otherwise it is inhibited and behaves as the identity. Table 6.2 summarizes the values of the controlling expressions depending on the values of the control inputs. As mentioned in [19], the controlling expressions are ordered

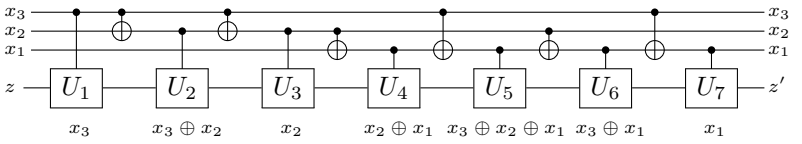


Figure 6.8. Abstract representation of a conjunction of three control variables based on [19] .

in a Gray code mode: every controlling expression has either one additional control input or deletes one from the preceding expression. This guarantees that there are no repeated controlling expressions. The "empty controlling expression", corresponding to the 000 of the Gray code, does not make a controlling sense and it is not included. This is why Table 6.2 has only 7 columns U_i. It is easy to see that the rows of Table 6.2 are different words of a 4-out-of-7 code: in each row there are four values 1 and three values 0. (A Gray code is balanced, but the exclusion of the "empty controlling expression" explains the missing value 0).

Remark 6.2. *By inspection of Table 6.2, for any control vector, if the four columns are considered, where the control vector produces an entry of 1, then all other rows contain two values* 1 *and two values* 0. *Similarly, if the three columns are considered, where the control vector produces an entry of* 0, *then all other rows contain two values* 1 *and one value* 0.

The former analysis and Remark 6.2 lead to the following circuit specification. For a given input control vector, read the corresponding row of Table 6.2. Replace every activated U-gate with a W-gate and replace the inhibited gates with W^+-gates. Since the inhibited gates behave as identity, the four activated gates will generate $W^4 = \texttt{NOT}$.

If the same circuit is driven with a different control vector, three gates will be inhibited (contributing the identity) and from the four activated gates, according to Remark 6.2, two will be W and the other two will be W^+. The cascade of these four gates (in any order) will also produce an identity, i.e., the circuit will produce NOT only for the selected control vector, and the identity for any other control vector.

Table 6.2. Relationship between input control values x_0, x_1, x_2 and activated/inhibited U-gates

	U_1	U_2	U_3	U_4	U_5	U_6	U_7
$x_3x_2x_1$	x_3	$x_3 \oplus x_2$	x_2	$x_2 \oplus x_1$	$x_3 \oplus x_2 \oplus x_1$	$x_3 \oplus x_1$	x_1
001	0	0	0	1	1	1	1
010	0	1	1	1	1	0	0
011	0	1	1	0	0	1	1
100	1	1	0	0	1	1	0
101	1	1	0	1	0	0	1
110	1	0	1	1	0	1	0
111	1	0	1	0	1	0	1

Not considered in the former analysis is the case of a $|000\rangle$ input control vector, since if all gates are inhibited, the whole circuit behaves as the identity. As discussed earlier for the case of two input variables, this has an indirect solution: if all U-gates are chosen to be W, every input vector different from $|000\rangle$ will activate 4 W's and will inhibit the other three, i.e., every non-$|000\rangle$ input control vector will produce `NOT`, meanwhile the $|000\rangle$ control vector will produce the identity. This corresponds to an `OR`-type of Toffoli gate with three control inputs. If the target line is driven with 1, the circuit will behave as `NOR`, and with the De Morgan Laws, this is equivalent to the conjunction of all negated control inputs.

The requirement of driving the target line with a 1 (or equivalently, adding a local inverter) is not severe if the gate is used in a circuit and shares the target line with it, because then the target line of the circuit would be driven by a 1. If the Toffoli gate does not share its target line with that of the circuit, and its output signal is used to control one or more additional gates before becoming a garbage-output, no local inverter is needed: the controlled gates should receive the controlling signal with a white dot. However, if the Toffoli gate does not share its target line with that of the circuit and its output signal is used to control one or more additional gates before becoming a significant output, then an additional inverter is required. This completes the

design of Toffoli gates with three control inputs and any combination of negated and non-negated inputs (including the case of all inputs negated) for a realization with a quantum cost of 13, neither requiring additional inverters (except in the special case mentioned earlier), nor ancillary lines.

6.2.3. Efficient Toffoli Gates with $n > 3$ Mixed Control Inputs

Assume that k variables are considered to build a Gray code starting with the word $00\ldots0$. This code will have 2^k words and since to move from one word to its neighbour, one variable must be complemented, 2^k-1 negations will be needed for the whole code. If now an additional variable is considered, a new Gray code may be generated by extending the former code with a mirror image of itself, (the last word of this extended code will be $00\ldots0$), and associating to the first part the $k+1^{st}$ variable with value 0, and to the second part, the $k+1^{st}$ variable with value 1. Notice that the first word of the resulting new Gray code will again be $00\ldots00$ meanwhile the last one will be $00\ldots01$.

Let be $n = k + 1$. The new Gray code will have 2^n words and $2^n - 1$ negations. If the words of the Gray code are used to structure controlling expressions of an extended Toffoli circuit, as a linear combination of the variables with coefficients from the Gray words, the first word $(00\ldots00)$ should be deleted, since it would generate an "empty" controlling expression. Therefore the circuit would have $2^n - 1$ controlling expressions associated to 2 qubit unitary matrices, and $2^n - 2$ negations leading to a total quantum cost of $2^{n+1} - 3$ and requiring neither additional inverters nor ancillary lines. This is a statement of the corresponding quantum cost, without claiming that this may be the minimum possible cost. Notice that this analysis corresponds to the induction step of an induction proof starting with a Toffoli gate with 2 control inputs and a quantum cost of 5.

Example 6.38. *The design of an extended Toffoli gate with 4 mixed control inputs can be realized based on the matrix X. The symbol X has been chosen, continuing the sequence initiated by V, followed by W. Since in the design of reversible circuits, the Pauli matrices are*

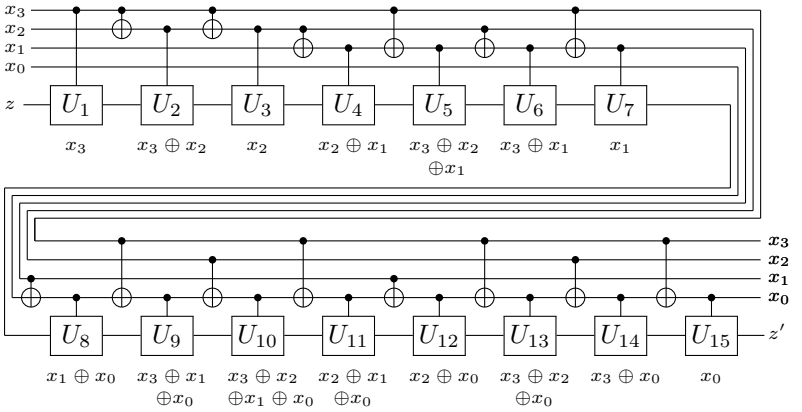


Figure 6.9. Extended Toffoli circuit with 4 mixed control units and without ancillary lines.

very rarely mentioned, there should be no confusion. The matrices X and X^{-1} are defined as follows:

$$X = \frac{1}{2}\begin{bmatrix} 1+i^{1/4} & 1-i^{1/4} \\ 1-i^{1/4} & 1+i^{1/4} \end{bmatrix}, \text{ and}$$

$$X^{-1} = \frac{1}{2}\begin{bmatrix} 1-i^{-1/4} & 1+i^{-1/4} \\ 1+i^{-1/4} & 1-i^{-1/4} \end{bmatrix}.$$

A similar analysis to the case of W shows that $X^* = X^{-1} = X^+$; therefore X is unitary. Furthermore, $X^8 =$ NOT. If under some quantum technology the matrix X is realizable, then the circuit shown in Figure 6.9 implements an extended Toffoli gate with 4 mixed control inputs.

Table 6.3 corresponding to the circuit of Figure 6.9, is built based on Table 6.2 and on the method explained above to obtain a Gray code from an existing one, with one additional variable (in this case chosen to be x_0). The entries for the columns U_1 to U_7 are taken from Table 6.2, duplicating each row, (since the behavior of these gates depend on x_3, x_2 and x_1, but not on x_0), and adding a row of 0's for the 000 control inputs. For any column U_{15-r}, $1 < r < 8$, the entry at a given row, is obtained as the addition modulo 2 of the entry at the column

Table 6.3. Relationship between input control values x_0, x_1, x_2, x_3 and activated/inhibited U-gates

	U_1	U_2	U_3	U_4	U_5	U_6	U_7
$x_3x_2x_1x_0$	x_3	x_3 x_2	x_2	x_2 x_1	x_3 x_2 x_1	x_3 x_1	x_1
0 0 0 1	0	0	0	0	0	0	0
0 0 1 0	0	0	0	1	1	1	1
0 0 1 1	0	0	0	1	1	1	1
0 1 0 0	0	1	1	1	1	0	0
0 1 0 1	0	1	1	1	1	0	0
0 1 1 0	0	1	1	0	0	1	1
0 1 1 1	0	1	1	0	0	1	1
1 0 0 0	1	1	0	0	1	1	0
1 0 0 1	1	1	0	0	1	1	0
1 0 1 0	1	1	0	1	0	0	1
1 0 1 1	1	1	0	1	0	0	1
1 1 0 0	1	0	1	1	0	1	0
1 1 0 1	1	0	1	1	0	1	0
1 1 1 0	1	0	1	0	1	0	1
1 1 1 1	1	0	1	0	1	0	1

	U_8	U_9	U_{10}	U_{11}	U_{12}	U_{13}	U_{14}	U_{15}
$x_3x_2x_1x_0$	x_1 x_0	x_3 x_1 x_0	x_3 x_2 x_1 x_0	x_2 x_1 x_0	x_2 x_0	x_3 x_2 x_0	x_3 x_0	x_0
0 0 0 1	1	1	1	1	1	1	1	1
0 0 1 0	1	1	1	1	0	0	0	0
0 0 1 1	0	0	0	0	1	1	1	1
0 1 0 0	0	0	1	1	1	1	0	0
0 1 0 1	1	1	0	0	0	0	1	1
0 1 1 0	1	1	0	0	1	1	0	0
0 1 1 1	0	0	1	1	0	0	1	1
1 0 0 0	0	1	1	0	0	1	1	0
1 0 0 1	1	0	0	1	1	0	0	1
1 0 1 0	1	0	0	1	0	1	1	0
1 0 1 1	0	1	1	0	1	0	0	1
1 1 0 0	0	1	0	1	1	0	1	0
1 1 0 1	1	0	1	0	0	1	0	1
1 1 1 0	1	0	1	0	1	0	1	0
1 1 1 1	0	1	0	1	0	1	0	1

U_r plus x_0. The entries of the column U_{15} correspond to the value of x_0. Below every U_i there is a list of the corresponding controlling variables.

For any given control vector, in the corresponding row of Tables 6.3, replace the activated U-gates with X matrices and the inhibited U-gates, with X^+ matrices. The 7 inhibited gates behave as identity and the 8 activated X gates build $X^8 = \texttt{NOT}$. Extending by induction Remark 6.2 done for Table 6.2 to the case of 4 control inputs, it may be concluded that any other input control vector applied to the circuit will activate a balanced number (4) of X and of X^+ gates building an identity transfer function. Finally, if all U-matrices are set to X, the circuit will behave as an `OR`-Toffoli gate. In this case, with the De Morgan Law it may be seen that if z is set to 1, the circuit will behave as the conjunction of all negated control inputs.

As it may be obtained from Figure 6.8, the circuit has a quantum cost of 29 and requires no ancillary line. Since in this example $n = 4$, this corresponds to $2^{n+1} - 3$ as stated at the beginning of this subsection. The above numerical results are basically the same as presented in [19], but they are here generalized to the realization of Toffoli gates with mixed control signals without requiring additional inverters (under the conditions discussed in the former subsection). A given architecture designed for one particular setting (all control inputs equal to 1) has been extended, preserving the quantum cost, to work with $2^n - 1$ additional control settings. In this sense the present proposal is efficient.

6.3. Reducing Quantum Cost of Pairs of Multi-Control Toffoli Gates

Marek Szyprowski Pawel Kerntopf

6.3.1. Reversible Circuits Synthesis

Many methods of reversible circuit synthesis have been developed [254]. Most of them construct circuits from multi-control Toffoli (in short `MCT`) gates, which are later decomposed into elementary quantum gates. A number of papers have recently been published on constructing such decompositions for any size of the reversible gates [203, 207, 208, 258]. An interesting question is how to construct the quantum circuit and what savings can be achieved by constructing circuit from elementary quantum gates. Such research have been already performed in [271]. However, its authors apply quantum decomposition directly only to small (3-bit) Toffoli gates, while larger Toffoli gates are first decomposed into equivalent circuits built from small Toffoli gates.

One of the well known cases of savings in quantum cost of the reversible circuit is the Peres gate [187], which reduces quantum cost from 6 (for a pair of Toffoli and controlled `NOT` gates) to 4 units (by constructing it directly from quantum gates), see Figure 6.10. Our main motivation for this work was to check if there exist pairs of 4-bit reversible gates being analogues to the 3-bit Peres gate. According to our knowledge nobody has published any results on this open problem. Such pairs, if they exist, might also contribute to the reversible circuit synthesis in the same way as Peres gate did [187].

In the presented approach the whole 4-bit reversible circuit is constructed directly from the quantum gates. In this way we have found new quantum decompositions for some pairs of `MCT` gates. Those pairs lead to significant savings in the number of elementary quantum gates required for their realizations. Our results show that the case of the Peres gate can be extended also to pairs of 3-bit and 4-bit Toffoli

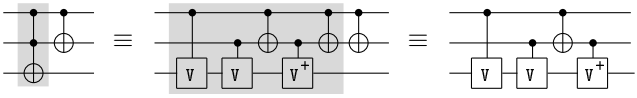


Figure 6.10. NCV quantum circuit for 3×3 Peres gate.

gates, which could be useful for improving reversible circuit synthesis and reducing quantum cost of reversible designs.

6.3.2. Background

Definition 6.32. *A completely specified Boolean function with n inputs and n outputs (referred to as $n \times n$ function) is called reversible if it maps each input assignment into a unique output assignment.*

Definition 6.33. *An n-input n-output ($n \times n$) gate (or circuit) is reversible if it realizes an $n \times n$ reversible function.*

In a reversible circuit fan-out of each gate output is always equal to 1. As a consequence $n \times n$ reversible circuits can be only built as a cascade of $k \times k$ reversible gates ($k \leq n$).

Definition 6.34. *A set of gates that can be used to build circuits is called a gate library.*

Many libraries of reversible gates have been examined in the literature. The most commonly used is NCT library which consists of multi-control Toffoli gates, which are defined below.

Definition 6.35. *Let $\oplus$ denote XOR operation. An $n \times n$ multi-control Toffoli gate (in short MCT gate) performs the following operation:*

$$(x_1, \ldots, x_n) \rightarrow \begin{cases} (x_1, \ldots, x_{n-1}, x_1, \ldots, x_{n-1} \oplus x_n), & \text{for } n \geq 2 \ , \\ (1 \oplus x_1), & \text{for } n = 1 \ . \end{cases} \tag{6.1}$$

1×1 `MCT`, 2×2 `MCT` and 3×3 `MCT` gates are called `NOT`, `CNOT` and *Toffoli* gates, respectively.

Every `MCT` gate inverts one input if and only if all others inputs are equal to 1, passing these inputs unchanged to corresponding outputs. Signals which are passed from input to corresponding output of the gate without a change are called *control* lines. The signal which can be modified by the gate is called *target*. Each `MCT` gate is self-inverse, i.e., equal to its own inverse.

The operation of each reversible gate can also be considered in the quantum gate level. It is well known that such $n \times n$ gates can be represented by a square matrix of dimension 2^n. For example, the simplest `MCT` gate, `NOT`, is described by the matrix:

$$\mathtt{N} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} . \tag{6.2}$$

It can be shown [258] that:

$$\mathtt{N}^{1/n} = \frac{1}{2} \begin{pmatrix} 1+i^{2/n} & 1-i^{2/n} \\ 1-i^{2/n} & 1+i^{2/n} \end{pmatrix} . \tag{6.3}$$

Let $\mathtt{V} = \mathtt{N}^{1/2}$ and $\mathtt{W} = \mathtt{N}^{1/4}$. Then, let $\mathtt{V}^+$ and $\mathtt{W}^+$ be a complex conjugate transpose of `V` and `W`, respectively. By transforming the above equations, one can find that [258]:

$$\mathtt{V} = \frac{1+i}{2} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} , \tag{6.4}$$

$$\mathtt{V}^+ = \mathtt{V}^{-1} = \frac{1-i}{2} \begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix} , \tag{6.5}$$

$$\mathtt{W} = \frac{1}{2} \begin{pmatrix} 1+\sqrt{i} & 1-\sqrt{i} \\ 1-\sqrt{i} & 1+\sqrt{i} \end{pmatrix} , \tag{6.6}$$

$$\mathtt{W}^+ = \mathtt{W}^{-1} = \frac{1}{2} \begin{pmatrix} 1-i\sqrt{i} & 1+i\sqrt{i} \\ 1+i\sqrt{i} & 1-i\sqrt{i} \end{pmatrix} . \tag{6.7}$$

$\mathtt{V} \circ \mathtt{V}^+ = Id$, $\mathtt{V} \circ \mathtt{V} = \mathtt{N}$, $\mathtt{V}^+ \circ \mathtt{V}^+ = \mathtt{N}$, as well as $\mathtt{W} \circ \mathtt{W}^+ = Id$, $\mathtt{W} \circ \mathtt{W} = \mathtt{V}$ and $\mathtt{W}^+ \circ \mathtt{W}^+ = \mathtt{V}^+$, where $\circ$ denotes matrix multiplication and Id denotes the identity matrix.

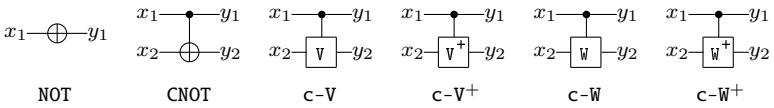


Figure 6.11. Graphical symbols for NCVW gates.

Definition 6.36. *A controlled-*V *(-*V$^+$*, -*W*, -*W$^+$*) gate is an elementary quantum gate which applies the transformation matrix* V *(-*V$^+$*, -*W*, -*W$^+$*), respectively, to the target line iff the value of the control line is 1.*

The gates *controlled-*V/V$^+$ and *controlled-*W/W$^+$ are also called *controlled-square-root-of-*NOT and *controlled-fourth-root-of-*NOT gates, respectively. It can be easily noticed that *controlled-*V and *controlled-*V$^+$ as well as *controlled-*W and *controlled-*W$^+$ gates are inverse of each other.

In the literature many libraries of quantum elementary gates have been considered. A subset of the gates introduced in [19] and recently analyzed in [258] consists of the above defined NOT, CNOT, *controlled-*V/V$^+$ and *controlled-*W/W$^+$ gates. Graphical symbols for those gates are depicted in Figure 6.11. In this chapter it will be called NCVW library. The first quantum gate library considered in the literature was NCV library, which limits quantum gates only to *controlled-*V/V$^+$ gates.

6.3.3. NCVW Quantum Circuits

Quantum operations performed by NCV and NCVW gates can be considered also as rotations around the x-axis of the Bloch sphere: NOT gate performs π rotation, V/V$^+$ performs $\pi/2$ rotation and W/W$^+$ corresponds to $\pi/4$ rotation, see diagram in Figure 6.12.

Reversible and quantum circuits are built as a cascades of gates. It is a common practice to limit states of control lines of controlled gates to a finite set of values. We assume that controlled gates can be used only iff the control line state is in the Boolean domain (0 or 1). This way quantum entanglement in the circuit can be avoided [258]. With such assumption we can introduce a simplified, eight-value algebra

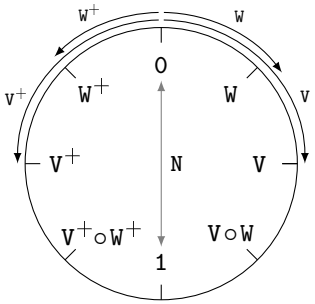


Figure 6.12. Quantum states and operations defined by N, V/V+ and W/W+ matrices on the cross-section of the Bloch sphere.

for describing the state of lines in the NCVW circuit. The transitions between the states of the lines after applying NCVW operations are shown in Table 6.4. Similar four-value logic can be introduced for NCV circuits.

Table 6.4. Eight-value logic for NCVW quantum operations

state	N	V	V+	W	W+
0	1	V	V+	W	W+
W	V+∘W+	V∘W	W+	V	0
V	V+	1	0	V∘W	W
V∘W	W+	V+∘W+	W	1	V
1	0	V+	V	V+∘W+	V∘W
V+∘W+	W	W+	V∘W	V+	1
V+	V	0	1	W+	V+∘W+
W+	V∘W	W	V+∘W+	0	V+

Definition 6.37. *Quantum cost of a reversible circuit is the number of elementary quantum gates required to build this circuit using the best known quantum mapping procedure for the reversible gates.*

Usually, each MCT gate in a reversible circuit is decomposed into elementary quantum gates and then the value of the quantum cost is calculated [203, 207, 208, 258]. It is known that some pairs of 3×3 Toffoli and 2×2 CNOT gates can be directly built from elementary quantum gates with lower quantum cost value than the sum of the

quantum costs of two single gates. An example of such a case is the Peres gate [187], see Figure 6.10.

6.3.4. Optimal Circuit Synthesis

In 2010 Golubitsky, Falconer and Maslov [120, 121] presented the implementation of an algorithm for finding a gate count optimal reversible circuit for any 4×4 reversible function. The algorithm uses several sophisticated tricks to reduce the memory complexity of calculations to fit into the resources available on nowadays computers. The main idea behind this algorithm is to use hash-tables to store a compact representation of the reversible functions for the optimal circuits up to 9 gates. Then, by combining the information about the circuits up to n gates, one can construct the optimal circuit for the function requiring up to $2 * n$ gates. This approach has been further extended in [311–314] to find circuits with reduced quantum cost and circuits built from generalized `NCT` gate libraries.

We adapted this methodology to quantum circuits characterized by multi-value logic. Our goal was to construct quantum circuit at least for a single 4-bit Toffoli gate. In case of `NCVW` gate library it is sufficient to construct 4×4 `NCVW` quantum circuit. However, in case of the `NCV` library, an additional ancillary line is required to construct such circuits, what leads to 5×5 `NCV` circuits.

We limit the states of all input signals to the Boolean domain, what gives 2^n Boolean states for an $n \times n$ circuit. `NCVW` quantum circuits require 3 bits to encode eight-value state of each line, `NCV` circuits - only 2 bits. This means that 4×4 `NCVW` quantum circuits require $4 * 3 * 2^4 = 192$ bits for storing the state of all lines for all input signal values. On the other hand, 5×5 `NCV` circuits require $5 * 2 * 2^5 = 320$ bits. In the database construction process one should also check the exact values of control lines before using the controlled gates. If the state of a control line might not be in the Boolean domain, then the given controlled gate need to be skipped in the given step of the algorithm. The most significant difference from the original [120, 121] approach is the method of handling the inversion on the function. By assuming that the control lines are allowed to have only Boolean values we break

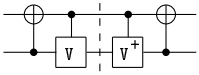


Figure 6.13. Identity circuit whose right and left subcircuits are inverse of each other.

the symmetry in the original approach. This means that it might be not possible to construct inverse circuits for some of the functions stored in the database because of the quantum entanglement, which might occur if one simply reverse the order of gates in the circuit and substitute each gate by its inversion.

Figure 6.13 shows an example of identity circuit whose right and left halves (subcircuits) are inverse of each other. However, the right subcircuit cannot be considered as a valid stand-alone quantum circuit in the presented approach, because the control line for the `CNOT` gate might be not in the Boolean domain because the circuit is entangled.

However, it is possible to compose two quantum functions if the result of the composition fits into the Boolean domain. This limits the algorithm only to reversible Boolean functions, but resolves the need for constructing the inverse function. From practical point of view this limitation is not a big problem, as the main goal for this algorithm is to find optimal quantum circuits useful in reversible (Boolean) circuit synthesis.

6.3.5. Experimental Results and Applications

We have constructed optimal quantum circuit databases up to 9 gates for 4×4 `NCVW` and 5×5 `NCV` circuits. The experiment have been performed on the IBM xSeries x3650 with 8x8-core Intel Xeon 6C X5650 2.66GHz CPUs and 320 GiB of RAM computer system with KVM virtualized RedHat RHEL6 64 bit operating system (62 logical CPUs and 300 GiB of RAM). The parameters for the database are presented in Table 6.5 and Table 6.6.

The first column shows the number of gates in an optimal circuit,

Table 6.5. Parameters of the optimal 4×4 quantum circuits database for functions requiring up to nine `NCVW` quantum gates.

gc	# of quantum functions	# of reduced functions	table size	time [s]
1	64	6	384 B	0
2	2,370	119	6 KiB	0
3	52,418	2,305	96 KiB	0
4	831,009	35,226	1,536 KiB	1
5	10,471,110	439,062	12 MiB	3
6	109,718,716	4,583,181	192 MiB	41
7	975,447,616	40,690,285	1,536 MiB	449
8	7,424,652,453	309,529,874	12 GiB	4,309
9	24,334,931,481	2,028,446,782	96 GiB	44,658

Table 6.6. Parameters of the optimal 5×5 quantum circuits database for functions requiring up to nine `NCV` quantum gates.

gc	# of quantum functions	# of reduced functions	table size	time [s]
1	65	4	640 B	0
2	2,230	46	5 KiB	0
3	53,910	626	40 KiB	1
4	1,027,085	9,729	640 KiB	4
5	16,290,196	142,296	10 MiB	63
6	221,536,465	1,880,785	80 MiB	860
7	2,625,081,980	22,045,601	1,280 MiB	10,713
8	27,341,529,065	228,634,538	10 GiB	138,241
9	251,564,135,255	2,099,850,539	160 GiB	1,373,407

the second column shows the total number of quantum functions requiring such number of quantum gates, and the third one presents the number of canonical representatives [120, 121, 314] of the functions actually stored in the database. The next column presents the amount of memory required to store the hash table with canonical

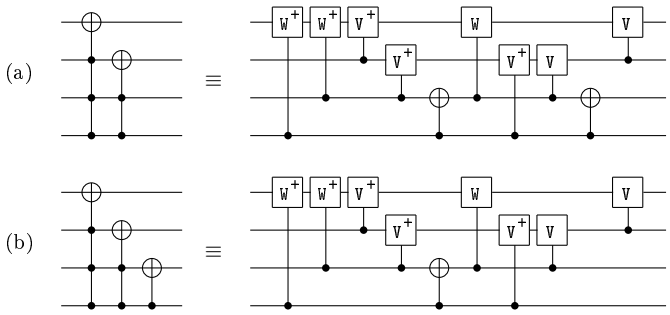


Figure 6.14. Reversible 4×4 circuits mapped to optimal NCVW quantum circuits: (a) pair of 4-bit and 3-bit MCT gates, (b) additional CNOT gate.

representatives for the circuits. Last column shows the time required to compute the given hash table (in CPU-seconds). Both databases can be used to construct quantum optimal circuits up to 18 gates for any reversible 4-bit Boolean function. Earlier approaches to quantum optimal circuit synthesis were able to construct optimal circuits up to 10 gates only [123, 138].

Let us compare the parameters of the constructed databases with the database from [121] for the 4-bit reversible circuits built from standard multi-control Toffoli gates. It can be easily noticed that the database for quantum circuits stores more functions for each gate count value. This is caused by one less symmetry used for finding canonical representatives since for quantum circuits it is not possible to use inversion (as illustrated in Figure 6.13) as well as the fact that there are more gates available (6 types: NOT, CNOT, controlled-V, controlled-V^+, controlled-W, controlled-W^+ instead of 4 NCT gates: NOT, CNOT, 3-bit and 4-bit Toffoli gates).

Using our tool we have constructed optimal quantum circuits for all pairs of the 2-bit, 3-bit and 4-bit MCT gates and compared the quantum cost counted as a sum of the quantum costs of each MCT gate with the number of quantum gates in the resulting circuits. In this way two interesting decompositions have been found using NCVW quantum

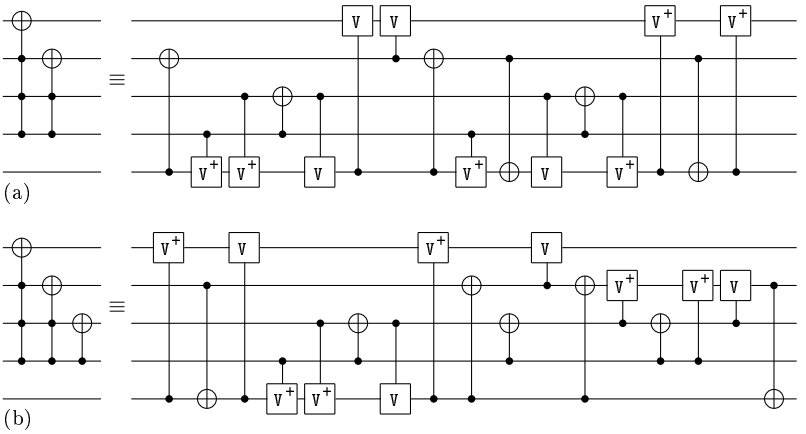


Figure 6.15. Optimal 5×5 **NCV** quantum circuits: (a) pair of 4-bit and 3-bit **MCT** gates, (b) Toffoli circuit from Figure 6.14 (b).

gate library, see Figure 6.14 (a) and Figure 6.16 (a). Namely, this experiment reveals that there exists a quantum decomposition of a pair of 4-bit and 3-bit **MCT** gates, which can be considered as a 4-bit analogue of Peres gate extended to 4-bit case. The difference in the quantum cost (QC) is significant. The QC of the pair of Toffoli gates of the left circuit in Figure 6.14 (a) is equal to $13 + 5 = 18$. The quantum circuit in right part of Figure 6.14 (a) has only QC=10.

One can notice that adding one more **CNOT** gate (the same as the last **CNOT** gate in the quantum circuit) to the both circuits in Figure 6.14 (a) gives us the circuit presented in Figure 6.14 (b). The pair of reversible and quantum circuits from Figure 6.14 (b) shows even higher savings in quantum cost. The quantum cost of the three Toffoli gates of Figure 6.14 (b) grows to QC=13+5+1=19 in comparison to Figure 6.14 (a), but the quantum cost for the equivalent quantum circuit in Figure 6.14 (b) is reduced to QC=9. This decomposition is specific to **NCVW** gate library.

The circuit found using 5×5 **NCV** database doesn't reveal such significant savings in quantum cost. Figure 6.15 (a) shows the circuit of a pair of 4-bit and 3-bit **MCT** gates with QC=14+5=19, and the equivalent optimal 5×5 **NCV** quantum circuit with the directly countable

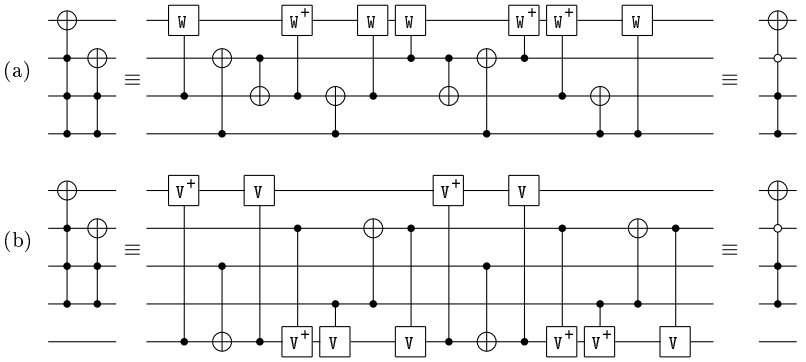


Figure 6.16. Best quantum circuits for the pair of 4-bit and 3-bit `MCT` gates: (a) 4×4 `NCVW`, (b) 5×5 `NCV`.

QC=16. Figure 6.15 (b) shows the extended circuit of three Toffoli gates with QC=14+5+1=20 and the equivalent optimal the 5×5 `NCV` circuit with QC=17.

The second interesting circuit that have been found is equivalent to the 4-bit mixed polarity Toffoli gate with one negative and two positives control lines, see Figure 6.16. These circuits have been found using both gate libraries: `NCVW` and `NCV`. The QC of the Toffoli gate pairs of Figure 6.16 are: (a) QC=13+5=18, and (b) QC=14+5=19. The resulting circuit of 13 `NCVW`- or 14 `NCV`-gates proves that our tool finds the optimal quantum circuit, because such mixed-polarity Toffoli gate can be also realized with the same number of elementary quantum gates as the standard positive-polarity Toffoli gate. An example of such realization, using the Barenco's decomposition and interchanging some `W` and `W`$^+$ gates is described in [281].

All other quantum circuits implementing the reversible functions of 4-bit circuits with pairs of Toffoli gates do not reveal significant savings in quantum cost – the reduction in those cases was 3 or less units.

Quantum circuits shown in Figure 6.14 (b) are particularly useful for reducing quantum cost in the existing reversible designs, because the quantum cost of such circuit is 4 units less than the quantum cost of the single 4-bit Toffoli gate itself. In some cases the quantum

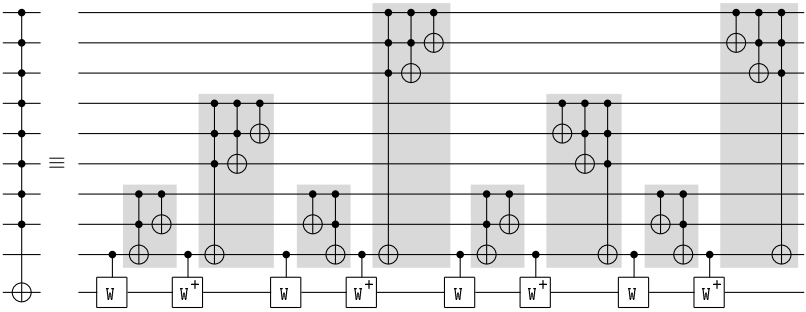


Figure 6.17. Application of the circuit from Figure 6.14 (b) to reduce quantum cost of the Toffoli gate with 8 control signals.

circuit and its inverse can replace 4-bit Toffoli gates. An example of such substitution is shown in Figure 6.17, where the above discussed quantum circuits reduced total quantum cost to $8{\times}1+4{\times}4+4{\times}9 = 60$ units for the circuit for 9-bit Toffoli gate. The so far known smallest quantum cost of the design from [258] is QC=64.

It can be checked by inspection that the values of the signals at the control lines for all gates in the presented circuits are only in the Boolean domain, i.e., having the values equal to 0 or 1.

In this section a 4-bit `NCVW` and 5-bit `NCV` optimal quantum circuits equivalent to the pairs of multi-control Toffoli gates have been presented. Those optimal circuits can be used for optimizing quantum decompositions of large multi-control Toffoli gates in a similar manner as in the case of Peres gates. An example of such optimization is depicted in Figure 6.17. The constructed database of optimal `NCV` and `NCVW` quantum circuits can also be used for optimizing quantum cost of reversible circuits, in a similar way as described in [312].

Bibliography

[1] "3GPP, Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and Channel Coding (Release 9) 3GPP Organizational Partners TS 36.212, Rev. 8.3.0". May 2008.

[2] M. Abramovici and M. A. Breuer. "Multiple Fault Diagnosis in Combinational Circuits Based on an Effect-Cause Analysis". In: *IEEE Transactions on Computers* C-29.6 (June 1980), pp. 451–460. ISSN: 0018-9340.

[3] M. Adamski. "Parallel Controller Implementation Using Standard PLD Software". In: *FPGAs : International Workshop on Field Programmable Logic and Applications*. Abingdon EE&CS Books, 1991, pp. 296–304.

[4] V. K. Agarwal and A. S. F. Fung. "Multiple Fault Testing of Large Circuits by Single Fault Test Sets". In: *IEEE Transactions on Computers* C-30.11 (Nov. 1981), pp. 855–865. ISSN: 0018-9340.

[5] A. Agrawal et al. "Compact and Complete Test Set Generation for Multiple Stuck-faults". In: *Digest of Technical Papers of the 1996 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD. Nov. 1996, pp. 212–219.

[6] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1974. ISBN: 0201000296.

[7] S. B. Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* 27.6 (June 1978), pp. 509–516. ISSN: 0018-9340.

[8] S. B. Akers. "Functional Testing with Binary Decision Diagrams". In: *Journal of Design Automation and Fault-Tolerant Computing* 2 (1978), pp. 311–331.

[9] F. Armknecht et al. "Efficient Computation of Algebraic Immunity for Algebraic and Fast Algebraic Attacks". In: *Advances in Cryptology - Eurocrypt 2006*. Vol. 4004. Berlin, Germany, Springer-Verlag, 2006, pp. 147–164. URL: `http://www.unilim.fr/pages_perso/philippe.gaborit/AI_main.pdf`.

[10] A. Artale et al. "Adding Weight to DL-Lite". In: *Proceedings of the 22nd Intnational Workshop on Description Logics*. DL. 2009.

[11] A. Artale et al. "DL-Lite in the Light of First-Order Logic". In: *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*. AAAI. 2007, pp. 361–366.

[12] P. Ashar, S. Devadas, and R. A. Newton. "A Unified Approach to the Decomposition and Re-Decomposition of Sequential Machines". In: *Proceedings of the 27th Design Automation Conference*. DAC. 1990, pp. 601–609.

[13] F. Baader, S. Brandt, and C. Lutz. "Pushing the $\mathcal{EL}$ Envelope". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. IJCAI. 2005, pp. 364–369.

[14] F. Baader, S. Brandt, and C. Lutz. "Pushing the $\mathcal{EL}$ Envelope Further". In: *Proceedings of OWL: Experiences and Directions*. OWLED. 2008.

[15] F. Baader et al., eds. *The Description Logic Handbook: Theory, Implementation, and Applications*. 2nd. Vol. 1. Cambridge University Press, 2003.

[16] R. I. Bahar et al. "Algebraic Decision Diagrams and Their Applications". In: *Digest of Technical Papers of the 1993 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD. Nov. 1993, pp. 188–191.

[17] Z. Banaszak, J. Kuś, and M. Adamski. *Sieci Petriego. Modelowanie, Sterowanie i Synteza Systemów Dyskretnych*. Zielona Góra: Wyższa Szkoła Inżynierska, 1993.

[18] S. Baranov. *Logic and System Design of Digital Systems*. Tallinn: TUT Press, 2008.

[19] A. Barenco et al. "Elementary Gates for Quantum Computation". In: *Physical Review A* 52.5 (Nov. 1995), pp. 3457–3467. DOI: 10.1103/PhysRevA.52.3457. eprint: arXiv:quant-ph/9503016.

[20] A. Barkalov and L. Titarenko. *Basic Principles of Logic Design*. Zielona Gora: University of Zielona Gora Press, 2010.

[21] C. Berge. *Graphs and Hypergraphs*. North-Holland; Elsevier, 1973. ISBN: 0444103996.

[22] *Berkeley Logic Interchange Format (BLIF)*. University of California, Berkeley, CA, USA. July 1992.

[23] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. URL: http://www.eecs.berkeley.edu/alanmi/abc/.

[24] J. Bern, C. Meinel, and A. Slobodova. "Efficient OBDD-Based Boolean Manipulation in CAD Beyond Current Limits". In: *32nd Conference on Design Automation*. DAC. 1995, pp. 408–413.

[25] A. Bernasconi, V. Ciriani, and R. Cordone. "On Projecting Sums of Products". In: *11th Euromicro Conference on Digital Systems Design: Architectures, Methods and Tools*. DSD. 2008, pp. 787–794.

[26] A. Bernasconi et al. "On Decomposing Boolean Functions via Extended Cofactoring". In: *Design, Automation and Test in Europe*. DATE. 2009, pp. 1464–1469.

[27] C. Berrou, A. Glavieux, and P. Thitimajshima. "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes". In: *IEEE International Conference on Communications*. ICC. May 1993, pp. 1064–1070.

[28] C. Berrou et al. "An IC for Turbo-Codes Encoding and Decoding". In: *IEEE International Solid-State Circuits Conference*. ISSCC. Feb. 1995, pp. 90–91.

[29] O. Beyersdorff et al. "The Complexity of Propositional Implication". In: *Information Processing Letters* 109.18 (2009), pp. 1071–1077. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2009.06.015. URL: http://www.sciencedirect.com/science/article/B6V0F-4WRD3N2-1/2/729176d318c75bb3c631828219e2bda9.

[30] O. Beyersdorff et al. "The Complexity of Reasoning for Fragments of Default Logic". In: *Proceedings of SAT 2009*. Vol. 5584. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 51–64.

[31] A. Biere. *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*. Tech. rep. 10/1. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, Aug. 2010.

[32] K. Biliński. "Application of Petri Nets in Parallel Controllers Design"". PhD thesis. University of Bristol, 1996.

[33] K. Biliński et al. "Parallel Controller Synthesis from a Petri Net Specification". In: *Proceedings of the European Design Automation Conference*. EDAC. Grenoble, France, 1994, pp. 96–101.

[34] A. G. Birger, E. T. Gurvitch, and S. Kuznetsov. "Testing of Multiple Faults in Combinational Circuits". In: *Avtomatika i Telemehanika* 8 (1975). (in Russian), pp. 113–120.

[35] M. Blanchard. *Comprendre, Maîtriser Et Appliquer Le Grafcet.* (in French). Toulouse: Cepadues, 1979.

[36] A. Blanksby and C. Howland. "A 690-mW 1-Gb/s 1024-b, Rate-1/2 Low-Density Parity-Check Code Decoder". In: *IEEE Journal of Solid-State Circuits* 37 (2002), pp. 404–412.

[37] E. Böhler et al. "Playing with Boolean Blocks, Part I: Post's Lattice with Applications to Complexity Theory". In: *SIGACT News* 34.4 (2003), pp. 38–52.

[38] B. Bollig and I. Wegener. "Improving the Variable Ordering of OBDDs is NP-Complete". In: *IEEE Transactions on Computers* 45.9 (Sept. 1996), pp. 993–1002. ISSN: 0018-9340.

[39] M. M. Bongard. *Pattern Recognition.* Rochelle Park, NJ, USA: Hayden Book Co., Spartan Books., 1970.

[40] S. Borkar. "Energy Management in Future Many-Core Microprocessors". TD Forum: Power Systems from the Gigawatt to the Microwatt - Generation, Distribution, Storage and Efficient Use of Energy. Feb. 2008.

[41] R. Bose and D. Ray-Chaudhuri. "On A Class of Error Correcting Binary Group Codes". In: *Information and Control* 3.1 (1960), pp. 68–79.

[42] L. E. Bourne. "An Inference Model for Conceptual Rule Learning". In: *Theories in Cognitive Psychology.* Ed. by R. Solso. Washington: Erlbaum, 1974, pp. 231–256.

[43] R. J. Brachman and H. J. Levesque. "The Tractability of Subsumption in Frame-Based Description Languages". In: *AAAI.* 1984, pp. 34–37.

[44] R. J. Brachman and J. G. Schmolze. "An Overview of the KL-ONE Knowledge Representation System". In: *Cognitive Science* 9.2 (1985), pp. 171–216. ISSN: 1551-6709. DOI: 10.1207/s15516709cog0902_1. URL: http://dx.doi.org/10.1207/s15516709cog0902_1.

[45] A. Braeken and B. Preneel. "Algebraic Immunity of Symmetric Boolean Functions". In: *Lecture Notes in Computer Science* 3797 (2005), pp. 35–48.

[46] T. Brandon et al. "A Scalable LDPC Decoder Architecture with Bit-Serial Message Exchange". In: *Integration, The VLSI Journal* 41.3 (2008), pp. 385–398.

[47] R. K. Brayton and A. Mishchenko. "ABC: an Academic Industrial Strength Verification Tool". In: *Proceedings of the 22nd International Conference on Computer Aided Verification*. CAV. Edinburgh, UK: Springer-Verlag, 2010, pp. 24–40. ISBN: 3-642-14294-X, 978-3-642-14294-9.

[48] R. K. Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA, USA: Kluwer Academic Publishers, 1984, p. 192. ISBN: 0898381649.

[49] F. Brglez and H. Fujiwara. "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran". In: *IEEE International Symposium on Circuits and Systems*. ISCAS. 1985, pp. 677–692.

[50] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* 35.8 (Aug. 1986), pp. 677–691. ISSN: 0018-9340.

[51] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* 35.8 (Aug. 1986), pp. 667–691. ISSN: 0018-9340.

[52] R. E. Bryant and Y.-A. Chen. "Verification of Arithmetic Functions with Binary Moment Diagrams". In: *31st Conference on Design Automation*. DAC. 1994, pp. 535–541.

[53] P. Buchholz and P. Kemper. "Hierarchical Reachability Graph Generation for Petri Nets". In: *Universität Dortmund, Fachbereich Informatik, Forschungsbericht Nr. 660*. 1997, p. 2002.

[54] A. Burg and O. Keren. "Functional Level Embedded Self-Testing for Walsh Transform Based Adaptive Hardware". In: *IEEE International On-Line Testing Symposium*. IOLTS. 2012, pp. 134–135.

[55] J. T. Butler and T. Sasao. "Logic Functions for Cryptography - A Tutorial". In: *Proceedings of the 10th Reed-Muller Workshop*. RM. Naha, Okinawa, Japan, May 2009, pp. 127–136.

[56] J. T. Butler et al. "On the Use of Transeunt Triangles to Synthesize Fixed-Polarity Reed-Muller Expansions of Functions". In: *Proceedings of the 10th Reed-Muller Workshop*. RM. Naha, Okinawa, Japan, May 2009, pp. 119–126.

[57] D. Calvanese et al. "DL-Lite: Tractable Description Logics for Ontologies". In: *Proceedings of the 20th National Conference on Artificial Intelligence*. AAAI. 2005, pp. 602–607.

[58] P. Camurati et al. "Improved Techniques for Multiple Stuck-at Fault Analysis Using Single Stuck-at Fault Test Sets". In: *Proceedings of the 1992 IEEE International Symposium on Circuits and Systems*. Vol. 1. ISCAS. May 1992, pp. 383–386.

[59] A. Chandrakasan, S. Sheng, and R. Brodersen. "Low-Power CMOS Digital Design". In: *IEEE Journal of Solid-State Circuits* 27.4 (1992), pp. 473–484.

[60] E. J. Chikofsky and J. H. Cross (II.) "Reverse Engineering and Design Recovery: a Taxonomy". In: *IEEE Transactions on Software Engineering* 7.1 (1990), pp. 13–17.

[61] M. Chudnovsky et al. "The Strong Perfect Graph Theorem". In: *Annals of Mathematics* 164.1 (2006), pp. 51–229.

[62] E. M. Clarke, M. Fujita, and X. Zhao. "Multi-Terminal Binary Decision Diagrams and Hybrid Decision Diagrams". In: *Representations of Discrete Functions*. Ed. by T. Sasao and M. Fujita. Springer US, 1996, pp. 93–108. ISBN: 978-1-4612-8599-1.

[63] D. Coppersmith and S. Winograd. "Matrix Multiplication via Arithmetic Progressions". In: *Journal Symbolic Computation* 9:3 (1990), pp. 251–280.

[64] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1994.

[65] R. Cornet and N. de Keizer. "Forty Years of SNOMED: a Literature Review". In: *BMC Medical Informatics and Decision Making* 8.Suppl 1 (2008), S2. ISSN: 1472-6947. DOI: 10.1186/1472-6947-8-S1-S2. URL: http://www.biomedcentral.com/1472-6947/8/S1/S2.

[66] O. Coudert. "Two-Level Logic Minimization: an Overview". In: *Integration, the VLSI Journal* 17.2 (Oct. 1994), pp. 97–140. ISSN: 0167-9260.

[67] N. Courtois. "Fast Algebraic Attacks on Stream Ciphers with Linear Feedback". In: *Advances in Cryptology - CRYPTO 2003*. Vol. LNCS 2729. Berlin, Germany, Springer-Verlag, 2003, pp. 176–194.

[68] N. Courtois and W. Meier. "Algebraic Attacks on Stream Ciphers with Linear Feedback". In: *Advances in Cryptology - EUROCRYPT 2003*. Vol. LNCS 2656. Berlin, Germany, Springer-Verlag, 2003, pp. 345–359.

[69] H. Cox and J. Rajski. "A Method of Fault Analysis for Test Generation and Fault Diagnosis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 7.7 (July 1988), pp. 813–833.

[70] N. Creignou et al. "Sets of Boolean Connectives that Make Argumentation Easier". In: *Proc. 12th JELIA*. Vol. 6341. Lecture Notes in Computer Science. Springer, 2010, pp. 117–129.

[71] N. Creignou et al. "The Complexity of Reasoning for Fragments of Autoepistemic Logic". In: *Circuits, Logic, and Games*. Ed. by B. Rossman et al. Dagstuhl Seminar Proceedings 10061. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2523.

[72] B. Crowley and V. C. Gaudet. "Switching Activity Minimization in Iterative LDPC Decoders". In: *Journal of Signal Processing Systems* (2011). DOI: doi10.1007/s11265-011-0577-y.

[73] T. W. Cusick and P. Stănică. *Cryptographic Boolean Functions and Applications*. Elsevier–Academic Press, 2009.

[74] T. Czajkowski and S. Brown. "Functionally Linear Decomposition and Synthesis of Logic Circuits for FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.12 (2008), pp. 2236–2249.

[75] A. Darabiha, A. C. Carusone, and F. R. Kschischang. "A Bit-Serial Approximate Min-Sum LDPC Decoder and FPGA Implementation". In: *IEEE International Symposium on Circuits and Systems*. ISCAS. May 2006, pp. 149–152. ISBN: 0-7803-9389-9. DOI: 10.1109/ISCAS.2006.1692544.

[76] C. Davies et al. "Adaptation of Tissue to a Chronic Heat Load". In: *ASAIO Journal* 40.3 (1994), pp. M514–M517. DOI: DOI:10.1097/00002480-199407000-00053.

[77] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[78] A. De Vos. *Reversible Computing*. Weinheim: Wiley-VCH, 2010.

[79] A. De Vos and S. De Baerdemacker. "Logics Between Classical Reversible Logic and Quantum Logic". In: *Proceedings of the 9th International Workshop on Quantum Physics and Logic*. Bruxelles. Oct. 2012, pp. 123–128.

[80] A. De Vos and S. De Baerdemacker. "The NEGATOR as a Basic Building Block for Quantum Circuits". In: *Open Systems & Information Dynamics* 20 (Mar. 2013), p. 1350004. ISSN: 1230-1612. DOI: 10.1142/S1230161213500042.

[81] A. De Vos and S. De Baerdemacker. "The Roots of the NOT Gate". In: *Proceedings of the 42nd International Symposium on Multiple-Valued Logic*. ISMVL. Victoria, BC, Canada, May 2012, pp. 167–172.

[82] A. De Vos, J. De Beule, and L. Storme. "Computing with the Square Root of NOT". In: *Serdica Journal of Computing* 3 (2009), pp. 359–370.

[83] A. De Vos, R. Van Laer, and S. Vandenbrande. "The Group of Dyadic Unitary Matrices". In: *Open Systems & Information Dynamics* 19 (2012), p. 1250003.

[84] D. Debnath and T. Sasao. "Fast Boolean Matching Under Permutation by Efficient Computation of Canonical Form". In: *IEICE Transactions of Fundamentals of Electronics, Communications and Computer Science* E87-A (2004), pp. 3134–3140.

[85] D. Deutsch, A. Ekert, and R. Lupacchini. "Machines, logic and Quantum Physics". In: *The Bulletin of Symbolic Logic* 3 (2000), pp. 265–283.

[86] F. J. O. Dias. "Fault Masking in Combinational Logic Circuits". In: *IEEE Transactions on Computers* 24.5 (May 1975), pp. 476–482. ISSN: 0018-9340.

[87] F. M. Donini et al. "The Complexity of Concept Languages". In: *Information and Computation* 134.1 (Apr. 1997), pp. 1–58.

[88] F. M. Donini et al. "The Complexity of Existential Quantification in Concept Languages". In: *Artificial Intelligence* 53.2-3 (Feb. 1992), pp. 309–327.

[89] R. Drechsler and B. Becker. *Binary Decision Diagrams - Theory and Implementations*. Kluwer Academic Publishers, 1998.

[90] R. Drechsler, A. Finder, and R. Wille. "Improving ESOP-Based Synthesis of Reversible Logic Using Evolutionary Algorithms". In: *Applications of Evolutionary Computation*. Vol. 6625. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 151–161.

[91] R. Drechsler et al. "Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams". In: *31st Conference on Design Automation*. DAC. June 1994, pp. 415–419.

[92] E. Dubrova. "A Polynomial Time Algorithm for Non-Disjoint Decomposition of Multi-Valued Functions". In: *International Symposium on Multiple-Valued Logic*. ISMVL. 2004, pp. 309–314.

[93] E. Eilam. *Reversing: Secrets of reverse engineering*. New York, NY, USA: Wiley, 2005.

[94] P. Elias. "Coding for Noisy Channels". In: *IRE Convention Record*. 1955.

[95] C. J. Etherington. "An Analysis of Cryptographically Significant Boolean Functions with High Correlation Immunity by Reconfigurable Computer". MA thesis. Monterey, CA: ECE Dept., Naval Postgraduate School, December 2010. URL: `http://www.dtic.mil/dtic/tr/fulltext/u2/a536393.pdf`.

[96] J. S. B. T. Evans, S. E. Newstead, and R. M. J. Byrne. *Human Reasoning: The Psychology of Deduction*. Hillsdale, NJ, USA: Erlbaum, 1993.

[97] J. Feldman. "Minimization of Boolean Complexity in Human Concept Learning". In: *Nature* 407 (2000), pp. 630–633.

[98] S. Fenner et al. *Rectangle Free Coloring of Grids*. 2009. URL: `http://www.cs.umd.edu/~gasarch/papers/grid.pdf`.

[99] P. Fišer and J. Hlavička. "BOOM, A Heuristic Boolean Minimizer". In: *Computers and Informatics* 22.1 (2003), pp. 19–51.

[100] P. Fišer and H. Kubatova. "Flexible Two-Level Boolean Minimizer BOOM-II and Its Applications". In: *Euromicro Conference on Digital Systems Design*. DSD. Cavtat, Croatia, 2006, pp. 369–376.

[101] P. Fišer and J. Schmidt. "How Much Randomness Makes a Tool Randomized?" In: *Proceedings of the International Workshop on Logic and Synthesis*. IWLS. San Diego, CA, USA, June 2011, pp. 17–24.

[102] P. Fišer and J. Schmidt. "Improving the Iterative Power of Resynthesis". In: *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits System*. DDECS. Apr. 2012, pp. 30–33. ISBN: 978-1-4673-1187-8.

[103] P. Fišer and J. Schmidt. "It Is Better to Run Iterative Resynthesis on Parts of the Circuit". In: *Proceedings of International Workshop on Logic and Synthesis*. IWLS. Irvine, CA, USA, June 2010, pp. 17–24.

[104] P. Fišer and J. Schmidt. "On Using Permutation of Variables to Improve the Iterative Power of Resynthesis". In: *Proceedings of the 10th International Workshop on Boolean Problems*. IWSBP. Freiberg, Germany, Sept. 2012, pp. 107–114.

[105] J. Fodor. "Concepts: A potboiler." In: *Cognition* 50 (1994), pp. 95–113.

[106] R. Forré. "The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition". In: *Advances in Cryptology - CRYPTO 1988*. Berlin, Germany: Springer-Verlag, 1988, pp. 450–468.

[107] H. E. Foundalis. *Phaeco: A Cognitive Architecture Inspired by Bongard's Problems*. 2006.

[108] A. Galindo and M. Martín-Delgado. "Information and Computation: Classical and Quantum Aspects". In: *Review of Modern Physics* 74 (2002), pp. 347–423.

[109] R. Gallager. "Low-Density Parity-Check Codes". In: *IRE Transactions on Information Theory* IT-8 (1962), pp. 21–28.

[110] E. M. Gao et al. "MVSIS: Multi-Valued Logic Synthesis System". In: *Notes of the International Workshop on Logic and Synthesis*. IWLS. Tahoe City, CA, USA, 2001.

[111] F. Garcia-Valles and J. M. Colom. "Parallel Controller Synthesis From a Petri Net Specification". In: *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*. SMC. 1995.

[112] V. C. Gaudet and W. Gross. "Switching Activity in Stochastic Decoders". In: *IEEE International Symposium on Multiple-Valued Logic*. ISMVL. May 2010.

[113] V. C. Gaudet, C. Schlegel, and R. Dodd. "LDPC Decoder Message Formatting Based on Activity Factor Minimization Using Differential Density Evolution". In: *IEEE Information Theory Workshop*. ITW. June 2007, pp. 571–576.

[114] M. Gebser et al. "clasp: A Conflict-Driven Answer Set Solver". In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Vol. LNAI 4483. LPNMR. Springer, 2007, pp. 260–265.

[115] D. Geer. "Chip Makers Turn to Multicore Processors". In: *IEEE Computer* 38.5 (2005), pp. 11–13.

[116] C. Girault and R. Valk. *Petri Nets for Systems Engineering. A Guide to Modeling, Verification, and Application*. Berlin Heidelberg: Springer-Verlag, 2003.

[117] F. Glover. "Tabu Search - Part I". In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206.

[118] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[119] E. I. Goldberg et al. "Negative Thinking by Incremental Problem Solving: Application to Unate Covering". In: *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD. San Jose, California, United States, 1997, pp. 91–98.

[120] O. Golubitsky, S. M. Falconer, and D. Maslov. "Synthesis of the Optimal 4-bit Reversible Circuits". In: *Proceedings of the 47th ACM/IEEE Design Automation Conference*. DAC. June 2010, pp. 653–656.

[121] O. Golubitsky and D. Maslov. "A Study of Optimal 4-Bit Reversible Toffoli Circuits and Their Synthesis". In: *IEEE Transactions on Computers* 61.9 (Sept. 2012), pp. 1341–1353. ISSN: 0018-9340.

[122] G. P. Goodwin and P. N. Johnson-Laird. "Conceptual Illusions". In: *Cognition* 114 (2010), pp. 263–308.

[123] D. Grosse et al. "Exact Synthesis of Elementary Quantum Gate Circuits for Reversible Functions with Don't Cares". In: *Proceedings of the 38th International Symposium on Multiple-Valued Logic*. ISMVL. May 2008, pp. 214–219.

[124] R. K. Guy. "A many-facetted problem of Zarankiewicz". In: *The Many Facets of Graph Theory*. Ed. by G. Chartrand and S. Kapoor. Vol. 110. Lecture Notes in Mathematics. University of Calgary Canada: Springer Berlin / Heidelberg, Oct. 1969, pp. 129–148. ISBN: 978-3-540-04629-5. URL: `http://dx.doi.org/10.1007/BFb0060112`.

[125] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Springer, 1996. ISBN: 978-0-387-31004-6.

[126] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. 1st. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN: 0-79239-746-0.

[127] M. Hack. *Analysis of Production Schemata by Petri Nets*. Corrections: Project MAC, Computation Structures Note 17 (1974). MIT Project MAC TR-94, 1972.

[128] R. Hamming. "Error Detecting and Error Correcting Codes". In: *The Bell System Technical Journal* 29 (1950).

[129] J. Han et al. "Toward Hardware-Redundant, Fault-Tolerant Logic for Nanoelectronics". In: *IEEE Design and Test of Computers* 22.4 (2005), pp. 328–339.

[130] M. A. Harrison. *Introduction to Switching Theory and Automata*. McGraw-Hill, 1965. ISBN: 978-0070268500.

[131] Harvard University Staff of the Computation Laboratory. *Synthesis of Electronic Computing and Control Circuits*. Ed. by H. H. Aiken. Cambridge, MA, USA: Harvard University Press, 1951.

[132] S. Hassoun and T. Sasao. *Logic Synthesis and Verification*. The Springer International Series in Engineering and Computer Science Series. Kluwer Academic Publishers, 2002. ISBN: 9780792376064.

[133] E. Hemaspaandra, H. Schnoor, and I. Schnoor. "Generalized Modal Satisfiability". In: *CoRR* abs/0804.2729 (2008), pp. 1–32. URL: http://arxiv.org/abs/0804.2729.

[134] J. Hlavička and P. Fišer. "BOOM, A Heuristic Boolean Minimizer". In: *Proceedings of the 2001 International Conference on Computer-Aided Design*. ICCAD. San Jose, CA, USA, 2001, pp. 493–442.

[135] C. V. Hoof. "Micro-Power Generation Using Thermal and Vibrational Energy Scavengers". TD Forum: Power Systems from the Gigawatt to the Microwatt - Generation, Distribution, Storage and Efficient Use of Energy. Feb. 2008.

[136] T. C. Hsiao and S. C. Seth. "An Analysis of the Use of Rademacher-Walsh Spectrum in Compact Testing". In: *IEEE Transactions on Computers* C-33.10 (1984), pp. 934–937.

[137] J. L. A. Hughes. "Multiple Fault Detection Using Single Fault Test Sets". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 7.1 (Jan. 1988), pp. 100–108. ISSN: 0278-0070.

[138] W. N. N. Hung et al. "Optimal Synthesis of Multiple Output Boolean Functions Using a Set of Quantum Gates by Symbolic Reachability Analysis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.9 (2006), pp. 1652–1663. ISSN: 0278-0070.

[139] M. Hunger and S. Hellebrand. "Verification and Analysis of Self-Checking Properties through ATPG". In: *14th IEEE International On-Line Testing Symposium*. IOLTS. July 2008, pp. 25–30.

[140] M. Hunger et al. "ATPG-Based Grading of Strong Fault-Secureness". In: *15th IEEE International On-Line Testing Symposium*. IOLTS. June 2009, pp. 269–274.

[141] "IEEE Standard 802.11n/D2.00, Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications". Feb. 2007.

[142] "IEEE Standard 802.16e, Air Interface for Fixed and Mobile Broadband Wireless Access Systems Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands and Corrigendum 1". Feb. 2006.

[143] "IEEE Standard 802.3an, Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications". Sept. 2006.

[144] "IEEE Standard 802.3ba, IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Amendment 4: Media Access Control Parameters, Physical Layers and Management Parameters for 40 Gb/s and 100 Gb/s Operation". June 2010.

[145] K. Ingle. *Reverse Engineering*. New York, NY, USA: McGraw-Hill, 1994.

[146] *International Energy Agency - Fast Facts*. URL: `http://www.iea.org/journalists/fastfacts.asp`.

[147] *International Technology Roadmap for Semiconductors, 2011 edition, Executive Summary*. URL: `http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf`.

[148] S. Kajihara, T. Sumioka, and K. Kinoshita. "Test Generation for Multiple Faults Based on Parallel Vector Pair Analysis". In: *Digest of Technical Papers of the 1993 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD. Nov. 1993, pp. 436–439.

[149] A. Karatkevich. *Dynamic Analysis of Petri Net-based Discrete Systems*. Berlin: Springer, 2007, xiii, 166 p. ISBN: 978-3-540-71464-4.

[150] Y. Karkouri et al. "Use of Fault Dropping for Multiple Fault Analysis". In: *IEEE Transactions on Computers* 43.1 (Jan. 1994), pp. 98–103.

[151] K. Karplus. *Using IF-THEN-ELSE DAGs For Multi-Level Logic Minimization*. Tech. rep. Santa Cruz, CA, USA, 1988.

[152] M. G. Karpovsky. "Error Detection for Polynomial Computations". In: *IEE Journal on Computer and Digital Techniques* C-26.6 (1980), pp. 523–528.

[153] M. G. Karpovsky and L. B. Levitin. "Universal Testing of Computer Hardware". In: *Spectral Techniques and Fault Detection*. Ed. by M. G. Karpovsky. Academic Press, 1985.

[154] M. G. Karpovsky, R. S. Stanković, and J. T. Astola. *Spectral Logic and Its Applications for the Design of Digital Devices*. Wiley, 2008. ISBN: 9780470289211. URL: `http://books.google.ee/books?id=Uxv7t7btTJYC`.

[155] M. G. Karpovsky and E. A. Trachtenberg. "Linear Checking Equations and Error-Correcting Capability for Computation Channels". In: *IFIP Congress*. 1977, pp. 619–624.

[156] U. Kebschull, E. Schubert, and W. Rosenstiel. "Multilevel Logic Synthesis Based on Functional Decision Diagrams". In: *Proceedings of the 3rd European Conference on Design Automation*. EDAC. Mar. 1992, pp. 43–47.

[157] O. Keren. "Adaptive Hardware Based on the Inverse Walsh Transform". In: *Proceedings of the 10th Reed-Muller Workshop*. RM. 2011, pp. 21–26.

[158] Y. C. Kim, V. D. Agrawal, and K. K. Saluja. "Multiple Faults: Modeling, Simulation and Test". In: *Proceedings of ASP-DAC 2002 7th Asia and South Pacific and the 15th International Conference on VLSI Design Automation Conference*. ASP-DAC. 2002, pp. 592–597.

[159] S. Kirkpatrick et al. "Optimization by Simulated Annealing". In: *Science* 220.4598 (May 1983), pp. 671–680.

[160] D. E. Knuth. "Art of Computer Programming". In: vol. 4. Fascicle 1, Section 7.1.4. Addison-Wesley Professional, 2009. Chap. The Bitwise Tricks & Techniques, Binary Decision Diagrams.

[161] N. Koda and T. Sasao. "LP Equivalence Class of Logic Functions". In: *IFIP 10.5 Workshop on Application of the Reed-Muller Expansion in Circuit Design*. RM. Hamburg, Germany, Sept. 1993, pp. 99–106.

[162] N. Koda and T. Sasao. "LP-Characteristic Vectors of Logic Functions and Their Applications". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* Part D-I, Vol. J76-D-1.6 (1993), pp. 260–268.

[163] I. V. Kogan. "Testing of Missing of Faults on the Node of Combinational Circuit". In: *Avtomatika i Vychislenie Tehnika, Automation and Computer Engineering* 2 (1976). (in Russian), pp. 31–37.

[164] A. N. Kolmogorov. "Three Approaches to the Quantitative Definition of Information". In: *Problemy Peredachi Informatisii* 1.1 (1965), pp. 3–11.

[165] I. Koren. *Computer Arithmetic Algorithms*. Natick, MA: A. K. Peters, 2002.

[166] A. V. Kovalyov. "Concurrency Relation and the Safety Problem for Petri Nets". In: *Proceedings of the 13th International Conference on Application and Theory of Petri Nets 1992, Lecture Notes in Computer Science*. Vol. 616. Springer-Verlag, June 1992, pp. 299–309.

[167] A. Kovalyov and J. Esparza. "A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs". In: *Proceedings of the International Workshop on Discrete Event Systems, (WODES)*. 1995, pp. 1–6.

[168] T. Kozłowski et al. "Parallel Controller Synthesis Using Petri Nets". In: *IEE Proceedings - Computers and Digital Techniques* 142.4 (1989), pp. 263–271.

[169] J. Kristensen and P. Miltersen. "Finding Small OBDDs for Incompletely Specified Truth-Tables is Hard". In: *Proceedings of the 12th Annual International Conference COCOON 2006*. Taipei, Taiwan, Aug. 2006, pp. 489–496.

[170] F. Kschischang, B. Frey, and H.-A. Loeliger. "Factor Graphs and the Sum-Product Algorithm". In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 498–519.

[171] G. Łabiak et al. "UML Modelling in Rigorous Design Methodology for Discrete Controllers". In: *International Journal of Electronics and Telecommunications* 58 (2012), pp. 27–34.

[172] Y.-T. Lai and S. Sastry. "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification". In: *Proceedings of the 29th ACM/IEEE Design Automation Conference*. DAC. Anaheim, CA, USA: IEEE Computer Society Press, 1992, pp. 608–613. ISBN: 0-89791-516-X. URL: http://dl.acm.org/citation.cfm?id=113938.149642.

[173] R. Landauer. "Irreversibility and Heat Generation in the Computing Process". In: *IBM Journal of Research and Development* 5 (1961), pp. 183–191.

[174] A. Lasota. "Modeling of Production Processes with UML Activity Diagrams and Petri Nets". PhD thesis. University of Zielona Góra, 2012.

[175] C. F. Laywine and G. L. Mullen. *Discrete Mathematics Using Latin Squares*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., 1998. ISBN: 978-0471240648.

[176] J. Le Coz et al. "Comparison of 65nm LP Bulk and LP PD-SOI with Adaptive Power Gate Body Bias for an LDPC Codec". In: *IEEE International Solid-State Circuits Conference*. ISSCC. Feb. 2011, pp. 336–337.

[177] C. Y. Lee. "Representation of Switching Circuits by Binary Decision Programs". In: *The Bell System Technical Journal* (1959), pp. 985–999.

[178] N. Y. L. Lee and P. N. Johnson-Laird. "A Synthetic Reasoning and Reverse Engineering of Boolean Circuits". In: *Proceedings of the Twenty-Seventh Annual Conference of the Cognitive Science Society*. Ed. by N. J. Mahwah. CogSci. Stresa, Italy, 2005, pp. 1260–1265.

[179] I. Levin, O. Keren, and H. Rosensweig. "Concept of Non-exactness in Science Education". In: *New Perspective for Science Education*. Florence, Italy, 2012.

[180] I. Levin, G. Shafat, and O. Keren. "Cognitive Complexity of Boolean Problems". In: *Proceedings of 10th International Workshop on Boolean Problems*. IWSBP. Freiberg, Germany, 2012, pp. 171–176.

[181] R. W. Lewis. *Programming Industrial Control Systems Using IEC 1131-3*. London: IEE, 1995.

[182] M. Lukac et al. "Decomposition of Reversible Logic Function Based on Cube-Reordering". In: *Proceedings of the Reed-Muller Workshop*. RM. Finland: TICSP Press, 2011, pp. 63–69.

[183] E. Macii and T. Wolf. "Multiple Stuck-at Fault Test Generation Techniques for Combinational Circuits Based on Network Decomposition". In: *Proceedings of the 36th Midwest Symposium on Circuits and Systems*. Vol. 1. MWSCAS. Aug. 1993, pp. 465–467.

[184] D. MacKay and R. Neal. "Near Shannon Limit Performance of Low Density Parity Check Codes". In: *Electronics Letters* 33.6 (1997), pp. 457–458.

[185] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Mathematical Library, 1977.

[186] D. Marx. "Graph Coloring Problems and Their Applications in Scheduling". In: *Periodica Polytechnica, Electrical Engineering* 48.1 (2004), pp. 11–16.

[187] D. Maslov and G. W. Dueck. "Improved Quantum Cost for n-Bit Toffoli Gates". In: *Electronics Letters* 39.25 (2003), pp. 1790–1791. ISSN: 0013-5194.

[188] D. Maslov, G. W. Dueck, and M. D. Miller. "Techniques for the Synthesis of Reversible Toffoli Networks". In: *ACM Transactions on Design Automation of Electronic Systems* 12.4 (2007).

[189] D. Maslov et al. "Quantum Circuit Simplification and Level Compaction". In: *IEEE Transactions on ComputerAided Design of Integrated Circuits and Systems*. 2008, pp. 436–444.

[190] M. Matsuura and T. Sasao. "Representation of Incompletely Specified Switching Functions Using Pseudo-Kronecker Decision Diagrams". In: *International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*. RM. Starkville, Mississippi, U.S.A, Aug. 2001, pp. 27–33.

[191] K. McElvain. *LGSynth93 Benchmark Set: Version 4.0*. Mentor Graphics. May 1993.

[192] D. L. Medin, E. B. Lynch, and K. O. Solomon. "Are there Kinds of Concepts?" In: *Annual Review of Psychology* 51 (2000), pp. 121–147.

[193] D. Medin and E. E. Smith. "Concepts and Concept Formation". In: *Annual Review of Psychology*. Ed. by M. R. Rosensweig and L. W. Porter. Stresa, Italy, 1984, pp. 113–118.

[194] A. Meier. "Generalized Complexity of $\mathcal{ALC}$ Subsumption". In: *CoRR* 1205.0722 (2012), pp. 1–10.

[195] A. Meier and T. Schneider. "Generalized Satisfiability for the Description Logic $\mathcal{ALC}$". In: *CoRR* abs/1103.0853 (Mar. 2011), pp. 1–37. URL: http://arxiv.org/abs/1103.0853.

[196] A. Meier and T. Schneider. "Generalized Satisfiability for the Description Logic $\mathcal{ALC}$". In: *Proceedings of the 8th International Conference on Theory and Applications of Models of Computation*. Vol. LNCS 6648. TAMC. Springer Verlag, 2011, pp. 552–562.

[197] A. Meier and T. Schneider. "The Complexity of Satisfiability for Sub-Boolean Fragments of ALC". In: *Proceedings of the 23rd International Workshop on Description Logics*. DL. CEUR-WS.org, 2010.

[198] W. Meier, E. Pasalic, and C. Carlet. "Algebraic Attacks and Decomposition of Boolean Functions". In: *Advances in Cryptology - CRYPTO 2004*. Vol. LNCS 3027. Berlin, Germany: Springer-Verlag, 2003, pp. 474–491.

[199] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

[200] K. Mielcarek. "Application of Perfect Graphs in Digital Devices Designing". (in Polish). PhD thesis. University of Zielona Góra, 2009.

[201] K. Mielcarek, M. Adamski, and W. Zajac. "Perfect Petri Nets". In: *Journal of Theoretical and Applied Computer Science* 77.3 (2010). (In Polish), pp. 169–176.

[202] M. D. Miller. "Decision Diagram Techniques for Reversible and Quantum Circuits". In: *Proceedings of the 8th International Workshop on Boolean Problems*. IWSBP. Freiberg. Sept. 2008, pp. 1–15.

[203] M. D. Miller. "Lower Cost Quantum Gate Realizations of Multiple-Control Toffoli Gates". In: *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. PacRim. Aug. 2009, pp. 308–313.

[204] M. D. Miller. "Multiple-Valued Logic Design Tools". In: *Proceedings of the 23rd International Symposium on Multiple-Valued Logic*. ISMVL. Sacramento, CA, USA, May 1993, pp. 2–11.

[205] M. D. Miller and J. C. Muzio. "Spectral Fault Signatures for Single Stuck-At Faults in Combinational Networks". In: *IEEE Transactions on Computers* C-33.8 (1984), pp. 765–769.

[206] M. D. Miller and Z. Sasanian. "Improving the NCV Realization of Multiple-Control Toffoli Gates". In: *Proceedings of the 9th International Workshop on Boolean Problems*. IWSBP. Freiberg. Sept. 2010, pp. 37–44.

[207] M. D. Miller and Z. Sasanian. "Lowering the Quantum Gate Cost of Reversible Circuits". In: *Proceedings of the 53rd IEEE International Midwest Symposium on Circuits and Systems*. MWSCAS. Aug. 2010, pp. 260–263.

[208] M. D. Miller, R. Wille, and Z. Sasanian. "Elementary Quantum Gate Realizations for Multiple-Control Toffoli Gates". In: *Proceedings of the 41st IEEE International Symposium on Multiple-Valued Logic*. ISMVL. May 2011, pp. 288–293.

[209] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. The Springer International Series in Engineering and Computer Science. Springer, 1995. ISBN: 9780792396529. URL: `http://books.google.ee/books?id=byDkfhnJdz8C`.

[210] S. Minato. "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems". In: *Proceedings of the 30th International Design Automation Conference*. DAC. Dallas, TX, USA: ACM, 1993, pp. 272–277. ISBN: 0-89791-577-1.

[211] S. Minato, N. Ishiura, and S. Yajima. "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. DAC. June 1990, pp. 52–57.

[212] A. Mishchenko and R. K. Brayton. "Scalable Logic Synthesis Using a Simple Circuit Structure". In: *Proceedings of the International Workshop on Logic and Synthesis*. IWLS. Vail, CO, USA, June 2006, pp. 15–22.

[213] A. Mishchenko, S. Chatterjee, and R. K. Brayton. "DAG-Aware AIG Rewriting: a Fresh Look at Combinational Logic Synthesis". In: *Proceedings of the 43rd Annual Design Automation Conference*. DAC. San Francisco, CA, USA: ACM, 2006, pp. 532–535. ISBN: 1-59593-381-6.

[214] A. Mishchenko, S. Chatterjee, and R. K. Brayton. "Improvements to Technology Mapping for LUT-Based FPGAs". In: *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 240–253. ISSN: 0278-0070.

[215] A. Mishchenko et al. "Combinational and Sequential Mapping with Priority Cuts". In: *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*. ICCAD. San Jose, CA, USA: IEEE Press, 2007, pp. 354–361. ISBN: 1-4244-1382-6.

[216] A. Mishchenko et al. "Scalable Don't-care-based Logic Optimization and Resynthesis". In: *ACM Transactions on Reconfigurable Technology and Systems* 4.4 (Dec. 2011), 34:1–34:23. ISSN: 1936-7406.

[217] C. W. Moon et al. "Technology Mapping for Sequential Logic Synthesis". In: *Proceedings of the International Workshop on Logic and Synthesis*. IWLS. NC, USA, 1989.

[218] C. Moraga. "Hybrid GF(2)-Boolean Expressions for Quantum Computing Circuits". In: *Reversible Computation*. Vol. 7165. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 54–63.

[219] C. Moraga. "Mixed Polarity Reed-Muller Expressions for Quantum Computing Circuits". In: *Proceeding of the Workshop Reed Muller*. RM. Finland: TICSP-Press, 2011, pp. 119–125.

[220] B. Motik, P. F. Patel-Schneider, and B. Parsia. *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*. `http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/`. 2009.

[221] T. Murata. "Petri Nets: Properties, Analysis and Applications". In: *Proceedings of the IEEE* 77 (Apr. 1989), pp. 541–580.

[222] R. Murgai et al. "Logic synthesis for programmable gate arrays". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. DAC. Orlando, FL, USA: ACM, 1990, pp. 620–625. ISBN: 0-89791-363-9.

[223] D. Nardi and R. J. Brachman. "An Introduction to Description Logics". In: *The Description Logic Handbook: Theory, Implementation, and Applications*. Ed. by F. Baader et al. 2nd. Vol. 1. Cambridge University Press, 2003. Chap. 1.

[224] R. M. Nosofsky et al. "Comparing Models of Rule-based Classification Learning: A Replication and Extension of Shepard, Hovland, and Jenkins (1961)". In: *Memory and Cognition* 3.22 (1994), pp. 352–369.

[225] OEIS. *Number of distinct $n \times n$ $(0, 1)$ matrices after double sorting: by row, by column, by row*. The On-Line Encyclopedia of Integer Sequences™ (OEIS™). Feb. 2013. URL: `http://oeis.org/A089006`.

[226] A. Oliveira et al. "Exact Minimization of Binary Decision Diagrams Using Implicit Techniques". In: *IEEE Transactions on Computers* 47.11 (1998), pp. 1282–1296.

[227] N. Onizawa, V. C. Gaudet, and T. Hanyu. "Low-Energy Asynchronous Interleaver for Clockless Fully Parallel LDPC Decoding". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 58.8 (2011), pp. 1933–1943.

[228] OpenCores. *Open Source Hardware Community*. URL: `http://opencores.org`.

[229] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011. ISBN: 978-0123742605.

[230] J. Pardey. "Parallel Controller Synthesis for VLSI Applications". PhD thesis. University of Bristol, 1993.

[231] M. Perkowski et al. "Multi-Level Logic Synthesis Based on Kronecker Decision Diagrams and Boolean Ternary Decision Diagrams for Incompletely Specified Functions". In: *VLSI Design* 3.3–4 (1995), pp. 301–313.

[232] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[233] C. A. Petri. *Kommunikation mit Automaten*. Bonn: Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, 1962.

[234] N. Pippenger. *Theories of Computability*. Cambridge University Press, 1997.

[235] I. Pomeranz and S. M. Reddy. "On Generating Test Sets that Remain Valid in the Presence of Undetected Faults". In: *Proceedings of the Seventh Great Lakes Symposium on VLSI*. GLSVLSI. Mar. 1997, pp. 20–25.

[236] C. Pommerenke. "Über die Gleichverteilung von Gitterpunkten auf m-dimensionalen Ellipsoiden". In: *Acta Mathematica* 5 (1959), pp. 227–257.

[237] D. Popel and R. Drechsler. "Efficient Minimization of Multiple-Valued Decision Diagrams for Incompletely Specified Functions". In: *Proceedings of the 33rd International Symposium on Multiple-Valued Logic*. Tokyo, Japan, May 2003, p. 241.

[238] D. A. Pospelov. *Logical Methods of Analysis and Synthesis of Circuits*. (in Russian). Energia, Moscow, 1974.

[239] E. Post. "The Two-Valued Iterative Systems of Mathematical Logic". In: *Annals of Mathematical Studies* 5 (1941), pp. 1–122.

[240] C. Posthoff and B. Steinbach. *Logic Functions and Equations - Binary Models for Computer Science*. Dordrecht, The Netherlands: Springer, 2004. ISBN: 978-1-4020-2937-0.

[241] B. Preneel et al. "Propagation Characteristics of Boolean Functions". In: *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*. Vol. EUROCRYPT. Springer-Verlag, New York, NY, 1991, pp. 161–173.

[242] A. Puggelli et al. "Are Logic Synthesis Tools Robust?" In: *Proceedings of the 48th Design Automation Conference*. DAC. San Diego, CA, USA: ACM, 2011, pp. 633–638. ISBN: 978-1-4503-0636-2.

[243] J. Rabaey (moderator). "Beyond the Horizon: The Next 10x Reduction in Power - Challenges and Solutions". Plenary Technology Roundtable. Feb. 2011.

[244] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. 2nd. Prentice-Hall, 2003.

[245] A. Rafiev, J. Murphy, and A. Yakovlev. "Quaternary Reed-Muller Expansions of Mixed Radix Arguments in Cryptographic Circuits". In: *Proceedings of the 39th International Symposium on Multiple-Valued Logic, (ISMVL)*. ISMVL. IEEE Computer Society Washington, May 2009, pp. 370–376.

[246] M. Rahman et al. "Two-qubit Quantum Gates to Reduce the Quantum Cost of Reversible Circuit". In: *Proceedings of the 41st IEEE International Symposium on Multiple-Valued Logic*. ISMVL. Tuusulla. May 2011, pp. 86–92.

[247] A. Raychowdhury et al. "Computing with Subthreshold Leakage: Device/Circuit/Architecture Co-Design for Ultralow-Power Subthreshold Operation". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13.11 (2005), pp. 1213–1224.

[248] I. Reiman. "Ueber ein Problem von K. Zarankiewicz". In: *Acta Mathematica Academiae Scientiarum Hungaricae* 9 (3 1958), pp. 269–279. DOI: 10.1007/BF02020254.

[249] S. Roman. "A problem of Zarankiewicz". In: *Journal of Combinatorial Theory, Series A* 18.2 (1975), pp. 187–198. ISSN: 0097-3165. DOI: 10.1016/0097-3165(75)90007-2. URL: http://www.sciencedirect.com/science/article/pii/0097316575900072.

[250] O. S. Rothaus. "On 'Bent' Functions". In: *Journal of Combinatorial Theory*. Vol. 20. Ser. A, 1976, pp. 300–305.

[251] P. Ruch et al. "Automatic Medical Encoding with SNOMED Categories". In: *BMC Medical Informatics and Decision Making* 8.Suppl 1 (2008), S6. ISSN: 1472-6947. DOI: 10.1186/1472-6947-8-S1-S6. URL: http://www.biomedcentral.com/1472-6947/8/S1/S6.

[252] R. Rudell. "Dynamic Variable Ordering for Ordered Binary Decision Diagrams". In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD. Santa Clara, CA, USA: IEEE Computer Society Press, 1993, pp. 42–47. ISBN: 0-8186-4490-7.

[253] B. Y. Rytsar. "A New Approach to the Decomposition of Boolean Functions. 4. Non-Disjoint Decomposition: the Method of P, Q-Partitions". In: *Cybernetics and Systems Analysis* 45.3 (2009), pp. 340–364.

[254] M. Saeedi and I. L. Markov. "Synthesis and Optimization of Reversible Circuits - A Survey". In: *ACM Computing Surveys* (2013). URL: http://arxiv.org/abs/1110.2574.

[255] K. K. Saluja and E. H. Ong. "Minimization of Reed-Muller Canonical Expansion". In: *IEEE Transactions on Computers* C-28.7 (1979), pp. 535–537.

[256] A. Sarabi et al. "Minimal Multi-level Realization of Switching Functions Based on Kronecker Functional Decision Diagrams". In: *International Workshop on Logic & Synthesis*. 1993.

[257] Z. Sasanian and M. D. Miller. "Mapping a Multiple-Control Toffoli Gate Cascade to an Elementary Quantum Gate Circuit". In: *Proceedings of the 2nd International Workshop on Reversible Computation*. RC. Bremen. July 2010, pp. 83–90.

[258] Z. Sasanian and M. D. Miller. "Transforming MCT Circuits to NCVW Circuits". In: *Proceedings of the 3rd International Workshop on Reversible Computation*. Vol. LNCS 7165. RC. Gent, Belgium: Springer, 2012, pp. 77–88. ISBN: 978-3-642-29516-4.

[259] T. Sasao. "Index Generation Functions: Recent Developments". In: *International Symposium on Multiple-Valued Logic* (2011), pp. 1–9.

[260] T. Sasao. *Switching Theory for Logic Synthesis*. Kluwer Academic Publishers, 1999.

[261] T. Sasao and P. Besslich. "On the Complexity of mod-2 Sum PLAs". In: *IEEE Transactions on Computers* C-39.2 (1990), pp. 262–266.

[262] T. Sasao and J. T. Butler. "The Eigenfunction of the Reed-Muller Transformation". In: *Proceedings of the 10th Reed-Muller Workshop*. RM. May 2007.

[263] T. Sasao and M. Fujita. *Representations of Discrete Functions*. Kluwer Academic Publishers, 1996.

[264] T. Sasao and M. Matsuura. "BDD Representation for Incompletely Specified Multiple-Output Logic Functions and its Applications to Functional Decomposition". In: *Proceedings of the 42nd Design Automation Conference*. DAC. June 2005, pp. 373–378.

[265] T. Sasao and M. Fujita, eds. *Representations of Discrete Functions*. Norwell, MA, USA: Kluwer Academic Publishers, 1996. ISBN: 0792397207.

[266] H. Savoj and R. K. Brayton. "The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. DAC. Orlando, FL, USA: ACM, 1990, pp. 297–301. ISBN: 0-89791-363-9.

[267] C. Schlegel and L. Perez. *Trellis and Turbo Coding*. John Wiley and Sons, 2004.

[268] H. Schnoor. "Algebraic Techniques for Satisfiability Problems". PhD thesis. Gottfried Wilhelm Leibniz Universität Hannover, 2007.

[269] I. Schnoor. "The Weak Base Method for Constraint Satisfaction". PhD thesis. Gottfried Wilhelm Leibniz Universität Hannover, 2008.

[270] C. Scholl et al. "Minimizing ROBDD Sizes of Incompletely Specified Boolean Functions by Exploiting Strong Symmetries". In: *Proceedings of the 1997 European conference on Design and Test*. EDTC. 1997.

[271] N. O. Scott and G. W. Dueck. "Pairwise Decomposition of Toffoli Gates in a Quantum Circuit". In: *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. GLSVLSI. Orlando, FL, USA: ACM, 2008, pp. 231–236. ISBN: 978-1-59593-999-9.

[272] A. Sedra and K. C. Smith. *Microelectronic Circuits*. 6th. Oxford University Press, 2011.

[273] E. Sentovich et al. *SIS: A System for Sequential Circuit Synthesis*. Tech. rep. UCB/ERL M92/41. EECS Department, University of California, Berkeley, 1992.

[274] G. Shafat and I. Levin. "Recognition vs Reverse Engineering in Boolean Concepts Learning". In: *International Conference on Cognition and Exploratory Learning in the Digital Age*. CELDA. 2012, pp. 65–72.

[275] J. L. Shafer et al. "Enumeration of Bent Boolean Functions by Reconfigurable Computer". In: *18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines*. FCCM. Charlotte, NC: IEEE Computer Society, 2010, pp. 265–272. URL: http://faculty.nps.edu/butler/PDF/2010/Schafer_et_al_Bent.pdf.

[276] C. E. Shannon. "The Synthesis of Two-Terminal Switching Circuits". In: *The Bell Systems Technical Journal* 28 (1949), pp. 59–98.

[277] C. E. Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27 (July 1948), pp. 379–423, 623–656.

[278] T. Shiple et al. "Heuristic Minimization of BDDs Using Don't Cares". In: *Tech. Rept. M93/58*. EECS Department, University of California, Berkeley, 1993.

[279] T. Siegenthaler. "Correlation Immunity of Nonlinear Combining Functions for Cryptographic Applications". In: *IEEE Transactions on Information Theory*. Vol. IT-30(5). 1984, pp. 776–780.

[280] J. E. Smith. "On Necessary and Sufficient Conditions for Multiple Fault Undetectability". In: *IEEE Transactions on Computers* C-28.10 (Oct. 1979), pp. 801–802. ISSN: 0018-9340.

[281] M. Soeken et al. "Optimizing the Mapping of Reversible Circuits to Four-Valued Quantum Gate Circuits". In: *Proceedings of the 42nd IEEE International Symposium on Multiple-Valued Logic*. ISMVL. May 2012, pp. 173–178.

[282] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.4.1*.

[283] R. Sridhar and S. Iyengar. "Efficient Parallel Algorithms for Functional Dependency Manipulations". In: *Proceedings of the Second International Conference on Databases in Parallel and Distributed Systems*. ICPADS. Dublin, Ireland: ACM, 1990, pp. 126–137. ISBN: 0-8186-2052-8. DOI: 10.1145/319057.319078.

[284] A. Srinivasan et al. "Algorithms for Discrete Function Manipulation". In: *Digest of Technical Papers of the 1990 IEEE International Conference on Computer-Aided Design*. ICCAD. Nov. 1990, pp. 92–95.

[285] M. Stan and W. Burleson. "Bus-Invert Coding for Low-Power I/O". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3.1 (1995), pp. 49–58.

[286] M. Stanković, S. Stojković, and R. S. Stanković. "Representation of Incompletely Specified Binary and Multiple-Valued Logic Functions by Compact Decision Diagrams". In: *Proceedings of the International Symposium on Multiple-Valued Logic*. ISMVL. Victoria, BC, Canada, May 2012, pp. 142–147.

[287] R. S. Stanković. "Some Remarks on Basic Characteristics of Decision Diagrams". In: *Proceedings of the 4th International Workshop on Applications of Reed-Muller Expansion in Circuit Design*. RM. Victoria, BC, Canada, Aug. 1999, pp. 139–146.

[288] R. S. Stanković et al. "Circuit Synthesis from Fibonacci Decision Diagrams". In: *VLSI Design* 14.1 (2002), pp. 23–34.

[289] R. S. Stanković et al. "Progress in Applications of Boolean Functions". In: ed. by T. Sasao and J. T. Butler. Morgan & Claypool Publishers, 2010. Chap. Equivalence Classes of Boolean Functions, pp. 1–31.

[290] B. Steinbach and C. Posthoff. "An Extended Theory of Boolean Normal Forms". In: *Proceedings of the 6th Annual Hawaii International Conference on Statistics, Mathematics and Related Fields*. Honolulu, Hawaii, 2007, pp. 1124–1139.

[291] B. Steinbach and C. Posthoff. "Artificial Intelligence and Creativity - Two Requirements to Solve an Extremely Complex Coloring Problem". In: *Proceedings of the 5th International Conference on Agents and Artificial Intelligence*. Ed. by J. Filipe and A. Fred. Vol. 2. ICAART. Valencia, Spain, 2013, pp. 411–418. ISBN: 978-989-8565-39-6.

[292] B. Steinbach and C. Posthoff. "Search Space Restriction for Maximal Rectangle-Free Grids". In: *10^{th} International Workshop on Boolean Problems*. IWSBP. 2012.

[293] B. Steinbach and C. Posthoff. "Solution of the Last Open Four-Colored Rectangle-free Grid - an Extremely Complex Multiple-Valued Problem". In: *Proceedings of the IEEE 43nd International Symposium on Multiple-Valued Logic*. ISMVL. Toyama, Japan, 2013, pp. 302–309. ISBN: 978-0-7695-4976-7. DOI: 10.1109/ISMVL.2013.51.

[294] B. Steinbach and C. Posthoff. "The Solution of Ultra Large Grid Problems". In: *21st International Workshop on Post-Binary ULSI Systems*. Victoria, BC, Canada, May 2012, pp. 1–10.

[295] B. Steinbach, C. Posthoff, and W. Wessely. "Approaches to Shift the Complexity Limitations of Boolean Problems". In: *Proceedings of the Seventh International Conference on Computer Aided Design of Discrete Devices*. CAD DD. Minsk, Belarus, Nov. 2010, pp. 84–91. ISBN: 978-985-6744-63-4.

[296] B. Steinbach, W. Wessely, and C. Posthoff. "Several Approaches to Parallel Computing in the Boolean Domain". In: *1st International Conference on Parallel, Distributed and Grid Computing*. PDGC. Jaypee University of Information Technology Waknaghat, Solan, H.P., India, Oct. 2010, pp. 6–11. ISBN: 978-1-4244-7672-5.

[297] B. Steinbach, ed. *10th International Workshop on Boolean Problems*. IWSBP. Freiberg, Germany, Sept. 2012. ISBN: 978-3-86012-438-3.

[298] B. Steinbach. "XBOOLE - A Toolbox for Modelling, Simulation, and Analysis of Large Digital Systems". In: *System Analysis and Modeling Simulation* 9.4 (1992), pp. 297–312.

[299] B. Steinbach and C. Posthoff. "Boolean Differential Calculus". In: *Progress in Applications of Boolean Functions*. San Rafael, CA, USA: Morgan & Claypool Publishers, 2010, pp. 55–78.

[300] B. Steinbach and C. Posthoff. "Boolean Differential Calculus - Theory and Applications". In: *Journal of Computational and Theoretical Nanoscience* 7.6 (2010), pp. 933–981. ISSN: 1546-1955.

[301] B. Steinbach and C. Posthoff. *Boolean Differential Equations*. Morgan & Claypool Publishers, June 2013. ISBN: 978-1627052412. DOI: 10.2200/S00511ED1V01Y201305DCS042.

[302] B. Steinbach and C. Posthoff. *Logic Functions and Equations - Examples and Exercises*. Springer Science + Business Media B.V., 2009. ISBN: 978-1-4020-9594-8.

[303] G. Strang. *Introduction to Linear Algebra (3rd ed.)* Wellesley, Massachusetts: Wellesley-Cambridge Press, 2003, pp. 74–76.

[304] V. Strassen. "Gaussian Elimination is Not Optimal". In: *Numerical Mathematics* 13 (1969), pp. 354–356.

[305] V. P. Suprun. "A Table Method for Polynomial Decomposition of Boolean Functions". In: *Kibernetika* 1 (1987). (in Russian), pp. 116–117.

[306] V. P. Suprun. "Fixed Polarity Reed-Muller Expressions of Symmetric Boolean Functions". In: *Workshop on Applications of the Reed-Muller Expansion in Circuit Design*. RM. 1995, pp. 246–249.

[307] V. P. Suprun. "Method of the Conversion DNF of the Boolean Functions to Canonical Zhegalkin Polynomial". In: *Automatika i vichislitelnaia technika* 2 (1984). (in Russian), pp. 78–81.

[308] V. P. Suprun. "Polynomial Expression of Symmetric Boolean Functions". In: *Izvestia akademii nauk SSSR. Tekhicheskaia Kibernetika* 4 (1985). (in Russian), pp. 123–127.

[309] *Synopsys Power Compiler*. URL: `http://www.synopsys.com/tools/implementation/rtlsynthesis/pages/powercompiler.aspx`.

[310] M. Szpyrka. *Petri Nets in Modeling and Analysis of Concurrent Systems*. Polish. Warszawa: WNT, 2008.

[311] M. Szyprowski and P. Kerntopf. "A Study of Optimal 4-Bit Reversible Circuit Synthesis from Mixed-Polarity Toffoli Gates". In: *Proceedings of the 12th IEEE Conference on Nanotechnology*. IEEE-NANO. Aug. 2012.

[312] M. Szyprowski and P. Kerntopf. "An Approach to Quantum Cost Optimization in Reversible Circuits". In: *Proceedings of the 11th IEEE Conference on Nanotechnology*. IEEE-NANO. Aug. 2011, pp. 1521–1526.

[313] M. Szyprowski and P. Kerntopf. "Optimal 4-Bit Reversible Mixed-Polarity Toffoli Circuits". In: *Proceedings of the 4th International Workshop on Reversible Computation*. RC. Copenhagen, Denmark: Springer, 2012, pp. 138–151. ISBN: 978-3-642-36314-6.

[314] M. Szyprowski and P. Kerntopf. "Reducing Quantum Cost in Reversible Toffoli Circuits". In: *Proceedings of the 10th Reed-Muller Workshop*. RM. 2011, pp. 127–136. eprint: `1105.5831`. URL: `http://arxiv.org/abs/1105.5831`.

[315] S. S. Tehrani, S. Mannor, and W. Gross. "Fully Parallel Stochastic LDPC Decoders". In: *IEEE Transactions on Signal Processing* 56.11 (2008), pp. 5692–5703.

[316] A. Thayse and M. Davio. "Boolean Differential Calculus and its Application to Switching Theory". In: *IEEE Transactions on Computers* C-22.4 (Apr. 1973), pp. 409–420.

[317] J. Tkacz and M. Adamski. "Wyznaczanie SM - pokrycia bezpiecznej Sieci Petriego Metoda Komputerowego Wnioskowania". In: *Pomiary, Automatyka, Kontrola* (Nov. 2011), pp. 1397–1400.

[318] T. Toffoli. "Reversible Computing". In: *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. ICALP. London, UK, UK: Springer-Verlag, 1980, pp. 632–644.

[319] C.-C. Tsai and M. Marek-Sadowska. "Boolean Function Classification via Fixed-Polarity Reed-Muller Forms". In: *IEEE Transactions on Computers* C-46.2 (1997), pp. 173–186.

[320] Z. Tu and Y. Deng. *Algebraic Immunity Hierarchy of Boolean Functions*. 2007. URL: `http://eprint.iacr.org/2007/259.pdf`.

[321] R. Ubar. "Complete Test Pattern Generation for Combinational Networks". In: *Proceedings Estonian Academy of Sciences, Physics and Mathematics* 4 (1982). (in Russian), pp. 418–427.

[322] R. Ubar. "Fault Diagnosis in Combinational Circuits by Solving Boolean Differential Equations". In: *Automatics and Telemechanics* 11 (1979). (in Russian), pp. 170–183.

[323] R. Ubar, S. Kostin, and J. Raik. "Multiple Stuck-at-Fault Detection Theorem". In: *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. DDECS. Apr. 2012, pp. 236–241.

[324] R. Ubar. "Test Generation for Digital Circuits Using Alternative Graphs". Russian. In: *Proceedings of Tallinn Technical University* 409 (1976), pp. 75–81.

[325] R. Ubar. "Test Synthesis with Alternative Graphs". In: *IEEE Design and Test of Computers* 13.1 (1996), pp. 48–57.

[326] R. Ubar, S. Kostin, and J. Raik. "About Robustness of Test Patterns Regarding Multiple Faults". In: *13th Latin American Test Workshop*. LATW. Apr. 2012, pp. 1–6.

[327] R. Ubar et al. "Structural Fault Collapsing by Superposition of BDDs for Test Generation in Digital Circuits". In: *2010 11th International Symposium on Quality Electronic Design*. ISQED. Mar. 2010, pp. 250–257.

[328] R. Vigo. "Categorical Invariance and Structural Complexity in Human Concept Learning". In: *Journal of Mathematical Psychology* 53 (2009), pp. 203–221.

[329] A. Viterbi. "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm". In: *IEEE Transactions on Information Theory* 13.2 (1967), pp. 260–269.

[330] H. Vollmer. *Introduction to Circuit Complexity*. Springer, 1999.

[331] A. Wang and A. Chandrakasan. "A 180mV FFT Processor Using Subthreshold Circuit Techniques". In: *IEEE International Solid-State Circuits Conference*. ISSCC. Feb. 2004, pp. 292–293.

[332] I. Webster and S. E. Tavares. "On the Design of S-Boxes". In: *Advances in Cryptology - CRYPTO'85*. Vol. 218. Springer-Verlag, Berlin, Germany, 1986, pp. 523–534.

[333] M. Werner. *Algorithmische Betrachtungen zum Zarankiewicz Problem*. Seminararbeit, TU Freiberg, Germany. May 2012. URL: http://11235813tdd.blogspot.de/2012/02/zarankiewicz-problem.html.

[334] N. Weste and D. Harris. *CMOS VLSI Design - A Circuits and Systems Perspective*. 4th. Boston: Pearson Education, 2011.

[335] Wikipedia. *Grigorchuk Group*. 2013. URL: http://en.wikipedia.org/wiki/Grigorchuk_group.

[336] Wikipedia. *Growth Rate (Group Theory)*. 2013. URL: http://en.wikipedia.org/wiki/Growth_rate_(group_theory).

[337] R. Wille and R. Drechsler. *Towards a Design Flow for Reversible Logic*. Dordrecht: Springer, 2010.

[338] R. Wille, M. Saeedi, and R. Drechsler. "Synthesis of Reversible Functions Beyond Gate Count and Quantum Cost". In: *Proceedings of the 18th International Workshop on Logic and Synthesis*. IWLS. Berkeley. Aug. 2009, pp. 43–49.

[339] C. Winstead et al. "An Error Correction Method for Binary and Multiple-Valued Logic". In: *IEEE International Symposium on Multiple-Valued Logic*. ISMVL. May 2011, pp. 105–110.

[340] M. Wiśniewska, R. Wiśniewski, and M. Adamski. "Usage of Hypergraph Theory in Decomposition of Concurrent Automata". In: *Pomiary, Automatyka, Kontrola* (July 2007), pp. 66–68.

[341] M. Wiśniewska. *Application of Hypergraphs in Decomposition of Discrete Systems*. Lecture Notes in Control and Computer Science, Vol. 23. Zielona Góra: University of Zielona Góra Press, 2012, p. 143.

[342] S. Wolfram. *Echelon Form*. 2007. URL: http://eprint.iacr.org/2007/259.pdf.

[343] S. Yamashita, S. Minato, and M. D. Miller. "Synthesis of Semiclassical Quantum Circuits". In: *Proceedings of the 2nd International Workshop on Reversible Computation*. RC. Bremen. July 2010, pp. 93–99.

[344] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide*. Tech. rep. Technical Report 1991 IWLS-UG-Saeyang, MCNC, Jan. 1991.

[345] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0*. User Guide. Microelectronic Center, 1991.

[346] S. Yanushkevich et al. *Decision Diagram Technique for Micro - and Nanoelectronic Design*. CRC Press, Taylor & Francis, Boca Raton, London, New York, 2006.

[347] T. Yasufuku et al. "Difficulty of Power Supply Voltage Scaling in Large Scale Subthreshold Logic Circuits". In: *IEICE Transactions on Electronics* E93-C.3 (2010), pp. 332–339.

[348] Z. Ying and L. Yanjuan. "A Multiple Faults Test Generation Algorithm Based on Neural Networks and Chaotic Searching for Digital Circuits". In: *2010 International Conference on Computational Intelligence and Software Engineering*. CiSE. Dec. 2010, pp. 1–3.

[349] A. D. Zakrevskij. *Parallel Algorithms of Logical Control*. (in Russian). Moscow: Second edition, URSS, 2003.

[350] A. D. Zakrevskij, Y. Pottosin, and L. Cheremisinova. *Design of Logical Control Devices*. Tallinn: TUT Press, 2009.

[351] Z. Zilic and Z. G. Vranesic. "Using Decision Diagrams to Design ULMs for FPGAs". In: *IEEE Transactions on Computers* C-47.9 (1998), pp. 971–982.

[352] K. Zuse. *Der Computer, mein Lebenswerk (The Computer, My Life's Work)*. Springer, Berlin, 1984, pp. 1–220. ISBN: 978-3-540-13814-3.

List of Authors

Jaakko Astola
Department of Signal Processing
Tampere University of Technology
Tampere, Finland
E-Mail: Jaakko.Astola@tut.fi

Anna Bernasconi
Department of Computer Science
Università di Pisa
Pisa, Italy
E-Mail: annab@di.unipi.it

Ariel Burg
Faculty of Engineering
Bar-Ilan University
Ramat-Gan, Israel
E-Mail: burgariel@gmail.com

Jon T. Butler
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California, U.S.A.
E-Mail: jbutler@nps.edu

Valentina Ciriani
Department of Computer Science
Università degli Studi di Milano
Milano, Italy
E-Mail: valentina.ciriani@unimi.it

Alexis De Vos
Elektronika en informatiesystemen
Universiteit Gent
Gent, Belgium
E-Mail: alex@elis.UGent.be

Petr Fišer
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
E-Mail: fiserp@fit.cvut.cz

Vincent C. Gaudet
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
E-Mail: vcgaudet@ecemail.uwaterloo.ca

Danila A. Gorodecky
United Institute of Informatic Problems
National Acacemy of Science of Belarus
Belarusian State University
Minsk, Belarus
E-Mail: danila.gorodecky@gmail.com

Andrei Karatkevich
Institute of Computer Engineering and Electronics
University of Zielona Góra
Zielona Góra, Poland
E-Mail: A.Karatkevich@iil.uz.zgora.pl

Osnat Keren
Faculty of Engineering
Bar-Ilan University
Ramat-Gan, Israel
E-Mail: osnat.keren@biu.ac.il

Pawel Kerntopf
Warsaw University of Technology
Warsaw, Poland
E-Mail: p.kerntopf@ii.pw.edu.pl

Ilya Levin
Science and Technology Education Department
Tel-Aviv University
Tel-Aviv, Israel
E-Mail: ilia1@post.tau.ac.il

M. Eric McCay
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California, U.S.A.
E-Mail: vobis132@gmail.com

Arne Meier
Institut für Theoretische Informatik
Leibniz Universität
Hannover, Germany
E-Mail: meier@thi.uni-hannover.de

Claudio Moraga
Unit of Fundamentals of Soft Computing
European Centre for Soft Computing
Mieres, Asturias, Spain
E-Mail: claudio.moraga@softcomputing.es

Christian Posthoff
Department of Computing and Information Technology
The University of the West Indies
Trinidad & Tobago
E-Mail: christian@posthoff.de

Hillel Rosensweig
Science and Technology Education Department
School of Education
Tel-Aviv University
Tel-Aviv, Israel
E-Mail: hillelro@yahoo.com

Jan Schmidt
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
E-Mail: schmidt@fit.cvut.cz

Gabi Shafat
Science and Technology Education Department
School of Education
Tel-Aviv University
Tel-Aviv, Israel
E-Mail: gash69@gmail.com

Pantelimon Stănică
Department of Applied Mathematics
Naval Postgraduate School
Monterey, California, U.S.A.
E-Mail: psanica@nps.edu

Milena Stanković
Department of Computer Science
University of Niš
Niš, Serbia
E-Mail: Milena.Stankovic@elfak.ni.ac.r40s

Radomir S. Stanković
Department of Computer Science
University of Niš
Niš, Serbia
E-Mail: Radomir.Stankovic@gmail.com

Stanislav Stanković
Department of Signal Processing
Tampere University of Technology
Tampere, Finland
E-Mail: Stanislav.Stankovic@tut.fi

Bernd Steinbach
Institute of Computer Science
Freiberg University of Mining and Technology
Freiberg, Germany
E-Mail: steinb@informatik.tu-freiberg.de

Suzana Stojković
Department of Computer Science
University of Niš
Niš, Serbia
E-Mail: Suzana.Stojkovic@elfak.ni.ac.rs

Valery P. Suprun
United Institute of Informatic Problems
National Acacemy of Science of Belarus
Belarusian State University
Minsk, Belarus
E-Mail: suprun@bsu.by

Marek Szyprowski
Warsaw University of Technology
Warsaw, Poland
E-Mail: m.szyprowski@ii.pw.edu.pl

Gabriella Trucco
Department of Computer Science
Università degli Studi di Milano
Milano, Italy
E-Mail: Gabriella.Trucco@unimi.it

Raimund Ubar
Tallinn University of Technology
Tallinn, Estonia
E-Mail: raiub@pld.ttu.ee

Steven Vandenbrande
Elektronika en informatiesystemen
Universiteit Gent
Gent, Belgium
E-Mail: Steven.Vandenbrande@UGent.be

Raphaël Van Laer
Elektronika en informatiesystemen
Universiteit Gent
Gent, Belgium
E-Mail: Raphael.VanLaer@UGent.be

Matthias Werner
Institute of Computer Science
Freiberg University of Mining and Technology
Freiberg, Germany
E-Mail: wmatthias@t-online.de

Remigiusz Wiśniewski
University of Zielona Góra
Zielona Góra, Poland
E-Mail: R.Wisniewski@iil.uz.zgora.pl

Index of Authors

Index

F

G

H

O

P

Q

R

S

T