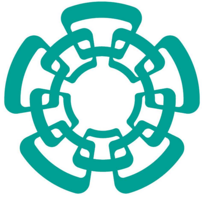


Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Laboratorio de Tecnologías de la Información

Un Algoritmo de Branch & Bound para Construir Pequeñas Instancias de Covering Arrays Binarios de Fuerza Variable

Tesis que presenta:

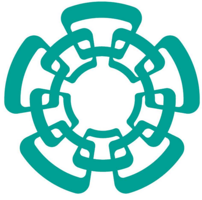
Josué Emmanuel Bracho Ríos

Para obtener el grado de:

**Maestro en Ciencias
en Computación**

Dr. José Torres Jiménez, Director
Dr. Eduardo A. Rodríguez Tello, Co-Director

Cd. Victoria, Tamaulipas, México. Enero, 2010



Research Center for Advanced Studies
of the National Polytechnic Institute

Information Technology Laboratory

A Branch & Bound Algorithm to Construct Small Instances of Binary Covering Arrays of Variable Strength

Thesis by:

Josué Emmanuel Bracho Ríos

as the partial fulfillment of the
requirement for the degree of:

**Master of Science
in Computer Science**

Dr. José Torres Jiménez, Director
Dr. Eduardo A. Rodríguez Tello, Co-Director

Cd. Victoria, Tamaulipas, México. January, 2010

© Copyright by
Josué Emmanuel Bracho Ríos
2010

This research was partially funded by the following projects: CONACyT 58554-Cálculo de Covering Arrays, 99276-Algoritmos para la canonización de Covering Arrays, 51623-Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas.

The thesis of Josué Emmanuel Bracho Ríos was approved by:

Dr. Gabriel Ramírez Torres

Dr. Manuel Valenzuela Rendón

Dr. José Torres Jiménez, Committte Chair

Dr. Eduardo A. Rodríguez Tello, Committte Co-chair

Cd. Victoria, Tamaulipas, México., January 15 2010

My family for their unconditional support, with all my love.
My friends for supporting me when there were difficult times.
And my advisors for all the help throughout the development of this research.

Acknowledgements

- I acknowledge to my family for their infinite comprehension and unconditional support, for all the advises and encourages to go ahead.

- I thank my friends that always encourage me to continue and prayed for me.

- I would like to thank to my supervisor and co-advisor, Dr. José Torres Jiménez and Eduardo A. Rodríguez Tello, for their support and advises for the development of this thesis.

- This work has been carried out with the economic support of the CONACyT and the administrative and technical support of the Laboratory of Information Technology (Cinvestav - Tamaulipas). Thanks specially to Dr. Arturo Díaz for giving me the opportunity to belong to this institution.

- I would also thank to CONACyT for their help as this work was partially funded by the following projects: CONACyT 58554-Cálculo de Covering Arrays, 99276-Algoritmos para la canonización de Covering Arrays, 51623-Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas.

Without the support of one or another way of all these people I could not have been able to carry out this work. To all them, many thanks.

Contents

Contents i

List of Figures v

List of Tables vii

Publications ix

Resumen xi

Abstract xiii

1 Introduction 1
- 1.1 Research Context . . . 1
  - 1.1.1 Relevance of Software Testing . . . 1
  - 1.1.2 Types of Software Testing . . . 4
  - 1.1.3 Interaction Testing . . . 6
    - 1.1.3.1 Code Coverage . . . 7
    - 1.1.3.2 Minimum Number of Tests . . . 8
    - 1.1.3.3 Types of Combinatorial Objects . . . 9
- 1.2 Research Goals and Contributions . . . 10
- 1.3 Thesis Overview . . . 11
- 1.4 Summary of the Chapter . . . 12

2 State of the art 13
- 2.1 Definitions and Examples . . . 13
  - 2.1.1 Latin Squares . . . 13
  - 2.1.2 Orthogonal Latin Squares . . . 14
  - 2.1.3 Orthogonal Arrays . . . 15
  - 2.1.4 Covering Arrays . . . 16
    - 2.1.4.1 Isomorphic Covering Arrays . . . 18
  - 2.1.5 Mixed Covering Arrays . . . 19
- 2.2 Covering Arrays Construction Methods . . . 21
  - 2.2.1 Constructing Optimal Covering Arrays within Polynomial Time . . . 22
    - 2.2.1.1 Case $t = 2$ . . . 23
    - 2.2.1.2 Case where $v = p^\alpha$ . . . 23
  - 2.2.2 Algebraic Transformations . . . 23
    - 2.2.2.1 TConfig . . . 23
    - 2.2.2.2 Combinatorial Test Services . . . 24

2.2.3 Deterministic Strategies . . . 24
2.2.3.1 Automatic Efficient Test Generator . . . 24
2.2.3.2 In Parameter Order (IPO) . . . 25
2.2.3.3 Test Case Generation . . . 25
2.2.3.4 Deterministic Density Algorithm . . . 26
2.2.4 Non-Deterministic Algorithms . . . 27
2.2.4.1 Simulated Annealing . . . 27
2.2.4.2 Tabu Search . . . 29
2.2.4.3 Genetic Algorithms . . . 30
2.2.4.4 Ant Colony Algorithm . . . 31
2.2.5 Other Methods . . . 32
2.2.5.1 Branch & Bound . . . 33
2.2.5.2 EXACT Method . . . 34
2.2.5.3 Constraint Programming . . . 34
2.3 Summary of the Chapter . . . 35

3 Methodology **37**
3.1 A New Backtracking Algorithm (NBA) . . . 37
3.2 Techniques for Improving the Efficiency of the Search . . . 38
3.2.1 Symmetry Breaking Techniques . . . 39
3.2.2 Partial t-Wise Verification . . . 40
3.2.3 Fixed Block . . . 42
3.3 Improved Backtracking Algorithm (IBA) . . . 43
3.4 Special Cases for the IBA . . . 45
3.5 Summary of the Chapter . . . 49

4 Experimental Results **55**
4.1 Computational Results . . . 55
4.1.1 Test Instances . . . 55
4.1.2 Comparative Criteria . . . 56
4.1.3 Comparison Between NBA and EXACT . . . 56
4.1.4 Comparison Between our IBA and the Exact . . . 58
4.1.5 Comparison Between IBA and IPOG-F . . . 59
4.1.6 Comparison Between NBA and IBA . . . 60
4.1.7 Comparison Against the Best Known Solutions . . . 61
4.2 Summary of the Chapter . . . 62

5 Conclusions and Future work **65**
5.1 Summary of the Thesis . . . 65
5.2 Discussion of the Methodology . . . 66
5.3 Challenges Confronted in the Development of this Thesis . . . 67
5.4 Scope of the Proposed Approach . . . 68
5.5 Future work . . . 68

Bibliography **71**

List of Figures

1.1 A basic work-flow of the software testing process [39] 3
1.2 The black-box testing diagram. 5
1.3 Fault detection at interaction levels 1 through 6 according to the type of application [3] 8

2.1 Classification scheme for combination strategies for constructing covering arrays . . 22
2.2 An example of the behavior of real ants choosing a road 32

3.1 Flow chart of the new backtracking algorithm. 50
3.2 Flow chart of the improved backtracking algorithm. 53

4.1 Graphic of performance between NBA and EXACT 57
4.2 Graphic of performance between IBA and EXACT 59
4.3 Graphic of performance between IBA and IPOG-F 61
4.4 Graphic of performance between IBA and NBA 63

List of Tables

1.1 System with 4 components 6
1.2 Percentage fault detection at interaction levels 1 through 6 according to the type of application 7
1.3 An example of a pairwise coverage 9
1.4 Number of tests required for diferent interaction levels 9

2.1 A simple latin square example of 3 rows and 3 columns 14
2.2 An example of two different latin squares of 3 rows and 3 columns 14
2.3 The configuration of two mutual orthogonally latin squares of size 3 15
2.4 Example of an OA_2(8;5,2,2) 16
2.5 Table summarizing the difference of required tests between OAs and CAs 16
2.6 Example of an CA(10;5,2,3) 18
2.7 An example of a CA(5;2,4,2) 19
2.8 An isomorphic CA(5;2,4,2) with permuted rows and columns based on table 2.7 . . 19
2.9 Example of an MCA(15;4,$\{2^2, 3^1, 5^1\}$,2) 20

3.1 An example of how to construct isomorphic covering arrays 39
3.2 Example of the current partial solution M after inserting 4 columns 40
3.4 Example of backtracks when the current column in the partial solution M is not a covering array 41
3.5 A CA(8;3,3,2) example. 42
3.7 The scheme of the equal counters of table 3.6. 44
3.9 Next valid move for the 4th column of the partial solution M when the 0 corresponds to a block of equal rows 46
3.10 Next valid move for the 4th column of the partial solution M when the zero moved is not the last one and belongs to a block of different rows 47
3.11 Next valid move for the 4th column of the partial solution M when the zero moved is not the last one and belongs to a block of equal rows 48
3.3 Example of invalid moves in the partial solution M when the rows does not remain ordered 51
3.6 Example of the initialization of the IBA using a CA(12;3,3,2). 52
3.8 Next valid move for the 4th column of the partial solution M 52

4.1 Performance comparison between NBA and EXACT. 56
4.2 Performance comparison between the improved B&B and EXACT. 58
4.3 Performance comparison between the algorithms IBA and IPOG-F. 60
4.4 Performance comparison between the IBA and the NBA. 62
4.5 Lower and upper bounds for binary alphabet 63

Publications

Josué Bracho-Ríos, José Torres-Jiménez and Eduardo Rodriguez-Tello. *A New Backtracking Algorithm for Constructing Binary Covering Arrays of Variable Strength*, in 8th Mexican International Conference on Artificial Intelligence, (MICAI 2009), Springer, Guanajuato, Mexico, November 2009.

Nelson Rangel-Valdez, José Torres-Jiménez, Josué Bracho-Ríos and Pedro Quiz-Ramos. *Problem and Algorithm Fine-Tuning a case of study using Bridge Club and Simulated Annealing*, in International Conference on Evolutionary Computation (ICEC 2009), Madeira, Portugal, October 2009.

Resumen

Un Algoritmo de Branch & Bound para Construir Pequeñas Instancias de Covering Arrays Binarios de Fuerza Variable

por

Josué Emmanuel Bracho Ríos
Maestro en Ciencias del Laboratorio de Tecnologías de la Información
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2010
Dr. José Torres Jiménez, Director
Dr. Eduardo A. Rodríguez Tello, Co-Director

Los Covering Arrays son estructuras combinatorias ampliamente usadas en el proceso de las pruebas de software. Un Covering Array puede ser usado como un conjunto de pruebas combinatorio en el cual todas las combinaciones de valores de cada t columnas (donde t es el nivel de interacción) están presentes. Los Covering Arrays son una alternativa muy útil cuando una prueba exhaustiva no es posible [32].

En este documento se propone el desarrollo de un algoritmo completo para construir Covering Arrays binarios de fuerza variable siempre y cuando mantengan la restricción de que el número de símbolos por columna se encuentren balanceados. Este algoritmo, basado en la técnica de Branch & Bound (B&B), garantiza siempre encontrar los Covering Arrays Óptimos con símbolos balanceados si existen o probar su no existencia para casos pequeños, cuando no hay restricciones de tiempo impuestas.

La efectividad del algoritmo de B&B propuesto es medida contra los mejores métodos completos para resolver el problema de construcción de Covering Arrays reportados en la literatura.

En cuanto a los resultados obtenidos, nuestro algoritmo logró encontrar las mejores cotas establecidas hasta el momento para el conjunto de instancias pequeñas tomadas de la literatura, así como ratificar que no existe un Covering Array con un número de símbolos balanceados por columna para otros casos. En términos del esfuerzo computacional, nuestro enfoque logró mejorar

en algunos casos al algoritmo EXACT.

Abstract

A Branch & Bound Algorithm to Construct Small Instances of Binary Covering Arrays of Variable Strength

by

Josué Emmanuel Bracho Ríos
Master of Science in Information Technology Laboratory
Research Center for Advance Study from the National Polytechnic Institute, 2010
Dr. José Torres Jiménez, Advisor
Dr. Eduardo A. Rodríguez Tello, Co-advisor

Covering Arrays are combinatorial structures extensively used in the software testing process. A Covering Array can be used as a combinatorial test suite in which all the combinations of the values of every t columns (where t is the level of interaction) are listed. Covering Arrays are a very useful alternative when exhaustive testing is not feasible [32].

In this document the development of a complete algorithm to construct binary covering arrays of variable strength is proposed. This algorithm, based on the Branch & Bound (B&B) technique, guarantees always to discover optimal Covering Arrays with a balanced number of symbols per column if they exist or prove their nonexistence for small instances, given that there are no computer time restrictions imposed.

The effectiveness of the proposed B&B algorithm is assessed against the best complete methods for solving the Covering Array construction problem reported in the literature.

With respect of the results obtained, our algorithm was able to obtain the best results reported for the set of small instances taken from the literature, as well as to corroborate that there is no Covering Array with a number of balanced symbols per column for other cases. In terms of computational effort, our approach was able to improve in some cases with respect to the EXACT algorithm.

1
Introduction

In this chapter we present in detail the importance of the development of this thesis, as well as an introduction to some terminology used in this manuscript. This chapter is divided in two main sections : research context, and goals and contributions. In research context some general aspects of the field, that this thesis is mainly related to, are described in order to have a better understanding of the context of this work. In the section "goals and contributions" the studies that were carried on in order to fulfill this work and the relevance of developing this thesis are explained. Finally a summary of this chapter is presented.

1.1 Research Context

1.1.1 Relevance of Software Testing

Within the recent years the use of software has become more and more important in our society [52]. Most of the enterprises nowadays depend on computer software, thus a failure on it can be catastrophic. As mentioned in a report of the National Institute of Standards and Technology (NIST),

the sales of software have reached approximately $180 billion dollars generating a significant and high-paid workforce, composed of 697,000 software engineers and 585,000 computer programmers. It shows clearly that the software industry is a really important part of the economy, moreover if the software presents errors, millionaire losses can occur affecting the economy. According to Hartman [22], the quality of the software relies strongly on the use of software testing.

Even though the use of software can give us a lot of benefits, an inadequate software testing process can lead to millionaire losses, or in other cases, where the software is applied for some machines with critical purposes, a software exception can cause more than monetary loss. An example is the Therac 25 crash [36], where it injured many patients and caused the death of some of them.

Figure 1.1 [39] represents a software test work-flow, in which 3 processes are shown, the informal iteration process, the formal iteration process and the In-Stage assessment process. The actors in the software testing process are:

- Software Developers
- Primary Developer Representative
- Testing Personnel

In Figure 1.1 we can clearly see two stages where tests are applied to the software, first a test is done by the local programmers, if it is not accepted, then a Test Incident Report (TIR) is created and the necessary updates to the program are elaborated, if it is accepted, then a second test is made by testers outside the software development team, at this stage a TIR is also developed for every test process that fails, and a Test Completion Report (TCR) is created for each test that is marked as ok. The acceptance test process is complete when a set of TCRs matching all tests identified in the Acceptance Test procedures of the module test plan have been generated and signed.

Considering the testing process in Figure 1.1, we can see that the software test process is an essential part for the software development process, and if the time and cost of the tests can be reduced, then software releases can be made quicker and with better quality.

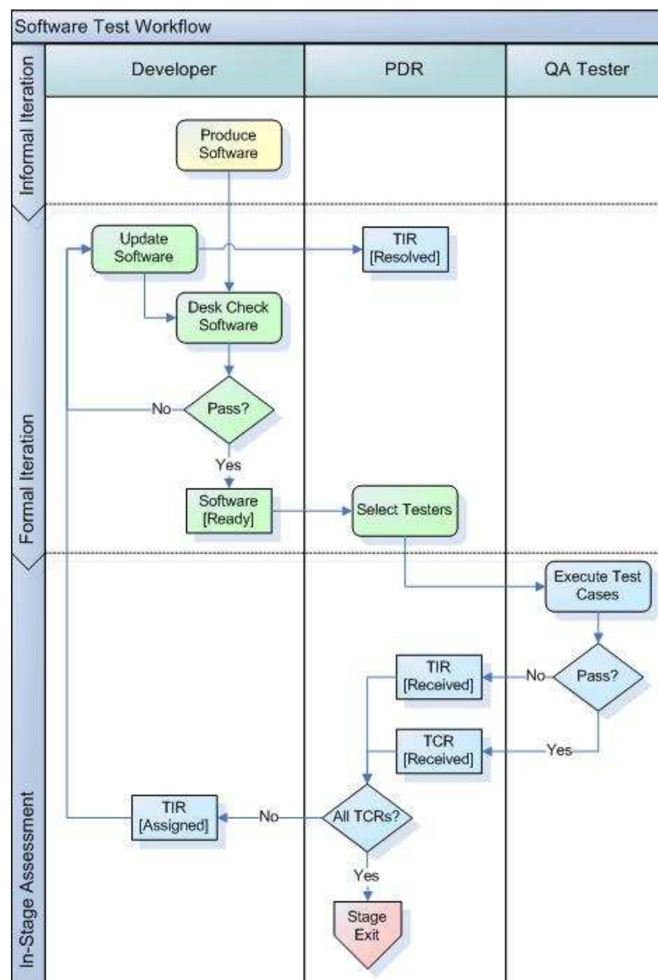


Figure 1.1: A basic work-flow of the software testing process [39]

We can conclude then that the software tests are an indispensable part of the software development process and it should not be taken lightly. Therefore, multiple techniques for software testing had been developed in order to improve the quality of the software. In the next section a general overview of the distinct types of software testing and their test design techniques are described.

1.1.2 Types of Software Testing

Two main types of software testing are reported: *White-box testing* and *Black-box testing*. The White-box testing (a.k.a. clear box testing, glass box testing, transparent box testing, translucent box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all execution paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs. In electronic hardware testing, every node in a circuit may be tested and measured; an example is in-circuit testing (ICT).

Since the tests are based on the actual implementation, if the implementation changes, the tests probably will need to change too. For example, ICT tests needs updates if component values change, and needs a whole new test if the circuit changes. This adds financial resistance to the change process, thus buggy products may stay buggy. Automated optical inspection (AOI) offers similar component level correctness checking without the cost of ICT fixtures, however changes still require test updates.

While White-box testing is applicable at the unit, integration and system levels of the software testing process, it is typically applied to the unit level testing. While it normally tests paths within a unit, it can also test paths between units during integration, and between subsystems during a system level test. Though this type of test design can uncover an overwhelming number of bugs, it might not detect unimplemented parts of the specification or missing requirements, but one can be sure that all paths through the test object are executed.

Typical white box test design techniques include:

- Control-Flow testing
- Data flow testing
- Branch testing
- Path testing

Figure 1.2: The black-box testing diagram.

On the contrary, Black-box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure. In Figure 1.2 a diagram of the Black-box testing strategy is represented.

This method of test design is applicable to all levels of software testing: unit testing, integration testing, functional testing, system testing and acceptance testing. The higher the level, and hence the bigger and more complex the box, the more one is forced to use Black-box testing to simplify the testing process. Even when this method can uncover unimplemented parts of the specification, one cannot be sure that all existent paths are tested.

There are some typical Black-box testing techniques as:

- Equivalence partitioning
- Boundary value analysis
- Decision table testing
- Interaction testing
- State transition tables
- Use case testing
- Cross-functional testing

This thesis is mainly related with Black-box testing and more specifically with interaction testing. In the following subsection the general aspects of the interaction testing, their advantages and a general definition of system coverage are detailed.

1.1.3 Interaction Testing

Many software systems today are built using components. Often, system faults are caused by an unexpected interaction among these [31]. Suppose we have an e-commerce system with 4 components and 2 different values for each component described in Table 1.1. A desired test will contain combinations among these parameters, such as (Firefox,Websphere,MasterCard,Db/2) and (Firefox,Apache,MasterCard,Oracle). In order to fully test the system we would require $2^4 = 16$ configurations.

Client	Web Server	Payment	Database
Firefox	WebSphere	MasterCard	Db/2
IE	Apache	Visa	Oracle

Table 1.1: System with 4 components

This may sound reasonable, but the number of necessary tests grows exponentially as the number of components increases. Suppose we had a system with 12 possible components and four possible settings each. We then need $4^{12} = 16,777,216$ test configurations, if we extrapolate these tests in terms of time, considering we could finish a test in one second, we would require 279,620 minutes (or the equivalent to 6 months) to finish all the tests. But a different approach is to use a technique called *interaction testing*.

Fisher in 1926 [43] was the pioneer in the use of interaction testing. Interaction testing implements a model-based testing approach using combinatorial design [4]. In this approach, all t-tuples of interactions in a system are incorporated into a test suite. In fact, interaction testing measures interactions rather than detecting interactions.

Interaction testing is based on the premise that many errors in software can only arise from the interaction of two or more parameters. Some of the advantages of using this technique are:

1. Code coverage

2. Minimum number of tests

1.1.3.1 Code Coverage

A number of studies have investigated the application of combinatorial methods to software testing [10]. Early research focused on pairwise testing, where all the possible interactions between two parameters are present at least once. Some of this research were focused on the percentage of code coverage when combinatorial methods were used [10, 18].

Many studies demonstrated the effectiveness of pairwise testing in a variety of applications. But, there is a possibility that some of the failures in a system are present when an interaction of more than 2 parameters occurs. Based on this, how can we determine the degree of interaction of the combinatorial object to test our system? Studies were made in order to show the percentage of failures a real system will present when distinct levels of interaction were used [31, 32]. The results of these tests are summarized in Figure 1.3 and Table 1.2.

In Figure 1.3 we can clearly see how the failure detection rate increases rapidly with the interaction level. With the server application, for example, 42% of the failures were triggered by only a single parameter value, 70% by 2-way combinations, and 89% by 3-way combinations. The detection rate curves for the other applications behaves in a similar way, reaching in some cases 100% of detection with 4 to 6-way interactions. This means that the interaction of size six or less parameters in these systems were causing 100% percent of the faults on the systems.

While not conclusive, these results suggest that combinatorial testing which exercises strength interaction of size two to six can be an effective approach to software assurance.

Interaction Level \ Application	Med Devices	Browser	Server	NASA Database	Network Security
1	66%	29%	42%	68%	17%
2	97%	76%	70%	93%	62%
3	99%	95%	89%	98%	87%
4	100%	97%	96%	100%	98%
5	100%	99%	96%	100%	100%
6	100%	100%	100%	100%	100%

Table 1.2: Percentage fault detection at interaction levels 1 through 6 according to the type of application

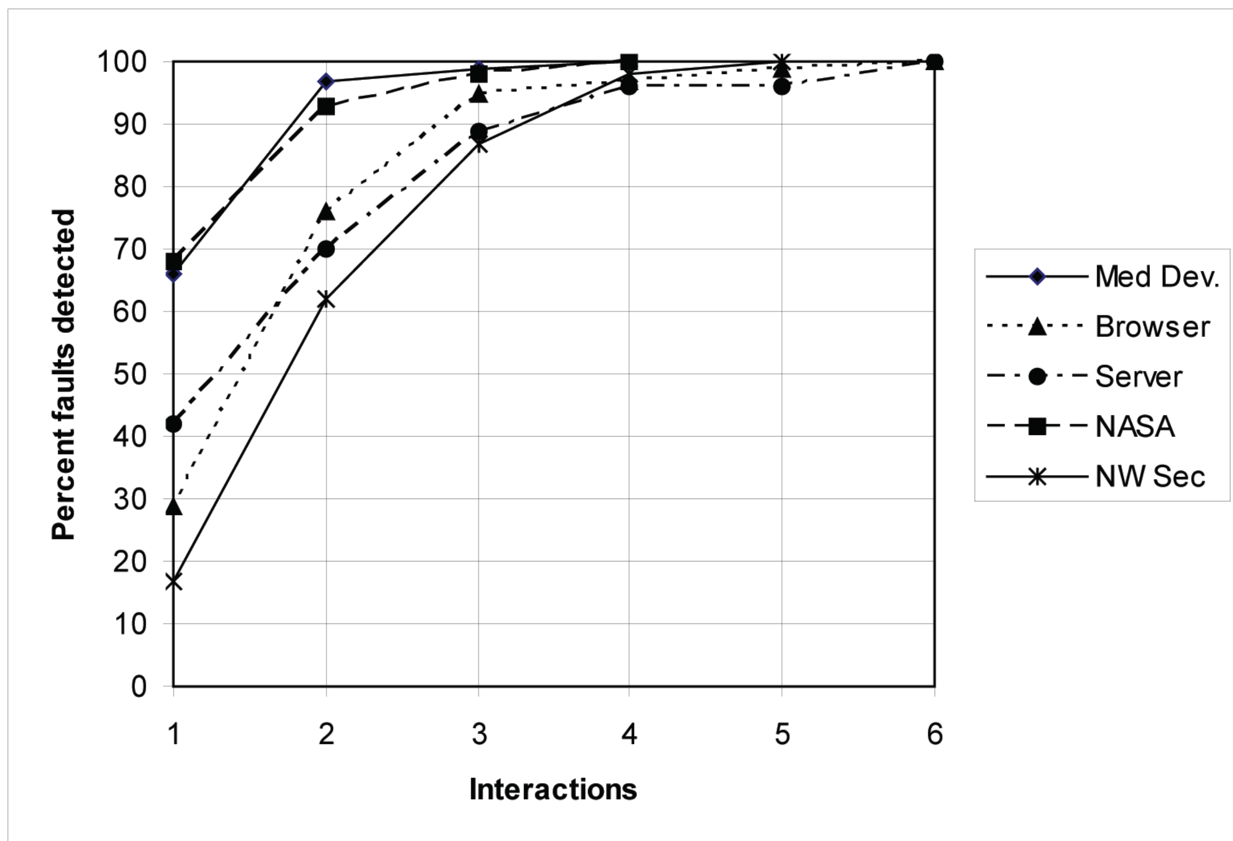


Figure 1.3: Fault detection at interaction levels 1 through 6 according to the type of application [3]

1.1.3.2 Minimum Number of Tests

Referring to the example in Table 1.1 where we have an e-commerce system with 4 components and two different configurations for each component. In order to fully test this system we would require $2^4 = 16$ configurations. But, using a pairwise approach we would only need 5 tests. This pairwise approach is presented in Table 1.3.

The difference in the number of tests may not sound really huge, but when the number of components increases so does the number of tests required to make a complete test. Suppose we had a system with 12 possible components and four possible settings each. We then need $4^{12} = 16,777,216$ test configurations. But, using different levels of interactions we can clearly see that the number of required tests are many times smaller than a full test approach. In Table 1.4 the number of tests required for an interaction level 2 through 6 are shown.

Client	Web Server	Payment	Database
Firefox	WebSphere	MasterCard	DB/2
Firefox	Apache	Visa	Oracle
IE	WebSphere	Visa	Oracle
IE	Apache	MasterCard	Oracle
IE	Apache	Visa	DB/2

Table 1.3: An example of a pairwise coverage

Interaction level	Number of components	Configurations per component	Number of tests
2	12	4	24
3	12	4	121
4	12	4	508
5	12	4	3,064
6	12	4	14,888

Table 1.4: Number of tests required for diferent interaction levels

In Table 1.4 we can observe that the number of tests increases when the interaction level increase, nevertheless even when the interaction level is 6 the $14,888$ tests required are nowhere near as the $16,777,216$ required to fully verify the system. Therefore, we can conclude that the advantage of using interaction testing increases when the dimensionality of the problem increases as well.

1.1.3.3 Types of Combinatorial Objects

One of the combinatorial objects that can be constructed in order to perform an interaction test is an *Orthogonal Array* (OA). An OA [29] is an $N \times k$ matrix with entries from a set of v distinct symbols arranged so that, for any set of t columns of the array, each of the v^t row vectors appears equally often. The notation of an OA is as follows: $OA_\lambda(N; t, k, v)$, where N represents the number of rows of the matrix. k represents the number of columns of the matrix. t represents the degree of interaction of the parameters, and λ is the number of times that each combination of size t must appear.

Due to the excessively large number of tests that are needed when $\lambda > 1$, and that there are a large number of values for v and k where the OA with $\lambda = 1$ does not exist, it is necessary to rely in less restrictive structures known as *Covering Arrays*.

Covering Arrays (CAs) are similar to OAs. A CA [25] of size N is an $N \times k$ array consisting of N vectors of length k with entries from $0, 1, \ldots, v-1$ (v is the size of the alphabet) such that every one of the v^t possible vectors of size t occurs at least once in every possible selection of t columns. The parameter t is referred to as the *covering strength*. The objective is to find the minimum N for which a CA with parameters k, v, t exists.

The minimum number of rows required to construct a CA for a specific value of v, k and t is denoted by $CAN(t, k, v)$ [8], being this an optimal CA.

1.2 Research Goals and Contributions

The algorithms published in the literature to construct CAs can be divided in two groups: complete and incomplete methods. Incomplete methods try to find good solutions as fast as possible, but they do not guarantee the optimality of the found solution. On the other hand, complete methods guarantee to always discover optimal solutions if they exist or to prove their nonexistence given that there are no computing time restrictions imposed.

While incomplete methods such as greedy algorithms or heuristic approaches can construct larger instances of CAs, there are no guarantee of the quality of the solution, as the solution found can be really close or far away of an optimal solution. Generating optimum CA (where N=CAN) is really important, as it reduces the number of tests, the cost and the time expended on the software testing process. For this reason the development of an algorithm that creates optimum CA is relevant [54]. Even though complete methods can guarantee the optimum value CAN, but the computational effort needed grows exponentially when the size of the problem increases, therefore, this approach is used to construct small instances of CAs.

Even though the clear limitation of complete methods, there are recursive constructions that use small CAs as ingredients to construct larger instances, therefore if the quality of the solution for the initial ingredient is near optimal or optimal the final solution will be good as well. This type of construction justifies the research of complete methods as they can guarantee to find optimal solutions or prove their nonexistence for small instances that can then be used as ingredients for

recursive methods.

In this work we propose to implement a complete method based on the Branch & Bound (B&B) technique in order to construct optimal binary CA of variable strength with balanced symbols per column. Since the whole search space to construct CAs is $v^{N \times k}$ this B&B approach will only be used to construct small instances.

With the aim of developing an algorithm with a good performance, several studies in different aspects were needed. The first task was to study and to implement deterministic methods reported in the literature to construct binary covering arrays. The main goal of this task was to construct as many optimal CAs already reported in the literature we could in order to find some patterns in their structure. Secondly, we studied matrix symmetries in order to reduce the search space of the problem and create some bound heuristics to improve the performance of our algorithm. Lastly, we studied bound heuristics already published in order to measure their effectiveness in the CA construction problem.

Considering that the CA construction problem is known to be NP-complete [35], it is unlikely to obtain optimum solutions in polynomial time. However, obtaining an optimum CA (for small instances) is really important, as it will guarantee to minimize the number of tests needed, reducing the time and cost of the whole software testing process. For this reason, constructing a B&B method capable of finding a CAN, given some parameters, will prove to be very useful for testing purposes.

1.3 Thesis Overview

This thesis is divided into 4 other chapters:

- State of the Art
- Methodology
- Experimental Results
- Conclusions and Future Works

In the second chapter some state of the art methods for constructing CAs are reviewed. Among them are algebraic, deterministic and non-deterministic approaches. In the methodology chapter, all steps involved in the development process of this thesis are described. In chapter 4 experimental results using a test suite composed of 14 well known optimal binary covering arrays of strength $3 \leq t \leq 5$ taken from the literature [14, 55] are used in order to compare the behavior of our algorithm against some methods published in the literature. In chapter 5 the conclusions, future works and also the contributions of this work are presented.

1.4 Summary of the Chapter

In this chapter a brief general context was described in order to get a better understanding of the terminology of the field that this thesis work is focused on. This context is divided in 3 subsections : relevance of software testing, types of software testing, and interaction testing. Finally, the research goals and contributions were described in terms of our principal objective in this work, a brief introduction of how we solved the problem proposed in this thesis, and why the development of this work is relevant. In the next chapter we will explain in detail some constructions that are proposed in the literature to solve the CA construction problem.

2

State of the art

In this chapter, the history of the CAs will be explained in detail. As well, types of construction including algebraic methods, polynomial methods, heuristic methods and other methods will be presented. In the first section of this chapter definitions and examples of fundamental theoretical concepts are written in detail. Lastly, some of the construction methods reported in the literature for solving the covering array constructing problem (CAC) are presented.

2.1 Definitions and Examples

2.1.1 Latin Squares

Definition *A Latin Square (LS) is an $n \times n$ table filled with n different symbols in such a way that each symbol occurs exactly once in each row and exactly once in each column.*

Latin Squares are combinatorial designs very antique and vastly studied. It is believed that Euler by 1782 was the first one to study them, and starting the XX century it was Fisher [19] who

1	2	3
2	3	1
3	1	2

Table 2.1: A simple latin square example of 3 rows and 3 columns

demonstrate their usefulness for the control of statistical agriculture experiments and Robert Mandl in 1985 [37] applied them in software testing.

An example of a LS can be seen in Table 2.1.

The name Latin squares originates from Leonhard Euler, who used Latin characters as the symbols used to construct the LS. A Latin square is said to be reduced (also, normalized or in standard form) if its first row and first column are in natural order. For example, the Latin square in Table 2.1 is reduced because both its first row and its first column are 1,2,3 (rather than 3,1,2 or any other order).

2.1.2 Orthogonal Latin Squares

Definition *Being n a positive integer and $A = [a_{ij}]$ and $B = [b_{ij}]$ latin squares of order n. Let be an array of $n \times n$ with entries $[(a_{ij}, b_{ij})]$; if within this array, each of the n^2 pairs of symbols occurs exactly once, then the LS A and B are orthogonal, and are denoted as: $A \perp B$.*

Let us represent two LS in Table 2.2. When we list all the possible combinations we can clearly see that all the possible n^2 pairs of symbols are present without any repetition (see Table 2.3).

Table 2.2: An example of two different latin squares of 3 rows and 3 columns

(a)

1	2	3
2	3	1
3	1	2

(b)

7	8	9
9	7	8
8	9	7

Definition *A set of LS $L_1, L_2, ..., L_m$ are a set of Mutually Orthogonally Latin Squares, or set of MOLS, if for each $1 \leq i \leq j \leq m$, L_i and L_j are orthogonal.*

The configuration of two MOLS of size 3 is presented in Table 2.3.

1 7	2 8	3 9
2 9	3 7	1 8
3 8	1 9	2 7

Table 2.3: The configuration of two mutual orthogonally latin squares of size 3

2.1.3 Orthogonal Arrays

Definition *An OA [29] is an $N \times k$ matrix with entries from a set of v distinct symbols arranged so that, for any set of t columns of the array, each of the v^t row vectors appears equally often. The notation of an OA is as follows: $OA_\lambda(N; k, v, t)$, where N represents the number of rows of the matrix. The parameter k represents the number of columns of the matrix. The parameter t represents the degree of interaction of the parameters, and λ is the number of times that each combination of size t must appear.*

The first works where combinatorial objects were applied to the designs of tests, were made through Orthogonal Arrays in disciplines like agriculture and medicine [24]. Mandl [37] was one of the first researchers to use OA in order to test software. He used OA to test if sorting operators on enumeration values are correct even when these enumeration values are ASCII characters for compilers written in ADA. He stated that the use of OA yields about as much useful information as the exhaustive approach.

A problem with OAs is that it can lead to tests excessively large when $\lambda > 1$. If there exists an OA with index $\lambda = 1$, then this OA is optimum as there is no OA that can be constructed with fewer rows. Nevertheless, there are a lot of values for v and k where the OA with $\lambda = 1$ does not exist, because of this it is necessary to rely in a less restrictive structure knows as *Covering Arrays*.

An example of an OA_2(8;5,2,2) is shown in Table 2.4.

In Table 2.4 we can clearly observe how all the possible combinations of 2^2 {(0,0),(0,1),(1,0),(1,1)} are present exactly 2 times for every pair of columns.

In order to show the difference in number of tests between OAs and CAs when their parameters k, v, t remain the same, a table with some examples of these constructions will be presented in Table 2.5. It is important to mention that the value of λ in Table 2.5, is the minimum value where an OA

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

Table 2.4: Example of an OA_2(8;5,2,2)

with the given parameters k, v, t can be constructed.

k	v	t	CA	OA	λ
3	2	2	4	4	1
11	2	2	7	12	3
19	2	2	8	20	5
4	2	3	8	8	1
12	2	3	15	24	3
16	2	3	17	32	4
6	2	4	21	80	5
16	2	4	35	256	8

Table 2.5: Table summarizing the difference of required tests between OAs and CAs

From Table 2.5 we can conclude that the size of OAs are greater than the size of a CA (even more when the size of t increase). Due to this fact in terms of software testing the CAs prove to be an easier combinatorial object to use.

2.1.4 Covering Arrays

Definition *A Covering Array CA*$(N; t, k, v)$ *of size* N *is an* $N \times k$ *array consisting of* N *vectors of length* k *(degree) with entries from an alphabet of size* v*, i.e.,* $\{0,\ 1,\ \ldots,\ v-1\}$*, such that every one of the* v^t *possible vectors of size* t *(t-wise) occurs at least once in every possible selection of* t *elements from the vectors. The parameter* t *is referred to as the strength or level of interaction. The minimum* N *for which a CA*$(N; t, k, v)$ *exists is known as the* covering array number *and it is defined according to* (2.1)*.*

$$\text{CAN}(t, k, v) = min\{N : \exists \, \text{CA}(N; t, k, v)\} \tag{2.1}$$

Researchers publish the best *upper bound* known for a specific CA. Stevens [51] made a summary of the results of strength 2, Sloane [47] presents an excellent summary of the results published for the CA of strength 2 and 3. Sherwood [45] maintains a web page of OA and CA constructions by permutations groups. Lastly, Hartman [22] presents a survey for CA with an uniform alphabet or a mixed alphabet and Colbourn maintain a web page with the best known solutions found [14].

A CA$(N; t, k, v)$ can be mapped to a software test suite as follows. In a software test we have k components, each of these has v configurations. A test suite is an $N \times k$ array where each row is a test case. Each column represents a component and the value in the column is the particular configuration.

Lei and Tai [35] demonstrated that the problem of generating the minimum pairwise test set belongs to the NP class (for non-binary alphabets). Then by reduction of the problem of vertex cover, they proved that the *pair-cover problem* is NP-complete. Colbourn [15] also showed that this problem is NP-complete by reducing it from the SAT problem.

Even though the general problem of finding a combinatorial test suite is NP [30], there are some isolated cases that can be solved in polynomial time [15]:

1. When the strength is 2 $(t = 2)$ and the alphabet is 2 $(v = 2)$. Sloane stated that this case was solved by Rényi with an even value of N and by Katona independently. Then, it was completely solved by Kleitman and Spencer for all N [47].

2. When the alphabet is a power of prime $v = p^\alpha$, $k \leqslant (p^\alpha + 1)$, and $p^\alpha > t$ where α is a natural number. This construction was proposed by Bush [6]. He used Finite Galois Fields in order to solve this case.

An example of a CA(10,5,2,3) is shown in Table 2.6.

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

Table 2.6: Example of an CA(10;5,2,3)

2.1.4.1 Isomorphic Covering Arrays

Given a $CA(N;t,k,v)$ permuting the rows and/or columns produces an equivalent CA [25]. The rows represent a set of test vectors, and their order is irrelevant. Permuting the columns does not affect since every subset of t columns contains all the combinations of v^t symbols.

Definition *Two Covering Arrays are isomorphic if one Covering Array can be obtained by the permutation of rows, columns, or symbols of the other.*

There are 3 types of symmetries in a CA: row symmetry, column symmetry and symbol symmetry. The row symmetry refers to the possibility to alter the order of the rows without affecting the CA properties. There are $N!$ possible row permutations of a CA. The column symmetry refers to permuting columns in the CA without altering it. There exist $k!$ possible column permutations of a CA. In the same way the symbol symmetry includes all the possible permutations of symbol per column, giving a number of $(v!)^k$ isomorphic CAs that can be constructed this way. By the previous analysis we can conclude that there are a total of $N! \times k! \times (v!)^k$ different isomorphic CAs to one specific CA.

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

Table 2.7: An example of a CA(5;2,4,2)

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

Table 2.8: An isomorphic CA(5;2,4,2) with permuted rows and columns based on table 2.7

An example of two isomorphic CAs can be seen in Table 2.7 and Table 2.8:

As seen the CAs in Tables 2.7 and 2.8 are isomorphic because the second CA can be produced by changing the first and last columns and changing the second and third columns of the first CA.

2.1.5 Mixed Covering Arrays

The research of Cohen et al. [11] states that frequently the number of parameters for a system is greater than 79 and the values that each parameter can take may vary. In order to apply a combinatorial test suite for this kind of system, a variation of the CA must be used. This variation is called *Mixed Covering Array*. On a Mixed Covering Array (MCA) the cardinality of the alphabet in each column may vary, it implies that the parameter v will not be a scalar number, in fact the parameter v will be a vector that contains the cardinality of each column $v_0, v_2, v_3, ..., v_{k-1}$.

Definition *Being N, k, v, t positive integers where $t \leq k$. A Mixed Covering Array of type $\prod_{i=0}^{k-1} v_i$ with strength t and size N, denoted by $MCA(N; k, \prod_{i=0}^{k-1} v_i, t)$, its an array A of size $N \times k$. Being $i_i, ..., i+t \subseteq 1, ..., k$, and B a sub-array of size $N \times t$ obtained by selecting the columns $i_1 ... i_t$ of the MCA. There are $\prod_{i=0}^{k-1} v_i$ distinct t-tuples that can appear as rows of B, a MCA requires that at least appear once.*

Little is known about the minimal sizes of MCA. In [24] Hedayat and Sloane present a study about the MCA. Sloane et al. [48] extend such work and using linear programming techniques they obtain minimum cases for orthogonal arrays. Stardom [50] and Chateauneuf [7] suggest the necessity of extending their works about CA of uniform alphabets to MCA, but, mostly of the works reported were only for $t = 2$. Recently, Moura et al. [41] reported some algebraic transformations to construct MCA. Such transformations were limited for strength 2, but they obtained the minimum N for cases where $k \leq 4$ and some cases where $k = 5$.

The vast majority of the literature corresponding to the CA applied to the design of software testing include methods to construct them efficiently while and with the minimum number of rows.

An example of a MCA is presented in Table 2.9.

0	1	1	3
0	0	0	0
0	0	1	0
0	0	0	4
0	1	2	1
0	1	0	2
1	0	2	3
1	0	2	2
1	0	0	1
0	1	1	4
1	1	2	4
1	1	2	0
1	0	1	1
1	0	1	2
1	1	0	3

Table 2.9: Example of an MCA(15;4,$\{2^2, 3^1, 5^1\}$,2)

As we can see in Table 2.9 the main difference of one MCA and one CA is that the number of symbols per column in the MCAs may vary, while in the CAs every column has the same number of symbols.

2.2 Covering Arrays Construction Methods

The objective of constructing a CA is to minimize the number of rows with the given parameters v, k, t. There are several ways to construct these combinatorial objects. Although, given the complexity of the problem, there are few complete methods. Most of them are incomplete methods [55] and according to Reneé Bryce et al. [5] the incomplete methods that have been more effective are Simulated Annealing and Tabu Search.

Grindal et al. [21] classified the methods to construct CAs as follows:

- Non-Deterministic
  - Heuristic
    - Automatic Efficient Test Generator (AETG)
    - Simulated Annealing (SA)
  - Bio-Inspired Metaheuristics (BIM)
    - Genetic Algorithm (GA)
    - Ant Colony Algorithm (ACA)
  - Random
- Deterministic
  - Iterative
    - Test Case Based
      - Constrained Array Test System (CATS)
      - Eeach Choice (EC)
      - Partly Pair-Wise (PPW)
      - k-bound
      - Anti Random (AR)

· k-perim

· Base Choice (BC)

· All Combinations (AC)

∗ Parameter Based

· In Parameter Order (IPO)

– Constructed in Polynomial Time

∗ OA

∗ CA

The full tree from Grindal et al. [21] is shown in Figure 2.1. Some of the construction methods shown in Figure 2.1 will be explained in the following sections.

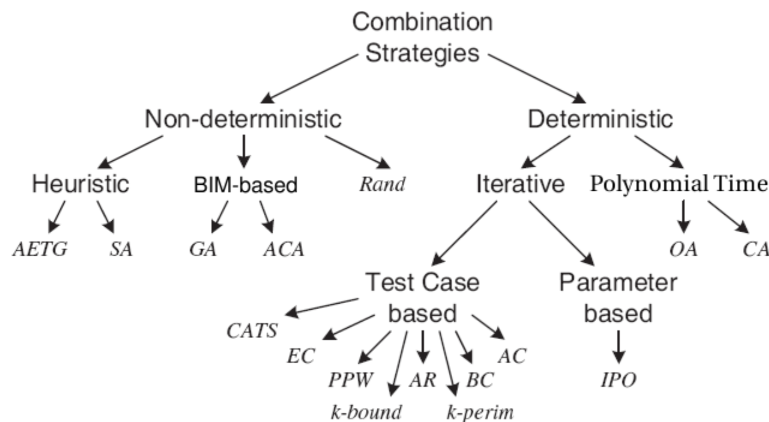


Figure 2.1: Classification scheme for combination strategies for constructing covering arrays

2.2.1 Constructing Optimal Covering Arrays within Polynomial Time

The construction of optimum CA in general its an NP-Complete problem [44], nevertheless there are 2 cases for which it is possible to construct optimum cases within polynomial time:

1. When the strength is 2 ($t = 2$) and the alphabet is 2 ($v = 2$). Sloane stated that this case was solved by Rényi with an even value of N and by Katona independently. Then, it was completely solved by Kleitman and Spencer for all N [47].

2. When the alphabet is a power of prime $v = p^\alpha$, $k \leqslant (p^\alpha + 1)$, and $p^\alpha > t$. This construction was proposed by Bush [6], he used Finite Galois Fields in order to solve this case.

2.2.1.1 Case $t = 2$

The polynomial algorithm follow these steps: fill the first row of the CA with 0's, calculate the total of remaining rows following the equation 2.2, the remaining 1's are calculated by column, $\lceil N/2 \rceil$, the remaining 0's per column are calculated $N - 1 - \lceil N/2 \rceil$, lastly the remaining rows are filled with the combinations of 1's and 0's obtained within the last steps.

$$k \leq \binom{N-1}{\lceil N/2 \rceil} \tag{2.2}$$

2.2.1.2 Case where $v = p^\alpha$

Bush [6] proposed a theorem to develop direct constructions through OA and Galois Finite Fields. This theorem requires that $v = p^\alpha$ be a power of prime and $(v > t)$, and for the case $CAN(k, v, t) = v^t$ for every $k \leq v + 1$. When the value of $\alpha = 1$ the construction is based in MOLS, for the cases where $\alpha > 1$ the CAs are constructed using finite fields.

2.2.2 Algebraic Transformations

2.2.2.1 TConfig

Tconfig is a tool developed by Williams and Pobert [53] that employs algebraic transformations based on OA to construct CA of strength $t = 2$ and uniform alphabet. It uses the philosophy of "divide and conquer". The theory of this construction is based on the work reported in [2] where it requires that $\mid v \mid$ be a power of prime.

2.2.2.2 Combinatorial Test Services

The Combinatorial Test Services (CTS) is a software library created by Hartman and Raskin [23] for the generation and manipulation of test cases, it is used for testing input data or configurations. CTS enables the user to generate small test suites with strong coverage properties, and perform other useful operations for the creation of systematic software test plans.

Hartman and Raskin used a set of theorems and mathematical lemmas that allows to generate in polynomial time distinct type of CA constructions, some of them are optimal or near optimal. They also deal with the *testing budget problem*, this problem refers to the construction of a testing array A of size at most b having largest possible coverage measure. The CTS package first constructs a covering array, and then orders its columns in such a way as to maximize the incremental coverage achieved by each new column. It does not explore all possible orderings of the columns, but rather takes a greedy approach, selecting the next column using an algorithm whose complexity is quadratic in the number of columns.

2.2.3 Deterministic Strategies

2.2.3.1 Automatic Efficient Test Generator

The Automatic Efficient Test Generator (AETG) is a tool originally developed by the Bellcore company (now Telecordia), that mechanically generates efficient test sets from user defined test requirements. It is based on algorithms that uses ideas from statistical experimental design theory to minimize the number of tests needed for a specific level of test coverage of the input test space. AETG has been used in Bellcore for screen testing, interoperability testing and for protocol conformance testing. Cohen et al. [11] were the first ones to publish a description of the algorithms used in the implementation of the AETG.

In this algorithm a CA is constructed using the technique “one row at a time”. It is important to mention that the results are not always optimal and generally the results are not obtained within a logarithmic time.

2.2.3.2 In Parameter Order (IPO)

In Parameter Order (IPO) was designed by Lei and Tai [35], it is a fast pairwise testing generator, but it does not guarantee the best solution possible. The IPO algorithm works as follows: it generates a pairwise test of the first two parameters, then it extends the test set to generate a pairwise test for the first three parameters, and this loop continues until the requested parameters are completed. To generate the next vectors of the CA two steps are followed:

1. Horizontal Growth, in this step different values that guarantee the maximum coverage between the test set already constructed are added in a greedy manner. It is to consider that many ties have to be broken consistently to ensure that the resulting test set is deterministic.

2. Vertical Growth, this step covers the remaining uncovered combinations, one at a time, either by changing an existing test or by adding a new test. When it changes an existing test it only modifies already established *don't care* positions. If there are no don't care positions a new test is added.

A *don't care* position is a position within the CA that can be changed for whichever valid value for the selected column and it will still remain as a valid CA. In this algorithm the don't care positions are represented with the symbol $\star$.

This system has two options, one is to generate a new pairwise test set without reusing an existing test. The other option involves keeping the existing set and generate additional tests, if necessary, such that the combined test set is a pairwise test. The combined set produced by the second option may be larger than the new set produced by the first option.

In [34] an improved version of this algorithm called IPOG-D is presented in which problems of strength $2 \leq t \leq 6$ can be solved.

2.2.3.3 Test Case Generation

The Test Case Generation (TCG) its an algorithm developed by Tung and Aldiwan [56]. It assumes that there is a system S with k input parameters $(F1, F2, ..., F_k)$. For each parameter $i, 1 \leq i \leq k$

there are $m(i)$ different values, $V_{i1}, V_{i2}, ..., V_{im(i)}$. It is to consider that $m(i)$ represents the size or cardinality of the vector V_i which it will be represented as $\mid V_i \mid$. The algorithm follow these 4 steps:

1. Ascendantly sort the input parameters i with respect to their cardinality $\mid V_i \mid$ such that $\mid V_1 \mid \leq \mid V_i \mid \leq \mid V_k \mid$.

2. M candidates vectors are generated such that they contain the missing combinations.

3. The candidate that presents the most missing combinations is selected.

4. The steps 1,2,3 loops until the array of vectors contains all the possible combinations of parameters.

This algorithm can generate CAs and MCAs of strength $t = 2$. Its advantage is the quickness of the algorithm to construct test cases guaranteeing that all combinations are present. But, in [12, 56] is indicated that TCG sometimes maintains the accuracy of AETG, but it can perform poorly with respect to the accuracy of AETG. In part, the fixed ordering of factors in TCG removes a degree of freedom that AETG exploits to certain extent.

2.2.3.4 Deterministic Density Algorithm

The Deterministic Density Algorithm (DDA) developed by Bryce et al. [5] employs the technique *one test at a time* improving the results obtained by other algorithms following the same technique. The DDA constructs one row of a covering array at a time using a steepest ascent approach. Factors are dynamically fixed one at a time in an order based on density. New rows are continually added until all interactions have been covered.

Four decisions must be made to instantiate this prototype:

1. Factor density, the manner in which densities are computed for factors.

2. Factor tie-breaking rule, a tie-breaking is done when two or more maximum densities for factors are equal.

3. Level density, the manner in which densities are calculated for levels.
4. Level tie-breaking rule, a tie-breaking is done when two or more maximum densities for levels are equal.

The basic principle of this algorithm is to have the least repeated combinations based on a given configuration.

2.2.4 Non-Deterministic Algorithms

2.2.4.1 Simulated Annealing

The Simulated Annealing Algorithm (SA) was first introduced by S. Kirkpatrick [27] and was inspired in statistical mechanics, in which first melting a solid and leading to its maximum energy value where the atoms can move freely, then cooling it slowly, so the atoms can find the best energy value to form a pure crystal. If there is no slow cooling factor, then the crystal may present flaws (locally optimal structures).

This analogy can be applied to optimization problems. The solid represents the optimization problem, the energy represents the objective function, and the optimum value is reached by a cooling process, in which an initial and final temperature as well as a reduction factor are previously defined. During the optimization process, random states are evaluated, if a state has a lower energy, then it is immediately accepted, but if an state with a higher energy is found then according to the Boltzman distribution it will be decided if it will be accepted or rejected. By accepting states where the energy value is higher, the probability of getting stuck on a local optimum is reduced.

The main parameters of the Simulated Annealing algorithm are:

1. Markov Chain Length
2. Initial Temperature
3. Final Temperature

4. Cooling Factor

5. Frozen Parameter

6. Number of Total Evaluations

7. Stop Criterion

8. Perturbation Function

The Markov Chain is a sequence of trials, where the probability of the outcome of a given trial depends only on the outcome of the previous trial. In the case of SA, a trial corresponds to a move, and the set of outcomes is given by a finite set of neighboring states. Each move depends only on the outcome of the previous attempt, so the concept of Markov chains applies. Since the number of the trial does not affect the probabilities, it is considered homogenous.[1]

The Initial Temperature is the temperature in which the SA must start, according to the Metropolis Monte Carlo simulation mentioned in the literature it must start in a high temperature state, and then the temperature must be lowered slowly until the final temperature is reached.

The Final Temperature is the lower bound of the temperature, when the temperature reaches this limit the SA algorithm ends and the best value reached is reported.

The Cooling Factor or Reduction Factor determines how slowly the temperature is lowered. If the value is too high, then the algorithm ends faster, but a good solution is not guaranteed. If this value is too low, then the algorithm could reach a frozen state.

The Frozen parameter tells us in how many iterations if no solution was accepted (better or worse) the algorithm reaches a frozen state and ends.

The Number of Evaluations determine how many evaluations the algorithm check before exiting.

Cohen et al. [9] was the first to propose the use of the Simulated Annealing (SA) for the covering array constructing problem. Their results improved the best results obtained with the *one test at a time* approach.

2.2.4.2 Tabu Search

Nurmela in 2004 published a work relating to the covering construction problem using Tabu Search (TS) [42].

TS is a general search procedure for solving combinatorial optimization problems of the following general type in equation 2.3:

$$Minimize\ f'(x),\ subject\,to\,x \in X \tag{2.3}$$

where f' is a cost function and X is a set of variable solutions. The procedure has its roots in the intelligent problem solving approach, found in artificial intelligence [40]. In this matter it can be said that there is a certain learning and the search is intelligent. In simple terms it can be said that TS works by moving within feasible solutions and better solutions than the current solution, but to escape from local optima it sometimes move to worst solutions (something like the SA).

To ensure that the process does not loops within local optima reached in a previous iteration, the TS stores a vector of length t as a tabu list. This tabu list is represented as a determined number of recent moves, which can not be repeated in a certain number of iterations.

Unfortunately, the tabu list may forbid certain moves, such as moves that lead to a better solution than the best one found so far. To solve this an *aspiration criterion* is created to cancel the tabu status of a move if it is judged to be proven useful.

Nurmela describes his implementation of Tabu Search to construct CAs as:

1. An $N \times k$ matrix M is constructed with random values.

2. A random combination that is not presented within M is chosen.

3. It is checked which rows requires a simple mutation from one of its elements in order to have the combination selected previously and it is added as a part of the neighborhood of the matrix M.

4. The neighborhood is evaluated in terms of missing combinations, and the matrix with the least

number of missing combinations is selected. In case of ties a random matrix within the tied matrices is chosen.

5. This process iterates until the cost of a matrix reaches 0.

Nurmela reported new upper bounds for CAs, but the downside of his approach was that the time required to solve some instances, as there were some cases that required even months to finish.

2.2.4.3 Genetic Algorithms

The Genetic Algorithms (GA) are optimization techniques first popularized by Holland [26] in 1975. The main idea in this technique is based in nature, as nature itself is an optimization process where species evolve to reach optimum or near optimum states in order to survive. So, by mimicking this process, individuals in optimization problems evolve to more suitable states based on an evaluation function in order to reach optimum or near optimal states.

Much of the terminology used in this technique is taken directly from biology. The algorithm begins with a chromosome *population* of size n chosen randomly, and then it proceeds through a sequence of generations. In each generation a new population with a higher average and maximum fitness than the previous ones is created using 3 distinct *genetic operators*. The 3 genetic operators are:

1. Selection, in this operator some chromosomes are selected through some techniques that check the fitness of each chromosome, where the probability to select certain chromosomes with less fitness than the previous chromosomes selected decreases.

2. Crossover, in this operator the population is randomly partitioned into pairs of chromosomes. The population size n is always even, so that there can be a whole number of pairs of chromosomes. Each chromosome is considered as a *parent* and both parents always give some information to their offsprings.

3. Mutation, in this operator depending on a certain probability, an offspring mutate some of its information depending on the mutation technique used in the algorithm.

The algorithm loops until the best solution is achieved or until the defined number of generations is reached.

According to the results applying genetic algorithms to the covering construction problem [50, 13] we can conclude that there is a lot of room to improve the GA implementations. The main issue using population based algorithms is that each individual is considered as a matrix of size $N \times k$, the cost of evaluating a matrix to be a CA is $N \times \binom{k}{t}$, and considering a population of size n, the whole cost of evaluate all the population is $\sum_{i=0}^{i<n} N \times \binom{k}{t}$. Considering this, the time spent only evaluating the initial population is huge compared to TS or SA.

2.2.4.4 Ant Colony Algorithm

The Ant Colony Algorithm (ACA) is a meta-heuristic algorithm for the approximate solution of combinatorial optimization problems that has been inspired by the foraging behavior of real ant colonies proposed in 1996 by Marco Dorigo et al. [17]. The structured behavior of an ant colony is possible by a chemical substance called *pheromone*, which establish the best possible route from the colony to their food source. Real ants are capable of finding the shortest trajectory from a food source to their nest, without using visual cues by exploiting pheromone information. While walking, ants deposit pheromone on the ground, and follow, in probability, pheromone previously deposited by other ants . A way ants exploit pheromone to find a shortest trajectory between two points is shown in Figure 2.2.

Consider Figure 2.2(a): Ants arrive at a decision point in which they have to decide whether to turn up or down. Since they have no clue about which is the best choice, they choose randomly. It can be expected that 50% of the ants choose at this moment the upper road and the other 50% choose the bottom road. Figure 2.2(b) and Figure 2.2(c) show what happens in the immediately following instants, supposing all ants walk at approximately the same speed. The number of dashed lines represents the amount of pheromone the ants have deposited on the ground. Since going to the bottom is shorter than taking the upper road, more ants will be passing by the bottom road, and therefore the amount of pheromone accumulated in the bottom road exceeds the upper one. This event can be seen in Figure 2.2(d) as more ants will take the bottom road and in a very short time

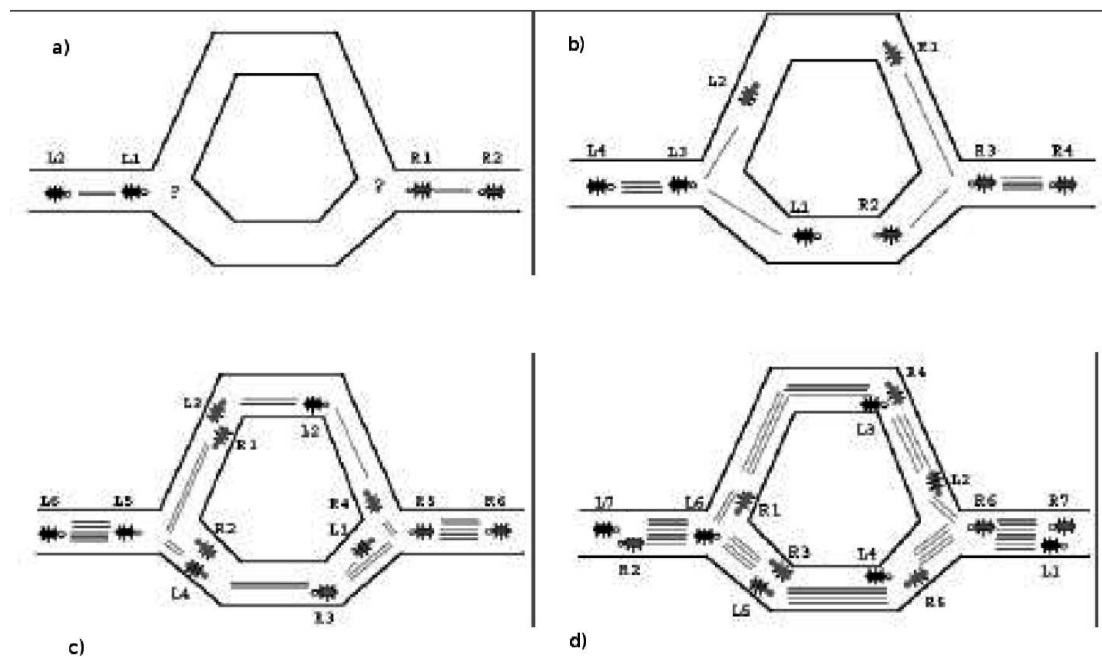


Figure 2.2: An example of the behavior of real ants choosing a road

all the ants will take the same road.

The computational method follows the ant behavior by giving more pheromone to better solutions. Shiba et al. [46] proposed the ACA to generate test cases using a "one test at a time" approach. In their algorithm a test case can be represented as a route from a starting point to the final objective. A given amount of ants start their travel to the final objective. Each time an ant reach to its final objective, it deposits a certain quantity of pheromone to each point visited. When a new ant starts, it will prefer those points where the scent of the pheromone is stronger.

2.2.5 Other Methods

The classification proposed by Grindal et al. [21] does not include exact methods. We are enlarging this classification by adding exact methods to the classification proposed by Grindal.

The next two constructions that are going to be described in the following sections are the EXACT method and the constraint programming. The EXACT methods uses a backtracking algorithm to solve the CA construction problem, while the constraint programming establishes constraints that

can be solved with a SAT solver algorithm to produce the CA.

In terms of backtracking algorithms the most popular technique is the Branch & Bound, which is also the technique selected to solve the CA construction problem in this work.

2.2.5.1 Branch & Bound

A B&B algorithm maintains feasible upper and lower bounds of the problem being solved, and it terminates with a certificate that the solution obtained is optimal [33].

A B&B implementation requires two procedures. The first one is a splitting procedure that, given a set S of candidates, returns two or more smaller sets $S_1, S_2, \ldots, S_w$ whose union covers S. Note that the minimum of $f(x)$ over a specific set S is $\min\{x_1, x_2, \ldots, x_w\}$, where each x_i is the minimum of $f(x)$ within the specific set. This step is called *branching*, since its recursive application defines a tree whose nodes are the subsets of S.

The second tool is a procedure that computes upper and lower bounds for the minimum value of $f(x)$ within a given subset S. This step is called *bounding*.

The main idea of the B&B algorithm is: if the lower bound for some tree node A is greater than the upper bound for some other node B, then A may be safely discarded from the search. This step is called *pruning*, and is usually implemented by maintaining a global variable m that records the minimum upper bound seen among all subregions examined so far. Thus, any node whose lower bound is greater than m can be discarded.

The recursion stops either when the current candidate set S is reduced to a single element; or when the upper bound for set S matches the lower bound. In this way, any element of S will be a minimum of the function within S.

According to Kohler et al. [28] and Smith [49] B&B methods have emerged as the principal general method for finding optimal solutions for solving discrete optimization problems. The temporal requirements of B&B [28] usually grows exponentially or as a high degree polynomial on the problem size. Thus, their usefulness is limited to small sized problems.

2.2.5.2 EXACT Method

Jun Yan and Jian Zhang [54] implemented a backtracking algorithm and published a tool called EXACT (EXhaustive seArch of Combinatorial Test suites). They applied rules in order to search only in solutions that are ordered by row and column in order to eliminate row and column symmetries so that the search space of the problem was reduced.

In order to further reduce the search space they also used a *miniblock* which they define as a $mb \times t$ sub-array, starting from the construction of a CA from a fixed space occupied by this miniblock they theoretically reduce the search space to the Equation 2.4 if the miniblock is fixed.

$$\frac{\prod_{i=1}^{k} v_i^N}{\prod_{i=1}^{t} v_i^{mb}} \tag{2.4}$$

They also used a novel pruning technique called SCEH (The Sub-Combination Equalization Heuristic) in order to reduce the search space. They use this technique because they noted that for many CA each symbol appears almost the same number of times.

In 2008 J. Yan et al. [55] published a modification to their previously published tool EXACT. In this modification they added a new rule. They stated that for each two rows i, j $(1 < i \leq mb$ and $j > mb)$ of a MCA or CA, if these two rows have the same first t values and $R_i >_{lex} R_j$, then they swap the two rows.

2.2.5.3 Constraint Programming

Hnich et al. [25] used a Constraint Programming (CP) approach to solve the covering test problem. They stated that imposing a constraint that each combination of parameter values must appear once will introduce a huge number of variables and reification constraints. Also they stated that the propagation of those constraints was inefficient and ineffective.

In order to solve some of the inefficiencies of their first proposed algorithm, they designed 3 new improvements for their previous algorithm.

In their first improvement the constraint specifies that every number in the range 0 to $2^t - 1$ should be presented at least once and at most $b - 2^t + 1$ times in the b test vectors in the column

corresponding to the t-tuple. This proved to be more efficient but it still has a large number of intersections, so they proposed an Integrating Model.

In the second improvement they merge the two models by using the variables of both approaches, linked by channeling constraints. With this approach there was an increase in the number of variables.

Finally the last improvement called the weakened matrix model is a modified version of the integrated matrix model, it was used with a SAT local search algorithm.

In this work they demonstrated that for moderate problem sizes their approach could find an optimal solution, and that a local search algorithm on a SAT-encoding of the CA construction problem can find improved solutions for somewhat larger instances. One of the advantages of their implementation is the easy handling of side constraints, on the other hand, one of the disadvantages is the bottleneck on the size of the problems that they are able to solve.

Even though the model created by this approach its considered a complete model, Hnich et al. [25] used an incomplete SAT solver to construct their results. Therefore, there are no guarantee of optimality in their work. However, if a complete SAT solver is used, this approach then transforms into a complete method.

2.3 Summary of the Chapter

In this chapter we explained in detail how the CAs were born, starting from latin squares, orthogonal latin squares, mutually orthogonal latin squares, orthogonal arrays and finally reaching the construction problem we were emphasizing on, the covering arrays. We also described in general distinct type of constructions : polynomial constructions, algebraic transformations, deterministic constructions, non-deterministic constructions and also some other constructions. One of the last two constructions called the EXACT method is the one used to do a comparative analysis of our implementation. The B&B technique used in this work is also described in general terms. In the next chapter we will explain in detail our proposed methodology to construct balanced binary covering arrays of variable strength.

3

Methodology

In this chapter, we present the specific details that were involved in the development of the B&B proposed to construct small instances of binary CA with balanced symbols per column. As well as step by step examples in order to have a better understanding of the techniques used and how they work.

The first approach called a New Backtracking Algorithm (NBA) was developed and its flowchart is presented in this chapter, the NBA was able to find the best solutions reported in the literature, but it has the main problem of the huge time required to find a valid configuration. The second approach called an Improved Backtracking Algorithm (IBA) tries to overcome the disadvantages of the NBA.

3.1 A New Backtracking Algorithm (NBA)

We can represent a CA as a 2-dimensional $N \times k$ matrix M. Each row can be regarded as a test case and each column represents some parameter of the system under test. Each entry in the matrix is called a cell, and we use M_{ij} to denote the cell at row i $(i > 0)$ and column j $(j > 0)$, i.e., the

value of parameter j in test case i.

We apply an exhaustive search technique to this problem. Our algorithm is based on the Branch & Bound technique. The main algorithm can be described as an iterative procedure as follows.

For a given matrix $N \times k$, and a strength t, we construct the first element l belonging to the set of all possible columns with $\lfloor \frac{N}{2} \rfloor$ zeros and it is inserted in the first column of the partial solution M. Then the next element $l_{i-1} + 1$ is constructed and if the row i is smaller than the row $i+1$ and it is a partial CA then the element is inserted, otherwise it tries with the next element. If no elements could be inserted in the current column of M, it backtracks to the last column inserted and tries to insert a new element. When k columns are inserted then the procedure finishes and the CA is generated. A flow chart of this algorithm is shown in Figure 3.1.

3.2 Techniques for Improving the Efficiency of the Search

The worst time complexity of the naive exhaustive search for CA(N, k, v, t) is shown in (3.1).

$$\binom{\binom{N}{\lfloor \frac{N}{2} \rfloor}}{k} \tag{3.1}$$

This worst time complexity can be greatly reduced by eliminating all the possible isomorphic CAs. There are 3 types of symmetries in a CA: row symmetry, column symmetry and symbol symmetry. The row symmetry refers to the possibility to alter the order of the rows without affecting the CA properties. There are $N!$ possible row permutations of a CA. The column symmetry refers to permuting columns in the CA without altering it. There exist $k!$ possible column permutations of a CA. In the same way the symbol symmetry includes all the possible permutations of symbol per column, giving a number of $(v!)^k$ isomorphic CAs that can be constructed this way. By the previous analysis we can conclude that there are a total of $N! \times k! \times (v!)^k$ number of symmetries in a CA. An example of two isomorphic CAs is shown in Table 3.1.

The covering array in Table 3.1(d) can be produced by the following steps over the covering array in Table 3.1(a): exchanging the symbols of the first column (Table 3.1(b)), then exchanging the

Table 3.1: An example of how to construct isomorphic covering arrays

(a)

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

(b)

1	**0**	**0**	**0**
1	**1**	**1**	**1**
0	0	1	1
0	1	0	1
0	1	1	0

(c)

1	1	**1**	**1**
1	0	**0**	**0**
0	0	**1**	**1**
0	1	**0**	**1**
0	1	**1**	**0**

(d)

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

first and second rows (Table 3.1(c)), finally exchanging the third and fourth columns.

The searching of only non-isomorphic CAs can significantly reduce the search space, symmetry breaking techniques have been previously applied in order to eliminate the row and column symmetries in [55] by Yan and Zhang. However, they only proposed an approach to eliminate row and column symmetries, and the symbol symmetries were not included. In the following sections we will describe the symmetry breaking techniques that we applied within our backtracking algorithm to search only over the non-isomorphic CAs.

3.2.1 Symmetry Breaking Techniques

In order to eliminate the row and column symmetries in our new backtracking algorithm the constraint that within the current partial solution M the column j must be smaller than the column $j+1$, and the row i must be smaller than the row $i+1$.

As we have mentioned above a CA$(N;t,k,v)$ has $N! \times k!$ row and column symmetries. This generates an exponential number of isomorphic CAs. Adding the constraints mentioned above we eliminate all those symmetries and reduce considerably the search space. Another advantage of this symmetry breaking technique is that we do not need to verify that the columns are ordered, we only need to verify that the rows are still ordered as we insert new columns. This is because we are generating an ordered set of columns such that the column l is always smaller than the column $l+1$.

Moreover, we propose a new way of breaking the symbol symmetry in CAs. We have observed, from previously experimentation, that near-optimal CAs have columns where the number of 0's and 1's are balanced or near balanced. For this reason we impose the restriction that through the whole

process the symbols in the CAs columns must be balanced. In the case where N is not even, the number of 0's must be exactly $\lfloor \frac{N}{2} \rfloor$ and the number of 1's $\lfloor \frac{N}{2} \rfloor + 1$. As long as we know this is the first work in which the symbol symmetry breaking is used. In Table 3.2 an example of our construction is shown.

Table 3.2: Example of the current partial solution M after inserting 4 columns

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

In Table 3.2 we can see clearly that the next column generated is automatically lexicographically greater than the previous one, so we do not need to verify for the column symmetry breaking rule. Even though that in Table 3.2 the rows are ordered as well, it does not happen in general so we still have to check that the rows remain ordered for each element that we try to insert in the partial solution M.

An example of invalid moves where the rows in the partial solution M does not necessarily are ordered is shown in Table 3.3

The Table 3.3 (a) its the first matrix accepted, as we can see it is balanced in number of symbols per column, and it is lexicographically ordered by columns and rows. But if we observe the Table 3.3 (b) and Table 3.3 (c), those two configurations are not valid as the rows are not lexicographically ordered. Is not until the move to Table 3.3 (d) that the rows are lexicographically ordered and the algorithm can continue to the next step the *Partial t-Wise Verification.*

3.2.2 Partial t-Wise Verification

Since a CA of strength $t-1$ is present within a CA of strength t we can bound the search space more quickly and efficiently if we partially verify for a CA instead of waiting until all k columns are inserted. We can test the first $t-1$ columns with strength i, where i is the current element inserted,

and when i is greater or equal to t then is tested with strength t. It is important to remark that a complete CA verification is more expensive in terms of time,[1] than making the partial evaluations described above due to the intrinsic characteristics of the backtracking algorithm.

A detailed example of partial t-wise verification is presented in Table 3.4.

Table 3.4: Example of backtracks when the current column in the partial solution M is not a covering array

(a)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(c)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(d)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

[1] The computational complexity of making a full verification of a CA$(N; t, k, v)$ is $N \times \binom{k}{t}$

From Table 3.4 we can make the following observations: in this example the objective is to find a CA(8;4,2,3), but the next valid moves from Table 3.4 (a) through Table 3.4 (c) when applying the partial t-wise verification the 4th column does not produce a valid CA. It is not until the matrix reaches the configuration seen in Table 3.4 (d) that the matrix M produces the CA(8;4,2,3).

3.2.3 Fixed Block

The search space of this algorithm can be greatly reduced if we use a *Fixed Block* (FB). We define a FB as matrix of size N and length t in which the first $\lfloor\frac{(N-v^t)}{2}\rfloor$ rows are filled with 0's, then a CA of strength t, $k = t$ and $N = v^t$ is inserted. This CA can be easily generated (in polynomial time) by creating all the v^t binary numbers and listing them in order. An example of a CA(8;3,3,2) is shown in Table 3.5. Finally, the last $\lceil\frac{(N-v^t)}{2}\rceil$ rows are filled with 1's. It can be easily verified that in a FB the rows and columns are already lexicographically ordered. This FB is construct in this way in order to preserve the symmetry breaking rules proposed in Sect. 3.2.1 and is used to initialize our algorithm as shown in Figure 3.1.

Table 3.5: A CA(8;3,3,2) example.

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

This type of construction has its advantages and disadvantages, some of them are:

- Advantages

1. The search space its reduced as it only checks balanced CAs.
2. The matrices are strictly lexicographically ordered by row and column.
3. It is partially verified to be a CA to reduce the computational effort.
4. The CAs are evaluated in an incremental way.

- Disadvantages

1. Every new column inserted must be verified to determine if the new column meets the ordered-by-row criteria.
2. The FB is not the best initialization possible so it has to backtrack most of the times to the first t columns.

In order to eliminate these disadvantages we proposed a new construction inspired by a work of Karen Meagher, this construction will guarantee to construct matrices ordered by row and column automatically. The objective to introduce this construction is to get rid of the time validating if the matrix is still ordered by row.

3.3 Improved Backtracking Algorithm (IBA)

The first step in the IBA is to identify new rules in order to apply valid symbol changes without altering the lexicographically ordered properties of the matrix.

Below is an example of the initialization in order to search a CA(12,11,3,2), as proposed in section 3.2.3 the first 3 columns are constructed as seen in Table 3.6.

The first step is to divide in blocks of equal and different rows. This will help to do symbol changes without altering the lexicographically ordered properties. In this algorithm an equal counter starting with 1 will represent all the first block of equal rows, after a different row is reached the counter is increased and the algorithm continues. For a row that has no equal rows it is represented with 0's. Table 3.7 represent a matrix with the equal counter (ECM) for each column. Each of the columns of Table 3.7 represents the equal rows of the CA until reaching that specific column,

such that if in the third column and second row there is a value of "1", that means that in the corresponding CA that row until the third column is part of the first block of equal rows.

1	1	1
1	1	1
1	1	1
1	1	0
1	2	0
1	2	0
2	3	0
2	3	0
2	4	0
2	4	2
2	4	2
2	4	2

Table 3.7: The scheme of the equal counters of table 3.6.

After creating the matrix of equal counters a simple rule applies. Within a block of equal rows there can not be a 0 after a 1, this will guarantee that the rows will remain ordered. Now in order to guarantee the lexicographically ordered by column rule, the only valid moves are to try to move the last 0 in a column to the next valid position below it. In order to generate the next possible column, the first step is to copy the $i-1$ column into the i column. After copying it we try to move the last 0, if it can not be moved, then the next 0 is searched and tried to moved, and if no 0's can be moved then it backtracks to the last column, this process iterates until a valid column is reached.

In Table 3.8 the next valid move for the 4th column is detailed in these steps: it first copies the 3rd column to the next column, then in the current column the last zero is tried to push downwards following the structure in Table 3.7 in order to maintain a matrix lexicographically ordered by row and column.

As we can see maintaining these simple rules we can make direct movements that guarantee that

the matrix is lexicographically sorted by rows and columns. Adding the previous techniques described in section 3.2 is not complicated as the construction is still generated column by column and the partial t-wise verification still applies. The previous example in fact has the columns balanced and if the algorithm is followed, the number of 0's and 1's will remain the same for each column.

Even though it is really simple to follow these rules, there are two different cases and a special case that needs to always be verified in order to guarantee that the full search space is verified in order to assure that our algorithm is complete. This two cases and the special case will be described in the following sections.

3.4 Special Cases for the IBA

There are two cases that can occur when creating the next valid move via the structure of the equal counters. The first case is when the 0 selected of an specific column belongs to a block of equal rows. The second case is the opposite when the 0 that is going to be moved corresponds to a block of different rows.

Using the structure of equal counters in Table 3.7 an example of the two cases will be explained in detail. In the first case when the 0 to be moved belongs to a block of equal rows, in order to move it to the next valid position, it needs to be verified that its final destination corresponds to a block of a different enumeration of the structure of equal counters. If there is no valid move, the next 0 in the column is selected. This process iterates until a valid move is reached or until there are no 0's that can be moved.

An example where the 0 corresponds to a block of equal rows and it can not be moved is shown in Table 3.9. Refer to Table 3.7 to check the values of the equal counters. In this example we can see that in Table 3.9 (a) the last zero is in italic, and it corresponds to a block of equal rows and there is no group with a different enumeration below it, this can be checked in the structure of equal counters in Table 3.7. Since the last zero can not be moved, the next zero is marked in bold font. This zero corresponds to a group of different rows and there is no restriction to move it, so the next valid move can be seen in Table 3.9 (b). It is important to mention that both cases were covered in

this example, as the zero in bold font was in a group of different rows.

Table 3.9: Next valid move for the 4th column of the partial solution M when the 0 corresponds to a block of equal rows

(a)

0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	**0**
1	1	0	1
1	1	1	*0*
1	1	1	1
1	1	1	1

(b)

0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	**0**
1	1	1	*0*
1	1	1	1
1	1	1	1

The special case mentioned at the beginning of this section needs to be checked every time a zero that is not in the last position is moved. In order to check within all the search space of the problem to guarantee that this algorithm remains as a complete method, every time a zero is moved two steps are followed : first if the position of the original zero is part of a block of equal rows, all the rows below that belongs to the same block are filled with 1's, then all the zeros below the initial position of the zero moved below need to be rearranged such that after the last symbol 1 inserted the remaining zeros must be listed subsequently, after there are no zeros remaining the column is filled with 1's.

An example of this special case where the selected zero belongs to a block of different rows is shown in Table 3.10.

Table 3.10: Next valid move for the 4th column of the partial solution M when the zero moved is not the last one and belongs to a block of different rows

(a)

0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1
0	1	0	**0**
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	*0*
1	1	1	*0*
1	1	1	1

(b)

0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1
0	1	0	**1**
0	1	1	*0*
1	0	0	*0*
1	0	1	*0*
1	1	0	1
1	1	1	1
1	1	1	1
1	1	1	1

As seen in Table 3.10 (a) the last two zeros in italic can not be moved, the zero in bold font is the one selected instead and it belongs to a group of different rows. Note that in Table 3.10 (b) the position of the zero in bold font of Table 3.10 (a) is changed to 1 and below this position are the 3 remaining 0's in italic. The remaining positions of the column were filled with 1's. After doing these steps we can clearly observe that the partial solution M is still lexicographically ordered by row and column.

The next example will involve the special case where the selected zero belongs to a block of equal rows. This example will be shown in Table 3.11.

Table 3.11: Next valid move for the 4th column of the partial solution M when the zero moved is not the last one and belongs to a block of equal rows

(a)

0	0	0	0
0	0	0	**0**
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	*0*
1	1	1	*0*
1	1	1	*0*
1	1	1	*0*

(b)

0	0	0	0
0	0	0	**1**
0	0	0	**1**
0	0	1	*0*
0	1	0	*0*
0	1	1	*0*
1	0	0	*0*
1	0	1	*0*
1	1	0	1
1	1	1	1
1	1	1	1
1	1	1	1

As seen in Table 3.11 (a) the last four zeros in italic can not be moved, the zero in bold font is the one selected instead and it belongs to a group of equal rows. Note that in Table 3.11 (b) the position of the zero in bold font of Table 3.11 (a), and because it belongs to a block of equal rows, the row below it was filled with 1's. Then the remaining five 0's in italic were added. The remaining positions of the column were filled with 1's. After doing these steps we can clearly observe that the partial solution M is still lexicographically ordered by row and column.

After seeing these examples, a flowchart of the improved version of the algorithm is shown in Figure 3.2.

The main difference in the IBA and the NBA is very clear seeing the flowcharts of Figure 3.1 and Figure 3.2, the loops made in the NBA until a valid matrix M were removed in the IBA with the construction of a matrix of equal counters (ECM). And the computational time expended in updating the ECM is of order $O(N)$.

3.5 Summary of the Chapter

In this chapter we explained that our goal in this methodology was to search within the non-isomorphic search space of covering arrays. We explained the techniques used to develop the NBA and also the new methodology proposed to develop the IBA. Two flow charts showing the description of each algorithm were shown in order to give a general idea of our implementation and also some examples of our constructions were presented step by step. In the next chapter the computational results for both the NBA and the IBA will be given in detail.

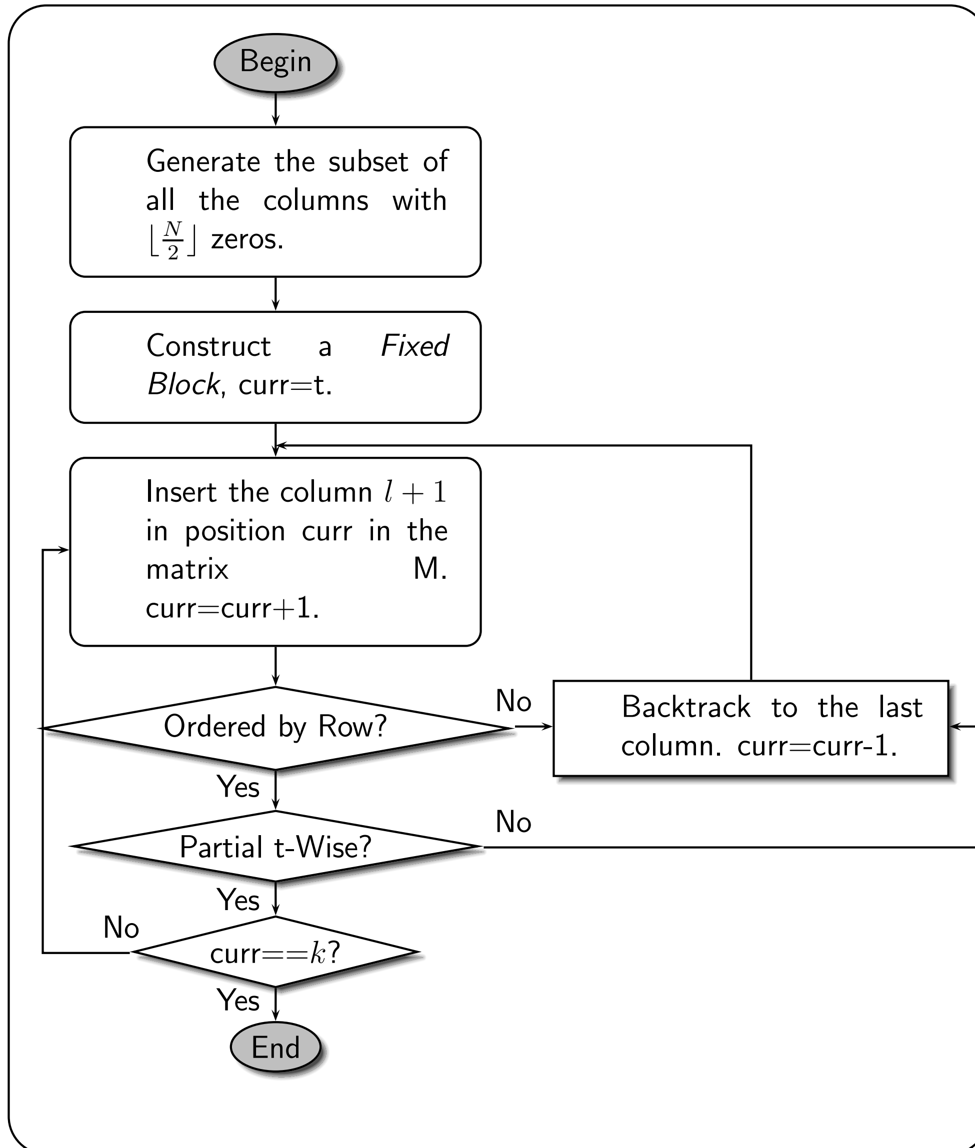


Figure 3.1: Flow chart of the new backtracking algorithm.

Table 3.3: Example of invalid moves in the partial solution M when the rows does not remain ordered

(a)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	**0**
1	1	1	**1**
1	1	1	1

(b)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
1	1	1	**0**
1	1	1	**1**

(c)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	1
1	1	1	1
1	1	1	**0**

(d)

l	$l+1$	$l+2$	$l+3$
0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1
1	1	1	1
1	1	1	1

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

Table 3.6: Example of the initialization of the IBA using a CA(12;3,3,2).

Table 3.8: Next valid move for the 4th column of the partial solution M

(a)

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

(b)

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

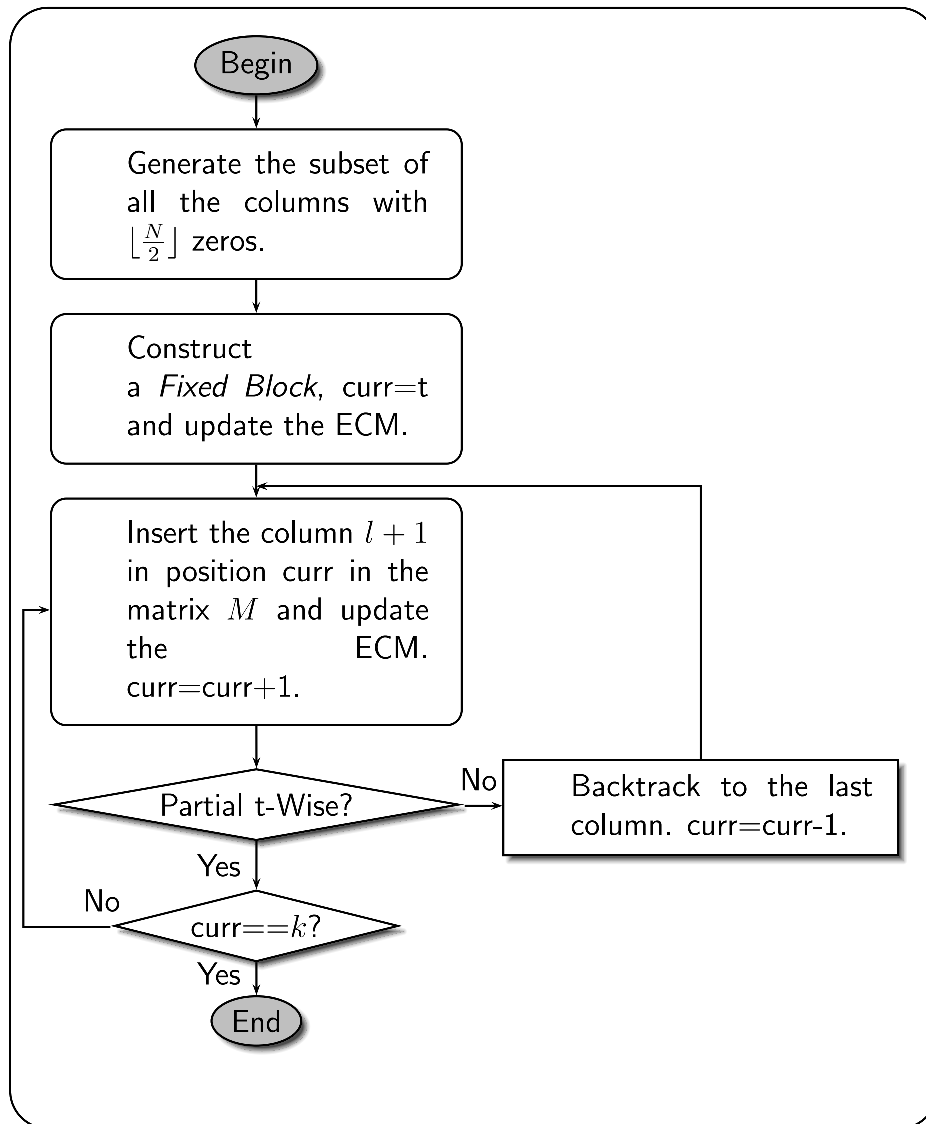


Figure 3.2: Flow chart of the improved backtracking algorithm.

4

Experimental Results

This chapter will present some comparisons between methods published to construct CAs in the literature and our algorithms proposed in Chapter 3. A table and a graph detailing the results are included in each one of the tests performed. The comparative criteria used was the one commonly used in the literature the best N, and for cases where the best N was achieved for both algorithms, we used as a criteria the computational effort expended in order to reach the solution.

4.1 Computational Results

4.1.1 Test Instances

In this chapter, we present a set of experiments used to evaluate the performance of the 2 proposed algorithms from Chapter 3. The algorithms were coded in C and compiled with gcc without optimization flags. They were run sequentially into a CPU Intel Core 2 Duo at 1.5 GHz, 2 GB of RAM with Linux operating system.

The test suite used for the experimentation is composed of 14 well known binary covering arrays

of strength $3 \leq t \leq 5$ taken from the literature [14, 55].

4.1.2 Comparative Criteria

The main criterion used for the comparison is the same as the one commonly used in the literature: the best size N found (smaller values are better) given fixed values for k, t, and v. When both algorithms reached the best N reported in the literature, the next criteria used is the computational effort. The timing of the algorithms were taken using the UNIX command *time*.

4.1.3 Comparison Between NBA and EXACT

The purpose of this experimentation is to carry out a performance comparison of the upper bounds achieved by our NBA with respect to those produced by the EXACT procedure [55]. For this comparison we have obtained the EXACT algorithm from the authors. Both algorithms were run in the computational platform described at the beginning of the chapter.

Table 4.1 displays the detailed computational results produced by this experiment. The first two columns in the table indicate the degree k, and strength t of the instance. Columns 3 and 5 show the best solution N found by B&B and the EXACT algorithms, while columns 4 and 6 depict the computational time T, in seconds, expended to find those solutions.

Table 4.1: Performance comparison between NBA and EXACT.

		NBA		EXACT	
k	t	N	T	N	T
4	3	8	**0.005**	8	0.021
5	3	10	**0.005**	10	0.021
6	3	12	**0.008**	12	0.023
7	3	12	**0.018**	12	0.024
8	3	12	0.033	12	0.023
9	3	12	0.973	12	0.022
10	3	12	0.999	12	0.041
11	3	12	0.985	12	0.280
12*	3	15	**1090.800**	15	1100.400
5	4	16	**0.020**	16	0.038
6	4	21	95.920	21	0.266
6	5	32	102.000	32	0.025
Average			107.64		91.76

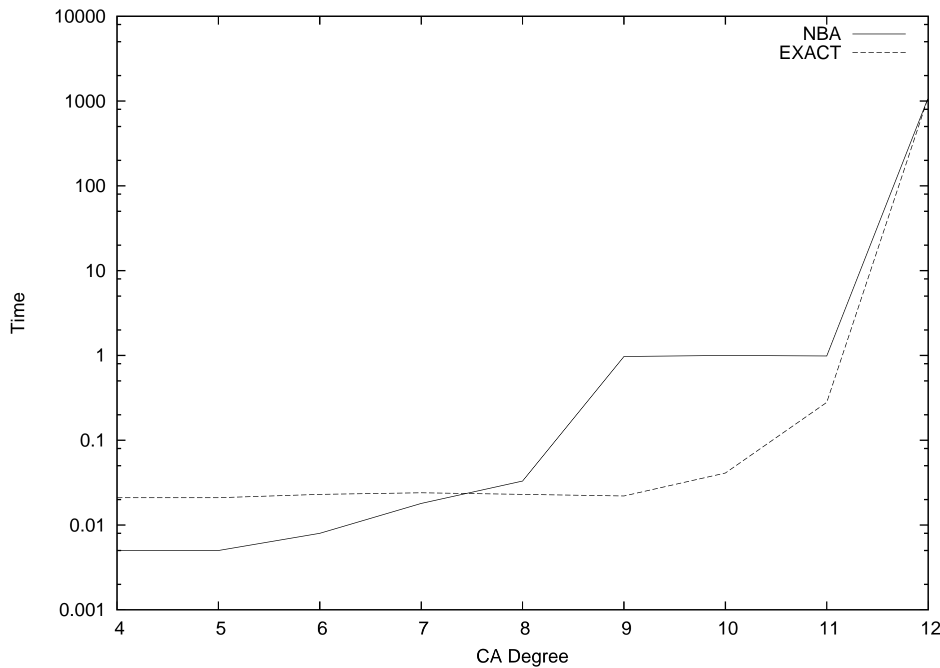


Figure 4.1: Graphic of performance between NBA and EXACT

From the data presented in Table 4.1 we can make the following main observations. First, the solution quality attained by the NBA is very competitive with respect to that produced by the state-of-the-art procedure EXACT. In fact, it is able to consistently equal the best-known solutions attained by the EXACT method (see columns 3 and 5). Secondly, regarding the computational effort, one observes that in this experiment the EXACT algorithm consumes slightly more computational time than B&B for 6 out 12 benchmark instances (shown in boldface in column 4).

We would like to point out that the instance marked with a star in Table 4.1 was particularly difficult to obtain using the EXACT algorithm. We have tried many different values for the parameter SCEH (Sub-Combination Equalization Heuristic), and only using a value of 1 the EXACT tool was able to find this instance consuming more CPU time than our NBA algorithm. For the rest of the experiments we have used the default parameter values recommended by the authors.

A graphic showing the performance of both algorithms in a logarithmic scale and for strength $t = 3$ is presented in Figure 4.1.

We can conclude with respect of the results shown in Figure 4.1, that our NBA performed better

against the EXACT algorithm in smaller instances, but when the instances started to grow as seen in Table 4.1, the EXACT algorithm outperformed our NBA.

4.1.4 Comparison Between our IBA and the Exact

Considering the last results we developed a new version of the algorithm described in the last chapter. The purpose of this experiment is to carry out a performance comparison between the improved version of our backtracking algorithm (IBA) and the EXACT procedure [55]. We will make the same comparisons as in Table 4.1 using the same architecture described in the beginning of this chapter. This comparison is presented in Table 4.2 and we can make the following main observations. First, the solution quality attained by the improved version of the backtracking algorithm still obtain the best results achieved by our initial algorithm and the EXACT.

Secondly, in terms of computational effort, we can clearly observe that this version of the algorithm outperforms our first algorithm the NBA and also consumed slightly less computational time in more cases than the EXACT algorithm. In fact, EXACT produces CAs using in average 86.596% more computational resources than our improved version of the algorithm.

Table 4.2: Performance comparison between the improved B&B and EXACT.

		IBA		EXACT	
k	t	N	T	N	T
4	3	8	**0.004**	8	0.021
5	3	10	**0.004**	10	0.021
6	3	12	**0.005**	12	0.023
7	3	12	**0.012**	12	0.024
8	3	12	**0.012**	12	0.023
9	3	12	0.420	12	0.022
10	3	12	0.200	12	0.041
11	3	12	**0.104**	12	0.280
$12^{\star}$	3	15	**17.145**	15	1100.400
5	4	16	**0.003**	16	0.038
6	4	21	44.067	21	0.266
6	5	32	**0.003**	32	0.025
Average			**5.164**		91.760

A graphic showing the performance of both the IBA and the EXACT algorithms in a logarithmic scale for strength $t = 3$ is presented in Figure 4.2.

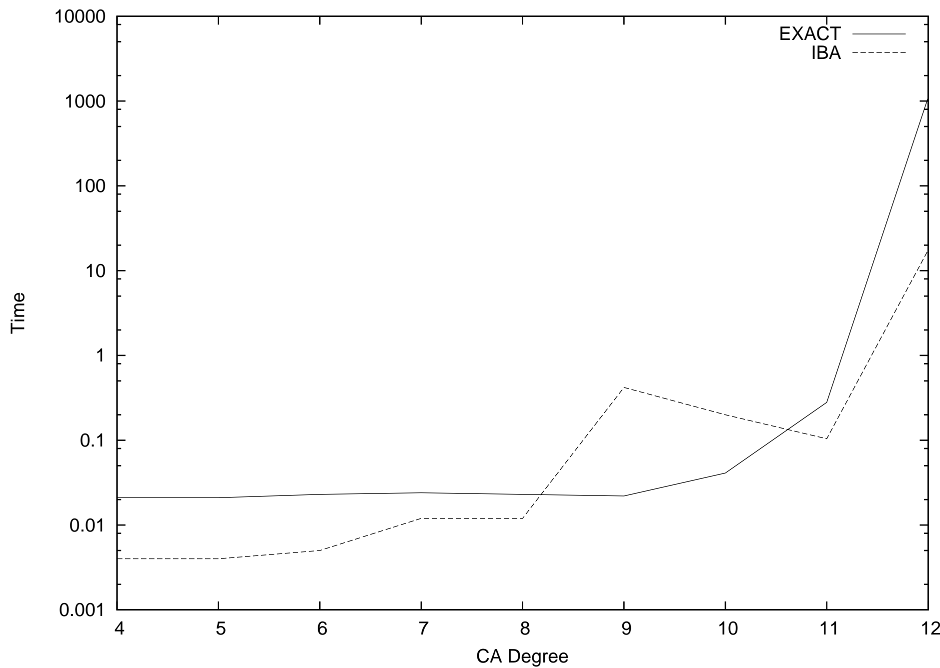


Figure 4.2: Graphic of performance between IBA and EXACT

We can conclude with respect of the results shown in Figure 4.2, that the overall performance of our IBA was better compared to the performance of the EXACT, but there are some peaks in Figure 4.2 that reflect some instances that were harder to solve.

4.1.5 Comparison Between IBA and IPOG-F

In a third experiment we have carried out a performance comparison of the upper bounds achieved by our IBA with respect to those produced by the state-of-the-art procedure called IPOG-F [20].

Table 4.3 presents the computational results produced by this comparison. Columns 1 and 2 indicate indicates the degree k, and strength t of the instance. The best solution N found by our B&B algorithm is depicted in column 3, while that reached by the IPOG-F algorithm is presented in column 4. Finally, the difference (Δ_N) between the best result produced by our IBA algorithm compared to that achieved by IPOG-F is shown in the last column.

From Table 4.3 we can clearly observe that in this experiment the IPOG-F procedure [20] consistently returns poorer quality solutions than our IBA. Indeed, IPOG-F produces covering arrays

Table 4.3: Performance comparison between the algorithms IBA and IPOG-F.

k	t	N IBA	N IPOG-F	Δ_N	Time IBA	Time IPOG-F
4	3	8	9	-1	0.004	1.724
5	3	10	11	-1	0.004	1.560
6	3	12	14	-2	0.005	1.623
7	3	12	16	-4	0.012	1.568
8	3	12	17	-5	0.012	1.720
9	3	12	17	-5	0.420	1.672
10	3	12	18	-6	0.200	1.487
11	3	12	18	-6	0.104	1.750
12	3	15	19	-4	17.45	1.764
13	3	16	20	-4	975.91	1.875
5	4	16	22	-6	0.003	1.280
6	4	21	26	-5	44.067	1.393
7	4	24	32	-8	1350.267	1.446
6	5	32	42	-10	0.003	1.487
Average		15.29	20.07	-4.79		

which are in average 31.26% worse than those constructed with B&B. But being the IBA a complete method, as the size of the instances begin to grow, the computing time grows as well.

A graphic showing the performance of both the IBA and the IPOG-F algorithms in terms of solution quality and for strength $t = 3$ is presented in Figure 4.3.

Figure 4.3 shows how the performance in terms of the solution quality. Our IBA can find the best solutions reported in the literature, while the IPOG-F algorithm in some instances was really far away from the best solution.

4.1.6 Comparison Between NBA and IBA

The purpose of this experiment is to carry out a performance comparison between the improved version of our backtracking algorithm (IBA) and the first approach called (NBA). We will make the same comparisons as in Table 4.1 using the same architecture described in the beginning of this chapter. This comparison is presented in Table 4.4 and we can make the following main observation. In terms of computational effort, we can clearly observe that this version of the algorithm outperforms our initial version the NBA algorithm. In fact, NBA produces CAs using in average 102.476 more computational resources than our improved version of the algorithm.

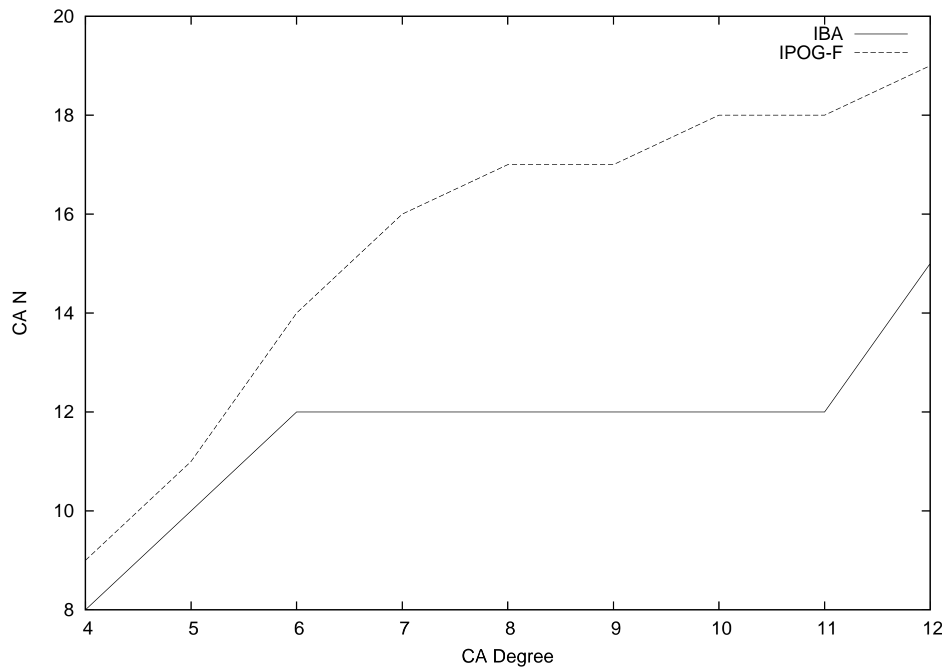


Figure 4.3: Graphic of performance between IBA and IPOG-F

A graphic showing the performance of both the IBA and the NBA algorithms in a logarithmic scale and for strength $t = 3$ is presented in Figure 4.4.

We can conclude with respect of the results shown in Figure 4.4, that our IBA outperform our first approach of the backtracking algorithm.

4.1.7 Comparison Against the Best Known Solutions

With respect to the quality of the solution Colbourn et al. [16] has recently written an article that establishes upper and lower bounds and in some cases the CAN for multiple alphabets and strength. Regarding our work the results for binary alphabet reported by Colbourn et al. [16] will be shown in Table 4.5 and also our best N found with our program will be presented in the table as well.

What we can see in Table 4.5 is that in all the cases where the optimum value was imposed, our B&B reached the same result as Colbourn et al. [14], and for the case where $k = 13$ and $k = 14$ for strength 3, the optimum value oscillates between 15 and 16, but our IBA could not found a CA with $N = 15$, it only found a CA with $N = 16$. Therefore, we can state that there is no balanced

Table 4.4: Performance comparison between the IBA and the NBA.

		IBA		NBA	
k	t	N	T	N	T
4	3	8	**0.004**	8	0.005
5	3	10	**0.004**	10	0.005
6	3	12	**0.005**	12	0.008
7	3	12	**0.012**	12	0.0018
8	3	12	**0.012**	12	0.033
9	3	12	**0.420**	12	0.0973
10	3	12	**0.200**	12	0.0999
11	3	12	**0.104**	12	0.985
$12^{\star}$	3	15	**17.145**	15	1090.800
5	4	16	**0.003**	16	0.020
6	4	21	**44.067**	21	95.920
6	5	32	**0.003**	32	102.000
Average			**5.164**		107.564

CA(15;13,3,2) or CA(15;14,3,2) and with a great probability the optimum case is $N = 16$.

4.2 Summary of the Chapter

In this chapter the experiments with our first version and improved version of the algorithm (NBA and IBA) were performed. We assessed our algorithms against the results of the EXACT algorithm and the famous IPOG-F algorithm, our proposed methodology proved to be very competitive in terms of quality of solution and computational time. Comparing our algorithms against the IPOG-F algorithm, the quality of the solution achieved with our algorithm were in most of the cases better. Lastly, we compare our NBA algorithm against our IBA with good results, proving the fact that for all the instances our IBA performed better than our NBA. The last section showed a comparison of the best known solutions for various strengths and in all of them we achieved the same results. And an important fact was that we demonstrate that there were no balanced solution for $k = 13$ and $k = 14$ for strength $t = 3$ with $N = 15$. In the next chapter we will discuss in detail our work, and also point out our limitations and future work.

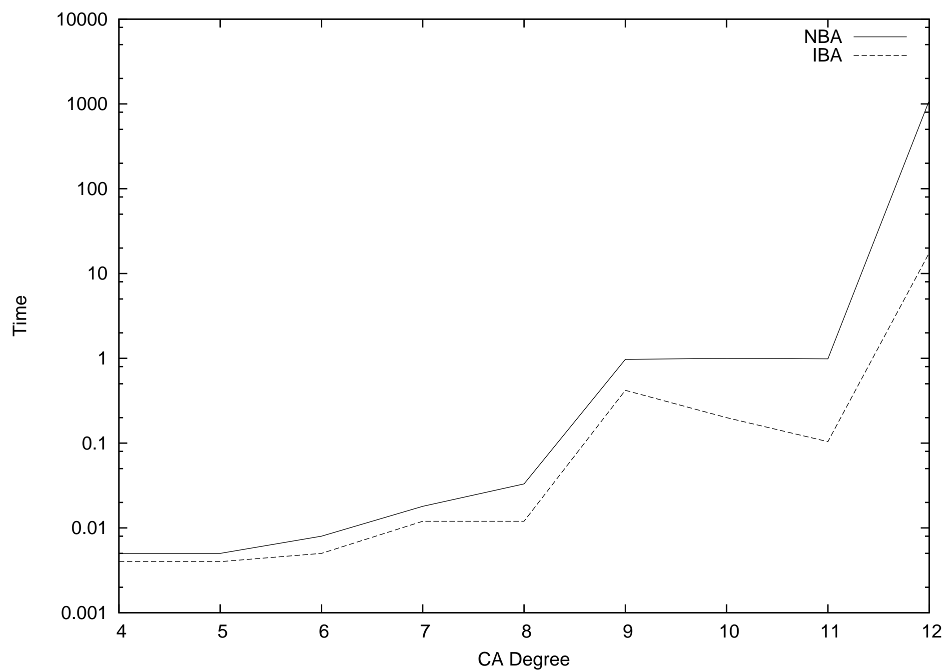


Figure 4.4: Graphic of performance between IBA and NBA

k	t	Colbourn	B&B
3	3	8	8
4	3	8	8
5	3	10	10
6	3	12	12
7	3	12	12
8	3	12	12
9	3	12	12
10	3	12	12
11	3	12	12
12	3	15	15
13	3	15-16	16
14	3	15-16	16
4	4	16	16
5	4	16	16
6	4	21	21
7	4	24	24
5	5	32	32
6	5	32	32

Table 4.5: Lower and upper bounds for binary alphabet

5

Conclusions and Future work

In previous chapters we have described the development of the proposed methodology, its implementation, and the results obtained with the algorithm proposed. This final chapter contains a general summary of the work presented previously, a brief discussion about the methodology, the difficulties in its development, and its limitations. In the last section some lines of future work are suggested related to our approach.

5.1 Summary of the Thesis

We proposed a new backtracking algorithm the NBA that implements some symmetry breaking techniques to reduce efficiently the search space. Additionally, we have presented a new technique for breaking the symbol symmetry which allow to reduce considerably the size of the search space as well as an algorithm to construct automatically lexicographically ordered matrices by rows and columns (IBA). Experimental comparisons were performed and they show that our IBA is able to match some of the best-known solutions for small instances of binary CAs, expending in some cases less computational time compared to another existent backtracking algorithm called EXACT. We

have also carried out a comparison of the upper bounds achieved by our backtracking algorithm with respect to those produced by a state-of-the-art procedure called IPOG-F. In this comparison the results obtained by our backtracking algorithm, in terms of solution quality, are better than those achieved by IPOG-F for all the studied instances. Finding optimum solutions for the CA construction problem in order to construct economical sized test-suites for software interaction testing is a very challenging problem. We hope that the work reported in this thesis could shed useful light on some important aspects that must be considered when solving this interesting problem. We also expect the results shown in this work incite more research on this topic. For instance, one fruitful possibility for future research is the design of new pruning heuristics in order to have the possibility to generate larger instances of CAs.

5.2 Discussion of the Methodology

The key aspects of this work are: 1) the rules in order to efficiently construct balanced and lexicographically ordered matrices, and 2) the guarantee to find a covering array or prove their nonexistence.

The main advantage of our proposal is that we efficiently generate lexicographically ordered matrices. In our first approach, in order to proceed with the partial t-wise verification the column must meet the requirement that it does not violate the ordered by row property. When a block of equal rows of size ρ is found, our first approach tried every 2^ρ movements until an specific configuration that meets the ordered by row property was found. Instead of trying every possible movement a new approach that automatically construct ordered matrices was developed. In this new approach a new matrix of size $N \times k$ is created for the sole purpose of storing the information of the equal rows per column. With the aid of this matrix we only generate the set of $\rho + 1$ valid movements. This lead us to an increase in the overall performance as shown in Chapter 4.

Another advantage is that we search only within the space of the balanced symbol matrices, this space reduction was inspired by observing the properties that optimal or near optimal CAs reported in the literature had exactly or near exactly the same number of symbols per column. Based on this

property we start our search of CAs with the restriction that each column must have exactly $\lfloor \frac{N}{2} \rfloor$ zeros. Using this property we were able to find every solution reported as optimal in the literature. Even though, we could not demonstrate that there is always a balanced symbol solution.

Another technique implemented in order to search more efficiently was the partial verification of the matrix. With the aid of this technique we could know beforehand if a certain valid column may produce a CA, since within a CA of strength t there exists always a CA of strength $2 \leq t \leq t-1$. Waiting until k columns were inserted to validate if the matrix is a CA requires a computational effort of $N \times \binom{k}{t}$, moreover, we would not have any knowledge of where to backtrack as there could be more than 1 column that causes conflicts. An alternative is to partially verify for a CA since the insertion of the third column, given that it help us to advance efficiently through feasible solutions.

Even though there were techniques that help us to reduce the search space, the problem grows exponentially as the strength t and the number of columns k increase. For this reason we could not work with large instances and we were limited to strength $2 \leq t \leq 5$. Another problem with our approach was the initialization. In order to preserve our algorithm as a complete method we proposed an initialization that guarantees to start from the first balanced CA of size N, strength t and with t columns. But, according to the results obtained, this initialization proved not to be the best one as there were certain cases where a substantial increase in terms of time was presented. When this occurred it meant that our algorithm had to backtrack to the first t columns, this huge number of backtracks leads to an exponential increase in time. We tried different initializations, these different initializations favored some CAs and worsen other CAs. We think this behavior is due to that these initialization approaches are skipping some regions of the search space.

5.3 Challenges Confronted in the Development of this Thesis

During the development of the proposed methodology, we face up to some difficulties, among them: our first challenge was to find an efficient way to construct lexicographically ordered matrices

automatically. While researching the literature different works were found that proposed methods to break the row and column symmetry [38, 54, 55], but in those works they also used a process to validate if the rows were ordered. In a technical report of Karen Meagher [38] she described some rules that will help to the development of an algorithm to construct lexicographically ordered matrices automatically. While these rules helped, the problem of finding an efficient algorithm that uses these rules was still missing. After studying more the problem, we identified the conflict zone that could lead to disordered matrices, the main problem was alternating 1's and 0's within a block of equal rows. After discovering this problem the next step was to develop an algorithm to find those block of equal rows and make use of them in order to efficiently search through lexicographically ordered matrices. Our solution was the creating of the *Equal Counter Matrix* described in Chapter 3, this matrix required a computational effort of N to keep it updated and proved to be an useful aid to construct balanced binary covering arrays of variable strength.

5.4 Scope of the Proposed Approach

The major limitation in this research is the huge search space, being the covering array construction problem a well known NP problem [35], generating larger instances was a major problem as the search space increases along with the number of columns and strength. Our algorithm is intended to be used for constructing small balanced instances.

5.5 Future work

In this section, we describe some lines of future work and we present some preliminary ideas about how they can be tackled.

1. *Improvement of the current methodology*. The following issues must be considered:

    - Finding new pruning techniques that can be added in order to reduce the search space to construct larger instances or validate its optimality

- Generate an efficiently initial solution, such that no relevant search space has been skipped and can continue to be called a complete method
- Perform a mathematical theorem in order to prove that there is always a CA with balanced symbols per column

Bibliography

[1] E.H. Aarts, J. Korst, and P.J. van Laarhoven. *Local Search in Combinatorial Optimization*. John Wiley and Sons Inc, 1997.

[2] R. Bryce and C. Colbourn. The density algorithm for pairwise interaction testing: Research articles. *Software Testing Verification and Reliability*, 17(3):159–182, 2007.

[3] R. Bryce, Y. Lei, D. Kuhn, and R. Kacker. *Combinatorial Testing*, chapter 14. Engineering Science Reference, 2009.

[4] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and contraints. *Information and Software Technology Journal (IST, Elsevier)*, 48:960–970, 2006.

[5] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. *International Conference on Software Engineering (ICSE 05)*, pages 146–155, 2005.

[6] K. A. Bush. Orthogonal arrays of index unity. *Annals of Mathematical Statistics*, 13:426–434, 1952.

[7] M. A. Chateauneuf. *Covering Arrays*. PhD thesis, Michigan Technological University, 2000.

[8] M. B. Cohen, C. J. Colbourn, and Ling A. C. H. Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics*, 308:2709–2722, 2008.

[9] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. *14th International Symposium on Software Reliability Engineering (ISSRE 03)*, pages 394–405, 2003.

[10] M. B. Cohen, S. R. Dalal, J. Parelius, and C. Gardner. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[11] M. B. Cohen, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[12] M. B. Cohen, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proccedings of the International Conference on Software Engineering (ICSE 2003)*, pages 38–48, 2003.

[13] C. Colbourn. Combinatorial aspects of covering arrays. Technical report, Arizona State University, 2004.

[14] C. J. Colbourn. Most recent covering arrays tables. Web Page, Last Time Accessed 4-Nov-2009. http://www.public.asu.edu/ ccolbou/src/tabby/catable.html.

[15] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. *Proceedings of the IASTED International Conference on Software Engineering*, 2004.

[16] C. J. Colbourn, G. Kéri, P. P. Rivas, and J. C. Schlage. Covering and radius-covering arrays: Construction and classification. *Elsevier*, 2009. preprint version.

[17] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:29–41, 1996.

[18] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing: experience report. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 205–215. ACM, 1997.

[19] R. Fisher and R. Aylmer. *Statistical methods for research workers / by R.A. Fisher*. Oliver and Boyd, Edinburgh, UK:, 11th ed.(rev.) edition, 1950.

[20] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113(5):287–297, 2008.

[21] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Software testing verification & reliability*, 15:167–199, 2005.

[22] A. Hartman. *Software and Hardware Testing Using Combinatorial Covering Suites*, volume 34 of *Operations Research/Computer Science Interfaces Series*, chapter 10, pages 237–266. Graph Theroy, Combinatorics and Algorithms, 2005.

[23] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284:149–156, 2004.

[24] A. S. Hedayat, N. J. A. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Springer-Verlag, New York, USA, 1999.

[25] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

[26] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, Massachusetts, USA, 1992.

[27] S. Kirkpatrick. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[28] W. H. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of ACM*, 21(1):140–156, 1974.

[29] C. Koukouvinos and E. Lappas. Construction of two level orthogonal arrays via solutions of linear systems. In *Computer Algebra in Scientific Computing*, pages 285–293, 2005.

[30] D. R. Kuhn and V. Okum. Pseudo-exhaustive testing for software. In *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 153–158, Washington, DC, USA, 2006. IEEE Computer Society.

[31] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. *Proceedings 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[32] D. R. Kuhn, D. R. Wallance, and A. M. Jr. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004.

[33] E. Lawler and D. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[34] Y. Lei, R. Kacker, R. D. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 07)*, pages 549–556, 2007.

[35] Y. Lei and K.C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *In Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering*, pages 254–261, Washington, DC, USA, 1998. IEEE Computer Society.

[36] N. G. Levenson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[37] R. Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Communications ACM*, 28(10):1054–1058, 1985.

[38] K. Meagher. Non-isomorphic generation of covering arrays. Technical report, University of Regina, 2002.

[39] Shell Method. The shell method software engineering process repository. Web Page, Last Time Accessed 4-Nov-2008. http://www.shellmethod.com.

[40] L. B. Morales. Scheduling a bridge club by tabu search. *Mathematics Magazine*, 1997.

[41] L. Moura, J. Stardom, B. Stevens, and A. Williams. Covering arrays with mixed alphabet sizes. *Journal of Combinatorial Designs*, 11(6):413–432, 2003.

[42] K. J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Appl. Math.*, 138(1-2):143–152, 2004.

[43] R. Fisher. The arrangement of field experiments. *Journal of Ministry of Agriculture of Great Britain*, 33:503–513, 1926.

[44] G. Seroussi and N. H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.

[45] G. Sherwood. On the construction of orthogonal arrays and covering arrays using permutations groups. web page. http://home.att.net/gsherwood/cover.htm.

[46] T. Shiba, T. Tsuchiya., Kikuno, and K. Tohru. Using artificial life techniques to generate test cases for combinatorial testing. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference*, pages 72–77. IEEE Computer Society, 2004.

[47] N. J. A. Sloane. Covering arrays and intersecting codes. *Journal of Combinatorial Designs*, 1:51–63, 1993.

[48] N. J. A. Sloane and J. Stufken. A linear programming bound for orthogonal arrays with mixed levels. *Journal of Statistical Planning and Inference*, 56:295–305, 1996.

[49] D. R. Smith. Random trees and the analysis of branch and bound procedures. *Journal of ACM*, 31(1):163–188, 1984.

[50] J. Stardom. Metaheuristics and search for covering and packing arrays. Master's thesis, Simon Fraser University, May 2001.

[51] B. Stevens. *Transversal Covers and Packings*. PhD thesis, University of Toronto, 1998.

[52] Gregory Tassey and RTI. The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD, USA, May 2002.

[53] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. In *AICCSA*, pages 304–312, 2001.

[54] J. Yan and J. Zhang. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. *Computer Software and Applications Conference(COMPSAC 06) 30th Annual International*, 1:385–394, 2006.

[55] J. Yan and J. Zhang. A backtracking search tool for constructing combinatorial test suites. *The journal of systems and software*, 81(10):1681–1693, 2008.

[56] T. Yu-Wen and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In IEEE Computer Society, editor, *Proceedings of the IEEE Aerospace Conference*, pages 431–437, 2000.