UT-T00061-SS1
DON.-7014

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Laboratorio de Tecnologías de Información,
CINVESTAV-Tamaulipas

Construcción de Torres de Covering Arrays

Tesis que presenta:

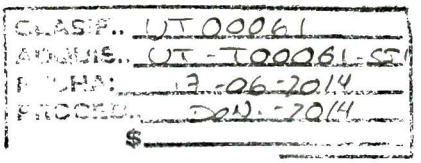
Idelfonso Izquierdo Márquez

Para obtener el grado de:

Maestro en Ciencias en Computación

Director de la Tesis:
Dr. José Torres Jiménez

Cd. Victoria, Tamaulipas, México. Agosto, 2013



ID: 213688-1001

RESEARCH CENTER FOR ADVANCED STUDY
FROM THE NATIONAL POLYTECHNIC INSTITUTE

Information Technology Laboratory,
CINVESTAV-Tamaulipas

Construction of Towers of Covering Arrays

Thesis by:

Idelfonso Izquierdo Márquez

as the fulfillment of the
requirement for the degree of:

Master of Science
in Computer Science

Thesis Director:
Dr. José Torres Jiménez

Cd. Victoria, Tamaulipas, México. August, 2013

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

The thesis of Idelfonso Izquierdo Márquez is approved by:

Dr. Arturo Díaz Pérez

Dr. Wilfrido Gómez Flores

Dr. José Torres Jiménez, Committe Chair

Cd. Victoria, Tamaulipas, México., August 19 2013

To my parents Uriel and María Magdalena,
and to my brothers and sisters Deisy, Andy, Brenda, Noel, Elida, and Nahum.

Acknowledgements

- I am very grateful with the CINVESTAV-Tamaulipas for accepting me in the Master of Science in Computer Science program and for providing me all the facilities to successfully complete the program.
- I greatly appreciate to the CONACYT the two years of economic support, and to the Juárez Autonomous University of Tabasco the partial economic support during the second year of the program.
- I acknowledge the support of access to the infrastructure of high performance computing of the Information Technology Laboratory Unit (Hidra) at CINVESTAV-Tamaulipas and to the hybrid cluster for supercomputing (Xiuhcoatl) at CINVESTAV.
- Finally, I thank to my advisor Dr. José Torres Jiménez his invaluable help in the development of this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Publications

Carlos Ansótegui, Idelfonso Izquierdo, Felip Manyà, and José Torres Jiménez. *A Max-SAT-based Approach to Constructing Optimal Covering Arrays*, accepted for oral presentation in the Sixteenth International Conference of the Catalan Association of Artificial Intelligence (CCIA 2013).

Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez. *Generation of Non-Isomorphic Covering Arrays*, submitted to Discrete Applied Mathematics. July, 2013.

Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez. *Towers of Covering Arrays*, submitted to Discrete Applied Mathematics. August, 2013.

Resumen

Construcción de Torres de Covering Arrays

por

Idelfonso Izquierdo Márquez
Laboratorio de Tecnologías de Información, CINVESTAV-Tamaulipas
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2013
Dr. José Torres Jiménez, Director

Un covering array CA($N; t, k, v$) es una matriz de tamaño $N \times k$ sobre el conjunto $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$ en la cual cada submatriz de tamaño $N \times t$ contiene al menos una vez todas las t-tuplas del conjunto $\mathbb{Z}_v^t$. Los parámetros N y k son las dimensiones del covering array, el parámetro v es el orden del covering array, y el parámetro t es la fuerza de cobertura de interacciones. Dado un covering array A = CA($N; t, k, v$), llamado covering array base, es posible en ocasiones generar un covering array B = CA($Nv; t+1, k+1, v$) mediante una construcción basada en yuxtaponer verticalmente v copias del covering array base A con algunas columnas trasladadas, y haciendo que la columna $k+1$ del covering array B esté conformada por N ceros, seguidos por N unos y así hasta terminar con N elementos iguales a $v-1$. El covering array B que se obtiene de esta construcción tiene con respecto al covering array base v veces más renglones, una columna más, una unidad más de fuerza y el mismo orden. La misma construcción que se aplicó al covering array A de fuerza t podría aplicarse al covering array B de fuerza $t+1$ (el cual sería ahora el covering array base) para generar un covering array C = CA($Nv^2; t+2, k+2, v$) de fuerza $t+2$. El proceso continuaría extendiendo el covering array C de fuerza $t+2$ a uno de fuerza $t+3$, y así sucesivamente, formando una *Torre de Covering Arrays* (*TCA*). Esta tesis tiene como objetivo demostrar que mediante la construcción de TCAs es posible producir covering arrays de calidad competitiva con los mejores reportados actualmente.

Abstract

Construction of Towers of Covering Arrays

by

Idelfonso Izquierdo Márquez
Information Technology Laboratory, CINVESTAV-Tamaulipas
Research Center for Advanced Study from the National Polytechnic Institute, 2013
Dr. José Torres Jiménez, Advisor

A covering array CA$(N; t, k, v)$ is an $N \times k$ matrix over the set $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$ with the property that every $N \times t$ submatrix covers at least once all the t-tuples of the set $\mathbb{Z}_v^t$. The parameters N and k are the dimensions of the covering array, the parameter v is the order of the covering array, and the parameter t is the strength of coverage of interactions. Given a covering array $A = \text{CA}(N; t, k, v)$, called the base covering array, it is sometimes possible to generate a covering array $B = \text{CA}(Nv; t+1, k+1, v)$ of strength $t+1$ by means of a construction based on juxtaposing vertically v copies of the base covering array A with some columns translated, and doing that the column $k+1$ of the covering array B will be conformed by N zeroes, followed by N ones, and so on until finish with N elements equal to $v-1$. The covering array B obtained from this construction has with respect to the base covering array v times more rows, one more column, one unit more of strength and the same order. The same construction applied to the covering array A of strength t might be applied to the covering array B of strength $t+1$ (which is now the base covering array) to generate a covering array $C = \text{CA}(Nv^2; t+2, k+2, v)$ of strength $t+2$. The process will continue extending the covering array C of strength $t+2$ to a covering array of strength $t+3$, and so on, forming a *Tower of Covering Arrays* (*TCA*). This thesis has the objective of demonstrate that it is possible to produce covering arrays of quality competitive with the currently best reported ones by means of the construction of TCAs.

Nomenclature

Acronyms

CA	Covering Array
CACP	Covering Array Construction Problem
CAN	Covering Array Number
MCA	Mixed Covering Array
OA	Orthogonal Array
TCA	Tower of Covering Arrays

Notation

$\mathrm{CA}(N; t, k, v)$	Covering array of dimensions $N \times k$, order v, and strength t
$\mathrm{CAN}(t, k, v)$	The minimum N for which a covering array $\mathrm{CA}(N; t, k, v)$ exists
$\mathrm{OA}_\lambda(N; t, k, v)$	Orthogonal array of dimensions $N \times k$, order v, strength t, and index λ
$\mathbb{Z}_v$	The set of the integers from 0 to $v - 1$
$\mathbb{Z}_v^t$	The cartesian product of t sets equal to $\mathbb{Z}_v$
π	Permutation of the columns of a covering array
$\mathcal{P}_A$	Matrix $\mathcal{P}$ of the covering array A
$\mathcal{E}$	The construction to expand the covering arrays in the towers
$\oplus$	The operation of column translation by a constant value
δ	Matrix or vector to translate the columns of the base covering array

1

Introduction

This introductory chapter presents the thesis problem and the main goals of the thesis. Section 1.1 provides a definition of the combinatorial objects called covering arrays. Section 1.2 describes the practical applications of the covering arrays in the design of experiments, which are the motivations to investigate a new way to construct covering arrays. Section 1.3 presents the research problem: the construction of towers of covering arrays; the section begins by defining the concept of tower of covering arrays and the construction proposed to generate them; after that, the thesis problem is stated, followed by the research hypothesis, the general objective, and the particular objectives of the thesis. An overview of the thesis contents is given in Section 1.4. Finally, Section 1.5 highlights the main points of the chapter.

1.1 Introduction

This thesis investigates a new method to construct the combinatorial designs called covering arrays. The covering arrays were derived from the designs known as *orthogonal arrays*. An orthogonal

array $\mathrm{OA}_\lambda(N; t, k, v)$ is an $N \times k$ array over the set $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$ with the property that every subarray of t distinct columns covers *exactly* $\lambda \geq 1$ times each tuple of the set $\mathbb{Z}_v^t$. When $\lambda = 1$ it is usually omitted in the notation. Figure 1.1 shows the orthogonal array $\mathrm{OA}_2(8; 2, 7, 2)$; in every combination of $t = 2$ distinct columns of this orthogonal array the tuples of the set $\mathbb{Z}_2^2 = \{0, 1\}^2 = \{(0,0), (0,1), (1,0), (1,1)\}$ appears exactly two times.

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

Figure 1.1: The orthogonal array $\mathrm{OA}_2(8; 2, 7, 2)$.

In an orthogonal array $\mathrm{OA}_\lambda(N; t, k, v)$ the parameters N and k are the dimensions of the orthogonal array; the parameter v is the order of the orthogonal array or the number of distinct symbols in every column; the parameter t is the strength of coverage of interactions; and the parameter λ is the number of times that each tuple of $\mathbb{Z}_v^t$ appears in every combination of t distinct columns (for covering arrays the parameters N, t, k, and v have the same meaning).

Much work has been done for orthogonal arrays, for example see [39] for theory and applications of the orthogonal arrays. However, the requirement that every combination of t columns covers all the tuples of $\mathbb{Z}_v^t$ exactly λ times is too restrictive in some applications. The covering arrays are combinational designs very similar to the orthogonal arrays; the difference is that in the covering arrays every combination of t distinct columns covers each tuple of the set $\mathbb{Z}_v^t$ *at least once*. Thanks to this relaxed requirement the covering arrays have a wider range of applications. The concept of covering array is defined formally in the following Definition 1, which was adapted from [38]:

DEFINITION **1** *Let be N, t, k, and v four positive integers, a covering array $CA(N; t, k, v)$ is an $N \times k$ array $A = (a_{i,j})$, $0 \leq i \leq N-1$, $0 \leq j \leq k-1$, over $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$ with the property*

that for any t distinct columns $0 \leq c_0 < c_1 \cdots < c_{t-1} \leq k-1$, and any member $(x_0, x_1, \ldots, x_{t-1})$ of $\mathbb{Z}_v^t$, there exists at least one row r such that $x_i = a_{r,c_i}$ for all $0 \leq i \leq t-1$.

Figure 1.2 shows the covering array CA$(12; 3, 11, 2)$. This covering array has $N = 12$ rows and $k = 11$ columns, its order is $v = 2$, and its strength is $t = 3$. Each one of the $\binom{11}{3} = 165$ subarrays of three distinct columns covers every member of $\mathbb{Z}_2^3$ at least once. The set $\mathbb{Z}_2^3$ is equal to

$$\begin{aligned}\mathbb{Z}_2^3 &= \{0,1\}^3 \\ &= \{0,1\} \times \{0,1\} \times \{0,1\} \\ &= \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}.\end{aligned}$$

↓	↓		↓							
0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	1	1	0	0
0	0	0	1	1	1	0	1	0	1	1
0	1	1	1	0	1	0	0	1	0	1
1	0	1	0	1	0	0	0	1	1	1
1	1	1	0	0	1	0	1	0	1	0
0	0	1	1	0	0	1	1	1	1	0
1	0	1	1	1	1	1	0	0	0	0
0	1	0	0	1	1	1	0	1	1	0
1	1	0	1	0	0	1	0	0	1	1
1	0	0	0	0	1	1	1	1	0	1
0	1	1	0	1	0	1	1	0	0	1

Figure 1.2: The covering array CA$(12; 3, 11, 2)$. In the columns marked with a down arrow the first occurrence of each tuple of the set $\{0,1\}^3$ is colored in purple.

To clarify the concept of strength of a covering array, consider the columns 0, 1, 3 of the covering array in Figure 1.2. In the twelve rows of this group of three columns the eight tuples of the set $\{0,1\}^3$ are covered at least once. From top to bottom in the table, the first occurrence of each tuple is colored in purple. This way, in a covering array of strength t every combination of t distinct columns covers each tuple of the set $\mathbb{Z}_v^t$ at least once.

Given the values of t, k, and v the problem of constructing covering arrays is the problem of generating a covering array CA$(N; t, k, v)$ with the minimum number of rows N. This problem is very hard for general values of t, k, and v, so that a number of different methods has been proposed to solve it (some of which are reviewed in Chapter 3). This thesis investigates a new method to construct covering arrays based on the construction of *Towers of Covering Arrays* (*TCAs*). The main objective of this research is to prove that the TCA approach can produce covering arrays of quality competitive with the best known covering arrays.

1.2 Motivations

Covering arrays have practical applications in the design of experiments for software and hardware testing [37, 44]. Consider a software component with k parameters, all of which have two possible values. To fully verify the proper operation of the component the 2^k combinations of values for the input parameters must be checked. However, this exhaustive approach is impractical for complex components having for example $k = 30$ parameters.

The alternative to the exhaustive approach is to check only the interactions of size t. In this case a covering array of strength t can be used as a test suite, because it ensures that all possible interactions among any t parameters are checked in the experiment. A covering array $A = (a_{i,j}) =$ CA$(N; t, k, v)$ is used as a test suite in the following way: the k columns represent the k factors, the order v is the number of possible values for each factor, the strength t is the degree of coverage of interactions, and the N rows are the test cases. The i-th row $(a_{i,0}\ a_{i,1}\ \cdots\ a_{i,k-1})$ corresponds to the test case in which the first factor takes its value number $a_{i,0}$, the second factor takes its value number $a_{i,1}$, and so on. Covering arrays with few rows are desired because the verification of all the interactions of size t among the k parameters is done with less test cases.

In the area of software testing the covering arrays are the base of the combinatorial testing methods [19, 35, 44]. In a serie of studies realized by the National Institute of Standards and

Technology (NIST) in a wide range of domains, it was found that all the failures in the software products under study were due to interactions involving at most six parameters [45, 46, 47, 73]. This results is not conclusive for all software products, but it suggests that the number of parameters involved in a failure is relatively low; therefore, testing all interactions of size t is very effective to detect failures in the software products. Another areas of applications of covering arrays are testing advanced materials [14], bioinformatics [61, 63], and data compression [67].

The objective of the methods to construct covering arrays is to satisfy the coverage properties of the covering arrays (i.e., every submatrix of t distinct columns covers at least once each tuple of $\mathbb{Z}_v^t$) using the least possible number of rows. In general, the optimum number of rows for a covering array of k columns, strength t, and order v is unknown; what it is generally known is the current upper bound for a combination of k, t, and v; so, the main objective of the new methods of construction is to improve the current upper bounds. The purpose of this thesis is to produce covering arrays that improve or equal some current upper bounds by means of the construction of TCAs.

1.3 Thesis Problem

The problem studied in this thesis is the construction of TCAs. This section begins introducing the concept of TCA; after that, the research problem, the research hypothesis, the general objective, and the particular objectives of the thesis are stated.

1.3.1 Towers of Covering Arrays

A TCA is defined as follows: a TCA of height h is a succession of $h+1$ covering arrays $C_0, C_1, \ldots, C_h$, where C_0 is a covering array of strength t called the *base* of the TCA, and for $i = 1, 2, \ldots, h$, C_i is a covering array of strength $t + i$.

The covering arrays in a TCA are created by the iterative application of one construction that we called $\mathcal{E}$. This construction takes a base covering array CA$(N; t, k, v)$ of strength t, N rows, k

columns, and order v, and *expands* it to a covering array CA(Nv; $t+1, k+1, v$) of strength $t+1$, Nv rows, $k+1$ columns and, order v, by juxtaposing vertically v copies of the base covering array, but translating the j-th column of the i-th copy by the (i, j) entry of matrix over $\mathbb{Z}_v$ called δ. The column $k+1$ of the covering array of strength $t+1$ is conformed by N zeroes, followed by N ones, a so on until finish with N elements equal to $v-1$.

The construction $\mathcal{E}$ might be applied to the covering array CA(Nv; $t+1, k+1, v$) of strength $t+1$ to produce a covering array CA(Nv^2; $t+2, k+2, v$) of strength $t+2$. The process would continue expanding the covering array of strength $t+2$ to a covering array of strength $t+3$, and so on, producing a TCA. Figure 1.3 illustrates the concept of TCAs.

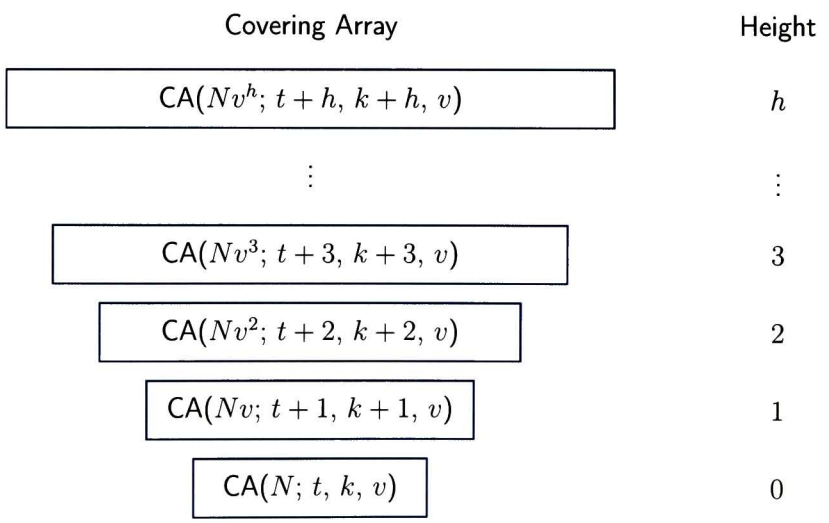


Figure 1.3: A TCA of height h is conformed by $h+1$ covering arrays of strengths t, $t+1$, ..., $t+h$.

An important concept for describing the construction $\mathcal{E}$ is the concept of translating the columns of a covering array, which is given in Definition 2.

DEFINITION **2** *Let be $A = CA(N; t, k, v)$ a covering array of order v, and let be A_j, $0 \leq j \leq k-1$, one column of the covering array A. To translate the column A_j by a value $c \in \{0, 1, \ldots, v-1\}$ means to add modulo v the value c to every element of A_j. The symbol $\oplus$ will be used to denote the operation of column translation (see Figure 1.4).*

$$A_j \oplus c = \begin{pmatrix} a_{0,j} \\ a_{1,j} \\ \vdots \\ a_{N-1,j} \end{pmatrix} \oplus c = \begin{pmatrix} (a_{0,j} + c) \bmod v \\ (a_{1,j} + c) \bmod v \\ \vdots \\ (a_{N-1,j} + c) \bmod v \end{pmatrix}$$

Figure 1.4: Translation of the column A_j by the value $c \in \{0, 1, \ldots, v-1\}$.

Now, the construction to generate a covering array of strength $t+1$ from a covering array of strength t (the construction $\mathcal{E}$) is defined formally in Definition 3. Figure 1.5 shows the definition of the construction $\mathcal{E}$ in schematic form.

DEFINITION 3 *Let be $A_0, A_1, \ldots, A_{k-1}$ the k columns of a covering array $A = CA(N; t, k, v)$; let be δ a matrix of dimensions $v \times k$ such that $\delta_{0,j} = 0$ and $\delta_{i,j} \in \{0, 1, \ldots, v-1\}$ for $1 \leq i \leq v-1$, $0 \leq j \leq k-1$; and let be $X_0, X_1, \ldots, X_{v-1}$ the v columns of the matrix $X = (x_{i,j})$ of dimensions $N \times v$ such that $x_{i,j} = j$ for $0 \leq i \leq N-1$, $0 \leq j \leq v-1$. The construction $\mathcal{E}$ to try to expand the base covering array A of strength t to a covering array B of strength $t+1$ consists in creating a matrix B of size $Nv \times (k+1)$ composed by v blocks of size $N \times (k+1)$ juxtaposed vertically. The j-th column of the i-th block $(0 \leq j \leq k-1,\ 0 \leq i \leq v-1)$ is the column j of the base covering array translated by the entry (i, j) of the matrix δ; the last column of the i-th block is the column X_i (see Figure 1.5).*

The concept of TCAs was introduced in some way for orthogonal arrays. In the book of Hedayat, Sloane, and Stufken [39] the theorem 2.24 says that an OA$(N; 2u, k, 2)$ exists if and only if an OA$(2N; 2u+1, k+1, 2)$ exists. The converse of this theorem is that the orthogonal array $A =$ OA$(N; 2u, k, 2)$ can be used to construct the orthogonal array $B =$ OA$(2N; 2u+1, k+1, 2)$. The way to construct B is to place the rows of A followed by 0, together with the rows of $\overline{A}$ (the complementary matrix of A) followed by 1, as shown next, where $\mathbf{0}$ is a constant column vector conformed by zeroes and $\mathbf{1}$ is a constant column vector conformed by ones:

$$B = \begin{pmatrix} A & \mathbf{0} \\ \overline{A} & \mathbf{1} \end{pmatrix}$$

$$A = \begin{pmatrix} A_0 & A_1 & \dots & A_{k-1} \end{pmatrix} \qquad \delta = \begin{pmatrix} 0 & 0 & \dots & 0 \\ \delta_{1,0} & \delta_{1,1} & \dots & \delta_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{v-1,0} & \delta_{v-1,1} & \dots & \delta_{v-1,k-1} \end{pmatrix}$$

$$X = \begin{pmatrix} 0 & 1 & \dots & v-1 \\ 0 & 1 & \dots & v-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & v-1 \end{pmatrix}$$

$$B = \begin{pmatrix} A_0 \oplus \delta_{0,0} & A_1 \oplus \delta_{0,1} & \dots & A_{k-1} \oplus \delta_{0,k-1} & X_0 \\ A_0 \oplus \delta_{1,0} & A_1 \oplus \delta_{1,1} & \dots & A_{k-1} \oplus \delta_{1,k-1} & X_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_0 \oplus \delta_{v-1,0} & A_1 \oplus \delta_{v-1,1} & \dots & A_{k-1} \oplus \delta_{v-1,k-1} & X_{v-1} \end{pmatrix}$$

Figure 1.5: The construction $\mathcal{E}$. The first k columns of the i-th block of the matrix B are the columns of the base covering array A translated by the value $\delta_{i,j}$, and the last column of the block is the column X_i.

This construction allows the generation of towers of orthogonal arrays of height $h = 1$, but they only exist for $v = 2$ and when the strength of the orthogonal array A is even ($t = 2u$).

It is important to mention that the construction $\mathcal{E}$ does not always produce a covering array of strength $t + 1$ based on a covering array of strength t; this depends on the base covering array A and on the matrix δ used. In this work we want to construct TCAs in which the base covering array can have any strength and any order; so, we check all matrices δ that exist for the base covering array to see if one of them produce a matrix B that is a covering array of strength $t + 1$.

1.3.2 Problem Statement

The problem investigated in this thesis is the construction of TCAs in which the base covering array can have any strength and any order, with the objective of generating covering arrays of quality competitive with the best known ones.

In this work a covering array A produced in the construction of TCAs will be considered of quality competitive with the best known one B (which is the current upper bound) if A satisfies the coverage properties in a smaller or equal number of rows than B.

1.3.3 Research Hypothesis

It is possible to construct TCAs of any order such that the covering arrays generated improve or equal some of the current upper bounds.

1.3.4 General Objective

To produce covering arrays that improve or equal some of the current upper bounds by means of the construction of TCAs

1.3.5 Particular Objectives

- To define a methodology for constructing TCAs.
- To develop the algorithms required for realizing efficiently the methodology.
- To perform a computational experimentation with the objective of constructing TCAs with the maximum height possible.

1.4 Thesis Contents

The thesis document is organized in six chapters; next is a brief summary of the content of each chapter:

- *Chapter 1 Introduction.* The first chapter of the thesis is an introduction to the thesis problem: the construction of TCAs.

- *Chapter 2 Background on Covering Arrays.* The second chapter introduces some concepts about covering arrays which are very important for the TCA approach. They include the coverage properties of covering arrays, isomorphism in covering arrays, and covering arrays of minimum rank.

- *Chapter 3 State of the Art of the Construction of Covering Arrays.* This chapter presents a number of methods developed to solve the problem of constructing covering arrays. The methods studied were grouped in four categories: exact, greedy, heuristic, and algebraic.

- *Chapter 4 Methodology to Construct the Towers of Covering Arrays.* This chapter describes the methodology followed to construct TCAs and the algorithms developed to realize the methodology.

- *Chapter 5 Computational Results.* This chapters describes the computational experimentation done to prove the research hypothesis and the relevant results obtained.

- *Chapter 6 Conclusions.* The last chapter presents a summary of the main contributions of the thesis, some directions for future work, and the final discussion of the work done.

1.5 Chapter Summary

This chapter introduced the thesis problem: the construction of TCAs. A covering array CA($N; t, k, v$) is an $N \times k$ matrix over $\mathbb{Z}_v$ with the property that every $N \times t$ subarray covers at least once the tuples in the set $\mathbb{Z}_v^t$. A TCA of height h is a succession of $h+1$ covering arrays $C_0, C_1, \ldots, C_h$, where C_0 is a covering array of strength t called the base of the TCA, and for $i = 1, 2, \ldots, h$, C_i is a covering array of strength $t+i$. The objective of constructing TCAs is to obtain covering arrays of quality competitive with the best known ones. The following chapter presents the concepts about covering arrays that are required to solve efficiently the thesis problem.

2

Background on Covering Arrays

This chapter presents some concepts about covering arrays which are very important for the subsequent chapters of the thesis and for the construction of TCAs. Section 2.1 explains the basic operation of verifying if a given matrix is a covering array of a certain strength; this operation is based in counting the number of times each t-tuple is covered in every combination of t columns. Section 2.2 introduces the concept of isomorphism in covering arrays and the three operations from which isomorphic covering arrays are derived: row permutation, column permutation, and symbol permutation in the columns; in addition, non-isomorphic covering arrays are introduced, and it is explained why they are important for the construction of TCAs. Section 2.3 presents the problem of constructing covering arrays; and Section 2.4 finalizes the chapter introducing a more general type of covering arrays called mixed covering arrays.

2.1 Verification of Covering Arrays

A covering array CA($N; t, k, v$) has $\binom{k}{t}$ different submatrices of dimensions $N \times t$, and each of them covers the v^t different tuples of the set $\mathbb{Z}_v^t$ at least once. One basic operation for covering arrays is to verify if a given matrix is a covering array of a certain strength t.

Consider the matrix A of Figure 2.1. To be a covering array of strength $t = 2$ every combination of two columns of matrix A must cover the four tuples of the set $\mathbb{Z}_2^2 = \{(0,0),(0,1),(1,0),(1,1)\}$ at least once. We use a matrix called $\mathcal{P}$ to record the number of times the t-tuples are covered in the $\binom{k}{t}$ combinations of columns of a matrix with k columns. This matrix $\mathcal{P}$ has $\binom{k}{t}$ rows and $|\mathbb{Z}_v^t|$ columns, and its (i,j) entry is the number of times the i-th combination of columns covers the j-th tuple of the set $\mathbb{Z}_v^t$. The matrix $\mathcal{P}_A$ for the array A of Figure 2.1 is shown to the right of the same Figure 2.1.

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$\mathcal{P}_A$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$\{0,1\}$	1	2	1	2
$\{0,2\}$	1	2	1	2
$\{0,3\}$	2	1	1	2
$\{0,4\}$	1	2	2	1
$\{1,2\}$	1	1	1	3
$\{1,3\}$	1	1	2	2
$\{1,4\}$	1	1	2	2
$\{2,3\}$	1	1	2	2
$\{2,4\}$	1	1	2	2
$\{3,4\}$	1	2	2	1

Figure 2.1: The matrix A and its associated matrix $\mathcal{P}_A$.

If the matrix M being checked is a covering array then its associated matrix $\mathcal{P}_M$ does not have any entry equal to zero, since one entry (i,j) equal to zero means that the i-th combination of columns does not cover the j-th tuple of the set $\mathbb{Z}_v^t$. Every entry equal to zero in the matrix $\mathcal{P}_M$ indicates a *missing tuple* in the matrix M. For the matrix A of Figure 2.1 all the elements of its associated matrix $\mathcal{P}_A$ are greater than zero, so the matrix A is a covering array of strength $t = 2$.

2.2 Isomorphism in Covering Arrays

For the same parameters N, k, t, and v there may exist several covering arrays with N rows, k columns, strength t, and order v. However, a number of these covering arrays would be equivalent due to the existence of three symmetries in the covering arrays.

Any of the following three operations produces a covering array equivalent to the one over which the operation is applied:

1. To permute the rows of the covering array.
2. To permute the columns of the covering array.
3. To permute the symbols in the columns of the covering array.

Furthermore, any combination of these three operations applied to a covering array produces an equivalent covering array. The following Definition 4 introduces the concept of isomorphism in covering arrays.

DEFINITION **4** *Two covering arrays A and B with the same parameters N, t, k, and v are isomorphic if B can be derived from A (and vice versa) by a combination of a row permutation, a column permutation, and a symbol permutation in a subset of columns.*

For a covering array CA$(N; t, k, v)$ of N rows, k columns, and order v, there are $N!$ row permutations, $k!$ column permutations, and $(v!)^k$ different combinations of symbol permutations. The number of different combinations of symbol permutation is $(v!)^k$ because there are $v!$ possible symbol permutations for each one of the k columns. The operation of symbol permutation in a column is also referred as *relabeling* the column. This way, excluding itself, the number of covering arrays which are isomorphic to one in particular is $N!\,k!\,(v!)^k - 1$; however, some of these covering arrays may be identical, and so the number of different isomorphic covering arrays may be less than $N!\,k!\,(v!)^k$.

Consider the covering array $A = \mathrm{CA}(6; 2, 5, 2)$ of Figure 2.2, which is the same as the covering array A of Figure 2.1. Let us number its $N = 6$ rows and its $k = 5$ columns sequentially starting from zero; so, its current permutation of rows is $\tau = (0\ 1\ 2\ 3\ 4\ 5)$, and its current permutation of columns is $\pi = (0\ 1\ 2\ 3\ 4)$.

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad \begin{array}{l} \tau = (0\ 1\ 2\ 3\ 4\ 5) \\ \pi = (0\ 1\ 2\ 3\ 4) \\ \phi = (0\ 0\ 0\ 0\ 0) \end{array}$$

Figure 2.2: Current row permutation, column permutation, and symbol permutations of the covering array A.

Moreover, let us denote its current combination of symbol permutations by $\phi = (0\ 0\ 0\ 0\ 0)$, where the elements of ϕ are indices of the symbol permutations for each column of A listed in lexicographic order. For a covering array of order v there are $v!$ symbol permutations or relabelings for each column, these relabelings will be represented by an index $j \in \{0, 1, \ldots, v! - 1\}$ considering the symbol permutations in lexicographic order. In this case is $v = 2$, and therefore the $2! = 2$ symbol permutations for each column of A are in lexicographic order $(0\ 1)$ and $(1\ 0)$. These permutations are indexed by 0 and 1 respectively. The symbol permutation $(0\ 1)$ is the identity, that is, it indicates to change the zeroes by zeroes and the ones by ones; the symbol permutation $(1\ 0)$ indicates to change the zeroes by ones and the ones by zeroes.

Let be $\tau' = (3\ 1\ 2\ 5\ 4\ 0)$ another permutation of the rows of the covering array A of Figure 2.2; let be $\pi' = (4\ 2\ 0\ 3\ 1)$ another permutation of the columns of A; and let be $\phi' = (0\ 1\ 0\ 1\ 0)$ another combination of symbol permutations for the columns of A. Figure 2.3 shows the covering array B resulting from applying to the covering array A the operations defined by τ', π', and ϕ'. The order in which these operations are applied is irrelevant for the final result. The covering array B is isomorphic to A because it was derived from A by one combination of the three operations that produce equivalent covering arrays.

$$B = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix} \qquad \begin{aligned} \tau' &= (3\,1\,2\,5\,4\,0) \\ \pi' &= (4\,2\,0\,3\,1) \\ \phi' &= (0\,1\,0\,1\,0) \end{aligned}$$

Figure 2.3: The isomorphic covering array B produced by the application of τ', π', and ϕ' to the covering array A.

Isomorphic covering arrays are equivalent because the operations of row permutation, column permutation, and symbol permutation do not change the coverage properties of the matrix over which they are applied. That is, if the initial matrix is a covering array, then the matrix after the operations is also a covering array; also, if the initial matrix has m missing tuples, then the matrix after the operations also has m missing tuples (but not necessarily the same missing tuples).

In more formal terms, the operations of permute the rows, permute the columns, and permute the symbols in the columns do not affect the qualitative t-independence of the columns of the covering array over which the operations are applied:

DEFINITION **5** *Let be N, t, and v be positive integers. t vectors $w_0, w_1, \ldots, w_{t-1} \in \mathbb{Z}_v^N$ are qualitatively independent if for each tuple $(x_0, x_1, \ldots, x_{t-1}) \in \mathbb{Z}_v^t$ there is an index $i \in \{0, 1, \ldots, N-1\}$ such that $(w_{0_i}, w_{1_i}, \ldots, w_{t-1_i}) = (x_0, x_1, \ldots, x_{t-1})$. A set of vectors is qualitatively t-independent if any t distinct vectors of the set are qualitatively independent.*

The equivalence of the isomorphic covering arrays is better appreciated in the matrix $\mathcal{P}$ of the covering arrays. Figure 2.4 shows the matrix $\mathcal{P}_B$ of the covering array B of Figure 2.3, which is isomorphic to the covering array A of Figure 2.2. The i-th row of $\mathcal{P}_B$ is a permutation of the j-th row of $\mathcal{P}_A$, where possibly $i \neq j$; moreover, every row of $\mathcal{P}_A$ is used to derive a row of $\mathcal{P}_B$ and there are not two rows of $\mathcal{P}_B$ derived from the same row of $\mathcal{P}_A$. For example, both matrices $\mathcal{P}_A$ and $\mathcal{P}_B$ have a row with one 3 and three 1's (fifth row of $\mathcal{P}_A$ and seventh row of $\mathcal{P}_B$), all the other rows have two 1's and two 2's.

$$B = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$\mathcal{P}_B$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$\{0,1\}$	2	1	2	1
$\{0,2\}$	1	2	2	1
$\{0,3\}$	2	1	1	2
$\{0,4\}$	1	2	1	2
$\{1,2\}$	2	2	1	1
$\{1,3\}$	2	2	1	1
$\{1,4\}$	1	3	1	1
$\{2,3\}$	1	2	2	1
$\{2,4\}$	1	2	1	2
$\{3,4\}$	1	2	1	2

Figure 2.4: The matrix $\mathcal{P}_B$ of the covering array B.

Contrary to the isomorphic covering arrays, the non-isomorphic covering arrays are those covering arrays which can not be transformed among them by permutations of rows, permutations of columns, and permutation of the symbols in the columns. Figure 2.5 shows another covering array $C = \text{CA}(6; 2, 5, 2)$, but this covering array is non-isomorphic to the covering arrays A and B of Figure 2.2 and Figure 2.3 respectively. The matrix $\mathcal{P}_C$ shows that the covering array C has a different pattern of coverage of the tuples in $\mathbb{Z}_2^2$.

$$C = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$\mathcal{P}_C$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$\{0,1\}$	1	3	1	1
$\{0,2\}$	1	3	1	1
$\{0,3\}$	1	3	1	1
$\{0,4\}$	3	1	1	1
$\{1,2\}$	1	1	1	3
$\{1,3\}$	1	1	1	3
$\{1,4\}$	1	1	3	1
$\{2,3\}$	1	1	1	3
$\{2,4\}$	1	1	3	1
$\{3,4\}$	1	1	3	1

Figure 2.5: The covering array C is non-isomorphic with the previous covering arrays A and B.

This characteristic is not true in general for all the non-isomorphic covering arrays, but it is very important for the construction of TCAs. Since some non-isomorphic covering arrays have different

patterns of coverage of the tuples in the set $\mathbb{Z}_v^t$, they may produce different TCAs for the same parameters N, t, k, and v.

Let us introduce a very important concept in this thesis: the *rank* of a covering array. The N rows of a covering array CA$(N; t, k, v)$ can be considered as numbers in base v, with the most significant digit to the left; the rank of a covering array is the ordered sequence conformed by these N numbers in base v; Figure 2.6 illustrates the concept of rank of a covering array.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 2 & 2 & 2 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

Rank:
$(0, 13, 26, 32, 42, 46, 61, 65, 75)$

Figure 2.6: The rank of the covering array shown in the figure is the ordered sequence (0, 13, 26, 32, 42, 46, 61, 65, 75) constructed by considering the nine rows of the covering array as numbers in base three.

Let be $\mathcal{S}$ the set conformed by one covering array $s = \text{CA}(N; t, k, v)$ and all its $N!\,k!\,(v!)^k - 1$ isomorphic covering arrays. In the set $\mathcal{S}$ there is a covering array $s' \in \mathcal{S}$ that has the smallest rank among all the covering arrays in $\mathcal{S}$; this covering array s' is the covering array of minimum rank equivalent to s. In the following Definition 6 the concept of covering array of minimum rank is formally defined.

DEFINITION **6** *A covering array CA$(N; t, k, v)$ with rank $(a_0, a_1, \ldots, a_{N-1})$ is of minimum rank only if none of its $N!\,k!\,(v!)^k - 1$ isomorphic covering arrays has a rank $(b_0, b_1, \ldots, b_{N-1})$ such that for some $i \in \{0, \ldots, N-1\}$ is $b_i < a_i$ and $b_j = a_j$ for $0 \leq j < i$.*

Figure 2.7 shows the covering arrays of minimum rank A^*, B^*, C^* equivalent to the covering arrays A, B, C of the Figures 2.2, 2.3, 2.5 respectively. The covering arrays A^* and B^* are identical

because they are the equivalent of minimum rank of the isomorphic covering arrays A and B. In the other hand, the covering array C^* is different from A^* and B^* because C is non-isomorphic to A and B. To get the covering array of minimum rank that is equivalent to a given covering array, all the isomorphic covering arrays for the given one are generated, and it is taken the covering array with the minimum rank.

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix} \qquad C = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$A^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \qquad B^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \qquad C^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Figure 2.7: The covering arrays A, B, C and their equivalents of minimum rank A^*, B^*, C^*.

Isomorphic covering arrays determine disjoint classes in the set of all covering arrays that exist for the same parameters N, t, k, and v, since any element in one class can be transformed into any other element of the same class by permutations of rows, permutations of columns, and permutations of symbols; but with these operations it is not possible to transform one element of one class into one element of a different class. In Figure 2.8 the set $\mathcal{U}$ is the set of all $N \times k$ matrices over $\mathbb{Z}_v$; in these matrices are included all the covering arrays CA$(N; t, k, v)$. The disjoint sets $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ represent the classes of isomorphic covering arrays. The number of different classes in which the set of all covering arrays with parameters N, t, k, and v is partitioned is equal to the number of distinct non-isomorphic covering arrays of minimum rank CA$(N; t, k, v)$.

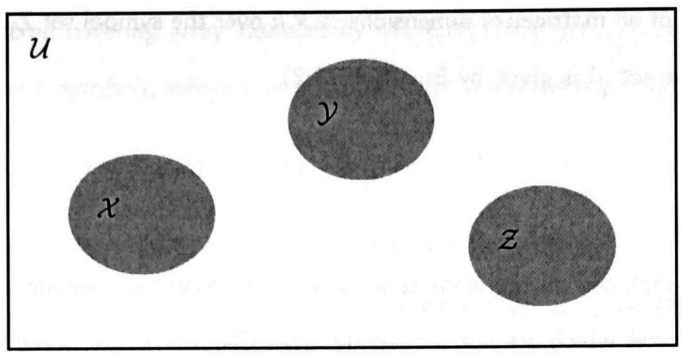


Figure 2.8: Islands of isomorphic covering arrays in the universe of the $N \times k$ matrices over $\mathbb{Z}_v$.

2.3 Construction of Covering Arrays

The covering array construction problem (CACP) consists in constructing a covering array CA$(N; t, k, v)$ given the parameters t, k, and v in such a way the number of rows N of the covering array is minimal. The smallest N for which a covering array exists is the covering array number (CAN) for the parameters t, k, and v, and it is denoted by

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

The CACP seems to be NP-complete because some related problems are; for example it is NP-complete to determine if there exists a row r that covers at least m missing tuples in a matrix [22]. Only for a few cases the CACP is solvable in polynomial time. The case $v = 2$ and $t = 2$ was solved by Rényi [59] for N even, and by Katona [42] and Kleitman and Spencer [43] for all N. The other case solvable in polynomial time is $v = p^n$ and $k = v + 1$, with p prime and $n \geq 1$, which was solved by Bush [11]. In addition to these cases, optimal solutions are known for some combinations of t, k, and v (see [26]); but for general values of t, k, and v the CACP appears to be infeasible except when the parameters are quite small [22]. Chapter 3 is devoted to analyze some of the most relevant methods to construct covering arrays.

Let be $\mathcal{A}$ the set of all matrices of dimensions $N \times k$ over the symbol set $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$. The cardinality of the set $\mathcal{A}$ is given by Equation (2.2).

$$|\mathcal{A}| = v^{Nk} \tag{2.2}$$

In the set $\mathcal{A}$ are included all the isomorphic and non-isomorphic covering arrays CA$(N; t, k, v)$, as well as those matrices which are not a covering array CA$(N; t, k, v)$. Some of these matrices do not have any possibility of being a covering array of strength t; for example the zero matrix. In addition, repeated rows are not desirable in covering arrays, because the main objective in the CACP is to minimize the number of rows.

Let be $\mathcal{B}$ the set of all $N \times k$ matrices over $\mathbb{Z}_v$ without repeated rows, and such that the rows are in ascending order according to their rank; the cardinality of $\mathcal{B}$ is given in Equation (2.3).

$$|\mathcal{B}| = \binom{v^k}{N} \tag{2.3}$$

The search space defined by $\mathcal{B}$ is more accurate than the defined by $\mathcal{A}$, because in $\mathcal{B}$ only are included those matrices which do not have repeated rows and whose rows are in ascending order according to their rank. The number of matrices in the set $\mathcal{B}$ is the number of ways of selecting N rows from the v^k different rows of length k over $\mathbb{Z}_v$. Therefore, the set $\mathcal{B}$ will be considered as the search space for the CACP.

2.4 Mixed Covering Arrays

Both the covering arrays seen so far and the orthogonal arrays of Chapter 1 have the same number of symbols in all their columns. Nonetheless, it is possible that each column has a different number of symbols, i.e., a different order. The mixed covering arrays are covering arrays in which every column may have a different order. The following definition for mixed covering arrays was taken from [21]:

DEFINITION **7** *A mixed covering array, denoted by MCA*$(N; t, k, (v_0, v_1, \ldots, v_{k-1}))$*, is a matrix of dimension* $N \times k$ *over* v *symbols, where* $v = \sum_{j=0}^{k-1} v_j$*, with the following properties:*

1. *Each column* j $(0 \leq j \leq k-1)$ *only contains symbols from the set* S_j*, where* $|S_j| = v_j$.

2. *Each submatrix of dimensions* $N \times t$ *conformed by the columns* $j_0, j_1, \ldots, j_{t-1}$*, where* $\{j_0, j_1, \ldots, j_{t-1}\} \subset \{0, 1, \ldots, k-1\}$*, covers at least once the* $\prod_{i=0}^{t-1} v_{j_i}$ *distinct* t*-tupes of the set* $\{0, 1, \ldots, v_{j_0} - 1\} \times \{0, 1, \ldots, v_{j_1} - 1\} \times \cdots \times \{0, 1, \ldots, v_{j_{t-1}} - 1\}$.

Generally, some v_j are the same, so the notation MCA$(N; t, k, (w_0^{k_0}, w_1^{k_1}, \ldots, w_{s-1}^{k_{s-1}}))$ is commonly used; in these notation $k = \sum_{i=0}^{s-1} k_i$ and $v = \sum_{i=0}^{s-1} k_i w_i$, that is, k_i is the number of columns with order w_i. Figure 2.9 shows the mixed covering array MCA$(9; 2, 5, 3^2 2^3)$; the first two columns of this MCA are of order three and the remaining three columns are of order two. Every pair of columns $i, j \in \{0, 1, 2, 3, 4\}$, $i \neq j$, covers at least once all the tuples of the set $\{0, 1, \ldots, v_i - 1\} \times \{0, 1, \ldots, v_j - 1\}$.

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

Figure 2.9: The mixed covering array MCA$(9; 2, 5, 3^2 2^3)$. The first two columns of the MCA have order three and the other three columns have order two.

Some of the methods to construct covering arrays revised in the following chapter are able to construct both mixed covering arrays and covering arrays with columns of the same order, which are called uniform covering arrays to distinguish them from the mixed covering arrays. The covering arrays constructed by the TCA approach are uniform covering arrays.

2.5 Chapter Summary

In this chapter were introduced the concepts about covering arrays required for the construction of TCAs. One basic operation for covering arrays is to verify if a given matrix is or not a covering array of a certain strength t. A matrix called $\mathcal{P}$ is used to record the number of times the tuples of the set $\mathbb{Z}_v^t$ are covered in the $\binom{k}{t}$ combinations of t distinct columns of the matrix being verified. Isomorphic covering arrays determine disjoint classes in the set of all covering arrays that exist for the same parameters N, t, k, and v; since any element in one class can be transformed into any other element of the same class by permutations of rows, permutations of columns, and permutations of symbols; but with these operations it is not possible to transform one element of one class into one element of a different class. The covering arrays in different classes are non-isomorphic among them. The covering array construction problem (CACP) consists in constructing a covering array CA$(N; t, k, v)$ given the parameters t, k, and v in such a way the number of rows N of the covering array is minimal. The search space for the CACP is $\binom{v^k}{N}$, i.e., the number of ways of selecting N different rows from the v^k rows of length k over $\mathbb{Z}_v$. For general values of t, k, and v the CACP appears to be infeasible except when the parameters are quite small; in the next chapter are reviewed some of the most important methods developed to solve the CACP in an exact way or in an approximate way.

3

State of the Art of the Construction of Covering Arrays

Due to the difficulty of solving the problem of constructing covering arrays a number of methods have been developed. This chapter gives a review of the state of the art of the methods to construct covering arrays. The methods analyzed were grouped in four categories: exact methods (Section 3.1), greedy methods (Section 3.2), metaheuristic methods (Section 3.3), and algebraic methods (Section 3.4). In addition to these methods Section 3.5 describes five useful operations for covering arrays previously constructed. The chapter ends with a summary of the methods analyzed, and a discussion of the methods with more similarity to the one proposed in this thesis: the construction of towers of covering arrays.

3.1 Exact Methods

Given the values k, t y v the exact methods construct a covering array CA($N; t, k, v$) in which the number of rows N is optimal. The exact methods have techniques for accelerating the search process, but even so they are only practical for constructing small covering arrays.

3.1.1 The Automatic Generator EXACT

Yan and Zhang developed in [74] an automatic generator of mixed covering arrays called EXACT (EXhaustive seArch of Combinatorial Test suites). This generator works by making assignments to the cells of the MCA while possible, and making backtrack when it reaches a point in which it is not possible to complete the MCA. The algorithm incorporates some strategies to accelerate the search process, one of which is do not explore arrays isomorphic to arrays previously explored. To do this, the algorithm only considers the arrays that are lexicographically ordered both by rows and by columns. Column relabeling is not considered by the generator, because its objective is to generate only one MCA for the given parameters; instead, a technique called SCEH (Sub-Combination Equalization Heuristic) is used. The SCEH technique reduces the search space by assuming that the symbols in the columns of the MCA are balanced, i.e., the symbols appear nearly the same number of times in a column.

3.1.2 New Backtracking Algorithm

A searching algorithm to construct binary covering arrays ($v = 2$) of strength t and dimensions $N \times k$ was given by Bracho-Rios et al. [9]. The algorithm constructs the covering arrays column by column imposing a lexicographic ordering of the columns to break the column and row symmetries. The columns to construct the covering arrays are balanced in symbols, they have $\lfloor \frac{N}{2} \rfloor$ zeroes and $N - \lfloor \frac{N}{2} \rfloor$ ones. Before starting the search, a block of t columns is made fixed, the first $N - 2^t$ rows

of the block are filled with zeroes and the last 2^t rows are filled with the 2^t tuples of size t over the symbol set $\{0, 1\}$. Suppose a partial solution with r columns ($t \leq r < k$) and let be l the last column of the partial solution. To construct a covering array of strength t and $r + 1$ columns, the algorithm checks all the columns l' greater than l in lexicographic order until find one which makes a covering array of strength t with the r columns of the partial solution such that the rows and the columns are ordered lexicographically. If no such column is found the algorithm backtracks to column $r - 1$. In this work it was demonstrated that there are no solution with balanced symbols for $k = 13$ and $k = 14$ for strength $t = 3$ with $N = 15$ rows. In Section 5.3 we prove that neither is there solution for these two cases with non balanced columns.

3.1.3 Constraint Programming

Hnich et al. [40] proposed four matrix models of constraint programming to construct covering arrays, called respectively naive matrix model, alternative matrix model, integrated matrix model, and weakened matrix model. The naive matrix model consists in an $N \times k$ matrix of integer variables $x_{r,i}$, $1 \leq r \leq N$, $1 \leq i \leq k$, such that $x_{i,r} = m$ if the value of the cell at row r and column i is m. The alternative matrix model consists in an $N \times \binom{k}{t}$ matrix of compound variables $y_{r,j}$, where each compound variable represents a tuple of variables $(x_{r,l_1}, x_{r,l_2}, \ldots, x_{r,l_t})$ of the naive matrix model. The integrated matrix model associates the compound variables in the alternative matrix model with their t corresponding variables in the original matrix model by means of channeling constraints. The weakened matrix model is a SAT encoding with several constraints omitted; this SAT encoding was developed to take advantage of the existing SAT local search methods.

3.1.4 SAT Encodings

Lopez-Escogido et al. [50] proposed a SAT encoding similar to the Hnich's encoding to generate covering arrays of strength two. For each element $m_{i,j}$ of the matrix M associated with the instance

are introduced v Boolean variables $m_{i,j,x}$, $0 \leq x < v$. The model of transformation uses two sets of clauses in conjunctive normal form (z_1 and z_2) to ensure each element of the matrix M takes exactly one value of the alphabet $\{0, 1, \ldots, v-1\}$; and uses a third set of clauses (z_3) in non-conjunctive normal form to guarantee the interactions among the columns of the covering array are satisfied. The three set of clauses are the following ones:

$$z_1 = \bigwedge_{i=0}^{N-1} \Big(\bigvee_{\forall j,x \,|\, 0 \leq j < k, 0 \leq x < v} m_{i,j,x} \Big)$$

$$z_2 = \bigwedge_{i=0}^{N-1} \Big(\bigvee_{\forall j,x,y \,|\, 0 \leq j < k, 0 \leq x < y < v} (\overline{m_{i,j,x}} \vee \overline{m_{i,j,y}}) \Big)$$

$$z_3 = \bigwedge_{\forall x,y \,|\, 0 \leq x < y < v} \Big(\bigwedge_{\forall j,l \,|\, 0 \leq j < l < k} \Big(\bigvee_{\forall i \,|\, 0 \leq i < N} (m_{i,j,x} \wedge m_{i,l,y}) \Big) \Big)$$

Taking the conjunction of these three set of clauses gives the complete SAT formula of the instances of covering arrays: $F = z_1 \wedge z_2 \wedge z_3$.

Banbara et al. [7] give another two SAT encodings to construct covering arrays, called respectively order encoding and mixed encoding.

3.1.5 Generation of Non-Isomorphic Covering Arrays

In a technical report Meagher [54] addressed the generation of non-isomorphic covering arrays of strength two and order two. The objective of this work is to find all the non-isomorphic covering arrays for given values of N, k, $t = 2$, and $v = 2$. The algorithm generates the covering arrays one column at a time from 0 columns to k columns rejecting the covering arrays which are not of minimum rank. The algorithm proposed can be used as an exact method to construct covering arrays because it can be stopped as soon as the first non-isomorphic covering array for the given parameters is constructed. The discarding of subarrays which are not of minimum rank acts as a pruning criteria in the exact method, because only the subarrays of minimum rank are extended.

3.2 Greedy Methods

For combinations of the parameters N, t, k, and v for which the exact methods are impractical, the searching methods to construct covering arrays can follow a greedy strategy to produce a good solution in low time. This section reviews four of the most relevant greedy methods to construct covering arrays.

3.2.1 The AETG System

The Automatic Efficient Test Generator (AETG system) is a generator of test suites (i.e., covering arrays) defined by the user. In [18] was developed a new method to generate the test suites in the AETG system. This method starts with an empty test suite and generates one test case at a time, until all the interactions of size t among the given parameters are covered. To obtain the following test case the algorithm generates several candidate test cases, and the test which covers the greater number of missing tuples is selected. A candidate test case is generated in the following way (let us consider $t = 2$ for simplicity):

1. A factor f is chosen, and for this factor is selected its value that appears in the greater number of uncovered pairs.

2. Let be $f_1 = f$, the remaining factors are sorted randomly $f_1, \ldots, f_k$.

3. Suppose the factors $f_1, \ldots, f_j$ have a value assigned. For $1 \leq i \leq j$ let be x_i the value of factor f_i. The value x_{j+1} for factor f_{j+1} is selected as follows: for every value v_{j+1} of factor f_{j+1} it is determined the number of new pairs in the set $\{(f_{j+1} = x_{j+1}, f_i = x_i)\}$ for $1 \leq i \leq j$; the value selected for factor f_{j+1} is the one which appears more times in the new pairs covered.

The set of candidate test cases is obtained exploiting the randomness in the ordering of the remaining factors in step 2, since a different factor ordering can yield a different test case.

3.2.2 Deterministic Density Algorithm

The deterministic density algorithm (DDA) was proposed by Bryce and Colbourn in [10] as a generator of test suites for interactions of size $t = 2$. One important characteristic of this algorithm is that it generates the test suites "one case at a time", i.e., the algorithm generates one test case ready for use, then generates the following test case, and so on until cover all the interactions of size two among the parameters. To construct the following test case the algorithm repeatedly fixes one level or value for each factor and updates the local and global densities. When all factors have been fixed to one level the algorithm emits the test case.

The following factor to fix is the one with the largest density among the non-fixed factors. The density $\delta_{i,j}$ for factors i and j is computed as follows: (a) $\delta_{i,j} = (r_{i,j}/l^2_{max})^2$ if both factors have more than one level left, where l_{max} is the largest cardinality of the two factors, and $r_{i,j}$ is the number of missing interactions among factors i and j; (b) $\delta_{i,j} = (r_{i,j}/l_{max})^2$ if only one factor has one level left; (c) $\delta_{i,j} = 1.0$ if both factors have exactly one level left and a new pair is covered; and (d) $\delta_{i,j} = 0$ if both factors have exactly one level and no new pair is covered. The density of factor i is the summation of the local densities over each factor $j \neq i$. If two or more factors have the largest density then a factor-tie-braking rule is used; the most simple rule is to choose the smaller factor in lexicographic order.

Having selected the factor to fix, the DDA algorithm assigns one level to it; and the level selected is the one with largest level density. The level densities are computed for a specific level v_i in relation to an individual factor k_j as follows: (a) $\delta = (r_{i,j}/l_{max})$ if k_j has more than one level left involved in uncovered pairs with v_i; (b) $\delta = 1.0$ if k_j has only one level left involved in uncovered pairs and a new pair is covered; and (c) $\delta = 0$ if k_j has only one level left involved in uncovered pairs and no new pair is covered. Similarly to the for factor-tie-breaking rules, the most simple level-tie-breaking rule is to select the smaller level in lexicographic order.

3.2.3 In-Parameter-Order Algorithm

In [49] Lei and Tai introduced the In-Parameter-Order algorithm (IPO) to generate test cases in which all the interactions of size t among the k factors are covered. One characteristic of this algorithm is that it expands the covering array both by rows and by columns. The IPOG algorithm [48] is a generalization for $t > 2$ of the IPO algorithm, and the IPOG-F algorithm [30] is an improvement of the IPOG algorithm.

These three algorithms share a common strategy based on using the covering array of $k-1$ factors to construct the covering array of k factors. The base of this recursive procedure is the covering array of $k = t$ factors, which is constructed trivially listing the t-tuples of the set $\{0, 1, \ldots, v-1\}$. Once the base has been constructed the algorithm adds the factors $t+1, t+2, \ldots, k$ one at a time. To add one factor the algorithms performs the following steps:

- *Horizontal Growth.* One additional column corresponding to the new factor is added; the cells of the new column are filled in such a way some tuples of the new covering array are covered.
- *Vertical Growth.* New rows are added to cover the missing tuples remaining after filling the cells of the new column.

There are several ways to implement the procedures of horizontal growth and vertical growth. To fill the cells of the new column the IPOG algorithm processes the rows from up to down and choose the symbol that covers the greatest number of missing tuples. In the other hand, the IPOG-F algorithm selects in a greedy manner both the row and the symbol for the row; this examination of all row/symbol paris gives to the IPOG-F algorithm the capacity of produce better results. The criterion for selecting the symbol is the same as the IPOG algorithm.

The vertical growth is similar in the three algorithms (IPO, IPOG, IPOG-F), and it consists in cover the missing tuples not covered by the horizontal growth. Given a missing tuple the algorithm tries to accommodate it in one of the existing rows; if there is no place then a new row is added

and the missing tuple is written in it. The $k - t$ remaining cells in the new row are free to try to accommodate the other missing tuples.

3.2.4 Coverage Inheritance

Calvagna and Gargantini [12] developed a new greedy algorithm to generate MCAs of any strength. As in the IPO algorithm, an MCA is constructed adding one parameter at a time until all the parameters have been added.

Suppose $t < k$. The algorithm begins with the creation of an MCA of strength t for the first t parameters, which is done listing the $\prod_{i=1}^{t} v_i$ tuples of the first t parameters (whose alphabet cardinalities are $v_1, v_2, \ldots, v_t$). It is assumed that the cardinalities of the k parameters are in descending order, so $v_i \geq v_j$ if $i \leq j$. Now, suppose that the first $j - 1$ parameters ($t \leq j \leq k$) have been added and they conform an MCA of strength t. The procedure to add the parameter j starts by copying one column $i < j$ in the new column j. Since $v_i \geq v_j$ the symbols s of the column i such that $s \geq v_j$ are mapped to a null symbol x in column j, while the symbols $s < v_j$ of column i are valid symbols for column j. A symbol x in column j indicates the cell is free to be assigned.

Since the new column j is a copy of a column $i < j$ every subarray of t columns satisfies the requirements to be an MCA of strength t, except the subarrays that contains both column i and column j. This is denominated the property of *coverage inheritance*. The tuples not yet covered are added to the matrix using the following procedure:

1. To initialize a set of flags, one for each cell of column j, to indicate that the cell can be freely modified.

2. To modify the free cells in column j to increment the coverage with respect to the column i.

3. If the operation destroys inherited tuples, then modify the free cells in order to restore the original coverage with column i.

4. If there are no free cells, then new rows are added to restore a tuple.

This procedure only modifies the cells of the new column j; so the previous columns (which are an MCA of strength t) are not modified. Both the selection of the column i to be copied in the new column j and the criteria to assign the free cells are implemented following greedy strategies.

3.3 Metaheuristic Methods

Like the greedy methods, the metaheuristic approaches do not guarantee to find the minimum number of rows for the covering array being constructed. I practice, the metaheuristic methods give very good results. This section describes the way in which the metaheuristics of simulated annealing, tabu search, and genetic algorithms have been used to construct covering arrays.

3.3.1 Simulated Annealing

The technique of simulated annealing is an analogy of the process of heating and cooling the metals in order to obtain a strong crystalline structure. The metal is heated until it reaches the liquid state, and after that the temperature is lowered slowly until the metal reaches its solid state again. During the process of cooling the particles of the metal are arranged in a crystalline structure such that the energy of the system is minimal [1]. The computational implementation of this technique requires three parameters for simulating the temperature scheduling, these parameters are the initial temperature T_0, the final temperature T_f, and the cooling rate α $(0 < \alpha < 1)$. At the beginning, the current temperature T_t is initialized with T_0, but in the process it is repeatedly reduced using $T_{t+1} = \alpha T_t$ until it reaches its final value T_f. To simulate the change of states in the metal the algorithm maintains a current state s_i with energy E_i. The next state s_j with energy E_j is obtained by perturbing the current state s_i. If the energy of the new state s_j is less than the energy of s_i then s_j becomes the new current state; otherwise state s_j is accepted with probability $e^{-(E_j - E_i)/k_B T_t}$, where k_B is the Boltzmann constant. A fourth parameter L, called the length of the Markov chain,

defines the number of perturbations performed at the same temperature T_t. The algorithm stops if a desired solution is found, if the final temperature was reached, or if for ϕ consecutive temperatures changes the global best solution was not improved.

Covarrubias-Flores [29] proposed an algorithm of simulated annealing to construct binary covering arrays of variable strength. The initial solution of the algorithm is generated randomly but has the characteristic that the number of ones and zeroes are balanced in every column of the initial matrix. The evaluation function is the number of missing tuples. The neighborhood function is composed by two functions N_2 and N_5, the function N_2 has a utilization rate of 60% and the function N_5 has the remaining 40%. The function N_2 consists in making randomly β operations of complement in the matrix and in selecting the operation which minimizes the evaluation function; an operation of complement changes the content of one cell from 0 to 1 and vice versa. The function N_5 randomly selects one column j of the matrix and performs all the possible exchanges among the cells of the column, the selected exchange is the one which minimizes the evaluation function.

Another algorithm of simulated annealing to construct binary covering arrays is proposed in [71] by Torres-Jimenez and Rodriguez-Tello. To generate the initial solution this algorithm uses a heuristic consisting in filling randomly every column with $N/2$ ones and $N/2$ zeroes if N is even, and $\lfloor N/2 \rfloor + 1$ ones and $\lfloor N/2 \rfloor$ zeroes if N is odd. So, the number of ones and zeroes is different in at most one unit in every column of the initial solution. Let be A the initial solution, from A is obtained a new solution A' by means of a neighborhood function based on the functions *switch*(A, i, j) and *swap*(A, i, j, l). The function *switch*(A, i, j) changes the value of the cell (i, j) of A by a different value of the alphabet. The function *swap*(A, i, j, l) interchanges the values of the cells (i, j) and (l, j) of the j-th column of A. The cost of a solution A is the number of missing tuples in A. The parameters used in this simulated annealing algorithm were $T_0 = 4.0$, $\alpha = 0.99$, $L = (Nkv)^2$, $T_f = 1.0 \times 10^{-10}$, and $\phi = 11$. The algorithm is stopped if the current solution is a covering array, if the final temperature was reached, or if ϕ becomes 11. An improvement of this algorithm is given by the same authors in [72].

Avila-George et al. [3, 6] proposed three parallel approaches of simulated annealing to construct covering arrays, denominated independent search, semi-independent search, and cooperative search. In the independent search each processor executes its own sequential simulated annealing, the only interaction among the processors is at the beginning when all of them receive the same initial solution, and at the end when they all report their best solution found, in order to obtain the final best solution. The semi-independent approach consists in dividing the Markov chains equitably among the available processors; the processors interchange intermediate solutions and everyone updates its current best solution with the best of the intermediate solutions. The cooperative approach maintains a unique global best solution, which is accessible to all processors when they finish their respective Markov chain. When a processor finishes its Markov chain it may update the global best solution, but in any case it starts the next Markov chain with the current global best solution without waiting that the other processors complete their chains; so not all the processors start their i-th Markov chain with the same solution, which favors the algorithm can scape from local optima.

3.3.2 Tabu Search

Tabu search is an optimization technique formalized by Glover in [31]. This technique owes its name to a list called tabu list used to record the last movements done. The tabu list is used to avoid the algorithm returns to a neighborhood recently explored, and it can go towards more promising zones of the search space. Given a solution x in a neighborhood X, tabu search usually explores all the neighborhood X in a deterministic way, which contrasts with other techniques where the exploration of the neighborhoods is performed at random. If a solution x' better than the current solution x is found, then x' becomes the new best current solution. In case of none solution better than the actual is found in the neighborhood, the best solution of the neighborhood is taken as the new best solution, inclusive if it is not better than the previous best solution; this is done to escape from local optima. At the end of each iteration, the tabu list is updated, and given that the size of the tabu list is fixed, some movements are forgotten as other movements enter into the list.

Tabu search has been applied to the construction of covering arrays. In [66] Stardom proposed an algorithm of tabu search which explores all the matrices in the neighborhood of the current solution. The neighborhood of a matrix A is defined as the set of all the matrices than can be obtained from A by changing the content of one random cell of A. The best element of the neighborhood becomes the new best solution, without considering that it is better or worse than the current solution. The cost of a solution is the number of missing tuples. The administration of the tabu list is done in the function that performs the movements; this function receives as parameters a row r, a column c, an old value i, and a new value j, which indicate to change the content of the entry (r, c) of the matrix from value i to value j. This way, for a movement with parameters (r, c, i, j) the tabu list is reviewed to see if in the last L movements there is one with parameters (r', c', i', j') such that $r = r'$, $c = c'$, and either $i = i'$ and $j = j'$, or $i = j'$ and $j = i'$; if one of these two cases occur then the movement is forbidden because it is one of the recently done or it is the inverse of one recently done.

Also Nurmela in [57] applied the technique of tabu search to construct covering arrays. The algorithms starts by initializing randomly the matrix for the covering array. The cost of the matrix is the number of missing tuples in the matrix. From the set of all missing tuples in the current matrix, the algorithm selects one tuple randomly, and search the rows of the matrix that can cover the tuple by modifying only one element of the row; these modifications are the movements in the current neighborhood. The cost of these movements are computed and the algorithm selects the one with lower cost, whenever it is not in the tabu list. In case of an empty neighborhood the algorithm selects any row and writes the missing tuple in it.

In [32] Gonzalez-Hernandez et al. is presented a new algorithm of tabu search to construct MCAs of strengths 2 through 6. In this algorithm the initial matrix M can be initialized randomly, with the maximum Hamming distance among its rows, or with the symbols balanced in every column. The changes in the current solution are performed by four neighborhood functions $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$, and $\mathcal{N}_4$, each of one has a probability value of being applied to change the current solution. The function $\mathcal{N}_1$

performs all possible symbol changes in one cell (i, j) selected randomly; the function $\mathcal{N}_2$ performs all possible symbol changes in every cell of a column j of the matrix; the function $\mathcal{N}_3$ performs all the symbol changes in all the cells of the current solution; and finally the function $\mathcal{N}_4$ writes a missing tuple in every row of the matrix. Every movement in an application of a neighborhood function is done independently, and it is selected the movement that minimizes the number of missing tuples. The tabu list is composed by tuples $(\mathcal{N}, v, i, j, m)$, where $\mathcal{N}$ is the neighborhood function and v is the symbol assigned to the cell $m_{i,j} \in M$ by $\mathcal{N}$; the movements inserted in the tabu list are those movements that produces exactly the same number l of missing tuples.

Gonzalez-Hernandez and Torres-Jimenenez introduced [33] a new approach of tabu search to construct MCAs called MiTS (Mixed Tabu Search). This approach is based on using a mixture of neighborhood functions, plus a fine tuning of the parameters for the tabu search algorithm and of the probability values of applying each of the neighborhood functions. The initial solution is created randomly or it is constructed one row at a time maximizing the Hamming distance of the new row with respect to the previous rows in the matrix. The tabu list of the MiTS algorithm is composed by movements (i, j, v, F) that generate the symbol v more than once using the same neighborhood function F in the cell (i, j) of the matrix. The evaluation function is the number of missing tuples of the matrix. MiTS uses three neighborhood functions F_1, F_2, and F_3; the function F_1 selects a random position (i, j) of the matrix and performs all possible symbol changes in that position; the function F_2 selects a random column of the matrix and performs all symbols changes in every cell of the column; and the function F_3 performs all symbol changes in every cell of the matrix.

3.3.3 Genetic Algorithms

The genetic algorithms were introduced by Holland in [41] to understand the adaptive process of the natural systems. These algorithms pertain to the class of evolutionary algorithms, which are inspired in the biological process of the evolution of the species. The general working of these algorithms is the following (see [68]): a population of individuals is generated randomly, where the individuals

represent possible solutions to the problem and each of them has an associated value that indicates its fitness. In every iteration of the algorithm the fittest individuals are selected to be the parents of the new generation of individuals. The selected parents produce new individuals by means of variation operators like crossover and mutation. Finally a mechanism of generational replacement determines which individuals from parents and children pass to the next iteration of the algorithm. The main characteristic of the genetic algorithms is that the crossover operator is applied to a pair of parents to produce a pair of children, and after that a mutation operator is applied to the children.

Stardom [66] a genetic algorithm to construct covering arrays. In this algorithm the population is a set of matrices with missing tuples; the fitness of the matrices is the number of missing tuples, so the fittest matrices are those which have less missing tuples. To select the parents of the new individuals, the set S of the individuals in the current population is divided in two groups of size $|S|/2$; in each group the individuals are sorted randomly and the i-th members of each group are combined with the crossover operator to produce $|S|$ new individuals. The crossover operator consists in selecting a set E of coordinates (i, j) from one parent and in copying these coordinates into one child, the remaining coordinates to complete the child are taken from the second parent. The set E that is taken from one of the parents can be the first i complete rows, the first j complete columns, or a block conformed by the cells in the first i rows and in the first j columns. In each case is $i < N$ and $j < k$, where N and k are the dimensions of the matrices in the population, in order to ensure that the individuals produced by the crossover operator have coordinates of both parents. After that, the mutation operator is applied to the new $|S|$ individuals; this operator consists in changing the content of one cell selected randomly by other element of the alphabet. Next, the mean of the fitness of the $|S|$ parents and of the $|S|$ children is computed; the T matrices with a fitness smaller than the mean pass to the next generation, plus $|S| - T$ matrices randomly selected from the matrices with fitness greater than or equal to the mean.

Shiba et al. [64] proposed another genetic algorithm to construct test suites. The approach of this algorithm is to generate one test case at a time. To generate a test case the algorithm creates

P random candidate solutions. These candidate solutions are evolved until the best solution is not improved in T consecutive generations. The fitness of a candidate solution S is defined as the number of tuples not covered in the initial test but covered in S. The crossover operator consists in exchanging with a probability of 0.5 the values of every position of two candidate solutions. The mutation operator is applied after the crossover operator and it replaces the value of one position of the candidate solution with the value of another position of the same candidate solution.

3.4 Algebraic Methods

The algebraic methods have the characteristic that in the process of the construction of the covering array CA($N; t, k, v$) are involved formulas or operations with mathematical objects such as vectors, finite fields, groups or another covering arrays with small values of t, k, v.

3.4.1 Case $t = 2$ and $v = 2$

For the case $t = 2$ and $v = 2$ there exists an algorithm that given the number of rows N constructs in polynomial time a covering array CA($N; 2, k, 2$) with the maximum number of columns k; the result is an optimal covering array. This algorithm was proposed independently by Rényi [59] for N even, and by Katona [42] and Kleitman and Spencer [43] for all N.

The optimal number of columns for a covering array CA($N; 2, k, 2$) of N rows is given by the maximum value of k that satisfies the Inequality (3.1).

$$k \leq \binom{N-1}{\lceil \frac{N}{2} \rceil} \tag{3.1}$$

To construct the covering array it is created a matrix of size $N \times k$ in which all the elements of the first rows are zero, and the remaining $N-1$ rows are filled column by column (starting from the second row) with the $\binom{N-1}{\lceil \frac{N}{2} \rceil}$ combinations of $\lceil \frac{N}{2} \rceil$ ones and $N - 1 - \lceil \frac{N}{2} \rceil$ zeroes.

As an example of the algorithm consider $N = 6$. The maximum value of k that satisfies the Inequality (3.1) is $k = 10$; so a matrix of size 6×10 is created and its first row is filled with zeroes. The remaining five rows are filled column by column with the $\binom{6-1}{\lceil \frac{6}{2} \rceil} = \binom{5}{3} = 10$ combinations of $\lceil \frac{6}{2} \rceil = 3$ ones and $6 - 1 - \lceil \frac{6}{2} \rceil = 2$ zeroes, as shown in Figure 3.1.

1	0	0	1	1	1
2	0	1	0	1	1
3	1	0	0	1	1
4	0	1	1	0	1
5	1	0	1	0	1
6	1	1	0	0	1
7	0	1	1	1	0
8	1	0	1	1	0
9	1	1	0	1	0
10	1	1	1	0	0

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

Figure 3.1: To the left are the 10 combinations of 3 ones and 2 zeroes and to the right is the covering array CA(6; 2, 10, 2). The covering array is constructed filling its first row with zeroes and filling its other five rows column by column with the 10 combinations of 3 ones y 2 zeroes.

3.4.2 The Bush's Construction

The Bush's construction [11] allows to obtain an orthogonal array OA(v^t; t, $v+1$, v) of index unity ($\lambda = 1$) when the order v of the OA is prime or is a prime power, and $k = v + 1$.

Let be p a prime number, it is defined $GF(p)$ as the set of all the integers modulo p [8]. For $q = p^n$ where n is a positive integer, the elements of $GF(q)$ can be expressed as polynomials $P(x)$ of degree $n - 1$ with coefficients in $GF(p)$, taking every combination for the coefficients:

$$P_i(x) = \sum_{r=0}^{n-1} a_{ir} x^r \quad \text{for} \quad i = 0, 1, \ldots, q-1 \tag{3.2}$$

The addition is closed in $GF(q)$, but the multiplication can produce polynomials of degree greater than $n-1$, which are not members of $GF(q)$. To make the multiplication closed in $GF(q)$ it is required

a primitive element of $GF(q)$ of degree less than or equal to $n-1$ to reduce the terms of degree greater than $n-1$ produced in the multiplication of two polynomials.

To construct the OA it is created a matrix with dimensions $v^t \times (v+1)$. The first v columns of this matrix are labeled with the elements e_j of $GF(v)$ and the last column is labeled with ∞. In other hand, the rows of the matrix are labeled with the v^t different polynomials $y_i = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \cdots + a_1x + a_0$ that can be formed with coefficients $a_j \in GF(v)$. The number of these polynomials is v^t because there are v different coefficients by each one of the t terms of the polynomial.

As an example consider $v = 4$ and $t = 2$; here v is a prime power because $4 = 2^2$, and $p = 2$ and $n = 2$ in the expression $v = p^n$. This way, the OA to be created is OA$(16; 2, 5, 4)$, since the number of rows of the OA given by the construction is v^t and the number of columns is $v+1$. The field $GF(2^2)$ is conformed by the elements $0x+0$, $0x+1$, $1x+0$, and $1x+1$, which are the 2^2 different polynomials of $t = 2$ terms and degree $t-1$ that can be formed with coefficients in $GF(2)$. In a simplified way these polynomials are 0, 1, x, and $x+1$, and they will be denoted by e_0, e_1, e_2, and e_3 respectively. Since $t = 2$, the polynomials y_i used to label the rows are of the form $y_i = a_1x + a_0$, where $a_1, a_0 \in GF(2^2) = \{e_0, e_1, e_2, e_3\}$. For this example, the labeling of the rows and the columns of the OA are as shown in Figure 3.2.

The cells of the OA are filled according to the next two rules, the first rule is for the first v columns of the OA, and the second rule is for the last column of the OA:

1. To assign the value u to every cell (i, j) with $0 \leq i \leq v^t - 1$ and $0 \leq j \leq v-1$ if $y_i(e_j) = e_u$; that is, evaluate the polynomial $y_i(x)$ with $x = e_i$ and determine the result in $GF(v)$.

2. To assign the value u to every cell (i, j) with $0 \leq i \leq v^t - 1$ and $j = v$ if e_u is the main coefficient of the polynomial $y_i(x)$; that is, $e_u = a_{t-1}$ in the polynomial $y_i(x)$.

For the example of Figure 3.2 one primitive element of $GF(2^2)$ is the polynomial $x+1$, and so the terms with degree grater than 1 obtained in the evaluation of the polynomials y_i will be reduced using

	e_0	e_1	e_2	e_3	∞
$y_0(x) = e_0x + e_0$	0	0	0	0	0
$y_1(x) = e_0x + e_1$	1	1	1	1	0
$y_2(x) = e_0x + e_2$	2	2	2	2	0
$y_3(x) = e_0x + e_3$	3	3	3	3	0
$y_4(x) = e_1x + e_0$	0	1	2	3	1
$y_5(x) = e_1x + e_1$	1	0	3	2	1
$y_6(x) = e_1x + e_2$	2	3	0	1	1
$y_7(x) = e_1x + e_3$	3	2	1	0	1
$y_8(x) = e_2x + e_0$	0	2	3	1	2
$y_9(x) = e_2x + e_1$	1	3	2	0	2
$y_{10}(x) = e_2x + e_2$	2	0	1	3	2
$y_{11}(x) = e_2x + e_3$	3	1	0	2	2
$y_{12}(x) = e_3x + e_0$	0	3	1	2	3
$y_{13}(x) = e_3x + e_1$	1	2	0	3	3
$y_{14}(x) = e_3x + e_2$	2	1	3	0	3
$y_{15}(x) = e_3x + e_3$	3	0	2	1	3

Figure 3.2: OA produced by the Bush construction for $v = 4$ and $t = 2$.

this primitive element. This way, the values of the cells (i, j) in the first $v = 4$ columns of the OA are obtained by evaluating the polynomial $y_i(x) = e_ax + e_b$ in the element $e_j \in GF(v)$, $0 \leq a, b, j \leq 3$. The evaluation produces an element $e_u \in GF(v)$ and the index u of this element will be the value for the cell (i, j). For example, the value of the entry $(11, 3)$ is given by $y_{11}(e_3) = e_2e_3 + e_3$. The product e_2e_3 is equal to $(x)(x + 1) = x^2 + x$, which using the primitive element $x + 1$ and taking the modulo 2 of the coefficients is reduced to $x^2 + x = (x + 1) + x = 2x + 1 = 0x + 1 = 1$. The final result is given by $1 + e_3$, which is equal to $1 + (x + 1) = x + 2 = x + 0 = x$. The element x is the element e_2 of $GF(v)$, so the value for the entry $(11, 3)$ is 2.

The most time consuming part of the construction is the evaluation of the polynomials $y_i(x)$, because $v^tv = v^{t+1}$ evaluations must be performed, one for each cell in the first v columns of the OA. Depending on the value of t, the evaluation of a polynomial requires several multiplications and additions of elements of $GF(v)$. Torres-Jimenez et al. [70] developed a method to evaluate efficiently the polynomials $y_i(x)$ by means of the tables of sum, logarithm and antilogarithm of $GF(v)$. The evaluation of a polynomial y_i only involves accesses to these tables and sum of elements in the tables.

3.4.3 Constant Weight Vectors

Tang and Woo [69] proposed a method to construct test cases for logic circuits based on vectors of a particular set of weights. The weight of a vector $s = (s_1, s_2, \ldots, s_k)$ is defined as the sum of its entries: $w = \sum_{i=1}^{k} s_i$. The method constructs the test suite (the covering array) concatenating sets of vectors in which the vectors in the same set have the same weight w. For $v = 2$ the construction of the covering array is defined by the following theorem:

THEOREM **1** *Given k and t, $k > t$, then a set T of binary vectors covers all the interactions of size t if it contains all the binary vectors of weight w such that $w \equiv c$ mod $(k - t + 1)$ for a constant $c \in \{0, 1, \ldots, k - t\}$.*

The theorem provides the weights w of the vectors to construct the covering array. The constant c can vary from 0 to $k - t$; so there are $n - k + 1$ sets of weights which are a solution. To find the solution with less number of rows, the $n - k + 1$ solutions are checked.

For example, let be $k = 6$ and $t = 3$. The theorem provides $k - t + 1 = 6 - 3 + 1 = 4$ solutions given by the set of weights obtained from the expression $w \equiv c$ mod $(k - t + 1)$ when c varies from 0 to $k - t = 6 - 3 = 3$. The vector with less weight is the vector $(0, 0, 0, 0, 0, 0)$ whose weight is $w = 0$, and the vector with greater weight is $(1, 1, 1, 1, 1, 1)$ whose weight is $w = 6$. In general, the smaller weight of a vector for a covering array with k columns and order v is $w = 0$, and the greater weight is $w = k(v - 1)$. Table 3.1 shows the four sets of weights that are solution for $k = 6$ and $t = 3$, the last column of the table contains the total number of vectors of the covering array obtained from the set of weights in the second column.

3.4.4 Trinomial Coefficients

In [51] and [52] Martinez-Pena et al. presented a construction for ternary covering arrays ($v = 3$) based on sets of rows that can be represented by trinomial coefficients. Let be a, b, c, k integers such

c	$w \equiv c \bmod (k-t+1)$	Vectors
0	$\{0,4\}$	16
1	$\{1,5\}$	12
2	$\{2\}$	15
3	$\{3\}$	20

Table 3.1: Set of weights that are a solution for $k = 6$, $t = 3$, and $v = 2$.

that $0 \leq a, b, c \leq k$ and $k = a + b + c$. A trinomial coefficient, denoted by $\binom{k}{a,b,c}$, is the coefficient given by the following Equation (3.3):

$$\binom{k}{a,b,c} = \frac{(a+b+c)!}{a!\,b!\,c!} \tag{3.3}$$

The trinomial coefficient $\binom{k}{a,b,c}$ is used to represent the set of all different rows of k elements that can be formed with a 0's, b 1's, and c 2's. The number of such rows is precisely the numeric value of the trinomial coefficient.

For any strength $t \leq k$ may be constructed a covering array of k columns by a vertical juxtaposition of subsets of rows represented by trinomial coefficients. Like the constant weight vectors for binary covering arrays, there are in general many combinations of trinomial coefficients that conform a covering array of k columns and strength t. For $t \in \{2,3,4,5\}$ the authors derived four direct constructions that explicitly give the trinomial coefficients to construct a covering array of k columns and strength $t \in \{2,3,4,5\}$; so, there is no need to perform a search (except for a few cases) in order to determine the combination of trinomial coefficients to produce the covering array with minimum number of rows. These four direct constructions are shown in Table 3.2.

For example, for $t = 2$ and $k = 4$ the covering array with less number of rows is conformed by the trinomial coefficients of the first row of Table 3.2, which are $\binom{4}{3,0,1}, \binom{4}{1,3,0}, \binom{4}{0,1,3}$. The result of concatenating these trinomial coefficients is the covering array of $3k = 12$ rows CA$(12; 2, 4, 3)$.

Restrictions	Coefficients	Total rows
$t = 2,\ k \geq 3$	$\{\binom{k}{k-1,0,1}, \binom{k}{1,k-1,0}, \binom{k}{0,1,k-1}\}$	$3k$
$t = 3,\ k \geq 5$	$\{\binom{k}{k-1,0,1}, \binom{k}{k-1,1,0}, \binom{k}{1,0,k-1}, \binom{k}{0,1,k-1}, \binom{k}{1,k-2,1}\}$	$k^2 + 3k$
$t = 4,\ k \geq 7$	$\{\binom{k}{k-2,0,2}, \binom{k}{0,2,k-2}, \binom{k}{k-2,2,0}, \binom{k}{k-2,1,1}, \binom{k}{1,1,k-2}, \binom{k}{1,k-2,1}\}$	$4.5k^2 - 4.5k$
$t = 5,\ k \geq 7$	$\{\binom{k}{k-2,0,2}, \binom{k}{0,2,k-2}, \binom{k}{2,k-2,0}, \binom{k}{k-3,2,1}, \binom{k}{2,1,k-3}, \binom{k}{1,k-3,2}\}$	$\frac{4.5(k-1)^3-4.5(k-1)^2}{3}$

Table 3.2: Direct constructions using trinomial coefficients for $t \in \{2, 3, 4, 5\}$.

3.4.5 Cyclotomy

Colbourn in [24] presented some constructions resulting from the analysis of the cyclotomic matrices. Let be q a prime power such that $q \equiv 1 \pmod{v}$, where v is the alphabet of the covering array, and let be $\mathbf{x}_{q,v} = (x_i : i \in GF(q))$ a vector of q elements of the set $\{0, 1, \ldots, v-1\}$. The vector $\mathbf{x}_{q,v} = (x_i : i \in GF(q))$ is constructed based on a primitive element ω of $GF(q)$ by doing $x_0 = 0$ and $x_i = j \bmod v$ when $i = \omega^j$ for $i \in GF(q)$. The cyclotomic vector $\mathbf{x}_{q,v}$ produces the cyclotomic matrix $A_{q,v} = (a_{i,j})$ of size $q \times q$ making $a_{i,j} = x_{j-i}$ (with $j - i$ computed in modulo q). Sometimes this matrix A is a covering array of strength t: CA$(q; t, q, v)$. The construction ensures that A is a covering array of strength t if $q > t^2 v^{4t}$. In the binary case A is a covering array if $q > t^2 2^{2t-2}$.

For some cases better bounds are possible. Consider $v = 2$, $t = 2$, and $q = 9$; in this case $q < t^2 2^{2t-2}$ but the construction produces a covering array. To determine the cyclotomic vector $\mathbf{x}_{9,2}$ it is constructed the logarithm table of $GF(9) = \{0, 1, 2, 3, 4, 5, 6, 7, 8\} = \{0,\ 1,\ 2,\ x,\ x+1,\ x+2,\ 2x,\ 2x+1,\ 2x+2\}$, raising the element $x \in GF(9)$ to the powers $p = 1, 2, \ldots, 8$, and reducing the powers of degree greater than 1 by substituting x^2 for the primitive element $x + 1 \in GF(9)$. Table 3.3 shows the logarithm table for $GF(9)$; the first two columns shows the computation of the powers of x; and the last column indicates the exponent to which x was raised to get the element of $GF(9)$ that appears in the third column.

Taking the modulo 2 of the logarithms $8, 4, 1, 2, 7, 5, 3, 6$ in the last column of Table 3.3 are obtained the values $0, 0, 1, 0, 1, 1, 1, 0$. These values will be the elements $i = 1, 2, \ldots, 8$ of the

p	x^p	$GF(9)$	$\log_{x+1}$
1	x	1	8
2	$x+1$	2	4
3	$2x+1$	x	1
4	2	$x+1$	2
5	$2x$	$x+2$	7
6	$2x+2$	$2x$	5
7	$x+2$	$2x+1$	3
8	1	$2x+2$	6

Table 3.3: Logarithm table for $GF(9)$.

cyclotomic vector $\mathbf{x}_{9,2}$; the first element of the cyclotomic vector is $\mathbf{x}_0 = 0$. Therefore, the cyclotomic matrix $A_{9,2} = (a_{i,j})$ conformed by the rotations of the cyclotomic vector $\mathbf{x}_{9,2} = (0\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 0)$ will be a covering array CA$(9; 2, 9, 2)$.

3.4.6 Constructions Using Groups

Chateauneuf et al. introduced in [16] a method to construct covering arrays of strength three by means of a group acting on a set of v symbols $\{0, 1, \ldots, v-1\}$. Later, Meagher and Stevens [55] adapted this method to produce covering arrays of strength two CA$(k(v-1)+1; 2, k, v)$ by means of a group $G < Sym_v$ and a *starter vector* $s \in \mathbb{Z}_v^k$.

The fundamental step in this method is to find a starter vector s given the group G. A vector $s \in \mathbb{Z}_v^k$ is a starter vector if all the sets $d_1, d_2, \ldots, d_{k-1}$, defined as $d_i = \{(s_j, s_{j+i}) \mid j = 0, 1, \ldots, k-1)\}$ with subscripts in modulo k, have at least one element from every orbit of the group action of G on pairs of $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$.

The starter vector s is used to form a matrix M of size $k \times k$ by juxtaposing vertically the rotations of s. After that, the group action of G over M produces $v-1$ matrices of $k \times k$; these matrices and the vector $(0\ 0\ \cdots\ 0)$ of size k are concatenated vertically to form the covering array. The vector $(0\ 0\ \cdots\ 0)$ is added to cover the tuples $(0, 0)$, since these tuples might not be covered by the concatenation of the $v-1$ matrices.

For example, let be $t = 3$, $k = 5$, and $v = 3$, a starter vector for the group $G = \langle(1,2)\rangle = \{e, (1,2)\}$ is $s = (0,1,1,1,2)$. With this starter vector is formed the following circulant matrix:

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 2 \\ 1 & 1 & 1 & 2 & 0 \\ 1 & 1 & 2 & 0 & 1 \\ 1 & 2 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \end{pmatrix}$$

The elements of G acting on M produce the following two matrices, which are concatenated vertically with the vector $(0\ 0\ 0\ 0\ 0)$ to produce the covering array $\mathsf{CA}(11; 2, 5, 3)$.

$$M_e = \begin{pmatrix} 0 & 1 & 1 & 1 & 2 \\ 1 & 1 & 1 & 2 & 0 \\ 1 & 1 & 2 & 0 & 1 \\ 1 & 2 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \end{pmatrix}; \qquad M_{(1,2)} = \begin{pmatrix} 0 & 2 & 2 & 2 & 1 \\ 2 & 2 & 2 & 1 & 0 \\ 2 & 2 & 1 & 0 & 2 \\ 2 & 1 & 0 & 2 & 2 \\ 1 & 0 & 2 & 2 & 2 \end{pmatrix}$$

The element e is the identity, so M_e is equal to M; the matrix $M_{(1,2)}$ is obtained by interchanging the symbols 1 and 2 in the matrix M.

3.4.7 Roux-type Constructions

In [65] appears the following theorem taken form the doctoral thesis of G. Roux [62]:

$$\mathsf{CAN}(3, 2k, 2) \leq \mathsf{CAN}(3, k, 2) + \mathsf{CAN}(2, k, 2).$$

The proof of this theorem is a way to construct a covering array with twice the number of columns of its two *ingredients*: given the covering arrays $A = \mathsf{CA}(N_3; 3, k, 2)$ and $B = \mathsf{CA}(N_2; 2, k, 2)$ it is possible to construct the covering array $C = \mathsf{CA}(N_3 + N_2; 3, 2k, 2)$ as shown next, where $\overline{B}$ is the complementary matrix of B:

$$C = \begin{pmatrix} A & A \\ B & \overline{B} \end{pmatrix}$$

Chateauneuf and Kreher [15] found three generalizations of the Roux theorem, also given as theorems. In the first two theorems it is mentioned the combinatorial object called *ordered design*. An ordered design $\mathrm{OD}(t,k,v)$ is an array of dimensions $\binom{v}{t}t! \times k$ on v symbols such that in every combination of t distinct columns each t-tuple of t distinct elements is covered exactly once [15].

THEOREM 2 *For* $v \geq 3$, $CAN(2,3k,v) \leq CAN(2,k,v) + v(v-1)$.

Proof: Let be $A = CA(N_2;2,k,v)$, let be $B = OD(2,3,v)$, and let be B_j the column j of B repeated k times. Then the following array C is a covering array $\mathrm{CA}(N_2 + v(v-1);2,3k,v)$:

$$C = \begin{pmatrix} A & A & A \\ B_1 & B_2 & B_3 \end{pmatrix}$$

□

THEOREM 3 *If there is an* $OD(2,m,u)$ *then for* $v \leq u$ *and* $2 \leq k \leq m$, $CAN(2,\ m(m-1)k,\ v) \leq CAN(2,k,v) + 2u(u-1)$.

Proof: Let be $A = \mathrm{CA}(N_2;2,k,v)$, let be $B = \mathrm{OD}(2,m,u)$, and let be B_j the j-th column of B repeated k times. For $i = 1,\ldots,m-1$, and subscripts in $\mathbb{Z}_m$, construct the next array C_i of size $(N_2 + 2u(u-1)) \times mk$:

$$C_i = \begin{pmatrix} A & A & \cdots & A \\ B_1 & B_2 & \cdots & B_m \\ \vdots & \vdots & \ddots & \vdots \\ B_{i+1} & B_{i+2} & \cdots & B_{i+m} \end{pmatrix}$$

Then $C = (C_1\ C_2\ \cdots\ C_{m-1})$ is the covering array $\mathrm{CA}(N_2 + 2u(u-1);\ 2,\ m(m-1)k,\ v)$. □

THEOREM 4 $CAN(3,2k,v) \leq CAN(3,k,v) + (v-1)\,CAN(2,k,v)$.

Proof: Let be $A = \mathrm{CA}(N_3;3,k,v)$, let be $B = \mathrm{CA}(N_2;2,k,v)$, and let be C_v the cyclic group of permutations generated by $\pi = (1,2,\ldots,v)$. Construct the covering array $C =$

CA($N_3 + (v-1)N_2; 3, 2k, v$) as follows, where B^g is the array obtained by applying the permutation g to the symbols in B:

$$C = \begin{pmatrix} A & A \\ B & B^{\pi^1} \\ B & B^{\pi^2} \\ \vdots & \vdots \\ B & B^{\pi^{v-1}} \end{pmatrix}$$

□

Generalizations for greater strengths and alphabets have been developed by Cohen et al. [20], Colbourn et al. [28], and Martirosyan and van Trung [53].

3.4.8 Product of Covering Arrays of Strength Two

In [27] it is shown a procedure to multiply two covering arrays of strength two. Let be $A = (a_{i,j})$ the covering array CA($N; 2, k, v$), and let be $B = (b_{i,j})$ the covering array CA($M; 2, l, v$). The product of A and B, denoted by $A \otimes B$, is the matrix $C = (c_{i,j})$ of size $(N + M) \times kl$ such that:

- $c_{i,(f-1)k+g} = a_{i,g}$ para $1 \leq i \leq N$, $1 \leq f \leq l$, $1 \leq g \leq k$.
- $c_{N+i,(f-1)k+g} = b_{i,f}$ para $1 \leq i \leq M$, $1 \leq f \leq l$, $1 \leq g \leq k$.

Basically k copies of the covering array B are pasted to l copies of the covering array A, as shown in Figure 3.3. The result of the product of A and B is the covering array C = CA($N + M; 2, kl, v$).

3.4.9 Power of a Covering Array

In [37] Hartman presented a method to squaring the number of columns k of a covering array A = CA($N; t, k, v$). In order to do the powering it is required an orthogonal array $B = (k^2; 2, T(v,t)+1, k)$ with k^2 rows, $T(v,t)+1$ columns, strength 2, and order k. The order of the orthogonal array should be k because of its elements will be used as indices for the columns of A. $T(v,t)$ is the Turán number for v and t, and it denotes the maximum number of edges in a v-partite graph with t nodes.

$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,k}$	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,k}$	$\cdots$	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,k}$
$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,k}$	$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,k}$	$\cdots$	$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,k}$
$\vdots$				$\vdots$				$\cdots$	$\vdots$			
$a_{N,1}$	$a_{N,2}$	$\cdots$	$a_{N,k}$	$a_{N,1}$	$a_{N,2}$	$\cdots$	$a_{N,k}$	$\cdots$	$a_{N,1}$	$a_{N,2}$	$\cdots$	$a_{N,k}$
$b_{1,1}$	$b_{1,1}$	$\cdots$	$b_{1,1}$	$b_{1,2}$	$b_{1,2}$	$\cdots$	$b_{1,2}$	$\cdots$	$b_{1,l}$	$b_{1,l}$	$\cdots$	$b_{1,l}$
$b_{2,1}$	$b_{2,1}$	$\cdots$	$b_{2,1}$	$b_{2,2}$	$b_{2,2}$	$\cdots$	$b_{2,2}$	$\cdots$	$b_{2,l}$	$b_{2,l}$	$\cdots$	$b_{2,l}$
$\vdots$				$\vdots$				$\cdots$	$\vdots$			
$b_{M,1}$	$b_{M,1}$	$\cdots$	$b_{M,1}$	$b_{M,2}$	$b_{M,2}$	$\cdots$	$b_{M,2}$	$\cdots$	$b_{M,l}$	$b_{M,l}$	$\cdots$	$b_{M,l}$

Figure 3.3: Product of the covering arrays $A = \mathrm{CA}(N; 2, k, v)$ and $B = \mathrm{CA}(M; 2, l, v)$. The result is the covering array $\mathrm{CA}(N + M; 2, kl, v)$.

The powering procedure consists in creating the matrix of blocks C of k^2 columns and $T(v,t)+1$ rows. Each element of C will contain one column of A. Let be $B[i,j]$ the entry (i,j) of the orthogonal array B, and let be A_i the i-th column of A. The cell (i,j) of C is the column $B[j,i]$ of A, i.e., $C[i,j] = A_{B[j,i]}$, as shown in Figure 3.4. The result is the covering array $C = \mathrm{CA}(N(T(v,t)+1);\, t,\, k^2,\, v)$.

3.5 Manipulation of Covering Arrays

There are some useful operations that can be applied to a covering array previously constructed. This section describes five of them: verification, maximization of constant rows, optimal reduction, wildcard detection, and fusion.

3.5.1 Verification of Covering Arrays

As mentioned in Chapter 2 the verification of covering arrays consists in determining if a given matrix is or not a covering array of a certain strength. This operation is based on counting the t-tuples covered in every combination of t columns. The matrix is a covering array only if every submatrix of t distinct columns covers each t tuple of the alphabet.

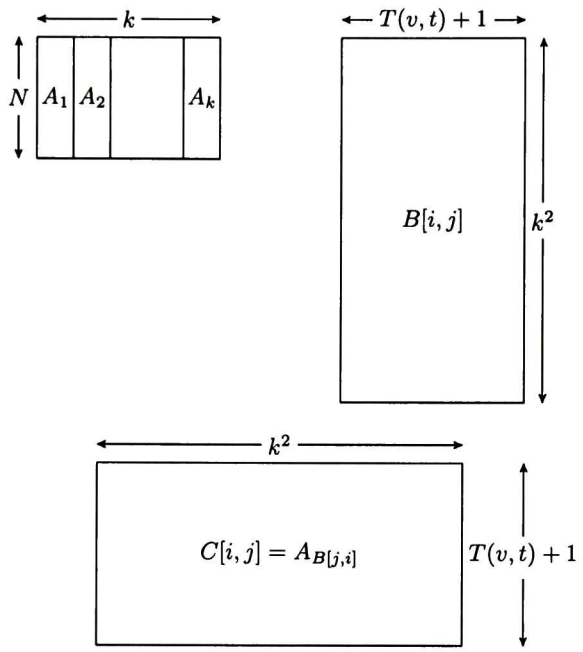


Figure 3.4: Construction of the array of blocks C from the columns of the covering array A and the indices of the orthogonal array B.

To describe in a general manner the process of verifying covering arrays let us describe how to verify mixed covering arrays. Remember that in a mixed covering array MCA$(N; t, k, v_0, v_1, \ldots, v_{k-1})$ every $N \times t$ submatrix conformed by the columns $c_0, c_1, \ldots, c_{t-1}$ covers at least once all the $\prod_{i=0}^{t-1} v_i$ tuples of the cartesian product of the sets with $v_{c_0}, v_{c_1}, \ldots, v_{c_{t-1}}$ symbols.

Avila-George [2] proposed a method to verify mixed covering arrays using a matrix called $\mathcal{P}$ of size $m \times n$, where $m = \binom{k}{t}$ and n is the cardinality of the cartesian product of the t larger alphabets. The matrix $\mathcal{P}$ is used to record how many times one t-tuple is covered in every combination of t columns of the MCA. Let be $C = \{c_0, c_1, \ldots, c_{t-1}\}$ a set of t columns and let be $S = (s_0, s_1, \ldots, s_{t-1})$ one tuple of the columns in C. For this tuple S it is associated an entry (i, j) of the matrix $\mathcal{P}$ which contains the number of times S is covered in C. The value of i and j are given by the following expressions, where J_j is defined recursively as $J_j = J_{j-1} \cdot v_{c_j} + s_j$ for $j \geq 1$ and $J_0 = s_0$:

$$i = \sum_{i=0}^{t-1} \binom{c_i}{i+1}, \qquad j = J_{t-1} \tag{3.4}$$

Next is given one example of the process to verify a MCA, the example was taken from [2]. Let be MCA$(6; 2, 4, 2^3 3^1)$ the MCA to be verified; the initial matrix $\mathcal{P}$ for this MCA is shown to the right in the Figure 3.5.

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

MCA$(6; 2, 4, 2^3 3^1)$

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Initial matrix $\mathcal{P}$

Figure 3.5: The initial matrix $\mathcal{P}$ for the verification of MCA$(6; 2, 4, 2^3 3^1)$ is a matrix of dimensions 6×6 initialized with zeroes.

The $m = 6$ rows of the matrix $\mathcal{P}$ correspond to the $\binom{k}{t} = \binom{4}{2} = 6$ combinations of $t = 2$ columns $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{1,3\}$, and $\{2,3\}$ of the MCA (although not in sequential order). The alphabets of the $k = 4$ columns of the MCA have respectively the cardinalities $2, 2, 2, 3$; so the number of columns of the matrix $\mathcal{P}$ is $n = 6$, since 3 and 2 are the cardinalities of the $t = 2$ larger alphabets.

Consider the combination of columns $\{0,1\}$. According to the expressions in (3.4) this combination of columns corresponds to the row $i = \binom{0}{1} + \binom{1}{2} = 0 + 0 = 0$ of the matrix $\mathcal{P}$. The number of t-tuples that must be covered in this combination of columns is $n' = 4$ because the cardinalities of the alphabets of these two columns is 2. The tuples that must be covered in the combination of columns $\{0,1\}$ are $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$, which are associated to the columns 0, 1, 2, and 3 of the row 0 of the matrix $\mathcal{P}$; for example, the tuple $(1,0)$ is associated to the column $j = (1)(2) + 0 = 2$. The tuples $(0,1)$ and $(1,0)$ are covered two times in the columns $\{0,1\}$, while the tuples $(0,0)$ and $(1,1)$ are covered once; so the content of row $i = 0$ of the matrix

$\mathcal{P}$ is as shown in the left part of the Figure 3.6. The shaded cells in the matrices of Figure 3.6 are the cells associated with a tuple that must be covered in the combination of columns associated with the particular row. The matrix under verification will be an MCA only if all the shaded cells have a value greater than zero.

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

Matrix $\mathcal{P}$ after processing the columns $\{0, 1\}$

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

Final matrix $\mathcal{P}$

Figure 3.6: To the left is the matrix $\mathcal{P}$ after processing the combination of columns $\{0, 1\}$ of MCA$(6; 2, 4, 2^3 3^1)$. To the right is the final matrix $\mathcal{P}$. The shaded cells are the cells associated with a tuple that must be covered.

The $\binom{k}{t}$ subsets of t columns can verified in parallel to reduce the time consumed by the verification process. In [2] and [5] are proposed parallel algorithms to verify covering arrays; these algorithms are based on dividing equitably the subset of t columns among the available processors. In [4] is proposed the verification of covering arrays using grid computing.

3.5.2 Maximization of Constant Rows

A constant row in a covering array is a row having the same symbol in all its elements. Formally, the i-th row of a covering array $A = (a_{i,j})$ of dimensions $N \times k$ is constant if $a_{i,j} = a_{i,0}$ for $j = 1, 2, \ldots, k-1$ [58].

By means of the three operations that produce isomorphic covering arrays it is possible to arrange the symbols of the covering array in order to make constant some of its rows. If the covering array has order v then the number of rows that can be made constant (without counting repeated rows) is at least 1 and at most v. The lower bound is 1 because it is always possible to make constant one row of the covering array by means of the operation of column relabeling. For example the Figure

3.7 shows the covering array CA(5; 2, 4, 2) in which the row 0 is made constant by relabeling the columns having the symbol 1 in the first row (columns 0 and 2).

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

CA(5; 2, 4, 2)

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

CA(5; 2, 4, 2) with one constant row

Figure 3.7: To the left is the covering array CA(5; 2, 4, 2), and to the right is one covering array isomorphic to it, in which the first row was made constant by relabeling the columns 0 and 2 of the covering array to the left.

The constant rows are very useful for the methods of multiplication and power of covering arrays, because if the covering arrays used have constant rows, then it is possible to delete some rows in the resulting covering array. However, to get more than one constant row in a covering array is not an easy task.

Quiz-Ramos in [58] developed four algorithms to maximize the number of constant rows in a covering array. Three of them try to maximize the constant rows by means of permutation of rows, permutation of columns, and permutation of symbols in the columns. But the fourth algorithm translates the problem to the graph domain.

For the input covering array CA($N; t, k, v$) it is generated a graph with N nodes, and there is an edge from node a to node b only if the rows a y b of the covering array have distinct symbols in every one of the k columns, that is $a_j \neq b_j$ for $j = 0, 1, \ldots, k-1$. Once the graph has been created, the problem to maximize the number of constant rows is transformed to the problem of finding the largest complete subgraph, i.e. the maximum clique problem. The rows of the covering array represented by the nodes that conform the maximum clique have different elements in each one of the k columns; so, these rows can be made constant by applying the operation of column relabeling. For example, suppose that for a certain covering array of $k = 7$ columns and order $v = 3$ the maximum clique is conformed by the nodes that represent the following rows:

$$a = (0\ 0\ 1\ 2\ 1\ 0\ 0)$$
$$b = (2\ 1\ 0\ 1\ 0\ 2\ 1)$$
$$c = (1\ 2\ 2\ 0\ 2\ 1\ 2)$$

It can be observed that in the seven columns all the symbols are different, in fact every column is a permutation of the symbols 0, 1, and 2. Therefore, it is possible to obtain the following three constant rows by permuting the symbols in the columns:

$$a = (0\ 0\ 0\ 0\ 0\ 0\ 0)$$
$$b = (1\ 1\ 1\ 1\ 1\ 1\ 1)$$
$$c = (2\ 2\ 2\ 2\ 2\ 2\ 2)$$

3.5.3 Optimal Shortening of Covering Arrays

Given a covering array A the Optimal Shortening of Covering ARrays (OSCAR) problem consists in finding a submatrix B of a determined size such that the number of missing tuples in B is minimized [13]. Let be A a matrix that may be or not a covering array CA$(N; t, k, v)$, and let be δ and Δ two integers such that $0 \leq \delta \leq N - v^t$, $0 \leq \Delta \leq k - t$, with the condition that at least one of them is greater than zero. The OSCAR problem consists in finding a submatrix B of A of size $(N - \delta) \times (k - \Delta)$ such that the number of missing tuples in B is minimal. In [13] Carrizales-Turrubiates proved that the OSCAR problem is NP-Complete by reducing the MAXCOVER problem to the OSCAR problem. The search space of the OSCAR problem is $\binom{N}{N-\delta}\binom{k}{k-\Delta}$.

The values δ y Δ are respectively the number of rows and the number of columns to eliminate from the input covering array. The domain of δ is $\{0, 1, \ldots, N - v^t\}$ because v^t is the lower bound for the number of rows of a covering array of strength t and order v. The domain of Δ is $\{0, 1, \ldots, k - t\}$ because the final matrix must have at least t columns in order to be a covering array of strength t. The problem is finding which δ rows and Δ columns to delete in order to the resulting submatrix has the minimum number of missing tuples.

As an example of the OSCAR problem let be $A = \text{CA}(6; 2, 7, 2)$, $\delta = 1$, and $\Delta = 3$ (see Figure 3.8). So the final matrix will have $N - \delta = 6 - 1 = 5$ rows and $k - \Delta = 7 - 3 = 4$ columns, and it is desired that this matrix has the minimum number of missing tuples. In this case there is an optimal solution: the elimination of the row 2 and of the columns 2, 4, 6 produces the matrix shown to the right of the Figure 3.8, which is a covering array $\text{CA}(5; 2, 4, 2)$.

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

$\text{CA}(6; 2, 7, 2)$

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

Row and columns to delete

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

$\text{CA}(5; 2, 4, 2)$

Figure 3.8: The deletion of row 2 and columns 2, 4, 6 of the covering array $\text{CA}(6; 2, 7, 2)$ produces the covering array $\text{CA}(5; 2, 4, 2)$.

In [13] are proposed twelve algorithms to solve the OSCAR problem, these algorithms select the rows and the columns to eliminate following exact, greedy, or metaheuristic strategies.

3.5.4 Wildcard Detection

Sometimes a covering array has entries that can be freely modified without affecting the coverage properties of the covering array, that is, without affecting the number of missing tuples of the array. These entries are called *wildcards* and are commonly represented by the symbol $*$. Figure 3.9 shows at the left the covering array $\text{CA}(7; 2, 8, 2)$, and shows at right the same covering array with the wildcards it contains.

Wildcard detection is very important for some postoptimization process for covering arrays. A postoptimization process is a process that tries to reduce the number of rows of a given covering array. One of these process that uses wildcards is the method of Nayeri et al. [56].

In [34] Gonzalez-Hernandez et al. introduced a methodology to maximize the number of wildcards in a covering array. This methodology is conformed by three main steps: a) to determine the tuples

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

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

Figure 3.9: Wildcards in the covering array CA$(7; 2, 8, 2)$.

covered only once, b) to determine the unfixed symbols, and c) to enumerate all the possible wildcard configurations.

In the first step are determined the tuples which are covered only once; for example, in the covering array of Figure 3.9 the tuples $(0, 1)$ and $(1, 0)$ appears only once in the combination of columns $\{0, 1\}$. The entries of the covering array associated with tuples covered only once in a submatrix do not have chance of being wildcards, and so they are the fixed symbols of the covering array. On the other hand, the entries of the covering array associated with tuples that are covered more than once in a submatrix are the unfixed symbols of the covering array (second step). For the covering array of Figure 3.9, the Figure 3.10 shows the entries associated with tuples covered only once, and the entries (marked with U) associated with tuples covered more than once.

0	U	U	U	1	U	U	0
0	1	0	0	U	0	1	U
1	1	1	1	0	0	1	0
0	0	1	U	U	1	1	U
1	1	0	1	1	1	0	1
1	0	0	0	0	1	U	0
0	0	U	1	0	U	U	1

Figure 3.10: The entries of the covering array CA$(7; 2, 8, 2)$ marked with U are the entries that may become a wildcard.

The third step consists in an algorithm of branch and bound which maximizes the number of wildcards in the covering array. The algorithm starts by identifying those tuples that are covered more than once and all its occurrences are conformed by at least one entry marked with U. For

example, the tuple $(0,0)$ is covered in rows 0 and 6 of the submatrix conformed by the columns $\{0,6\}$; in both rows the occurrence of the tuple is conformed by at least one entry marked with U. Another example is the tuple $(0,1)$ which occurs in the rows 0, 1, and 3 of the submatrix conformed by the columns $\{3,4\}$, and every of these occurrences has at least one entry marked with U.

In order to the covering array can retain its coverage properties, the algorithm fixes one occurrence of the tuples that are covered more than once and which have at least one cell marked with U. For example, there are two occurrences of the tuple $(0,0)$ that have at least one entry marked with U in the columns $\{0,6\}$; these occurrences are in rows 0 and 6. Similarly for the tuple $(0,1)$ in the columns $\{3,4\}$ one of the occurrences in the rows 0, 1, and 3 must be fixed. After fixing one occurrence of the tuple, the remaining unfixed symbols in the other occurrences become wildcards. This way, to maximize the number of wildcards the algorithm checks all possible configurations.

3.5.5 Fusion Operator

In [23] Colbourn established the following bounds:

1. $\mathsf{CAN}(2,k,v-1) \leq \mathsf{CAN}(2,k,v) - 2$
2. For v a prime power $\mathsf{CAN}(2,v+1,v-1) \leq v^2 - 3$

And after in [26] Colbourn et al. grouped these bounds under the name of *fusion*:

$$\mathsf{CAN}(t,k,v) \leq \mathsf{CAN}(t,k,v+1) - \begin{cases} 3 \text{ if } t=2,\ k \leq v+1,\ v \text{ is a prime power} \\ 2 \text{ otherwise} \end{cases} \tag{3.5}$$

The basic mechanism of the fusion operator is to obtain from a covering array $A = \mathsf{CA}(N;t,k,v)$ another covering array $B = \mathsf{CA}(M;t,k,v-1)$ of smaller size by replacing the occurrences of the symbol v in A for symbols of the set $\{0,1,\ldots,v-2\}$, and by deleting three or two rows according the cases of expression (3.5).

For the first case let be v a prime power and let be A the orthogonal array OA$(v^2; 2, v+1, v)$. It is always possible to create a constant row in A with the symbol $v-1$, by permuting the symbols in each column of A. This constant row is deleted and the remaining occurrences of the symbol $v-1$ are replaced by symbols in $\{0, 1, \ldots, v-2\}$ (this is the covering array B). Since A is an OA, every 2-tuple is covered exactly one time in B, except the tuple $(v-1, v-1)$. By means of the procedure described in [23] it is possible to delete another two rows of the covering array B.

In the second case one row can be deleted by making a constant row with the symbol $v-1$. To eliminate the second row, a row r is selected from the remaining $N-1$ rows and each entry (r', c), with $r' \neq r$, equal to $v-1$ is replaced by the content of the entry (r, c). These replacements ensure that each tuple $((r, c_1), (r, c_2))$ with $(r, c_1) \neq v-1$ and $(r, c_2) \neq v-1$ of the deleted row is covered in other row $r' \neq r$.

In [60] Rodriguez-Cristerna generalized the fusion operator to MCAs constructed using greedy methods. These MCAs have a lot of redundancy in the coverage of interactions, which means that they have several wildcards. The approach followed was to combine the fusion operator with the elimination of redundant interactions based on wildcard detection; this way, from a MCA$(N; t, k, (v_0, v_1, \ldots, v_{k-1}))$ is constructed a MCA$(M; t, k, (v'_0, v'_1, \ldots, v'_{k-1}))$ such that $v_i \leq v'_i$ for $0 \leq i < k$.

3.6 Chapter Summary

In this chapter were analyzed some of the relevant methods developed to construct covering arrays. The methods studied were classified in four categories: exact, greedy, metaheuristic, and algebraic. The exact methods produce optimal covering arrays, but they are practical only to generate covering arrays of moderate size given that all the search space is explored. The greedy methods do not guarantee a covering array with the minimum number of rows; their advantage is that they can generate covering arrays of greater size and in much less time than the exact methods. Like the

greedy methods, the metaheuristic methods do not guarantee to find optimal covering array, but most of the times they provide very good solutions. Finally, the algebraic methods have the characteristic that in the process of constructing a covering array are used formulas and operations with other mathematical objects; this class of methods do not perform a searching process to construct the covering array, rather the search (if exists) is devoted to generate the data or mathematical objects with which the final covering array is constructed. At the end of the chapter five useful operations for covering arrays were reviewed.

Among the methods studied, the ones with greater similarity to the method proposed in this thesis are the Roux-type constructions (Section 3.4.7), the product of covering arrays (Section 3.4.8), and the power of a covering array (Section 3.4.9). In the TCA approach the covering arrays are constructed by translating the columns of the previous covering array in the TCA, and the above three methods use smaller covering arrays to produce the final one. The next chapter shows the way in which the columns of a covering array of strength t are used to tray to construct a covering array of strength $t + 1$.

4

Methodology to Construct the Towers of Covering Arrays

This chapter describes the methodology proposed to construct the towers of covering arrays. The methodology proposed was devised with the objective that the TCAs are conformed by competitive quality covering arrays. Section 4.1 presents the general vision of the methodology, and explains why to search for TCAs of maximum height. Section 4.2 analyzes the construction $\mathcal{E}$ introduced in Chapter 1, and shows how this construction is used to generate the TCAs. Section 4.3 studies the problem of generating all the non-isomorphic covering arrays of minimum rank for a particular combination of the parameters N, t, k, v, in order to use these covering arrays as the bases of the TCAs.

4.1 Overview of the Methodology

This section presents an overview of the methodology proposed to generate competitive quality covering arrays through the TCA approach. Given a combination of the parameters N, t, k, and v, the methodology to construct the TCAs consists in the following three main steps:

1. To generate all the non-isomorphic covering arrays of minimum rank CA$(N; t, k, v)$ with the objective of using them as the bases of the TCAs.

2. To apply iteratively the construction $\mathcal{E}$ to the non-isomorphic covering arrays of minimum rank.

3. To deliver the TCA with the greatest height among all the constructed.

Figure 4.1 shows graphically the methodology proposed to solve the research problem. The covering arrays of strength t at the base of the TCAs are the non-isomorphic covering arrays of minimum rank for the given parameters N, t, k, and v. The bases are expanded iteratively by the construction $\mathcal{E}$. For each covering array at floor $j \geq 0$ of the TCA all the possible matrix in the following strength are generated in order to see which of them are covering arrays; these covering arrays will conform the floor $j+1$ of the TCA. After that, the construction $\mathcal{E}$ is applied to all the covering arrays at floor $j+1$ in order to extend the TCA to the floor $j+2$. At the end of the process, the TCA with the greatest height among all the constructed is reported as the TCA of maximum height for the parameters N, t, k, and v.

In addition, Figure 4.2 shows a high level flow diagram of the proposed methodology. The function *nextNonIsoCA*() computes and returns the following non-isomorphic covering for the input parameters N, t, k, and v. If the function succeeded it returns a covering array A; and when all the non-isomorphic covering arrays have been computed the function returns *Null*. Every non-isomorphic base is expanded iteratively by applying to it the construction $\mathcal{E}$; the function *apply$\mathcal{E}$Iteratively*() encapsulates this operation. If the TCA T returned by the function *apply$\mathcal{E}$Iteratively*() is higher

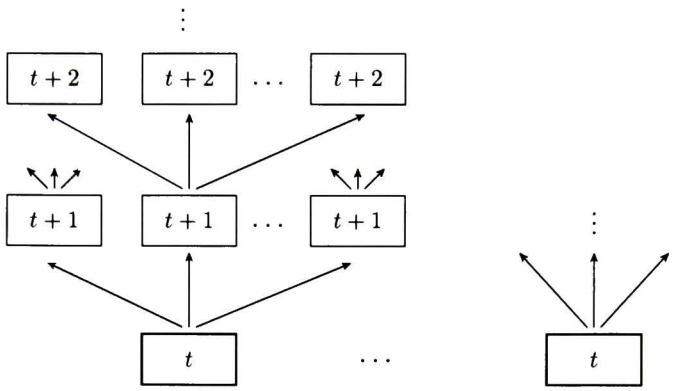


Figure 4.1: Methodology to construct the TCAs. The covering arrays at the base of the TCAs are non-isomorphic among them.

than the current best TCA T^*, then T becomes the new current best TCA; the function $h()$ is used to get the height of a TCA. Finally, the function *write*() writes the highest TCA found.

The motivation to construct TCAs of maximum height is because for TCAs of height $h > 0$ the ratio between the number of rows of the last covering array in the TCA and the number of rows of the best known covering array for the same values of t, k, and v decreases as h grows.

In [22] Colbourn established the following bound:

$$CAN(t-1, k-1, v) \leq \frac{1}{v} CAN(t, k, v). \tag{4.1}$$

Let be $A = \mathrm{CA}(N; t, k, v)$ a covering array of strength t and order v, let be j any column of A, and let be $x \in \{0, 1, \ldots, v-1\}$ any symbol. Then the $N' \times (k-1)$ subarray obtained by deleting column j from A and keeping only those rows of A that have symbol x in column j is a covering array $B = \mathrm{CA}(N'; t-1, k-1, v)$, where N' is the number of occurrences of x in column j [22].

Since A is a covering array of strength t, each symbol x of column j occurs at least once in conjunction with each $(t-1)$-tuple of the set $\mathbb{Z}_v^{t-1}$; so B is a covering array of strength $t-1$. Now, consider that symbol x is the symbol with less number of occurrences in column j, then the covering array B will have at most $N' = N/v$ rows. Therefore $CAN(t-1, k-1, v) \leq CAN(t, k, v)/v$.

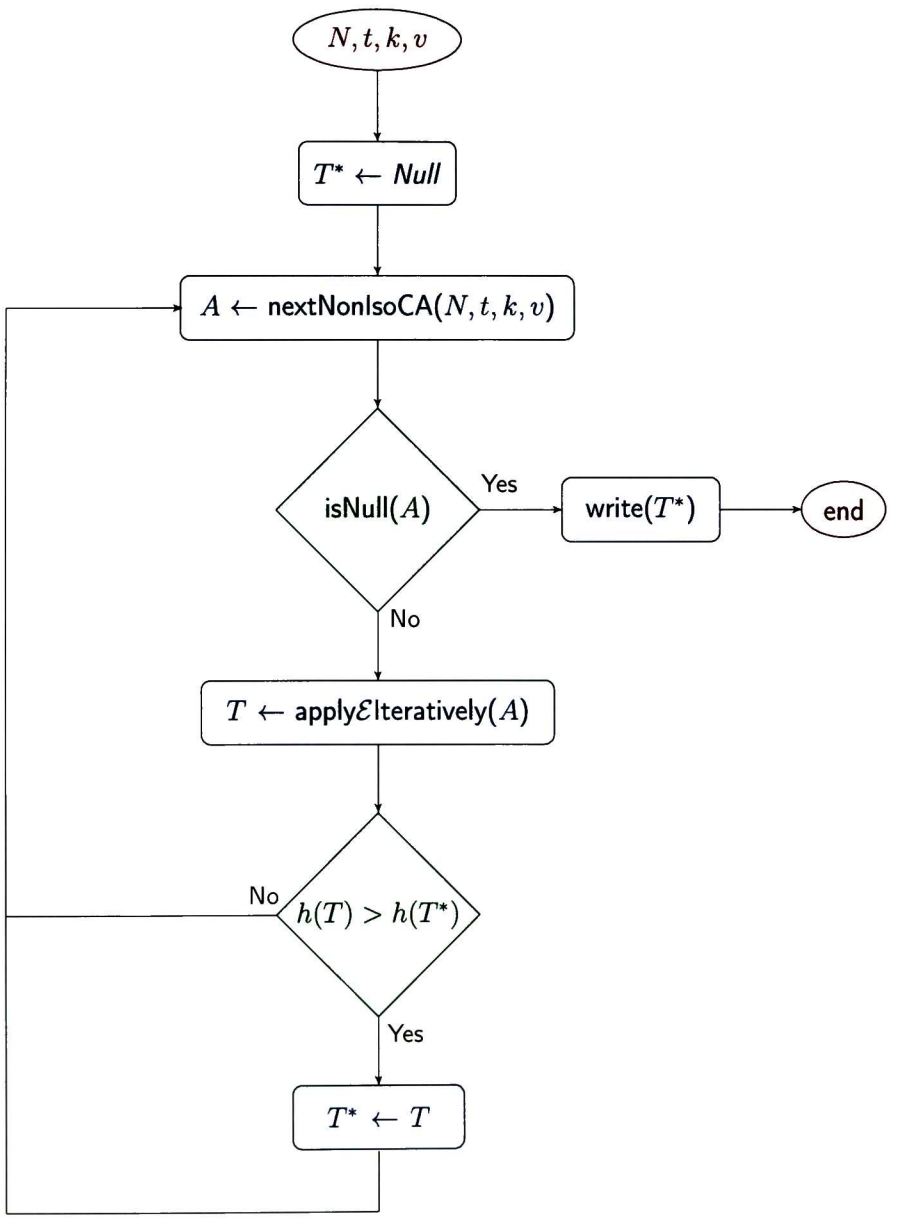


Figure 4.2: Flow diagram of the proposed methodology.

The Inequality 4.1 can be rewritten as follows:

$$CAN(t,k,v) \geq v \cdot CAN(t-1,k-1,v). \tag{4.2}$$

Inequality (4.2) says that the number of rows of the best covering array of strength t, k columns, and order v, is *at least* v times the number of rows of the best covering array of strength $t-1$, $k-1$ columns, and order v. In other words, if the covering array CA$(N; t-1, k-1, v)$ is optimum, then if exists the covering array CA$(M; t, k, v)$ has at least $M = Nv$ rows.

Figure 4.3 compares the covering arrays produced by the TCA approach with the best known covering arrays for the same parameters t, k, and v. In the TCA approach each covering array, other than the base, has exactly v times the number of rows of the previous covering array in the TCA; but in the best known covering arrays it may happen that $M_i > v\, M_{i-1}$ for some i between 1 and h in the Figure 4.3.

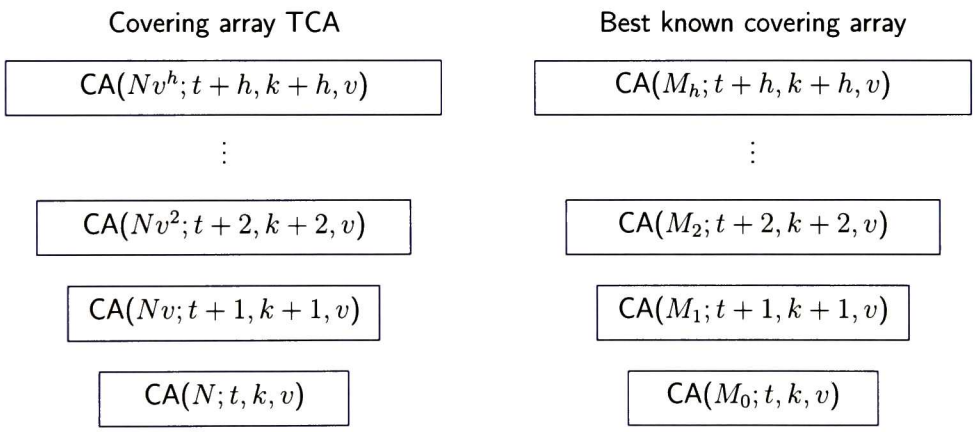


Figure 4.3: To the left are the covering arrays obtained with the TCA approach, and to the right are the best known covering arrays for the same parameters t, k, and v.

From the covering arrays in Figure 4.3 we have that the ratio between the number of rows of the covering arrays obtained with the TCA approach and the number of rows of the best known covering arrays satisfies the following inequalities:

$$\frac{N}{M_0} \geq \frac{Nv}{M_1} \geq \frac{Nv^2}{M_2} \geq \cdots \geq \frac{Nv^h}{M_h} \tag{4.3}$$

Therefore, as the height of a TCA increases the covering arrays obtained are more competitive with the best known covering arrays for the same parameters t, k, and v.

In the following two sections the first two points of the proposed methodology are further explained, but they are presented in inverted order: the construction $\mathcal{E}$ is presented in Section 4.2 and the generation of the non-isomorphic bases of minimum rank is presented in Section 4.3.

4.2 The Construction $\mathcal{E}$

This section further analyzes the construction $\mathcal{E}$ used to generate the TCAs. Section 4.2.1 explains why the construction $\mathcal{E}$ can generate (in some cases) a covering array of strength $t+1$ from a covering array of strength t; Section 4.2.2 shows an efficient way to apply the construction $\mathcal{E}$ to all matrices δ for the base covering array; Section 4.2.3 contains the pseudocode of the algorithms sketched in Section 4.2.2; and Section 4.2.4 describes the way in which the construction $\mathcal{E}$ is applied iteratively.

4.2.1 Strategy of the Construction $\mathcal{E}$

The generation of the towers of covering arrays is done by means of the construction $\mathcal{E}$. This construction can sometimes generate a covering array of strength $t+1$ from a covering array of strength t. The construction $\mathcal{E}$ was defined in Chapter 1, but we repeat its definition here:

DEFINITION 8 *Let be $A_0, A_1, \ldots, A_{k-1}$ the k columns of a covering array $A = CA(N; t, k, v)$; let be δ a matrix of dimensions $v \times k$ such that $\delta_{0,j} = 0$ and $\delta_{i,j} \in \{0, 1, \ldots, v-1\}$ for $1 \leq i \leq v-1$, $0 \leq j \leq k-1$; and let be $X_0, X_1, \ldots, X_{v-1}$ the v columns of the matrix $X = (x_{i,j})$ of dimensions $N \times v$ such that $x_{i,j} = j$ for $0 \leq i \leq N-1$, $0 \leq j \leq v-1$. The construction $\mathcal{E}$ to try to expand*

the base covering array A of strength t to a covering array B of strength $t+1$ consists in creating a matrix B of size $Nv \times (k+1)$ composed by v blocks of size $N \times (k+1)$ juxtaposed vertically. The j-th column of the i-th block $(0 \leq j \leq k-1,\ 0 \leq i \leq v-1)$ is the column j of the base covering array translated by the entry (i,j) of the matrix δ; the last column of the i-th block is the column X_i. Figure 4.4 shows the definition of the construction $\mathcal{E}$ in schematic form.

$$A = \begin{pmatrix} A_0 & A_1 & \dots & A_{k-1} \end{pmatrix} \qquad \delta = \begin{pmatrix} 0 & 0 & \dots & 0 \\ \delta_{1,0} & \delta_{1,1} & \dots & \delta_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{v-1,0} & \delta_{v-1,1} & \dots & \delta_{v-1,k-1} \end{pmatrix}$$

$$X = \begin{pmatrix} 0 & 1 & \dots & v-1 \\ 0 & 1 & \dots & v-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & v-1 \end{pmatrix}$$

$$B = \begin{pmatrix} A_0 \oplus \delta_{0,0} & A_1 \oplus \delta_{0,1} & \dots & A_{k-1} \oplus \delta_{0,k-1} & X_0 \\ A_0 \oplus \delta_{1,0} & A_1 \oplus \delta_{1,1} & \dots & A_{k-1} \oplus \delta_{1,k-1} & X_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_0 \oplus \delta_{v-1,0} & A_1 \oplus \delta_{v-1,1} & \dots & A_{k-1} \oplus \delta_{v-1,k-1} & X_{v-1} \end{pmatrix}$$

Figure 4.4: The construction $\mathcal{E}$. The first k columns of the i-th block of the matrix B are the columns of the base covering array A translated by the values at row i of matrix δ, and the last column of the block is the column X_i.

To appreciate how the construction $\mathcal{E}$ can produce a covering array of strength $t+1$ from a covering array of strength t, consider the covering array $A = \mathrm{CA}(4;2,3,2)$ and the matrix δ of Figure 4.5. The application of the construction $\mathcal{E}$ to this base covering array A and to this matrix δ produces the covering array $B = \mathrm{CA}(8;3,4,2)$ of strength three shown in the same Figure 4.5.

Let us explain the main idea of the construction $\mathcal{E}$, starting with the following Theorem 5.

THEOREM **5** *Let be A_j a column of a covering array of N rows and order v, then every translation of A_j is equivalent to one permutation of symbols in A_j.*

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \qquad \delta = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad X = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \qquad B = \left(\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{array}\right)$$

Figure 4.5: The construction $\mathcal{E}$ applied to the base covering array $A = \mathrm{CA}(4; 2, 3, 2)$ and to the matrix δ of the figure, produces the covering array of strength three $\mathrm{CA}(8; 3, 4, 2)$.

Proof: Let be $c \in \mathbb{Z}_v$ and let be $A'_j = A_j \oplus c$. Moreover, let be $a'_{i,j}$ the i-th element of A'_j and let be $a_{i,j}$ the i-th element of A_j, where $0 \leq i \leq N-1$. From the definition of the operation of column translation (Definition 2, Chapter 1) we have $a'_{i,j} = (a_{i,j} + c) \bmod v$ for $i = 0, 1, \ldots, N-1$.

Let be $a_{l_1,j}, a_{l_2,j} \in A_j$ such that $a_{l_1,j} = a_{l_2,j}$. Then,

$$a'_{l_1,j} = (a_{l_1,j} + c) \bmod v = (a_{l_2,j} + c) \bmod v = a'_{l_2,j}.$$

So, the operation of column translation maps equal elements to the same value. On the other hand, any two equal elements of A'_j are the image of equal elements in A_j: let be $a'_{l_1,j}, a'_{l_2,j} \in A'_j$ such that $a'_{l_1,j} = a'_{l_2,j}$, then $(a_{l_1,j} + c) \bmod v = (a_{l_2,j} + c) \bmod v$. Since $0 \leq a_{l_1,j}, a_{l_2,j}, c \leq v-1$ we have $a_{l_1,j} = a_{l_2,j}$. Therefore, the translation of A_j by $c \in \mathbb{Z}_v$ is equivalent to the following permutation of the symbols in A_j:

$$\begin{pmatrix} 0 & 1 & \ldots & v-1 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ (0+c) \bmod v & (1+c) \bmod v & \ldots & ((v-1)+c) \bmod v \end{pmatrix}.$$

□

It follows from Theorem 5 that the set of columns derived from the translations of a column A_j is a subset of the set of columns derived from the permutation of symbols in A_j. There are v possible operations of column translation, resulting from the translation of A_j by the values $0, 1, \ldots, v-1$; and there are $v!$ possible permutations of symbols in A_j. So, the cardinality of set of columns derived by permuting the symbols in A_j is greater than or equal to the cardinality of the set of columns derived by translating A_j; if fact, the cardinality of these sets is equal only when $v = 2$.

THEOREM **6** *Let be A a covering array CA$(N; t, k, v)$, let be $c_0, c_1, \ldots, c_{k-1} \in \mathbb{Z}_v$, and let be B the matrix derived from A by translating the column A_j of A by the value c_j for $j = 0, 1, \ldots, k-1$. Then, the matrix B is a covering array isomorphic to A.*

Proof: From Theorem 5 each operation of column translation is equivalent to one symbol permutation. Let be $\phi_j \in \{0, 1, \ldots, v! - 1\}$ the index, in lexicographic order, of the symbol permutation equivalent to the translation by c_j. This way, the matrix B can be derived from A by applying to A the permutation of rows $\tau = (0\ 1\ \cdots\ N-1)$, the permutation of columns $\pi = (0\ 1\ \cdots\ k-1)$, and the combination of symbol permutations $\phi = (\phi_0\ \phi_1\ \cdots\ \phi_{k-1})$. So, the matrix B is a covering array isomorphic to A. □

Given a base covering array $A =$ CA$(N; t, k, v)$ the first k columns of the matrix B produced by the construction $\mathcal{T}$ are formed by juxtaposing vertically v covering arrays $\overline{A^0}, \overline{A^1}, \ldots, \overline{A^{v-1}}$ isomorphic to A, where each covering array $\overline{A^i}$ is derived from A by translating the columns of A by the values at row i of the matrix δ. The Figure 4.6 shows this structure of the matrix B, the numbers 0, 1, ..., $v-1$ in boldface represent a constant column of N elements equal to 0, 1, ..., $v-1$ respectively.

The fact that the first k columns of each block of the matrix B are a covering array isomorphic to A is very important for the strategy of the construction $\mathcal{E}$, because it ensures that a number of combinations of $t+1$ columns satisfy the requirements to be part of a covering array of strength $t+1$, as formalized in Theorem 7.

$$B = \begin{pmatrix} \overline{A^0} & \mathbf{0} \\ \overline{A^1} & \mathbf{1} \\ \vdots & \vdots \\ \overline{A^{v-1}} & \mathbf{v\text{-}1} \end{pmatrix}$$

Figure 4.6: Structure of the matrix B produced by the construction $\mathcal{E}$.

THEOREM **7** *In the matrix B produced by the construction $\mathcal{E}$ for a given base covering array $A = CA(N; t, k, v)$ and a given matrix δ, every combination of $t+1$ columns conformed by t columns from the first k columns of B and the last column of B covers each tuple of the set $\mathbb{Z}_v^{t+1}$.*

Proof: By Theorem 6 we can partition the first k columns of matrix B in v covering arrays of strength t $\overline{A^0}, \overline{A^1}, \ldots, \overline{A^{v-1}}$ as shown in Figure 4.6. Every of these covering arrays covers at least once each tuple $x = (x_0, x_1, \ldots, x_{t-1}) \in \mathbb{Z}_v^t$. For $i = 0, 1, \ldots, v-1$ let be C^i the matrix conformed by appending to the covering array $\overline{A^i}$ the constant column of N elements equal to i. This way, every submatrix of $t+1$ columns of C^0 conformed by t columns from the first k columns of C^0 and the last column of C^0 covers each tuple $(x_0, x_1, \ldots, x_{t-1}, 0) \in \mathbb{Z}_v^{t+1}$. Similarly, every submatrix of $t+1$ columns of C^1 conformed by t from the first k columns of C^1 and the last column of C^1 covers each tuple $(x_0, x_1, \ldots, x_{t-1}, 1) \in \mathbb{Z}_v^{t+1}$. The same applies for the matrices $C^2, \ldots, C^{v-1}$. Given that the matrix B is conformed by juxtaposing vertically the matrices $C^0, C^1, \ldots, C^{v-1}$ we have that every submatrix of $t+1$ columns of B conformed by t columns from the first k columns of B and the last column of B covers each tuple of the set $\mathbb{Z}_v^{t+1}$. □

To illustrate the Theorem 7 consider the base covering array $A = \mathrm{CA}(4; 2, 3, 2)$ and the matrix δ of Figure 4.7. For this base covering array and this matrix δ the construction $\mathcal{E}$ produces the matrix B to the right of the same Figure 4.7. In this case $\overline{A^0}$ is the covering array derived from A by translating the three columns of A by the values 0, 0, and 0 respectively; and $\overline{A^1}$ is the covering array derived from A by translating the three columns of A by the values 1, 0, and 1 respectively.

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \qquad \delta = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \qquad B = \left(\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right)$$

Figure 4.7: Example to illustrate the Theorem 7.

Since A is a covering array of strength $t = 2$, every combination of two distinct columns covers the four tuples of the set $\mathbb{Z}_2^2 = \{(0,0),(0,1),(1,0),(1,1)\}$. So, every two columns of $\overline{A^0}$ plus the constant column of zeroes covers the tuples $(0,0,0)$, $(0,1,0)$, $(1,0,0)$, and $(1,1,0)$. Similarly, every two columns of $\overline{A^1}$ plus the constant column of ones covers the tuples $(0,0,1)$, $(0,1,1)$, $(1,0,1)$, and $(1,1,1)$. Therefore, every submatrix of B conformed by two of the first three columns and the last column, covers each tuple of the set $\mathbb{Z}_2^3$. The three different submatrices of B conformed by two of the first three columns and the last column are shown in Figure 4.8.

$$\left(\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right) \qquad \left(\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right) \qquad \left(\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right)$$

Figure 4.8: The three submatrices of B conformed by two of the first three columns and by the last column. By Theorem 7 each of these submatrices covers each tuple of $\mathbb{Z}_2^3$ at least once.

The only submatrix conformed by three columns from the first three columns of matrix B is the one shown in Figure 4.8. Notice that this submatrix does not cover all the tuples of the set $\mathbb{Z}_2^3$. So, the translation of the columns of the base covering array of strength t are done with the objective of each submatrix of B conformed by $t+1$ columns from the first k columns can be a covering array

$$\left(\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right)$$

Figure 4.9: The only submatrix conformed by three columns from the first three columns of B.

The number of submatrices of B conformed by t columns from the first k columns and by the last column of B is $\binom{k}{t}$; by the Theorem 7 these submatrices satisfy the requirements to be part of a covering array of strength $t+1$. And the number of submatrices of B conformed by $t+1$ columns from the first k columns is $\binom{k}{t+1}$; for these submatrices the translation of the base covering array are required in order to see if one combination of translations of columns make that these matrices also satisfy the requirements to be part of a covering array of strength $t+1$.

4.2.2 Applying the Construction $\mathcal{E}$ to Every Matrix δ

In general, the matrix B resulting from the application of the construction $\mathcal{E}$ might be or not a covering array of strength $t+1$, this depends on the base covering array and the matrix δ used For a covering array CA$(N; t, k, v)$ there exist $v^{k(v-1)}$ different matrices δ (since all the elements of the first row are zero), but not all of them produce a covering array of strength $t+1$; in fact, for some covering arrays of strength t none matrix δ produces a covering array of strength $t+1$; but in other cases there are several matrix δ that produce a covering array of strength $t+1$. So, in order to construct TCAs with maximum height it will be necessary to check all matrices δ for a base covering array CA$(N; t, k, v)$, and to expand the matrices B for which the construction $\mathcal{E}$ produces a covering array of strength $t+1$. For example, for the covering array $A = \text{CA}(4; 2, 3, 2)$ of Figure 4.5 there are $v^{k(v-1)} = 2^{3(1)} = 8$ different matrices δ, which are listed next, and the construction $\mathcal{E}$ produces a covering array of strength three for the matrices δ^1, δ^2, δ^4, and δ^7.

$$\delta^0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}; \quad \delta^1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}; \quad \delta^2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}; \quad \delta^3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix};$$

$$\delta^4 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}; \quad \delta^5 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}; \quad \delta^6 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}; \quad \delta^7 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}.$$

Figure 4.10 shows the flow diagram of the application of the construction $\mathcal{E}$ to every matrix δ that exist for a base covering array $A = \mathsf{CA}(N; t, k, v)$. The matrices δ are generated one at a time by means of the function *nextMatrixDelta*() and when all of them have been generated this function returns *Null*. The function $\mathcal{E}()$ applies the construction $\mathcal{E}$ to the base covering array A and to the current matrix δ. If the resulting matrix B is a covering array then the function *process*() process it. As we will see later the processing of the covering array B is the application of the construction $\mathcal{E}$ to every matrix δ that exist for the new base covering array B.

Now, we analyze the computational cost of applying the construct $\mathcal{E}$ to all matrices δ for a base covering array $A = \mathsf{CA}(N; t, k, v)$. One application of the construction $\mathcal{E}$ includes to create the matrix B and to verify if the matrix B is a covering array of strength $t + 1$. The time required to create the matrix B is linear in the size of the matrix B, i.e., $O(Nv \times (k + 1))$, assuming that the operation of column translation requires a linear time. The verification of the matrix B consists in checking that every submatrix of $t + 1$ columns from the first k columns of B covers all the tuples of the set $\mathbb{Z}_v^{t+1}$; remember that by Theorem 7 submatrices involving the last column of B do not need to be checked. Each submatrix has dimensions $(Nv) \times (t + 1)$ and there are $\binom{k}{t+1}$ of such submatrices. Because of the submatrix verification is done in linear time the computational cost of verify the matrix B is the one given in (4.4).

$$O\left(\binom{k}{t+1} Nv(t+1)\right) \tag{4.4}$$

Then, the cost of creating and verifying the matrix B is given by (4.5).

$$O\left(Nv(k+1) + \binom{k}{t+1} Nv(t+1)\right) \tag{4.5}$$

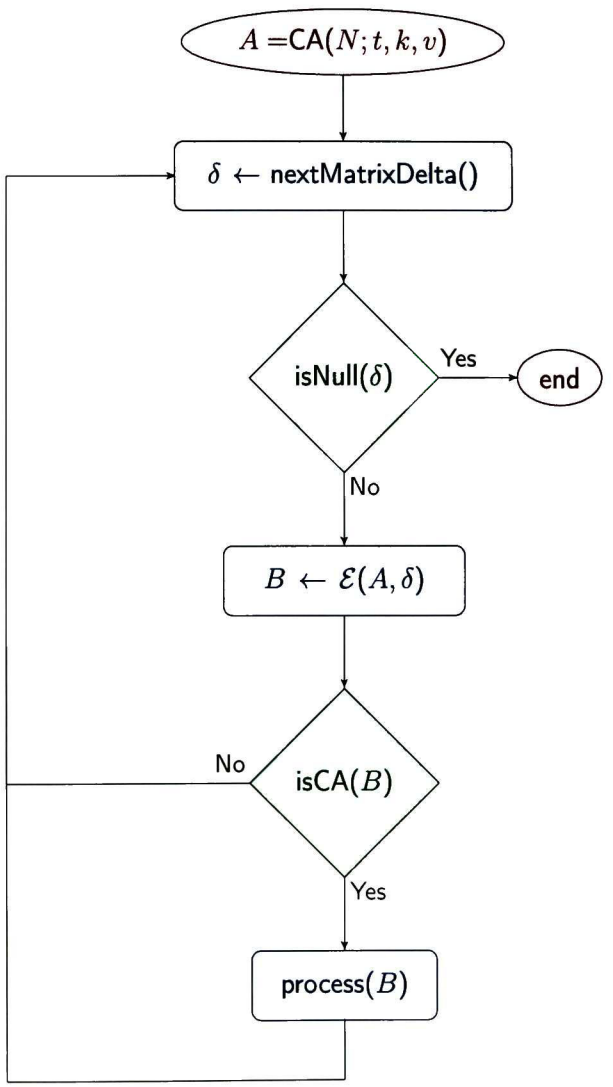


Figure 4.10: Flow diagram of the application of the construction $\mathcal{E}$ to every matrix δ that exist for a base covering array $A = \mathsf{CA}(N; t, k, v)$.

And therefore, the application of the construction $\mathcal{E}$ to all the $v^{k(v-1)}$ different matrices δ requires the time given in the expression (4.6).

$$O\left(v^{k(v-1)}\left[Nv(k+1)+\binom{k}{t+1}Nv(t+1)\right]\right) \tag{4.6}$$

To improve this execution time, the approach followed was to represent the matrices δ as linear vectors concatenating the rows of the matrices δ in order, but eliminating the first row of the matrices δ because all its elements are zero. Moreover, vectors δ are not generated sequentially, but rather they are generated in v-ary Gray code using the algorithm proposed in [36], where v is the order of the base covering array. For example, the eight matrices δ for the base covering array $A = \mathrm{CA}(4;2,3,2)$ of Figure 4.5 are represented by the following vectors δ, listed in the order they are generated in v-ary Gray code:

$$\delta^0 = (0\ 0\ 0),\ \delta^1 = (1\ 0\ 0),\ \delta^2 = (1\ 1\ 0),\ \delta^3 = (0\ 1\ 0),$$

$$\delta^4 = (0\ 1\ 1),\ \delta^5 = (1\ 1\ 1),\ \delta^6 = (1\ 0\ 1),\ \delta^7 = (0\ 0\ 1).$$

Generating the vectors δ in v-ary Gray code has the following advantages:

1. The application of the construction $\mathcal{E}$ to all vectors δ is accelerated notably, since the matrix B for a vector δ is different from the matrix B for the previous vector δ in only one column.

2. In some circumstances the verification of the matrix B can be omitted, since there is the assurance that the matrix B can not be a covering array of strength $t+1$ based on the verification of a previous matrix B.

Let be $A = \mathrm{CA}(N;t,k,v)$ the base covering array and let be $\delta^i = (\delta_0\ \delta_1\ \cdots\ \delta_{k(v-1)-1})$ the current vector δ. The matrix B for this vector, denoted by B^{δ^i}, is this one:

$$B^{\delta^i} = \begin{pmatrix} A_0 & A_1 & \dots & A_{k-1} & X_0 \\ A_0 \oplus \delta_0 & A_1 \oplus \delta_1 & \dots & A_{k-1} \oplus \delta_{k-1} & X_1 \\ A_0 \oplus \delta_k & A_1 \oplus \delta_{k+1} & \dots & A_{k-1} \oplus \delta_{2k-1} & X_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_0 \oplus \delta_{(v-2)k} & A_1 \oplus \delta_{(v-2)k+1} & \dots & A_{k-1} \oplus \delta_{(v-1)k-1} & X_{v-1} \end{pmatrix}$$

The following vector δ, denoted as δ^{i+1}, is equal to δ^i except in one position j, $0 \leq j \leq k(v-1)-1$. So, the matrix $B^{\delta^{i+1}}$ can be obtained from the matrix B^{δ^i} by replacing only one entry of the matrix B^{δ^i}. If j is the position of vector δ^{i+1} which has changed with respect to the vector δ^i, then the entry (x, y) of the matrix B^{δ^i} that must be updated is:

$$(x, y) = (\lfloor j/k \rfloor + 1, \; j \bmod k). \tag{4.7}$$

This entry (x, y) of the matrix B^{δ^i} is the column A_y translated by the value δ^i_j (i.e., $A_y \oplus \delta^i_j$). Therefore, in the matrix $B^{\delta^{i+1}}$ the entry (x, y) must be replaced by $A_y \oplus \delta^{i+1}_j$. By doing only this change the matrix $B^{\delta^{i+1}}$ is obtained from the matrix B^{δ^i}.

Now, suppose the matrix B^{δ^i} was checked to see if it is a covering array of strength $t+1$ and the result was false because the submatrix conformed by the $t+1$ columns $B^{\delta^i}_{l_0}, B^{\delta^i}_{l_1}, \dots, B^{\delta^i}_{l_t}$ does not cover all the tuples of $\mathbb{Z}_v^{t+1}$. As before, let be j the position of the vector δ^{i+1} that is different from the previous vector δ^i, and let be $y = j \bmod k$. If $y \notin \{l_0, l_1, \dots, l_t\}$ then the matrix $B^{\delta^{i+1}}$ can not be a covering array of strength $t+1$ because its submatrix conformed by the columns $B^{\delta^{i+1}}_{l_0}, B^{\delta^{i+1}}_{l_1}, \dots, B^{\delta^{i+1}}_{l_t}$ is identical to the submatrix of B^{δ^i} conformed by the columns $B^{\delta^i}_{l_0}, B^{\delta^i}_{l_1}, \dots, B^{\delta^i}_{l_t}$, and this submatrix does not cover all the tuples of $\mathbb{Z}_v^{t+1}$.

For example, consider the base covering array $A = \mathrm{CA}(5; 2, 4, 2)$ of Figure 4.11. Let be $\delta = (1\ 0\ 1\ 0)$ the current vector δ. The matrix B resulting from the application of the construction $\mathcal{E}$ to the base covering array A and to the vector $\delta = (1\ 0\ 1\ 0)$ is the one shown to the right of the Figure 4.11.

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \qquad B^{\delta} = \left(\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

Figure 4.11: The matrix B^{δ} for the vector $\delta = (1\ 0\ 1\ 0)$.

The vector δ' that follows the current vector $\delta = (1\ 0\ 1\ 0)$ in v-ary Gray code is $\delta' = (1\ 0\ 1\ 1)$. Since these vectors are different in the third position their respective matrices B are different in the third column. Substituting $k = 4$ and $j = 3$ in expression (4.7), the column y of the copy x of the base covering array that must be updated is:

$$(x, y) = (\lfloor j/k \rfloor + 1,\ j \bmod k) = (\lfloor 3/4 \rfloor + 1,\ 3 \bmod 4) = (1, 3)$$

So, the elements of matrix B to be updated are the elements in the column $y = 3$ of the copy $x = 1$ of the base covering array. These elements are shaded in the matrices B for the vectors $\delta = (1\ 0\ 1\ 0)$ and $\delta' = (\ 1\ 0\ 1\ 1)$ shown in the Figure 4.12.

$$B^{\delta} = \left(\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right) \qquad B^{\delta'} = \left(\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{array}\right)$$

Figure 4.12: The matrices B for the vectors $\delta = (1\ 0\ 1\ 0)$ and $\delta' = (\ 1\ 0\ 1\ 1)$.

In other hand, the matrix B^{δ} for the vector $\delta = (1\ 0\ 1\ 0)$ is not a covering array of strength $t+1 = 3$, because the combination of columns $\{0,1,2\}$ does not cover the tuples $(0,1,1)$ and $(1,1,0)$. So, the matrix $B^{\delta'}$ can not be a covering array of strength $t = 3$ because its columns 0, 1, 2 are identical to the columns 0, 1, 2 of the matrix B^{δ}. Figure 4.13 shows the submatrix conformed by the columns 0, 1, and 2 in both matrices B^{δ} and $B^{\delta'}$.

$$B^{\delta} = \left(\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right) \qquad B^{\delta'} = \left(\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{array}\right)$$

Figure 4.13: The submatrix conformed by the columns 0, 1, and 2 in the matrices B^{δ} and $B^{\delta'}$.

Following this approach the cost of generating the matrices B is $O(Nv \times (k+1))$ for the first vector δ, and $O(N)$ for all the other vectors δ. So, the cost of generating the matrices B for all vectors δ is:

$$O(Nv(k+1) + N(v^{k(v-1)} - 1)) \tag{4.8}$$

The number of times the verification of the matrix B is omitted depends on the base covering array. The worst case is the same: $O\big(v^{k(v-1)}\binom{k}{t+1}Nv(t+1)\big)$, but it is very improbable that this case occurs. In Chapter 5 this cost is estimated empirically.

4.2.3 Algorithms for the Construction $\mathcal{E}$

In this section are described the algorithms to apply the construction $\mathcal{E}$ to every vector δ for a base covering array CA$(N; t, k, v)$. Algorithm 1 is the main algorithm for the implementation of the construction $\mathcal{E}$; it receives as input data the base covering array $A =$ CA$(N; t, k, v)$ and applies the

construction $\mathcal{E}$ to each one of the $v^{k(v-1)}$ vectors δ. In lines 1-4 the algorithm declares the next four auxiliary arrays:

- B, the matrix of dimensions $Nv \times (k+1)$ to store the result of the application of the construction $\mathcal{E}$.
- δ, a vector of k elements to store the current vector δ.
- S, a vector of k elements used to generate the vectors δ in v-ary Gray code.
- D, a vector to store the first combination of $t+1$ columns of matrix B which does not cover all the tuples of the set $\mathbb{Z}_v^{t+1}$.

Algorithm 1 apply_$\mathcal{E}(A, N, t, k, v)$

Require: the base covering array $A = \mathsf{CA}(N, t, k, v)$
Ensure: the application of the construction $\mathcal{E}$ to every vector δ for the base covering array A

```
1: B is a matrix of dimensions Nv × (k + 1)
2: δ is a vector of k elements to store the Gray code
3: S is a vector of k elements to store +1 or −1 for each element of δ
4: D is a vector of t + 1 elements
5: initialize_matrix_B(A, B, N, k)
6: initialize_vector_delta(δ, S, k)
7: D[0] ← −1
8: i ← 0
9: while i ≠ −1 do
10:    apply_vector_delta(A, N, k, B, δ, i)
11:    if is_covering_array(B, t, v, i, D) then
12:       {do something with the new covering array B of strength t + 1}
13:    end if
14:    i ← next_vector_delta(δ, S, v, k)
15: end while
```

In lines 5-7 these arrays are initialized. The function *initialize_matrix_B*(), implemented in Algorithm 2, is called in line 5 to initialize the matrix B; next in line 6 the function *initialize_vector_delta*(), implemented in Algorithm 3, is called to generate the first vector δ; and

in line 7 the first position of vector D is set to -1, where -1 is an invalid column index used to signal that vector D currently does not contain a valid combination of $t+1$ columns of matrix B.

In line 8 the variable i is set to 0, this variable is updated in line 14 with the value returned by the function *next_vector_delta*() implemented in Algorithm 4. The **while** loop of line 9 is executed while this variable i is distinct of -1. In line 10 the current vector δ is applied to the matrix B (Algorithm 5) to produce the matrix B for the current vector δ. Next, in line 11 the matrix B is verified to see if it is a covering array of strength $t+1$ (Algorithm 6). In case of the matrix B is a covering array of strength $t+1$, the comment in line 12 will be replaced by a recursive call of the same function *apply_*$\mathcal{E}$() in order to apply the construction $\mathcal{E}$ to the new base covering array B; this operation will be explained in more detail in Section 4.2.4. In line 14 the invocation of the function *next_vector_delta*() updates the vector δ with the following vector in v-ary Gray code and returns the index i modified in vector δ. When all vectors δ have been generated, the function *next_vector_delta*() returns -1 as the modified index in order to break the **while** loop of line 9.

Algorithm 2 initializes the matrix B. The initialization of the matrix B consists in filling its first k columns with v exact copies of the base covering array $A = \mathrm{CA}(N; t, k, v)$; also, the last column is filled with N zeroes, followed by N ones, and so on until finish with N elements equal to $v-1$. The initialization of the matrix B is done only one time, before enter into the **while** loop of Algorithm 1; the resulting matrix B corresponds to the vector δ in which all its elements are equal to 0.

Vectors δ are generated in v-ary Gray code using the algorithm developed in [36]. In Algorithm 3 the vector δ and its auxiliary vector S are initialized. Next, in Algorithm 4 the following vector δ in v-ary Gray code is computed following the algorithm described in [36]. In each call, he function *next_vector_delta*() computes the following vector δ in v-ary Gray code and returns the index i modified to obtain the current vector δ. If the returned value is distinct of -1 then the vector δ is valid, so the construction $\mathcal{E}$ can be applied with this vector δ. When all vectors δ have been generated, the function returns -1 as the changed index to finalize the **while** loop of Algorithm 1.

Algorithm 2 initialize_matrix_$B(A, B, N, k)$

Require: the covering array $A = \text{CA}(N; t, k, v)$, the matrix B
Ensure: the initialization of the matrix B

```
1: for l = 0 to v − 1 do
2:    {set the first k columns of the l-th copy}
3:    for i ← 0 to N − 1 do
4:       for j ← 0 to k − 1 do
5:          B[l * N + i][j] ← A[i][j]
6:       end for
7:    end for
8:    {set the last column of the l-th copy}
9:    for i = 0 to N − 1 do
10:      B[l * N + i][k] ← l
11:   end for
12: end for
```

The function *apply_vector_delta*() implemented in Algorithm 5 obtains the matrix B for the current vector δ. To do this, the function updates the column of the matrix B affected by the index changed in vector δ. Let be i de index changed in vector δ, then the column of matrix B that must be updated is $c = i \bmod k$, and the copy of the base covering array to be updated is $\lfloor i/k \rfloor + 1$ (see expression 4.7). Since the base covering array has N rows, the elements of the column c of matrix B to be updated are the elements from the row $start = (\lfloor i/k \rfloor * N) + N$ to the row $start + N - 1$. The **for** loop in line 5 of Algorithm 5 replaces these elements by the column c of the base covering array translated by the value $\delta[i]$, where i is the index of the element modified in vector δ.

Algorithm 3 initialize_vector_delta(δ, S, k)

Require: the vector δ, the vector S, the number of columns k of the base covering array
Ensure: the initialization of vectors δ and S

```
1: for i ← 0 to k − 1 do
2:    δ[i] ← 0
3:    S[i] ← 1
4: end for
```

Algorithm 4 next_vector_delta(δ, S, v, k)

Require: the vector δ, the vector S, the order v, and the number of columns k of the base covering array
Ensure: the next vector δ in v-ary Gray code

```
1: i ← k − 1
2: j ← δ[k − 1] + S[k − 1]
3: while (j ≥ v or j < 0) and (i ≠ −1) do
4:    S[i] ← −S[i]
5:    i ← i − 1
6:    if i ≠ −1 then
7:       j ← δ[i] + S[i]
8:    end if
9: end while
10: if i ≠ −1 then
11:    δ[i] ← j
12: end if
13: return i
```

Algorithm 5 apply_vector_delta(A, N, k, B, δ, i)

Require: the base covering array $A = \mathsf{CA}(N; t, k, v)$, the matrix B, the vector δ, the index i modified in vector δ
Ensure: the matrix B for the current vector δ

```
1: start ← (⌊i/k⌋ * N) + N
2: end ← start + N − 1
3: c ← i mod k
4: l ← 0
5: for j ← start to end do
6:    B[j][c] ← (A[l][c] + δ[i]) mod v
7:    l ← l + 1
8: end for
```

Finally, Algorithm 6 implements the function *is_covering_array*() to verify if the matrix B for the current vector δ is a covering array of strength $t + 1$. The strategy of this function is to check that every combination of $t + 1$ columns from the first k columns covers all the tuples of the set $\mathbb{Z}_v^{t+1}$ at least once. By theorem 7 the combinations of columns involving the last column of matrix B do not require to be verified. The verification can be skipped when the vector D contains a valid

combination of $t+1$ columns and the column recently updated in matrix B is not in vector D, because in this case the matrix B does not have any possibility of being a covering array of strength $t+1$.

The function *is_covering_array*() receives the index i returned by the function *next_vector_delta*(). With this value i the function computes the index j of the column of matrix B that was updated by the function *apply_vector_delta*(). If j is not in D then the matrix B is not a covering array of strength $t+1$ and the function *is_covering_array*() returns **false** in line 4. If j is one of the columns stored in vector D, then the function performs a normal verification of the matrix B. When in line 9 it is found a combination C of $t+1$ columns with missing tuples, the combination C is stored in vector D and the verification is stopped because the matrix B is not a covering array of strength $t+1$. In case of every combination of $t+1$ columns covers all the tuples of the set $\mathbb{Z}_v^{t+1}$ the matrix B is a covering array of strength $t+1$, so the first element of vector D is set to -1 in order to perform a normal verification in the following call to the function *is_covering_array*().

4.2.4 Iterative Application of the Construction $\mathcal{E}$

In Algorithm 1 was not defined the action to perform when the matrix B is a covering array of strength $t+1$. The application of the construction $\mathcal{E}$ to all vectors δ can generate several covering arrays of strength $t+1$ based on a covering array of strength t; and the construction $\mathcal{E}$ should be applied to all of them as defined in the methodology to construct the TCAs (Section 4.1).

Therefore, the function *apply_*$\mathcal{E}$() implemented in Algorithm 1 should be called recursively in order to apply the construction $\mathcal{E}$ to the covering arrays in the previous floor of the TCA. Algorithm 7 shows the recursive implementation of the function *apply_*$\mathcal{E}$(), this implementation requires a set of auxiliary arrays B, δ, S, and D for each *floor* of the TCA, so we use the following arrays:

- $Tower$, an array of matrices, its content is the current TCA.

Algorithm 6 is_covering_array(B, t, v, i, D)

Require: the matrix B of size $Nv \times (k+1)$, the index i changed in vector δ, and vector D
Ensure: **true** if matrix B is a covering array of strength $t+1$ and **false** otherwise

```
1: j ← i mod k
2: if D[0] ≠ −1 then
3:    if the the column index j is not in D then
4:       return false {matrix B can not be a covering array}
5:    end if
6: else
7:    {to perform a normal verification of the matrix B}
8:    for all combinations of t + 1 columns C from the first k columns do
9:       if columns C[0], C[1], ..., C[t] do not cover all the tuples in the set Z_v^{t+1} then
10:         D ← C          {save this combination of columns in vector D}
11:         return false {B is not a covering array}
12:      end if
13:   end for
14:   D[0] ← −1 {indicates to perform a normal verification in the next call}
15:   return true
16: end if
```

- $BestTower$, an array of matrices, its content is the best (the highest) TCA found until now.
- Δ, an array of vectors, it contains a vector δ for each floor of the TCA.
- $Sign$, an array of vectors, they are used to generate the vectors δ.
- $Comb$, an array of vectors, a vector for each floor to store one combination of columns which does not meet the requirements to be a covering array.

Algorithm 7 uses global variable h^* to store the maximum height reached until now. Initially h^* is equal to 0, but it is incremented each time a new best height h is reached. Notice that the parameter h for the function *apply_$\mathcal{E}$_rec*() is the height to tray to reach, so the global variable h^* is updated only when $h > h^*$, also the best tower found is updated in this case (line 7). The way in which Algorithm 7 is implemented allows the iterative application of the construction $\mathcal{E}$ to all the covering arrays in the TCA derived from the input covering array of strength t.

Algorithm 7 apply_$\mathcal{E}$_rec(A, N, t, k, v, h)

Require: the base covering array $A = \text{CA}(N, t, k, v)$ and the height h to try to reach
Ensure: the application of the construction $\mathcal{E}$ to every vector δ for the base covering array A

```
1: B ← Tower[h]
2: δ ← Δ[h]
3: S ← Sign[h]
4: D ← Comb[h]
5: if h > h* then
6:    h* ← h
7:    BestTower ← Tower
8: end if
9: initialize_matrix_B(A, B, N, k)
10: initialize_vector_delta(δ, S, k)
11: D[0] ← −1
12: i ← 0
13: while i ≠ −1 do
14:    apply_vector_delta(A, N, k, B, δ, i)
15:    if is_covering_array(B, t, v, i, D) then
16:       apply_E_rec(B, Nv, t + 1, k + 1, v, h + 1)
17:    end if
18:    i ← next_vector_delta(δ, S, v, k)
19: end while
```

In the final program the input covering array of strength t for the function *apply_$\mathcal{E}$_rec*() will be a non-isomorphic covering array of minimum rank for the parameters N, t, k, and v. Therefore, the methodology proposed will be implemented by calling the function *apply_$\mathcal{E}$_rec*() for every non-isomorphic covering array of minimum rank CA(N, t, k, v), as explained in Figure 4.1 of Section 4.1.

4.3 Non-Isomorphic Bases

The first step in the methodology proposed to construct the TCAs is the construction of all the non-isomorphic covering arrays for the given parameters N, t, k, and v. The objective of this is to use the non-isomorphic covering arrays as the bases of the TCAs, in order to ensure the construction of the

TCA with the greatest height, but without exploring all the equivalent bases. Section 4.3.1 proves that only the non-isomorphic covering arrays need to be checked as the bases of the TCAs; Section 4.3.2 to Section 4.3.5 describe the algorithm developed in this thesis to generate the non-isomorphic covering arrays.

4.3.1 Bases for the Towers

As was said in Chapter 2 there may be several covering arrays for the same parameters N, t, k, and v, some of which are isomorphic among them, while others are non-isomorphic among them. Isomorphic covering arrays have equivalent coverage properties, and in this section it is proved that this equivalence implies that all the isomorphic covering arrays produce TCAs with equal maximum height.

Consider the following isomorphic covering arrays A_1 and A_2 with parameters $N = 6$, $k = 5$, $t = 2$, and $v = 2$; these covering arrays are isomorphic because A_2 can be derived from A_1 by means of the row permutation $\tau = (3\ 1\ 2\ 5\ 4\ 0)$, the column permutation $\pi = (4\ 2\ 0\ 3\ 1)$, and the column relabeling $\phi = (0\ 1\ 0\ 1\ 0)$:

$$A_1 = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Now we apply the construction $\mathcal{E}$ to every vector δ for the covering arrays A_1 and A_2, and record the number of missing tuples in every matrix B resulting from the application of the construction $\mathcal{E}$. Table 4.1 contains the results obtained.

Let be *lmiss*(x) a function that applies the construction $\mathcal{E}$ to every vector δ for the covering array x and returns a list of the number of missing tuples in the matrix B for each vector δ. For examples:

$$\textit{lmiss}(A_1) = (24, 10, 9, 10, 4, 8, 4, 9, 9, 4, 4, 4, 4, 8, 4, 9, 4, 8, 12, 8, 4, 0, 4, 4, 4, 8, 12, 8, 9, 12, 9, 10)$$

Vector δ	A_1 Missing tuples in B^δ	A_2 Missing tuples in B^δ
(0 0 0 0 0)	24	24
(0 0 0 0 1)	10	9
(0 0 0 1 1)	9	4
(0 0 0 1 0)	10	10
(0 0 1 1 0)	4	9
(0 0 1 1 1)	8	8
(0 0 1 0 1)	4	4
(0 0 1 0 0)	9	10
(0 1 1 0 0)	9	4
(0 1 1 0 1)	4	4
(0 1 1 1 1)	4	4
(0 1 1 1 0)	4	8
(0 1 0 1 0)	4	4
(0 1 0 1 1)	8	4
(0 1 0 0 1)	4	9
(0 1 0 0 0)	9	9
(1 1 0 0 0)	4	4
(1 1 0 0 1)	8	4
(1 1 0 1 1)	12	4
(1 1 0 1 0)	8	8
(1 1 1 1 0)	4	12
(1 1 1 1 1)	0	0
(1 1 1 0 1)	4	4
(1 1 1 0 0)	4	8
(1 0 1 0 0)	4	9
(1 0 1 0 1)	8	8
(1 0 1 1 1)	12	12
(1 0 1 1 0)	8	12
(1 0 0 1 0)	9	9
(1 0 0 1 1)	12	8
(1 0 0 0 1)	9	4
(1 0 0 0 0)	10	10

Table 4.1: Number of missing tuples in matrix B when the construction $\mathcal{E}$ is applied to each vector δ for the covering arrays A_1 and A_2.

$$\mathit{lmiss}(A_2) = (24, 9, 4, 10, 9, 8, 4, 10, 4, 4, 4, 8, 4, 4, 9, 9, 4, 4, 4, 8, 12, 0, 4, 8, 9, 8, 12, 12, 9, 8, 4, 10)$$

We notice that *lmiss*(A_1) is a permutation of *lmiss*(A_2), and we claim that this is true for any pair of isomorphic covering arrays. To prove empirically this asseveration we developed a program that takes a covering array $A = \mathsf{CA}(N; t, k, v)$ and computes *lmiss*(A); next, for every covering array A' isomorphic to A the program computes *lmiss*(A') and checks if it is a permutation of *lmiss*(A). One way to check if *lmiss*(A') is a permutation of *lmiss*(A) is to sort the lists and compare them entry by entry. We execute this program with a number of covering arrays $A = \mathsf{CA}(N; t, k, v)$; in all cases the list of any covering array isomorphic to A was a permutation of *lmiss*(A).

Besides we found another important property of the isomorphic covering arrays: suppose the construction $\mathcal{E}$ produces a covering array of strength $t + 1$ when applied to a covering array A of strength t and to a particular vector δ; then for the same vector δ the construction $\mathcal{E}$ produces a covering array of strength $t + 1$ for any covering array isomorphic to A. Thereby, it is sufficient to check only one of the $N!\, k!\, (v!)^k$ isomorphic covering arrays as the base of the TCA, because they all produce TCAs with the same maximum height.

For non-isomorphic covering arrays the lists of the number of missing tuples are not in general a permutation of each other. For example, the next covering array A_3 is another covering array for $N = 6$, $k = 5$, $t = 2$, and $v = 2$, but it is non-isomorphic to the coverings arrays A_1 and A_2:

$$A_3 = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The list of the number of missing tuples for this covering arrays is *lmiss*$(A_3) = (26, 12, 5,$ $10, 9, 4, 5, 10, 6, 10, 12, 8, 6, 10, 5, 10, 9, 4, 12, 8, 4, 8, 12, 8, 6, 10, 12, 8, 6, 10, 5, 10)$, which is not a permutation of *lmiss*(A_1) or *lmiss*(A_2). In addition, the construction $\mathcal{E}$ does not produce any covering array of strength $t + 1$ for this covering array A_3, as it does for the covering arrays A_1

and A_2. Therefore, we need to prove as the base of the TCA one element of each group of non-isomorphic covering arrays in order to construct the highest TCA. In Chapter 2 was viewed that the covering arrays of minimum rank can be used as the representative elements of the classes of isomorphic covering arrays. The following section starts the description of the algorithm developed in this thesis to generate all the non-isomorphic covering arrays of minimum rank that exist for a given combination of the parameters N, t, k, and v.

4.3.2 The NonIsoCA Algorithm

We developed an algorithm called NonIsoCA to generate all the non-isomorphic covering arrays of minimum rank for the parameters N, t, k, and v. The strategy of the NonIsoCA algorithm is to construct the covering arrays column by column (from 1 column to k columns) restricting column i to be smaller than column j in lexicographic order if $i < j$. The columns are generated in this manner because we are looking for covering arrays of minimum rank, and one property of these covering arrays is that their rows and their columns are increasingly ordered in lexicographic order. So, the arrays whose columns are not in lexicographic order can not be of minimum rank, and therefore the algorithm avoids to generate such arrays.

The algorithm starts the construction of the non-isomorphic covering arrays of k columns with the construction of the first column. In order to be the first column of a covering array of minimum rank a column c must satisfy the following properties, suppose order v:

- The column c is conformed by a block of b_0 zeroes, followed by a block of b_1 ones, and so on until finish with a block of b_{v-1} elements equal to $v-1$.
- $b_i \geq v^{t-1}$ for $i = 0, 1, \ldots, v-1$, where t is the strength of the covering arrays.
- $b_i \geq b_{i+1}$ for $i = 0, 1, \ldots, v-2$.

For example, suppose $N = 8$, $k = 6$, $t = 2$, and $v = 2$; then $v^{t-1} = 2^{2-1} = 2$, and so there must be at least two occurrences of each symbol 0 and 1 in the first column of the covering arrays

of minimum rank. The only three columns that satisfy the above requirements are these ones, where the super-index T means transpose:

$$(0\,0\,0\,0\,0\,0\,1\,1)^T$$
$$(0\,0\,0\,0\,0\,1\,1\,1)^T$$
$$(0\,0\,0\,0\,1\,1\,1\,1)^T$$

The reason for the above three rules for the first column of a covering array of minimum rank is that in other case (i.e., some of the rules are not satisfied) there exists a column relabeling (symbol permutation) followed by a row sorting that produces a column of a smaller rank. For example, the column $(0\,0\,0\,1\,1\,1\,1\,1)^T$ can not be the first column of covering array of minimum rank for $N = 8$, $k = 6$, $t = 2$, $v = 2$, because the symbol permutation $(1\ 0)$, which means to replace the zeroes by ones and the ones by zeroes, followed by a row sorting gives the column $(0\,0\,0\,0\,0\,1\,1\,1)^T$, which has a smaller rank.

Algorithm 8 implements the function *first_column*() that generates the first columns of the covering arrays of minimum rank. The function receives the column c of size N to store the result, the parameters N, t, v, and a vector called q of size v used to store at position i the number of occurrences of symbol i in column c. Initially $q[0] = -1$ to recognize the first time the function *first_column*() is called. The return value of the function is **true** if a column c was constructed and **false** if all columns c of minimum rank have been constructed.

The function *first_column*() is conformed by the **if-else** sentence at lines 1-27, the **if** sentence at lines 28-35, and the **return** sentence at line 36. The **if-else** sentence obtains the values of the vector q of size v, where $q[i]$ is the number of occurrences of symbol i in column c; this way the column c will be conformed by $q[0]$ zeroes, followed by $q[1]$ ones, and so on until finish with $q[v-1]$ elements equal to $v-1$. If a valid column c was found then the variable *found* is set to true, and the **if** sentence of lines 28-35 fills the column c according to the values in vector q.

In the first call to the function *first_column*() the value of $q[0]$ is equal to -1, so the **if** part of the **if-else** sentence is executed. For a covering array with parameters N, k, t, v each symbol of $\mathbb{Z}_v$

Algorithm 8 first_column(c, N, t, v, q)

Require: the column c; the parameters N, t, v; and the vector q
Ensure: **true** if one column c of minimum rank was constructed; **false** otherwise

1: **if** $q[0] = -1$ **then**
2: $q[i] \leftarrow v^{t-1}$ for $i = 1, 2 \ldots, v-1$
3: $q[0] \leftarrow N - (q[1] + q[2] + \cdots + q[v-1])$
4: $found \leftarrow$ **true**
5: **else**
6: $found \leftarrow$ **false**; $i \leftarrow 0$
7: **while** $i < v-1$ **and** $found =$ **false do**
8: **if** $q[i] \geq q[i+1] + 2$ **then**
9: $q[i+1] \leftarrow q[i+1] + 1$
10: $q[j] \leftarrow q[i+1]$ for $j = 1, 2, \ldots, i$
11: $q[0] \leftarrow N - (q[1] + q[2] + \cdots + q[v-1])$
12: $found \leftarrow$ **true**
13: **end if**
14: $i \leftarrow i + 1$
15: **end while**
16: **if** $found =$ **false then**
17: $i \leftarrow 0$
18: **while** $i < v-2$ **and** $found =$ **false do**
19: **if** exists $j \geq i+2$ s.t. $q[i] - q[i+1] = 1$, $q[j-1] - q[j] = 1$, $q[i+1] = \cdots = q[j-1]$ **then**
20: $q[i] \leftarrow q[i] - 1$
21: $q[j] \leftarrow q[j] + 1$
22: $found \leftarrow$ **true**
23: **end if**
24: $i \leftarrow i + 1$
25: **end while**
26: **end if**
27: **end if**
28: **if** $found$ **then**
29: $l \leftarrow 0$
30: **for** $i \leftarrow 0$ **to** $v-1$ **do**
31: **for** $j \leftarrow 1$ **to** $q[i]$ **do**
32: $c[l] \leftarrow i$; $l \leftarrow l + 1$
33: **end for**
34: **end for**
35: **end if**
36: **return** $found$

occurs at least v^{t-1} times in every column; so the fist column c of minimum rank is that column in which the symbols $1, 2, \ldots, v-1$ occur exactly v^{t-1} times and the symbol 0 occurs $N - [(v-1)v^{t-1}]$ times; this is what the operations in lines 2-3 do. In the subsequent calls the value of $q[0]$ is distinct of -1, so the **else** part at line 5 is executed to generate the other columns c of minimum rank. These columns c are given by one of the following two operations:

1. To search sequentially if the there exists an index i such that $q[i] \geq q[i+1] + 2$; if such index exists then:
   - increment $q[i+1]$ in one unit;
   - make $q[j]$ equal to the updated value of $q[i+1]$ for $j = 1, \ldots, i$; and
   - make $q[0] = N - \sum_{i=1}^{v-1} q[i]$.
2. If none index i was found in the previous operation, then to search sequentially if there exists a pair of indices i and j such that $j \geq i+2$, $q[i] - q[i+1] = 1$, $q[j-1] - q[j] = 1$, and $q[i+1] = q[i+2] = \cdots = q[j-1]$; if such indices exist then:
   - decrement the value of $q[i]$ in one unit; and
   - increment the value of $q[j]$ in one unit.

To exemplify the operations consider $N = 20$, $t = 2$, and $v = 4$. In this case $v^{t-1} = 4^{2-1} = 4$; so each symbol 0, 1, 2, 3 must appear at least 4 times in the column c. The first column c of minimum rank has exactly four occurrences of the symbols 1, 2, 3; and $20 - 4(3) = 8$ occurrences of the symbol 0; this column and the current vector q are shown in the first row of Table 4.2. From this column the operation 1 is applied three times to obtain another three columns c. Finally, the operation 2 is applied to obtain the last column c that can be the first column of a covering array of minimum rank.

Now that we have a way to construct the first column of the covering arrays of minimum rank, the Algorithm 9 shows the NonIsoCA algorithm. In lines 1-2 the algorithm declares the matrix A

Column c	Vector q	Operation
$(0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,2\,2\,2\,2\,3\,3\,3\,3)^T$	$(8,4,4,4)$	–
$(0\,0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,2\,2\,2\,2\,3\,3\,3\,3)^T$	$(7,5,4,4)$	1: $i=0$
$(0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,1\,2\,2\,2\,2\,3\,3\,3\,3)^T$	$(6,6,4,4)$	1: $i=0$
$(0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,2\,2\,2\,2\,2\,3\,3\,3\,3)^T$	$(6,5,5,4)$	1: $i=1$
$(0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,2\,2\,2\,2\,2\,3\,3\,3\,3\,3)^T$	$(5,5,5,5)$	2: $i=0$, $j=3$

Table 4.2: Columns of minimum rank for $N=20$, $t=2$, and $v=4$; the third column describes the rule applied to the vector q of the second column.

to store the covering arrays and the auxiliary vector q to compute the first columns of the covering arrays. The k columns of A are denoted by $A_0, A_1, \ldots, A_{k-1}$. The **while** loop of line 3 is executed for every column of minimum rank that the function *first_column*() constructs.

Algorithm 9 NonIsoCA(N,t,k,v)

Require: the parameters N, t, k, v
Ensure: the construction of the non-isomorphic covering arrays of minimum rank CA$(N;t,k,v)$

```
 1: A is a matrix of size N × k; its columns are A_0, ..., A_{k-1}
 2: q is a vector of v elements with q[0] = −1
 3: while first_column(A_0, N, t, v, q) do
 4:   A_1 ← A_0
 5:   r ← 1
 6:   while r ≥ 1 do
 7:     if extend(A, r) then
 8:       if is_minimum(A, r + 1) then
 9:         if r + 1 = k then
10:           write(A, r + 1)
11:         else
12:           r ← r + 1
13:           A_r ← A_{r−1}
14:         end if
15:       end if
16:     else
17:       r ← r − 1
18:     end if
19:   end while
20: end while
```

The central part of the NonIsoCA algorithm is the backtracking process to extend the covering arrays from 1 column to k columns; this process is done in the **while** loop of lines 6-19. This loop is executed while the current number of columns r of the current covering array of minimum rank is greater than or equal to 1. We describe the backtracking process next: suppose the algorithm has constructed a covering array of minimum rank with r columns A_0, A_1, ..., A_{r-1}, where $1 \leq r < k$. To extend this covering array to $r+1$ columns the algorithm checks in column A_r all the columns lexicographically greater than A_{r-1}, until one of the following two cases occur:

1. A column conforming a covering array (not necessarily of minimum rank) with the first r columns was found.

2. All the columns greater than A_{r-1} were checked and none of them conformed a covering array with the first r columns.

In the first case the covering array of $r+1$ columns $(A_0\ A_1\ \cdots\ A_{r-1}\ A_r)$ is tested for minimum rank (Algorithm 11 or Algorithm 12). If it is of minimum rank then the algorithm has constructed a covering array of minimum rank with $r+1$ columns, and the extension process tries to expand this covering array to $r+2$ columns. But if the covering array is not of minimum rank it is rejected and the extension process to $r+1$ columns is executed again, continuing from the last column checked in the previous call. If none column produces a covering array of $r+1$ columns, the algorithm decrements r in one unit (backtracks) in order to try to generate another covering array of minimum rank with r columns, which perhaps can be extended to $r+1$ columns.

For example, suppose that for $N = 6$, $k = 7$, $t = 2$, $v = 2$, the algorithm has constructed the following covering array of minimum rank with $r = 3$ columns:

$$\begin{pmatrix} 0 & 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * & * \\ 0 & 1 & 1 & * & * & * & * \\ 0 & 1 & 1 & * & * & * & * \\ 1 & 0 & 1 & * & * & * & * \\ 1 & 1 & 0 & * & * & * & * \end{pmatrix}$$

In this subarray the algorithm checks all columns lexicographically greater than the last one until:

1. It finds a column that conforms a covering array with the first $r = 3$ columns. The following step in this case is to verify if the covering array of $r = 4$ columns is of minimum rank, and in affirmative case the algorithm tries to extend this covering array to $r = 5$ columns.

2. It determines that it is not possible to extend the covering array. In this case the last column of the subarray is removed, and the algorithm tries to generate another covering array of minimum rank with $r = 3$ columns.

In line 7 of Algorithm 9 the function *extend*() tries to extend the current covering array of r columns to $r + 1$ columns. The candidate columns for position r are the columns lexicographically greater than A_{r-1}; so the column A_{r-1} is copied into column A_r before starting the search of the column A_r (line 4 and line 13). The value returned by the function *extend*() is **true** if the extension process success, and it is **false** otherwise. When the function returns **true** it is checked if the covering array of $r + 1$ columns is of minimum rank (line 8). If it is not of minimum rank the algorithm tries to generate another covering array with $r + 1$ columns; but in other case the algorithm checks if the number of columns is equal to k. In the affirmative case a new non-isomorphic covering array of minimum rank CA$(N; t, k, v)$ has been constructed, so it is reported by the function *write*() in line 10; but in other case (i.e., $r + 1 < k$) the value of r is incremented in one unit because the number of columns of the current subarray is now $r + 1$. When the function *extend*() returns **false** the value of r is decremented in one unit, or in other words, the algorithm backtracks to column $r - 1$; when r becomes zero the current iteration of the **while** loop of lines 6-19 finalizes.

The hard work of the NonIsoCA algorithm is done by the functions *extend*() and *is_minimum*(). The function *extend*() is described in Section 4.3.3, and the function *is_minimum*() is described in Section 4.3.4.

4.3.3 Extension of the Covering Arrays

For each candidate column A_r the function *extends*() checks if the array $(A_0\ A_1\ \cdots\ A_{r-1}\ A_r)$ is a covering array of strength t. But since $(A_0\ A_1\ \cdots\ A_{r-1})$ is a covering array of strength t, the algorithm only needs to check that every subarray conformed by A_r and $t-1$ columns from the columns $A_0, A_1, \ldots, A_{r-1}$ covers each t-tuple of $\mathbb{Z}_v^t$. The number of such subarrays is $\binom{r}{t-1}$, which is smaller than $\binom{r+1}{t}$, the number of subarrays that would be checked no taking advantage of the fact that the first r columns of A are a covering array of strength t.

In addition, the verification of the array $(A_0\ A_1\ \cdots\ A_{r-1}\ A_r)$ should be done only for the candidate columns that really have possibilities to conform a covering array of strength t with the previous r columns. In a covering array of strength t and order v each symbol $0, 1, \ldots, v-1$ occurs at least v^{t-1} times in every column of the covering array; so the algorithm performs the verification only for the candidate columns in which the symbols in $\mathbb{Z}_v$ occur at least v^{t-1} times.

Another restriction for the candidate columns A_r is possible given that we are constructing covering arrays of minimum rank: in a covering array of minimum rank every column c has the following characteristic:

$$c[i] \leq i \text{ for } i = 0, 1, \ldots, v-1$$

This way, when it is detected a position $0 \leq i \leq v-1$ such that $c[i] > i$ we can skip all the columns until the column equal to $c[i-1]$ incremented in one unit and $c[j] = 0$ for $j \geq i$. For example when the *extend*() function reaches the next column for $N = 16$ and $v = 4$:

$$(0\,0\,3\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0)^T$$

All the columns until $(0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0)^T$ can be skipped because they can not be part of a covering array of minimum rank. Notice that in this column the content of the position i

is less than or equal to i for $i = 0, 1, 2, 3$. The operation of skipping columns can be chained; for example when the extension process reaches the column:

$$(0\,1\,3\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0)^T$$

The algorithm detects that the content of the position 2 is greater than 2, so it increments the position 1 in one unit and writes zeroes in all the positions $j \geq 2$. The results is the column:

$$(0\,2\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0)^T$$

But in this column the content of the position 1 is greater than 1, so the content of the position 0 is incremented in one unit and a zero is placed in the positions $j \geq 1$. The results is:

$$(1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0)^T$$

In this column the content of the position 0 is greater than 0, so there are no more columns to check and the *extend*() function returns **false**.

The reason to impose $c[i] \leq i$ for $i = 0, 1, \ldots, v-1$ is that in other case there exists a symbol permutation (relabeling) of column c that produces a column c' of a smaller rank. For example consider $v = 3$, Table 4.3 shows in its first column all possible combinations for the first $v = 3$ elements of column c. In the second columns it is shown a symbol permutation of the symbols in c such that the resulting column c', shown in the third column of the table, has a rank smaller than or equal to the rank of c. A symbol permutation $(x\ y\ z)$ indicates to replace in column c the occurrences of 0 by x, the occurrences of 1 by y, and the occurrences of 2 by z. It can be noted that the first $v = 3$ elements of every column c' satisfies $c'[i] \leq i$ for $i = 0, 1, 2$.

All this facts are very important for the efficient implementation of the *extend*() function, given in Algorithm 10. This algorithm is conformed by a **while** loop that is executed until a **return**

First 3 elements of column c	Symbol permutation	First 3 elements of column c'
(0 0 0)	(0 1 2)	(0 0 0)
(0 0 1)	(0 1 2)	(0 0 1)
(0 0 2)	(0 2 1)	(0 0 1)
(0 1 0)	(0 1 2)	(0 1 0)
(0 1 1)	(0 1 2)	(0 1 1)
(0 1 2)	(0 1 2)	(0 1 2)
(0 2 0)	(0 2 1)	(0 1 0)
(0 2 1)	(0 2 1)	(0 1 2)
(0 2 2)	(0 2 1)	(0 1 1)
(1 0 0)	(1 0 2)	(0 1 1)
(1 0 1)	(1 0 2)	(0 1 0)
(1 0 2)	(1 0 2)	(0 1 2)
(1 1 0)	(1 0 2)	(0 0 1)
(1 1 1)	(1 0 2)	(0 0 0)
(1 1 2)	(2 0 1)	(0 0 1)
(1 2 0)	(2 0 1)	(0 1 2)
(1 2 1)	(2 0 1)	(0 1 0)
(1 2 2)	(2 0 1)	(0 1 1)
(2 0 0)	(1 2 0)	(0 1 1)
(2 0 1)	(1 2 0)	(0 1 2)
(2 0 2)	(1 2 0)	(0 1 0)
(2 1 0)	(2 1 0)	(0 1 2)
(2 1 1)	(2 1 0)	(0 1 1)
(2 1 2)	(2 1 0)	(0 1 0)
(2 2 0)	(1 2 0)	(0 0 1)
(2 2 1)	(2 1 0)	(0 0 1)
(2 2 2)	(1 2 0)	(0 0 0)

Table 4.3: All possible combinations for the first 3 positions of a column of order $v = 3$. The second column of the table shows the symbol permutations used to relabel c and produce c'.

sentence is executed in line 20 or in line 26; in the first case the value **false** is returned because all the candidate columns for position r were explored and none of them made a covering array with the previous columns; in the second case the value **true** is returned because it was found a column for position r that conformed a covering array of $r + 1$ columns.

The first sentence inside the **while** loop is the computation of the following column in

Algorithm 10 extend(A, r, v, t)

Require: the current covering array A of $r-1$ columns; the values r, v, t
Ensure: **true** if A was extended to $r+1$ columns; **false** otherwise

```
1: while true do
2:   A_r ← A_r + 1, where A_r + 1 is the column following A_r in lexicographic order
3:   i ← v − 2
4:   while i ≥ 0 and A_r[i] ≤ i do
5:     i ← i − 1
6:   end while
7:   if i ≥ 0 then
8:     A_r[i] ← 0
9:     i ← i − 1;
10:    while i ≥ 0 do
11:      A_r[i] ← A_r[i] + 1
12:      if A_r[i] > i then
13:        A_r[i] ← 0
14:        i ← i − 1
15:      else
16:        break
17:      end if
18:    end while
19:    if i < 0 then
20:      return false
21:    end if
22:  end if
23:  if every symbol in Z_v occurs at least v^{t−1} times in A_r then
24:    t' ← min(r, t)
25:    if the array (A_0 ··· A_r) is a covering array of strength t' then
26:      return true
27:    end if
28:  end if
29: end while
```

lexicographic order; for this operation we use the notation $A_r + 1$. In lines 3 to 6 it is checked if there is one position $0 \leq i \leq v-1$ whose content is greater than i. In the affirmative case, the index of this position is the value of the variable i, this variable is initialized with $v-2$ in line 3 and it is updated in line 5; also in this case the value of the variable i will be greater than or equal to zero after the **while** loop of lines 4-6, and so the **if** sentence of lines 7-22 is executed. If there is not

an index $0 \leq i \leq v-1$ such that $A_r[i] > i$ for the current column A_r then the value of the variable i will be -1 after the **while** loop of lines 4-6, and the **if** sentence of line 7 will not be executed.

Lines 8-18 implements the mechanism to adjust the column in order to the position i contains a symbol less than or equal to i. In case of this adjustment sets the content of the variable i to -1 the **if** sentence of lines 19-21 returns **false** to the caller.

If the current column A_r satisfies the requirement that $A_r[i] \leq i$ for $i = 0, 1, \ldots, v-1$ the control flow of the *extend*() function reaches the line 23. In this line it is checked if each symbol of $\mathbb{Z}_v$ occurs at least v^{t-1} times in A_r. If it is not the case the current iteration of the outer **while** loop finalizes and a new iteration starts at line 2 with the computation of the column $A_r + 1$. But in case of each symbol of the alphabet occurs at least v^{t-1} times in A_r, then it is checked if the subarray $(A_0\ A_1\ \cdots\ A_r)$ is a covering array of strength t', where t' is the minimum between r and t. If $r \geq t$ then the subarray is checked to see if it is a covering array of strength t, but if $r < t$ then it is checked if the subarray of r columns is a covering array of strength r, since in other case its columns can not be part of a covering array of strength $t > r$.

4.3.4 Minimum Rank Test

For a covering array A with N rows and r columns a naive algorithm to verify if A is of minimum rank has to check, in the worst case, all the $N!\,r!\,(v!)^r - 1$ covering arrays isomorphic to A. Algorithm 11 implements the naive verification of minimum rank. This algorithm checks all the $(v!)^r$ column relabelings for each one of the $r!$ column permutations, and the $N!$ row permutations for each column relabeling. The covering array A' in the **if** sentence of line 4 is the covering array derived from A by a particular column permutation, a particular combination of symbol permutations, and a particular row permutation.

Let be C the covering array derived from A (the covering array under testing) by a particular column permutation and a particular column relabeling of the column permutation; for this covering array C the naive algorithm checks the $N!$ permutations of rows. However, it is not necessary

Algorithm 11 is_minimum(A)

Require: one covering array A
Ensure: **true** if A is of minimum rank; **false** otherwise

```
1: for all permutations of columns do
2:   for all permutations of symbols in the columns do
3:     for all permutations of rows do
4:       if rank(A') < rank(A) then
5:         return false
6:       end if
7:     end for
8:   end for
9: end for
10: return true
```

to check all of them because the row permutation that produces the covering array of minimum rank is that one in which the rows of C are lexicographically ordered. Therefore, to find the best permutation of rows, the rows of C must be sorted lexicographically. Moreover, sometimes it will not be necessary to sort all the N rows of C in order to determine if A is of minimum rank. By using an ascending sorting algorithm over the rows of C it can be concluded that A is not of minimum rank as soon as one row of C is lexicographically smaller than the corresponding row of A. This way, the $N!$ permutations of rows are reduced to one row sorting done in $O(N^2)$ time.

With this improvement the permutations of rows have a fixed cost of $O(N^2)$, so we only concentrate in checking the column relabelings and the column permutations. These two symmetries are checked jointly by taking advantage of the fact that a covering array A of r column is a covering array of minimum rank if and only if each subarray of A conformed by its first $s = 1, 2, \ldots, r$ columns is also a covering array of minimum rank; if the first s columns of A are not of minimum rank then A is not of minimum rank.

In what follows we assume the existence of the function *rank*(x), which computes the rank of the covering array x. In Definition 9 we define the result of comparing the rank of two covering arrays.

DEFINITION **9** *Let be x and y two covering arrays with dimensions $N \times k$ over $\mathbb{Z}_v$, and let be*

$rank(x) = (x_0, x_1, \ldots, x_{N-1})$ and $rank(y) = (y_0, y_1, \ldots, y_{N-1})$. Then:

- *$rank(x)$ is equal to $rank(y)$ iff $x_i = y_i$ for $i = 0, 1, \ldots, N-1$.*
- *$rank(x)$ is smaller than $rank(y)$ iff there exists an index $0 \leq i \leq N-1$ such that $x_i < y_i$ and $x_j = y_j$ for $j < i$.*
- *$rank(x)$ is greater than $rank(y)$ iff there exists an index $0 \leq i \leq N-1$ such that $x_i > y_i$ and $x_j = y_j$ for $j < i$.*

We add one parameter to the *rank*() function to indicate how many columns of the covering array to take into account for computing the rank. So, *rank*(x, s) computes the rank of the covering array x considering only the first s columns of x. If x has r columns, the values of *rank*$(x, 1)$, *rank*$(x, 2)$, ..., *rank*(x, r) will be different. This is done to allow the comparison of the ranks of two covering arrays x and y with the same number of rows and order but with different number of columns.

The strategy followed by this smarter algorithm consists in constructing the permutations of columns one column at a time and checking the $v!$ relabelings for the new column appended to the permutation of columns. Let be $A_0, A_1, \ldots, A_{r-1}$ the r columns of the covering array under testing A, and let be $\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{v!-1}$ the $v!$ relabelings for one column of A, where relabeling ε_i corresponds to the i-th permutation of the symbols $0, 1, \ldots, v-1$ in lexicographic order. Relabeling one column A_j using $\varepsilon_i = (\varepsilon_{i_0}\ \varepsilon_{i_1}\ \cdots\ \varepsilon_{i_{v-1}})$ implies, for each $u \in \mathbb{Z}_v$, to replace the occurrences of the symbol u in A_j by the symbol ε_{i_u}. For example consider $v = 3$; Table 4.4 shows the $3! = 6$ possible relabelings for each column of a covering array of order three; to relabel the column j of a covering array of order $v = 3$ according to ε_4 implies to replace in column j the 0's by 2's, the 1's by 0's, and the 2's by 1's.

The current permutation of columns π will be represented by a list of indices $0 \leq j \leq r-1$, where $\pi_i = j$ means that column A_j is the i-th column of the covering array resulting from permuting the columns of A according to π. At the beginning of the process π is empty, $\pi = (\,)$, and since

A has r columns there are r possible columns to append to the permutation π. The order in which π is expanded is controlled by a permutation $\rho = (\rho_0\ \rho_1\ \cdots\ \rho_{r-1})$ of the column indices $0, 1, \ldots, r-1$, where r is the number of columns of A. Thereby, the first partial permutation of columns is $\pi = (\pi_0) = (\rho_0)$.

Let be D the covering array conformed by the columns of A taken in the order in which its indices are in π (at the beginning $D = (A_{\rho_0})$). Over the last column of D the $v!$ distinct relabelings $\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{v!-1}$ start to be performed sequentially. After each relabeling, the rows of D are sorted and the rank of D is compared with the rank of the first s columns of A, where s is the number of columns of the covering array D. The purpose of these comparisons is to check if the first s columns of A are a of minimum rank. There are three possible results for each relabeling ε_i of the last column of D:

1. $\mathit{rank}(D) < \mathit{rank}(A, s)$. In this case A is not of minimum rank because its first s columns are not of minimum rank, so the verification process terminates.

2. $\mathit{rank}(D) = \mathit{rank}(A, s)$. Relabeling ε_i is appended to a list L_{s-1} of the relabelings of column π_{s-1} that produce a covering array D with rank equal to the rank of the first s columns of A.

3. $\mathit{rank}(D) > \mathit{rank}(A, s)$. Relabeling ε_i is not stored in the list L_{s-1}.

As soon as one relabeling produces $\mathit{rank}(D) < \mathit{rank}(A, s)$ the verification process concludes that A is not of minimum rank. If no such relabeling exists then all the $v!$ relabelings are performed

Relabeling	Symbol Permutation
ε_0	(0 1 2)
ε_1	(0 2 1)
ε_2	(1 0 2)
ε_3	(1 2 0)
ε_4	(2 0 1)
ε_5	(2 1 0)

Table 4.4: The $3! = 6$ possible relabelings for a column of a covering array of order $v = 3$.

because with s columns is not possible to determine if A is of minimum rank. The relabelings which produce $rank(D) = rank(A, s)$ are stored because for each of them all the possible expansions of the current permutation of columns π must be checked to determine if A is of minimum rank. The algorithm employs r lists L_0, L_1, ..., L_{r-1} to store respectively the relabelings of the columns π_0, π_1, ..., π_{r-1} for which $rank(D) = rank(A, s)$.

Consider the initial permutation of columns $\pi = (\pi_0) = (\rho_0)$ and suppose that the cardinality of the list L_0, denoted as $|L_0|$, is greater than zero; so there was at least one relabeling for column π_0 which makes $rank(D) = rank(A, 1)$. Let be $L_0 = (\varepsilon_0^0, \varepsilon_1^0, \ldots, \varepsilon_{n_0-1}^0)$, where ε_i^0 denotes the i-th relabeling stored in L_0 and $n_0 = |L_0|$. For each relabeling ε_i^0 the current permutation of columns $\pi = (\rho_0)$ is expanded to the permutations of columns $(\rho_0\ \rho_1)$, $(\rho_0\ \rho_2)$, ..., $(\rho_0\ \rho_{r-1})$, as shown in the search tree of Figure 4.14.

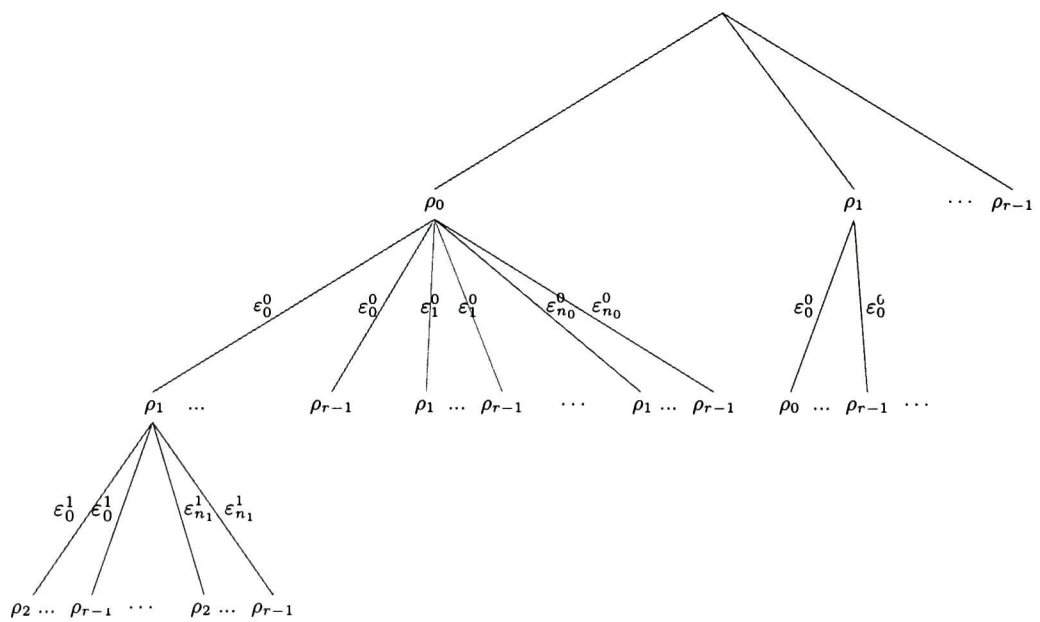


Figure 4.14: Search tree to verify if a covering array is of minimum rank. For all relabelings of the column ρ_j that make $rank(B) = rank(A, s)$ all the possible following columns are checked.

On the other hand, suppose that $rank(D) > rank(A, 1)$ for all the $v!$ relabelings of column ρ_0,

then $|L_0| = 0$. In this case, we do not expand the permutation of columns $\pi = (\rho_0)$ anymore because all the permutations starting with column ρ_0, $\pi = (\rho_0 \ * \ * \ \cdots \ *)$, will produce an array D with a rank greater than the rank of A up to $s = 1$ columns. As a consequence, a great number of permutations of columns, plus their associated column relabelings are not checked (a pruning in the search tree of Figure 4.14).

The search tree of Figure 4.14 is explored in a depth-first search manner; that is, the algorithm expands one permutation of s columns to $s+1, s+2, \ldots$ columns before than considering the other permutations of s columns. In the same way, the algorithm considers a relabeling ε_i^j of column π_j only after all the permutations of columns have been checked for the relabeling ε_{i-1}^j. At each time of the process, the first $s-1$ columns of the covering array D, which are $D_0, D_1, \ldots, D_{s-2}$, are relabeled according to one relabeling of the lists L_0, L_1, ..., L_{s-2} respectively. In every node of the search three is done the operation of row sorting, which is equivalent to the $N!$ permutation of rows; so the search tree contemplates the three symmetries of the covering arrays.

Retaking the example for the initial permutation of columns $\pi = (\rho_0)$, suppose that $|L_0| > 0$ and that the algorithm expands the permutation π to the first permutation with two columns $\pi = (\rho_0 \ \rho_1)$. In the following list are described the steps done to explore the search tree of Figure 4.14:

- The relabeling ε_0^0 is applied to column $\pi_0 = \rho_0$, and the $v!$ relabelings are preformed over the column recently added $\pi_1 = \rho_1$ without modifying column π_0.

- As done for column $\pi_0 = \rho_0$, if one relabeling of $\pi_1 = \rho_1$ makes $\mathit{rank}(D) < \mathit{rank}(A, 2)$, then A is not of minimum rank because its subarray $(A_0 \ A_1)$ is not of minimum rank. If all the relabelings produce $\mathit{rank}(D) > \mathit{rank}(A, 2)$, then the permutation $(\rho_0 \ \rho_1)$ is not expanded any more and the process continues with the permutation of columns $(\rho_0 \ \rho_2)$. If $\mathit{rank}(D) = \mathit{rank}(A, 2)$ for some relabelings of ρ_1, then these relabelings are stored in the list L_1 as shown in Figure 4.15.

- If $|L_1| > 0$ then the algorithm tries to expand the current permutation of columns $\pi = (\rho_0 \ \rho_1)$

to $s = 3$ columns. The algorithm relabels columns D_0 and D_1 according to the first relabelings in L_0 and L_1, which are ε_0^0 and ε_0^1 respectively. With D_0 and D_1 fixed, the next step is to append the column $\pi_2 = \rho_2$ to the permutation π and perform the $v!$ relabelings over π_2.

- Suppose that $\mathit{rank}(D) > \mathit{rank}(A, 3)$ for all relabelings of column $\pi_2 = \rho_2$, then the permutation $\pi = (\rho_0\ \rho_1\ \rho_2)$ is truncated and the algorithm takes the following candidate column ρ_3 as the new column for π_2; so the current permutation of columns is now $\pi = (\rho_0\ \rho_1\ \rho_3)$, and the $v!$ relabelings are performed over $\pi_2 = \rho_3$. The same is done for the other candidate columns $\rho_4, \rho_5, \ldots, \rho_{r-1}$. After exhausting the search for relabeling ε_0^1 of column $\pi_1 = \rho_1$, the algorithm takes the following relabeling in column L_1 (which is ε_1^1) and applies it to D_1.

- With column D_1 relabeled according to ε_1^1 the algorithm again tries to expand the permutation π to $s = 3$ columns by checking the columns $\rho_2, \rho_3, \ldots, \rho_{r-1}$ as candidates for the third column of the permutation. The same is repeated for the relabelings $\varepsilon_2^1, \varepsilon_3^1, \ldots, \varepsilon_{n_1-1}^1$.

- At this point only the permutation $\pi = (\rho_0\ \rho_1)$ has been checked, so the process continues with the permutation $\pi = (\rho_0\ \rho_2)$. Over the last column added, $\pi_1 = \rho_2$, the $v!$ relabelings are performed, and those producing $\mathit{rank}(D) = \mathit{rank}(A, 2)$ are stored in the list L_1. So the content of the list L_1 varies depending of the current relabeling for column π_0 and of the current column π_1. For the new relabelings in list L_1 the process to expand the current permutation

	L_0	L_1	L_2	$\cdots$	L_{r-1}
0	ε_0^0	ε_0^1			
1	ε_1^0	ε_1^1			
	$\vdots$	$\vdots$			
$\vdots$	$\varepsilon_{n_0-1}^0$	$\varepsilon_{n_1-1}^1$			
$v! - 1$					

Figure 4.15: L_0 and L_1 store respectively the relabelings of the columns π_0 and π_1 that make $\mathit{rank}(D) = \mathit{rank}(A, s)$. The relabelings in L_1 are dependent of the current relabeling of column π_0.

of columns to $s = 3$ columns is done as explained before.

- When all the permutations for $s = 2$ columns $(\rho_0\ \rho_1)$, $(\rho_0\ \rho_2)$, ..., $(\rho_0\ \rho_{r-1})$ are checked, the subtree for the first relabeling of column π_0 has been exhausted. To continue, the following relabeling in list L_0, which is ε_1^0, is applied to column π_0, and the entire process is repeated. That is, all the permutations π with $s = 2$ columns $(\rho_0\ \rho_1)$, $(\rho_0\ \rho_2)$, ..., $(\rho_0\ \rho_{r-1})$ are checked, but now the column $\pi_0 = \rho_0$ is relabeled according to ε_1^0.

- After repeating the process for the subsequent relabelings in L_0, ε_2^0, ε_3^0, ..., $\varepsilon_{n_0-1}^0$, the first column of the permutation $\pi_0 = \rho_0$ is changed to $\pi_0 = \rho_1$. Now, the entire process done for $\pi = (\rho_0)$ is repeated for $\pi = (\rho_1)$; for example if at least one relabeling for the current column π_0 makes $\mathit{rank}(D) = \mathit{rank}(A, 1)$, then the current permutation $\pi = (\rho_1)$ is extended to all the permutations with $s = 2$ columns $(\rho_1\ \rho_0), (\rho_1\ \rho_2), \ldots, (\rho_1\ \rho_{r-1})$. In the subsequent steps π_0 gets the columns ρ_2, ρ_3, ..., ρ_{r-1}. And when all the relabelings for $\pi_0 = \rho_{r-1}$ are checked the verification process finishes.

The search tree for the verification process may be very large, but in the worst case it checks the $(v!)^r$ column relabelings for each one of the $r!$ column permutations, as done by the naive verification algorithm (Algorithm 11), but has the advantage of reducing the $N!$ row permutations to a sorting of the rows done in $O(N^2)$. Furthermore, any time a relabeling for column π_j produces $\mathit{rank}(D) < \mathit{rank}(A, s)$, the verification process terminates and reports that A is not of minimum rank because its first s columns are not of minimum rank. In other case, it will be uncommon that all relabelings for column π_j produce $\mathit{rank}(D) = \mathit{rank}(A, s)$, and therefore some relabelings for the current permutation of columns are not explored. But the major time savings occur when $\mathit{rank}(D) > \mathit{rank}(A, s)$ for the $v!$ relabelings of the column recently added to the current permutation of columns, because in this case a large portion of the tree may be pruned. Anyway, the worst case of the algorithm is $O(N^2\, k!\, (v!)^k)$.

Algorithm 12 implements this smarter verification algorithm. The algorithm is controlled by the

Algorithm 12 is_minimum(A, r)

Require: one covering array A with r columns
Ensure: **true** if A is of minimum rank; **false** otherwise

```
1: ρ ← a permutation of the symbol set {0, 1, ..., v − 1} to define the order of expansion
2: π ← ( )
3: L_0, L_1, ..., L_{r−1} are empty lists
4: s ← 0
5: replace_column ← true
6: while s ≥ 0 do
7:    if replace_column then
8:       π_s ← next_candidate_column(ρ, s)
9:       D ← column_permutation(A, π)
10:      for all relabelings ε of the last column of D do
11:         sort_rows(D)
12:         if rank(D) < rank(A, s) then
13:            return false
14:         else if rank(D) = rank(A, s) then
15:            append(L_s, ε)
16:         end if
17:      end for
18:   end if
19:   if |L_s| > 0 then
20:      ε ← remove_first(L_s)
21:      relabel_column(D, s, ε)
22:      sort_rows(D)
23:      replace_column ← true
24:      s ← s + 1
25:   else
26:      if remains_candidate_columns(ρ, s) then
27:         replace_column ← true
28:      else
29:         replace_column ← false
30:         s ← s − 1
31:      end if
32:   end if
33: end while
34: return true
```

while loop of line 6. The algorithm expands and reduces the current permutation of columns π until an array D with smaller rank than A is found, or until the search tree is exhausted. Within the

while loop there are two main parts, the **if** sentence of lines 7-18 and the **if-else** sentence of lines 19-32. The boolean variable $replace\_column$ controls the execution of the sentences in the **if** of line 7. Each time the variable $replace\_column$ is **true** the following candidate column for position s of π is added to the current permutation of columns, the $v!$ relabelings are performed over it, and those producing $rank(D) = rank(A, s)$ are stored in the list L_s.

The **if** part of the **if-else** sentence in line 19 is executed for each relabeling $\varepsilon \in L_s$; in this block of sentences the last column of array D is relabeled with each of the relabelings in L_s (one at a time), the value of s is incremented in one unit, and the variable $replace\_column$ is set to **true** in order to check the first candidate column for position $s + 1$ in the next iteration of the **while** loop. The function *remove_first*() deletes and returns the first element of the list L_s received as a parameter. When all relabelings in list L_s are checked, the **else** block in line 25 is executed. In this block is asked if there are more candidate columns for the current position s; in the affirmative case the variable $replace\_column$ is set to **true**; but in other case all the candidate columns have been checked, and so the variable $replace\_column$ is set to **false**, and the algorithm backtracks to column $s - 1$. When $replace\_column$ is **false** the current column in π_s is not replaced in the current iteration of the **while** loop; instead, the following relabeling for that column is checked in the **if** part of the **if-else** sentence of line 19.

4.3.5 Examples of Non-Isomorphic Covering Arrays

As an example of the non-isomorphic covering arrays generated by the NonIsoCA algorithm consider the three covering arrays of Figure 4.16. These are all the non-isomorphic covering arrays of minimum rank for $N = 6$, $k = 7$, $t = 2$, and $v = 2$. Any other covering array for such parameters is isomorphic to exactly one of these covering arrays. Therefore, it is sufficient to check only these three covering arrays as base of the TCA to guarantee the construction of the highest TCA.

Figure 4.17 shows another example of non-isomorphic covering arrays. In this case $N = 6$, $k = 5$, $t = 2$, and $v = 2$. Table 4.5 shows the cardinality of the seven classes in which the set of all covering

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}; \quad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}; \quad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

Figure 4.16: The three non-isomorphic covering arrays of minimum rank CA$(6; 2, 7, 2)$. Any other covering array for $N = 6$, $k = 7$, $t = 2$, and $v = 2$, is isomorphic to exactly one of these arrays.

arrays CA$(6; 2, 5, 2)$ is partitioned. The representative of minimum rank of these classes are the covering arrays $A, B, \ldots, G$ of Figure 4.17.

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}; \quad B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}; \quad C = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}; \quad D = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix};$$

$$E = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix}; \quad F = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}; \quad G = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Figure 4.17: The seven non-isomorphic covering arrays of minimum rank CA$(6; 2, 5, 2)$.

4.4 Chapter Summary

In this chapter was described the methodology proposed to construct the TCAs given a combination of the parameters N, t, k, and v, and also were described the algorithms to efficiently implement the methodology. The methodology consists in the following three main steps: (1) to generate all the non-isomorphic covering arrays of minimum rank CA$(N; t, k, v)$ with the objective of using

Class	Representative of minimum rank	Distinct elements
1	A	23,040
2	B	230,400
3	C	172,800
4	D	691,200
5	E	345,600
6	F	345,600
7	G	276,480

Table 4.5: Cardinality of the seven classes of isomorphic covering arrays in the universe of the 6×5 matrices over $\{0, 1\}$. The representative of minimum rank of the classes are the covering arrays $A, B, \ldots, G$ of Figure 4.17.

these arrays as the bases of the TCAs; (2) to apply iteratively the construction $\mathcal{E}$ to each one of the above covering arrays; and (3) to take at the end of the process the TCA with the greatest height among all the constructed in the process. The motivation to generate TCAs of maximum height is that the more greater the height of the TCA the more competitive are the covering arrays in the TCA. To generate the TCAs the construction $\mathcal{E}$ is applied to every matrix δ for the base covering array; the computational work required for this operation can be reduced if the matrices δ are represented as linear vectors and if these vectors are generated in v-ary Gray code. Finally, the algorithm to generate the non-isomorphic bases constructs the non-isomorphic covering arrays extending the current subarray one column at a time, discarding the arrays which are not of minimum rank; every time a subarray with k columns is constructed the algorithm reports a new non-isomorphic covering array for the input parameters N, t, k, and v. The next chapter shows the computational results for the methodology developed in this chapter.

5

Computational Results

This chapter presents the relevant results of the proposed approach to construct covering arrays. Section 5.1 introduces a very important result of this thesis: the infinite TCAs, which are TCAs that can grow infinitely. To construct TCAs other than the infinite ones we perform a computational search taking into account the combinations of the values of N, t, k, and v for which it was possible to construct the non-isomorphic bases CA($N; t, k, v$); Section 5.2 describes the relevant TCAs constructed in the computational experimentation, i.e., the TCAs conformed by covering arrays that improve or equal a current upper bound. In addition to the construction of TCAs, a computational experimentation for the NonIsoCA algorithm was performed in Section 5.3; the most important result of this experimentation was the determination of the exact value of five lower bounds which had not been found before neither by computational search nor by algebraic analysis. Finally, Section 5.4 provides an empirical estimation of the execution time of the functions *apply_*$\mathcal{E}$() and *is_minimum*() introduced in Chapter 4.

5.1 Infinite Towers

An infinite TCA is a TCA that can grow infinitely, or in other words it is always possible to extend the top of the TCA to a covering array with one more unit of strength using the construction $\mathcal{E}$. To see how this is possible, let us introduce the covering arrays of strength one: a covering array $\mathsf{CA}(N; 1, k, v)$ of strength $t = 1$ is a matrix that contains in every column the symbols $0, 1, \ldots, v-1$ at least once. For example the next covering array $\mathsf{CA}(4; 1, 2, 4)$ is a covering of strength $t = 1$.

$$\mathsf{CA}(4; 1, 2, 4) = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{pmatrix}$$

A covering array of strength $t = 1$ can have any number of columns $k \geq 1$, with the condition that every column contains the symbols $0, 1, \ldots, v-1$ at least once. If the order is v then the optimal number of rows N for a covering array of strength $t = 1$ is $N = v$; in this case every column is a permutation of the symbol set $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$. For the infinite TCAs of this section the bases are optimal covering arrays of strength $t = 1$ and $k = 2$ columns, like the covering array $\mathsf{CA}(4; 1, 2, 4)$ previously shown. This way, for any order $v \geq 2$ the bases of the infinite TCAs are of the form $\mathsf{CA}(v; 1, 2, v)$.

To extend the covering array $\mathsf{CA}(v; 1, 2, v)$ of strength $t = 1$ to a covering array of strength $t = 2$ the normal procedure will apply the construction $\mathcal{E}$ to all vectors δ of length $k(v-1) = 2(v-1)$, generated in v-ary Gray code, until find one vector δ that produces a covering array $\mathsf{CA}(v^2; 2, 3, v)$ of strength $t = 2$. However, it is not necessary to generate all vectors δ, because for any covering array $\mathsf{CA}(v^t; t, t+1, v)$ of strength t in an infinite tower, there exists one vector δ that always produce a covering array $\mathsf{CA}(v^{t+1}; t+1, t+2, v)$ of strength $t+1$. This vector δ of length $k(v-1)$, where $k = t+1$, is the vector δ in which its i-th element δ_i, $0 \leq i \leq k(v-1)-1$, is equal to:

$$\delta_i = \begin{cases} \lceil i/k \rceil & \text{if } i+1 \equiv 0 \pmod{k} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

For example, the vector δ to expand the covering array CA$(4;1,2,4)$ of strength $t=1$ to a covering array CA$(16;2,3,4)$ of strength $t=2$ is the vector $\delta = (0\ 1\ 0\ 2\ 0\ 3)$. This covering array CA$(16;2,3,4)$ of strength $t=2$ is shown next:

$$\text{CA}(16;2,3,4) = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 2 & 0 \\ 3 & 3 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 1 \\ 2 & 3 & 1 \\ 3 & 0 & 1 \\ 0 & 2 & 2 \\ 1 & 3 & 2 \\ 2 & 0 & 2 \\ 3 & 1 & 2 \\ 0 & 3 & 3 \\ 1 & 0 & 3 \\ 2 & 1 & 3 \\ 3 & 2 & 3 \end{pmatrix}$$

Taking this covering array of strength $t=2$ as the base covering array, the vector δ that produces a covering array of strength $t=3$, according to (5.1), is the vector $\delta = (0\ 0\ 1\ 0\ 0\ 2\ 0\ 0\ 3)$, and the covering array produced is CA$(64;3,4,4)$. Table 5.1 shows the first 10 covering arrays in the infinite TCA for $v=4$. To the right of each covering array is the vector δ for which the construction $\mathcal{E}$ produces the covering array to the left. As another example of infinite TCA the Table 5.2 shows the first 10 covering arrays of the infinite TCA for $v=2$; in this case only the last column of the base covering array is translated to produce the following covering array of the TCA.

In Section 4.1 it was mentioned that the increment in the number of rows in the covering arrays of a TCA is the minimum possible if the base covering array is optimal. For the infinite TCAs we

Covering Arrays	Vectors δ
⋮	⋮
CA(1048576; 10, 11, 4)	(0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 3)
CA(262144; 9, 10, 4)	(0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 3)
CA(65536; 8, 9, 4)	(0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 3)
CA(16384; 7, 8, 4)	(0 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 3)
CA(4096; 6, 7, 4)	(0 0 0 0 0 1 0 0 0 0 0 0 2 0 0 0 0 0 3)
CA(1024; 5, 6, 4)	(0 0 0 0 1 0 0 0 0 2 0 0 0 0 3)
CA(256; 4, 5, 4)	(0 0 0 1 0 0 0 2 0 0 0 3)
CA(64; 3, 4, 4)	(0 0 1 0 0 2 0 0 3)
CA(16; 2, 3, 4)	(0 1 0 2 0 3)
CA(4; 1, 2, 4)	–

Table 5.1: Infinite tower for $v = 4$.

Covering Arrays	Vectors δ
⋮	⋮
CA(1024; 10, 11, 2)	(0 0 0 0 0 0 0 0 0 1)
CA(512; 9, 10, 2)	(0 0 0 0 0 0 0 0 1)
CA(256; 8, 9, 2)	(0 0 0 0 0 0 0 1)
CA(128; 7, 8, 2)	(0 0 0 0 0 0 1)
CA(64; 6, 7, 2)	(0 0 0 0 0 1)
CA(32; 5, 6, 2)	(0 0 0 0 1)
CA(16; 4, 5, 2)	(0 0 0 1)
CA(8; 3, 4, 2)	(0 0 1)
CA(4; 2, 3, 2)	(0 1)
CA(2; 1, 2, 2)	–

Table 5.2: Infinite tower for $v = 2$.

have that the base covering array of strength $t = 1$ is optimal; so for each $v \geq 2$ the infinite TCA of order v is composed by an infinite number of optimal covering arrays.

Some coverings arrays in the infinite TCAs are also the best known ones with respect to the number of columns. One covering array that is optimum in the number of rows is not necessarily optimum in the number of columns. For example the covering array CA$(9; 2, 3, 3)$, which is the second member of the infinite TCA for $v = 3$, is optimal with respect to the number of rows, because there not exists a covering array with $N < 9$ rows for $t = 2$, $k = 3$, and $v = 3$. However, this covering array is not the best known with respect to the number of columns, because there exists the covering array CA$(9; 2, 4, 3)$, which has one more column. Next are given the covering arrays of strength $2 \leq t \leq 6$ in the infinite TCAs for $2 \leq v \leq 25$ that equal a best known covering arrays with respect to the number of columns. The current best known covering arrays were taken from the covering arrays tables at [25]:

CA$(4; 2, 3, 2)$,
CA$(8; 3, 4, 2)$,
CA$(16; 4, 5, 2)$,
CA$(32; 5, 6, 2)$,
CA$(64; 6, 7, 2)$,
CA$(27; 3, 4, 3)$,
CA$(81; 4, 5, 3)$,
CA$(243; 5, 6, 3)$,
CA$(729; 6, 7, 3)$,
CA$(256; 4, 5, 4)$
CA$(1024; 5, 6, 4)$,
CA$(4096; 6, 7, 4)$,
CA$(3125; 5, 6, 5)$,
CA$(15625; 6, 7, 5)$,
CA$(36; 2, 3, 6)$,
CA$(216; 3, 4, 6)$,
CA$(1296; 4, 5, 6)$,
CA$(7776; 5, 6, 6)$,
CA$(46656; 6, 7, 6)$,
CA$(1000; 3, 4, 10)$,
CA$(10000; 4, 5, 10)$,
CA$(100000; 5, 6, 10)$,
CA$(1000000; 6, 7, 10)$,
CA$(20736; 4, 5, 12)$,
CA$(248832; 5, 6, 12)$,
CA$(2985984; 6, 7, 12)$,
CA$(2744; 3, 4, 14)$,
CA$(38416; 4, 5, 14)$,
CA$(537824; 5, 6, 14)$,
CA$(7529536; 6, 7, 14)$,
CA$(50625; 4, 5, 15)$,
CA$(759375; 5, 6, 15)$,
CA$(11390625; 6, 7, 15)$,
CA$(5832; 3, 4, 18)$,
CA$(104976; 4, 5, 18)$,
CA$(1889568; 5, 6, 18)$,
CA$(34012224; 6, 7, 18)$,
CA$(160000; 4, 5, 20)$,
CA$(3200000; 5, 6, 20)$,
CA$(64000000; 6, 7, 20)$,
CA$(194481; 4, 5, 21)$,
CA$(4084101; 5, 6, 21)$,
CA$(85766121; 6, 7, 21)$,
CA$(10648; 3, 4, 22)$,
CA$(234256; 4, 5, 22)$,
CA$(5153632; 5, 6, 22)$,
CA$(113379904; 6, 7, 22)$,
CA$(331776; 4, 5, 24)$,
CA$(7962624; 5, 6, 24)$,
CA$(191102976; 6, 7, 24)$.

5.2 TCAs from Non-Isomorphic Bases

This section describes the computational experimentation done to prove the methodology developed in Chapter 4 and the relevant TCAs constructed. Section 5.2.1 shows the combinations of the parameters N, t, k, and v for the computational experimentation; Section 5.2.2 describe the TCAs that allow the improvement of a current upper bound; and Section 5.2.3 describes the TCAs that allow to equal one or more current upper bounds.

5.2.1 Computational Experimentation

The algorithms developed in Chapter 4 were coded in two programs in standard C. The first program implements the NonIsoCA algorithm to generate all the non-isomorphic covering arrays of minimum rank CA($N; t, k, v$). The second program implements the iterative application of the construction $\mathcal{E}$ to generate the TCAs. The first program was named `NonIsoCA.c` and the second program was named `TCA.c`.

The program `NonIsoCA.c` receives as input parameters one combination of the parameters N, t, k, and v. With these parameters the program `NonIsoCA.c` generates the non-isomorphic covering arrays of minimum rank CA($N; t, k, v$) and writes them into an output file named `N*k*t*v*.txt`, where each * is replaced by the particular value of N, t, k, and v respectively. The `NonIsoCA.c` program is invoked in the following way:

```
./NonIsoCA -N <rows> -k <columns> -t <strength> -v <order>
```

An example of a real invocation is:

```
./NonIsoCA -N 6 -k 7 -t 2 -v 2
```

The results for this invocation is the output file `N6k7t2v2.txt`, whose content is:

```
N 6, k 7, t 2, v 2
0 15 51 60 85 106
```

```
0 7 56 91 109 118
0 7 57 90 109 118
```

The first line of the file shows the value of the input parameters separated by commas. The remaining lines are the rank of the non-isomorphic covering arrays found, one covering array per line. Since we know the order and the number of columns of the covering arrays, it is possible to get the final matrices from their ranks.

The program `TCA.c` takes the output file of the program `NonIsoCA.c` and executes the function *apply_$\mathcal{E}$_rec*() (Algorithm 7, Section 4.2.4) over each covering array in the file. The covering arrays in the file are read one at a time, converted to their matrix form, and expanded iteratively by means of the construction $\mathcal{E}$. The filename of the file with the non-isomorphic bases is the only parameter for the program `TCA.c`:

```
./TCA -file <filename>
```

In the following invocation, the program `TCA.c` receives the name of a file generated by the program `NonIsoCA.c` when it is executed with the parameters $N = 6$, $k = 7$, $t = 2$, and $v = 2$:

```
./TCA -file N6k7t2v2.txt
```

The output of the program `TCA.c` is a file with the TCA of greatest height constructed in the execution. Both programs `NonIsoCA.c` and `TCA.c` were compiled with GCC 4.4.6 using the optimization flag -O3, and executed sequentially in 256 processors AMD Opteron™ 6274 (1.4 GHz) of the hybrid supercomputing cluster "Xiuhcoatl" at CINVESTAV.

Next are given the 315 combinations of parameters N, t, k, and v executed in the computational experimentation, together with the height (denoted as Max h) of the greatest TCA constructed for each parameter combination. The values for the parameters N, t, k, and v were selected based on the viability in terms of the execution time of using the NonIsoCA algorithm to construct all non-isomorphic bases for the given parameters:

N	t	k	v	Max h
4	2	2	2	10
4	2	3	2	10
5	2	2	2	10
5	2	3	2	10
5	2	4	2	1
6	2	2	2	10
6	2	3	2	10
6	2	4	2	1
6	2	5	2	1
6	2	6	2	0
6	2	7	2	0
6	2	8	2	0
6	2	9	2	0
6	2	10	2	0
7	2	2	2	10
7	2	3	2	10
7	2	4	2	5
7	2	5	2	5
7	2	6	2	1
7	2	7	2	1
7	2	8	2	0
7	2	9	2	0
7	2	10	2	0
7	2	11	2	0
7	2	12	2	0
8	2	2	2	10
8	2	3	2	10
8	2	4	2	10
8	2	5	2	5
8	2	6	2	5
8	2	7	2	5
8	2	8	2	5
8	2	9	2	5
8	2	10	2	5
8	2	11	2	5
8	2	12	2	0
9	2	2	2	10

N	t	k	v	Max h
9	2	3	2	10
9	2	4	2	10
9	2	5	2	5
9	2	6	2	5
9	2	7	2	5
9	2	8	2	5
9	2	9	2	5
9	2	10	2	5
9	2	11	2	5
9	2	12	2	0
10	2	2	2	10
10	2	3	2	10
10	2	4	2	10
10	2	5	2	10
10	2	6	2	5
10	2	7	2	5
10	2	8	2	5
10	2	9	2	5
10	2	10	2	5
10	2	11	2	5
10	2	12	2	1
10	2	13	2	0
11	2	2	2	10
11	2	3	2	10
11	2	4	2	10
11	2	5	2	10
11	2	6	2	5
11	2	7	2	5
11	2	8	2	5
11	2	9	2	5
11	2	10	2	5
11	2	11	2	5
11	2	12	2	1
11	2	13	2	0
12	2	2	2	10
12	2	3	2	10
12	2	4	2	10

N	t	k	v	Max h
12	2	5	2	10
12	2	6	2	5
12	2	7	2	5
12	2	8	2	5
12	2	9	2	5
12	2	10	2	5
12	2	11	2	5
12	2	12	2	1
12	2	13	2	1
13	2	2	2	10
13	2	3	2	10
13	2	4	2	10
13	2	5	2	10
13	2	6	2	5
13	2	7	2	5
13	2	8	2	5
13	2	9	2	5
13	2	10	2	5
13	2	11	2	5
13	2	12	2	1
13	2	13	2	1
14	2	2	2	10
14	2	3	2	10
14	2	4	2	10
14	2	5	2	10
14	2	6	2	5
14	2	7	2	5
14	2	8	2	5
14	2	9	2	5
14	2	10	2	5
14	2	11	2	5
14	2	12	2	1
14	2	13	2	1
15	2	2	2	10
15	2	3	2	10
15	2	4	2	10
15	2	5	2	10

N	t	k	v	Max h
15	2	6	2	5
15	2	7	2	5
15	2	8	2	5
15	2	9	2	5
15	2	10	2	5
15	2	11	2	5
15	2	12	2	1
15	2	13	2	1
16	2	2	2	10
16	2	3	2	10
16	2	4	2	10
16	2	5	2	10
16	2	6	2	5
16	2	7	2	5
17	2	2	2	10
17	2	3	2	10
17	2	4	2	10
17	2	5	2	10
17	2	6	2	5
17	2	7	2	5
18	2	2	2	10
18	2	3	2	10
18	2	4	2	10
18	2	5	2	10
18	2	6	2	5
18	2	7	2	5
19	2	2	2	10
19	2	3	2	10
19	2	4	2	10
19	2	5	2	10
19	2	6	2	5
19	2	7	2	5
20	2	2	2	10
20	2	3	2	10
20	2	4	2	10
20	2	5	2	10
20	2	6	2	5

N	t	k	v	Max h
20	2	7	2	5
21	2	2	2	10
21	2	3	2	10
21	2	4	2	10
21	2	5	2	10
21	2	6	2	5
21	2	7	2	5
8	3	3	2	10
8	3	4	2	10
9	3	3	2	10
9	3	4	2	10
10	3	3	2	10
10	3	4	2	10
10	3	5	2	0
11	3	3	2	10
11	3	4	2	10
11	3	5	2	1
12	3	3	2	10
12	3	4	2	10
12	3	5	2	2
12	3	6	2	1
12	3	7	2	1
12	3	8	2	1
12	3	9	2	1
12	3	10	2	1
12	3	11	2	1
13	3	3	2	10
13	3	4	2	10
13	3	5	2	2
13	3	6	2	1
13	3	7	2	1
13	3	8	2	1
13	3	9	2	1
13	3	10	2	1
13	3	11	2	1
14	3	3	2	10
14	3	4	2	10
14	3	5	2	4
14	3	6	2	4
14	3	7	2	1
14	3	8	2	1
14	3	9	2	1
14	3	10	2	1
14	3	11	2	1
15	3	3	2	10
15	3	4	2	10
15	3	5	2	10
15	3	6	2	4
15	3	7	2	1
15	3	8	2	1
15	3	9	2	1
15	3	10	2	1
15	3	11	2	1
15	3	12	2	0
16	3	3	2	10
16	3	4	2	10
16	3	5	2	10
16	3	6	2	8
16	3	7	2	4
16	4	4	2	10
16	4	5	2	10
17	4	4	2	10
17	4	5	2	10
18	4	4	2	10
18	4	5	2	10
19	4	4	2	10
19	4	5	2	10
20	4	4	2	10
20	4	5	2	10
21	4	4	2	10
21	4	5	2	10
21	4	6	2	1
22	4	4	2	10
22	4	5	2	10

N	t	k	v	Max h
22	4	6	2	1
23	4	4	2	10
23	4	5	2	10
23	4	6	2	1
24	4	4	2	10
24	4	5	2	10
24	4	6	2	1
24	4	7	2	0
24	4	8	2	0
24	4	9	2	0
24	4	10	2	0
24	4	11	2	0
24	4	12	2	0
9	2	2	3	10
9	2	3	3	10
9	2	4	3	0
10	2	2	3	10
10	2	3	3	10
10	2	4	3	0
11	2	2	3	10
11	2	3	3	10
11	2	4	3	1
11	2	5	3	1
12	2	2	3	10
12	2	3	3	10
12	2	4	3	1
12	2	5	3	1
12	2	6	3	0
12	2	7	3	0
13	2	2	3	10
13	2	3	3	10
13	2	4	3	3
13	2	5	3	1
13	2	6	3	0
13	2	7	3	0
14	2	2	3	10
14	2	3	3	10
14	2	4	3	3
14	2	5	3	1
14	2	6	3	0
14	2	7	3	0
15	2	2	3	10
15	2	3	3	10
15	2	4	3	3
15	2	5	3	3
15	2	6	3	0
15	2	7	3	0
16	2	2	3	10
16	2	3	3	10
16	2	4	3	3
16	2	5	3	3
16	2	6	3	1
16	2	7	3	0
17	2	2	3	10
17	2	3	3	10
17	2	4	3	3
17	2	5	3	3
17	2	6	3	1
17	2	7	3	0
18	2	2	3	10
18	2	3	3	10
18	2	4	3	3
18	2	5	3	3
18	2	6	3	1
18	2	7	3	0
16	2	2	4	10
16	2	3	4	5
16	2	4	4	0
16	2	5	4	0
17	2	2	4	10
17	2	3	4	5
17	2	4	4	0
17	2	5	4	0
18	2	2	4	10

N	t	k	v	Max h
18	2	3	4	5
18	2	4	4	1
18	2	5	4	0
19	2	2	4	10
19	2	3	4	5
19	2	4	4	1
19	2	5	4	0
20	2	2	4	10
20	2	3	4	5
20	2	4	4	1

N	t	k	v	Max h
20	2	5	4	0
21	2	2	4	10
21	2	3	4	5
21	2	4	4	1
21	2	5	4	0
22	2	2	4	10
22	2	3	4	5
22	2	4	4	1
22	2	5	4	0

In Section 5.2.2 we show in more detail the TCAs of height greater than zero having at least one covering array that improve a current upper bound; and in Section 5.2.3 we show the TCAs of height greater that zero conformed by one or more covering arrays that equal a current upper bound.

5.2.2 Upper Bounds Improved

This section presents the TCAs that allow the improvement of a current upper bound. In total eleven current upper bounds were improved. Next are listed the TCAs that made possible the improvement:

- For $t = 7$, $k = 10$, and $v = 2$, the number of rows was lowered from $N = 274$ to $N = 224$, which is an improvement of 50 rows. The base for the TCA which made possible this improvement is the following covering array CA$(7; 2, 5, 2)$:

$$\text{CA}(7; 2, 5, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

From this covering array the iterative application of the construction $\mathcal{E}$ produces the TCA of height 5 shown in the left part of the next table:

TCA	Vector δ	Height	Best Known CA	Reference
CA(224; 7, 10, 2)	(0 0 0 0 0 1 1 0 1)	5	CA(274; 7, 10, 2)	[26]
CA(112; 6, 9, 2)	(0 1 1 0 0 0 1 0)	4	CA(108; 6, 9, 2)	[25]
CA(56; 5, 8, 2)	(0 0 1 1 0 1 0)	3	CA(52; 5, 8, 2)	[25]
CA(28; 4, 7, 2)	(0 0 0 1 1 1)	2	CA(24; 4, 7, 2)	[25]
CA(14; 3, 6, 2)	(1 1 0 0 1)	1	CA(12; 3, 6, 2)	[25]
CA(7; 2, 5, 2)	–	0	CA(6; 2, 5, 2)	[25]

In the right part of the table are the best known covering arrays reported for the same parameters t, k, and v of the covering arrays in the TCA at the left of the table. The column "Vector δ" shows the vector δ for which the construction $\mathcal{E}$ produced the covering array to the left, using the covering array in the row below as the base covering array. For example, with the base covering array CA(7; 2, 5, 2) and the vector $\delta = (1\ 1\ 0\ 0\ 1)$ the construction $\mathcal{E}$ produces the covering array CA(14; 3, 6, 2). The character "–" in the last rows indicates that the base covering array was not generated by the construction $\mathcal{E}$.

The reference [25] is the covering array tables maintained by C. J. Colbourn at `http://www.public.asu.edu/~ccolbou/src/tabby/catable.html`. The reference [26] contains the current upper bound for several combinations of t, k, and v. In these references is indicated the particular method used to construct the best known covering arrays.

- For $t = 7$, $k = 11$, and $v = 2$, the number of rows was lowered from $N = 386$ to $N = 256$, an improvement of 130 rows. The base covering array for the TCA that allow this improvement is the next covering array CA(8; 2, 6, 2):

$$\text{CA}(8; 2, 6, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

And this is the table with the TCA:

TCA	Vector δ	Height	Best Known CA	Reference
CA(256; 7, 11, 2)	(0 0 0 0 0 0 1 1 0 1)	5	CA(386; 7, 11, 2)	[26]
CA(128; 6, 10, 2)	(0 1 1 0 0 1 1 0 0)	4	CA(118; 6, 10, 2)	[25]
CA(64; 5, 9, 2)	(0 0 1 1 0 0 1 0)	3	CA(54; 5, 9, 2)	[25]
CA(32; 4, 8, 2)	(0 0 0 1 1 1 1)	2	CA(24; 4, 8, 2)	[25]
CA(16; 3, 7, 2)	(1 1 0 0 1 1)	1	CA(12; 3, 7, 2)	[25]
CA(8; 2, 6, 2)	–	0	CA(6; 2, 6, 2)	[25]

- For $t = 7$, $k = 12$, and $v = 2$, the number of rows was lowered from $N = 506$ to $N = 256$, an improvement of 250 rows. These are the base covering array CA(8; 2, 7, 2) and the TCA:

$$\text{CA}(8; 2, 7, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
CA(256; 7, 12, 2)	(0 0 0 0 0 0 1 1 1 0 1)	5	CA(506; 7, 12, 2)	[26]
CA(128; 6, 11, 2)	(0 1 1 0 0 1 1 1 0 0)	4	CA(118; 6, 11, 2)	[25]
CA(64; 5, 10, 2)	(0 0 1 1 0 1 0 1 0)	3	CA(56; 5, 10, 2)	[25]
CA(32; 4, 9, 2)	(0 0 0 1 1 1 1 1)	2	CA(24; 4, 9, 2)	[25]
CA(16; 3, 8, 2)	(1 1 0 0 1 0 0)	1	CA(12; 3, 8, 2)	[25]
CA(8; 2, 7, 2)	–	0	CA(6; 2, 7, 2)	[25]

- For $t = 7$, $k = 13$, and $v = 2$, the number of rows was lowered from $N = 634$ to $N = 256$, an improvement of 378 rows. Here is the base covering array CA(8; 2, 8, 2) and the TCA:

$$\mathrm{CA}(8;2,8,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathrm{CA}(256;7,13,2)$	$(0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,0\,1)$	5	$\mathrm{CA}(634;7,13,2)$	[26]
$\mathrm{CA}(128;6,12,2)$	$(0\,1\,1\,0\,0\,1\,1\,1\,1\,0\,0)$	4	$\mathrm{CA}(128;6,12,2)$	[25]
$\mathrm{CA}(64;5,11,2)$	$(0\,0\,1\,1\,0\,1\,0\,1\,1\,0)$	3	$\mathrm{CA}(64;5,11,2)$	[25]
$\mathrm{CA}(32;4,10,2)$	$(0\,0\,0\,1\,1\,1\,1\,0\,1)$	2	$\mathrm{CA}(24;4,10,2)$	[25]
$\mathrm{CA}(16;3,9,2)$	$(1\,1\,0\,0\,1\,0\,0\,1)$	1	$\mathrm{CA}(12;3,9,2)$	[25]
$\mathrm{CA}(8;2,8,2)$	–	0	$\mathrm{CA}(6;2,8,2)$	[25]

- For $t = 7$, $k = 14$, and $v = 2$, the number of rows was lowered from $N = 762$ to $N = 256$, an improvement of 506 rows. Below is the base covering array $\mathrm{CA}(8;2,9,2)$ and the TCA constructed from this base covering array:

$$\mathrm{CA}(8;2,9,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathrm{CA}(256;7,14,2)$	$(0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,0\,1)$	5	$\mathrm{CA}(762;7,14,2)$	[26]
$\mathrm{CA}(128;6,13,2)$	$(0\,1\,1\,0\,0\,1\,1\,1\,0\,1\,0\,0)$	4	$\mathrm{CA}(128;6,13,2)$	[25]
$\mathrm{CA}(64;5,12,2)$	$(0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,0)$	3	$\mathrm{CA}(64;5,12,2)$	[25]
$\mathrm{CA}(32;4,11,2)$	$(0\,0\,0\,1\,1\,1\,1\,0\,1\,1)$	2	$\mathrm{CA}(24;4,11,2)$	[25]
$\mathrm{CA}(16;3,10,2)$	$(1\,1\,0\,0\,1\,0\,0\,1\,1)$	1	$\mathrm{CA}(12;3,10,2)$	[25]
$\mathrm{CA}(8;2,9,2)$	–	0	$\mathrm{CA}(6;2,9,2)$	[25]

- For $t = 7$, $k = 15$, and $v = 2$, the number of rows was lowered from $N \geq 762$ to $N = 256$. The exact current bound for these values of t, k, v is not given explicitly in [26], but it must be greater than or equal to 762, which is the current upper bound for $t = 7$, $k = 14$, and $v = 2$. This follows from the inequality $\mathrm{CAN}(t, k, v) \geq \mathrm{CAN}(t, k-1, v)$. These are the base covering array $\mathrm{CA}(8; 2, 10, 2)$ and the TCA:

$$\mathrm{CA}(8; 2, 10, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathrm{CA}(256; 7, 15, 2)$	(0 0 0 0 0 0 1 1 1 1 1 1 0 1)	5	$\mathrm{CA}(\geq 762; 7, 15, 2)$	[26]
$\mathrm{CA}(128; 6, 14, 2)$	(0 1 1 0 0 1 1 1 0 0 1 0 0)	4	$\mathrm{CA}(128; 6, 14, 2)$	[25]
$\mathrm{CA}(64; 5, 13, 2)$	(0 0 1 1 0 1 0 1 1 1 1 0)	3	$\mathrm{CA}(64; 5, 13, 2)$	[25]
$\mathrm{CA}(32; 4, 12, 2)$	(0 0 0 1 1 1 1 0 1 0 1)	2	$\mathrm{CA}(24; 4, 12, 2)$	[25]
$\mathrm{CA}(16; 3, 11, 2)$	(1 1 0 0 1 0 0 1 1 1)	1	$\mathrm{CA}(12; 3, 11, 2)$	[25]
$\mathrm{CA}(8; 2, 10, 2)$	–	0	$\mathrm{CA}(6; 2, 10, 2)$	[25]

- For $t = 7$, $k = 16$, and $v = 2$, the number of rows was lowered from $N \geq 762$ to $N = 256$. Here is the base covering array $\mathrm{CA}(8; 2, 11, 2)$ and the TCA:

$$\mathrm{CA}(8; 2, 11, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
CA(256; 7, 16, 2)	(0 0 0 0 0 0 1 1 1 1 0 1 1 0 1)	5	CA($\geq$ 762; 7, 16, 2)	[26]
CA(128; 6, 15, 2)	(0 1 1 0 0 1 1 1 0 0 1 1 0 0)	4	CA(128; 6, 15, 2)	[25]
CA(64; 5, 14, 2)	(0 0 1 1 0 1 0 1 1 1 0 1 0)	3	CA(64; 5, 14, 2)	[25]
CA(32; 4, 13, 2)	(0 0 0 1 1 1 1 0 1 0 1 1)	2	CA(32; 4, 13, 2)	[25]
CA(16; 3, 12, 2)	(1 1 0 0 1 0 0 1 1 1 1)	1	CA(15; 3, 12, 2)	[25]
CA(8; 2, 11, 2)	–	0	CA(7; 2, 11, 2)	[25]

- The last TCA of in this section allows the improvement of the current upper bound for $\{t = 8, k = 11, v = 2\}$, $\{t = 9, k = 12, v = 2\}$, $\{t = 10, k = 13, v = 2\}$, and $\{t = 11, k = 14, v = 2\}$. These are the base covering array and the TCA:

$$\text{CA}(16; 3, 6, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
CA(4096; 11, 14, 2)	(0 0 0 0 0 0 1 1 1 1 1 1 0)	8	CA(5190; 11, 14, 2)	[26]
CA(2048; 10, 13, 2)	(0 0 0 0 0 0 0 0 0 1 1 1)	7	CA(2491; 10, 13, 2)	[26]
CA(1024; 9, 12, 2)	(0 0 0 0 1 1 0 0 0 0 1)	6	CA(1230; 9, 12, 2)	[26]
CA(512; 8, 11, 2)	(0 0 0 0 0 1 1 1 1 1)	5	CA(563; 8, 11, 2)	[26]
CA(256; 7, 10, 2)	(0 0 1 1 0 0 1 1 0)	4	CA(274; 7, 10, 2)	[25]
CA(128; 6, 9, 2)	(0 0 0 1 1 1 1 1)	3	CA(108; 6, 9, 2)	[25]
CA(64; 5, 8, 2)	(1 1 0 0 0 0 1)	2	CA(52; 5, 8, 2)	[25]
CA(32; 4, 7, 2)	(0 1 1 1 1 1)	1	CA(24; 4, 7, 2)	[25]
CA(16; 3, 6, 2)	–	0	CA(12; 3, 6, 2)	[25]

Notice that the covering array CA$(256; 7, 10, 2)$ at height 4 improves the current upper bound for $\{t = 7, k = 10, v = 2\}$; however the covering array CA$(224; 7, 10, 2)$ at height 5 of the first TCA given in this section is a better improvement of the current upper bound for $\{t = 7, k = 10, v = 2\}$.

Table 5.13 summarizes the current upper bounds improved by means of the TCAs described in this section.

t	k	v	Previous N	New N
7	10	2	274	224
7	11	2	386	256
7	12	2	506	256
7	13	2	634	256
7	14	2	762	256
7	15	2	$\geq$762	256
7	16	2	$\geq$762	256
8	11	2	563	512
9	12	2	1230	1024
10	13	2	2491	2048
11	14	2	5190	4096

Table 5.13: Upper bounds improved.

5.2.3 Upper Bounds Equaled

The TCAs presented in the previous section also contains some covering arrays that equal a current upper bound. Next are listed the covering arrays members of the TCAs of Section 5.2.2 that equal a current upper bound:

CA$(64; 5, 11, 2)$, CA$(128; 6, 12, 2)$, CA$(64; 5, 12, 2)$,
CA$(128; 6, 13, 2)$, CA$(64; 5, 13, 2)$, CA$(128; 6, 14, 2)$,
CA$(32; 4, 13, 2)$, CA$(64; 5, 14, 2)$, CA$(128; 6, 15, 2)$.

In addition to the TCAs of the previous section, there were constructed other TCAs whose covering array at the top equals a current upper bound. These TCAs are given in the following list. The items of the list are the parameters N, t, k, and v of the covering arrays at the base of the TCAs. We use the current upper bounds at [25] for comparison:

- $N = 5$, $k = 4$, $t = 2$, $v = 2$

$$\mathrm{CA}(5; 2, 4, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
CA(10; 3, 5, 2)	(1 1 1 1)	1	CA(10; 3, 5, 2)	[25]
CA(5; 2, 4, 2)	–	0	CA(5; 2, 4, 2)	[25]

- $N = 6$, $k = 5$, $t = 2$, $v = 2$

$$\mathrm{CA}(6; 2, 5, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
CA(12; 3, 6, 2)	(1 1 1 1 1)	1	CA(12; 3, 6, 2)	[25]
CA(6; 2, 5, 2)	–	0	CA(6; 2, 5, 2)	[25]

- $N = 12,\ k = 6,\ t = 3,\ v = 2$

$$\mathsf{CA}(12;3,6,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathsf{CA}(24;4,7,2)$	$(1\ 1\ 1\ 1\ 1\ 1)$	1	$\mathsf{CA}(24;4,7,2)$	[25]
$\mathsf{CA}(12;3,6,2)$	–	0	$\mathsf{CA}(12;3,6,2)$	[25]

- $N = 12,\ k = 7,\ t = 3,\ v = 2$

$$\mathsf{CA}(12;3,7,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathsf{CA}(24;4,8,2)$	$(1\ 1\ 1\ 1\ 1\ 1\ 1)$	1	$\mathsf{CA}(24;4,8,2)$	[25]
$\mathsf{CA}(12;3,7,2)$	–	0	$\mathsf{CA}(12;3,7,2)$	[25]

- $N = 12,\ k = 8,\ t = 3,\ v = 2$

$$\mathsf{CA}(12;3,8,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathsf{CA}(24;4,9,2)$	(1 1 1 1 1 1 1 1)	1	$\mathsf{CA}(24;4,9,2)$	[25]
$\mathsf{CA}(12;3,8,2)$	–	0	$\mathsf{CA}(12;3,8,2)$	[25]

- $N = 12,\ k = 9,\ t = 3,\ v = 2$

$$\mathsf{CA}(12;3,9,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathsf{CA}(24;4,10,2)$	(1 1 1 1 1 1 1 1 1)	1	$\mathsf{CA}(24;4,10,2)$	[25]
$\mathsf{CA}(12;3,9,2)$	–	0	$\mathsf{CA}(12;3,9,2)$	[25]

- $N = 12,\ k = 10,\ t = 3,\ v = 2$

$$\mathrm{CA}(12;3,10,2) = \begin{pmatrix} 0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&1&1&1&1\\ 0&0&1&1&1&0&0&0&1&1\\ 0&1&0&1&1&0&1&1&0&0\\ 0&1&1&0&1&1&0&1&0&1\\ 0&1&1&1&0&1&1&0&1&0\\ 1&0&0&1&1&1&1&0&0&1\\ 1&0&1&0&1&0&1&1&1&0\\ 1&0&1&1&0&1&0&1&0&0\\ 1&1&0&0&1&1&0&0&1&0\\ 1&1&0&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&0&0&1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathrm{CA}(24;4,11,2)$	$(1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1)$	1	$\mathrm{CA}(24;4,11,2)$	[25]
$\mathrm{CA}(12;3,10,2)$	–	0	$\mathrm{CA}(12;3,10,2)$	[25]

- $N = 12,\ k = 11,\ t = 3,\ v = 2$

$$\mathrm{CA}(12;3,11,2) = \begin{pmatrix} 0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&1&1&1&1&1\\ 0&0&1&1&1&0&0&0&1&1&1\\ 0&1&0&1&1&0&1&1&0&0&1\\ 0&1&1&0&1&1&0&1&0&1&0\\ 0&1&1&1&0&1&1&0&1&0&0\\ 1&0&0&1&1&1&1&0&0&1&0\\ 1&0&1&0&1&0&1&1&1&0&0\\ 1&0&1&1&0&1&0&1&0&0&1\\ 1&1&0&0&1&1&0&0&1&0&1\\ 1&1&0&1&0&0&0&1&1&1&0\\ 1&1&1&0&0&0&1&0&0&1&1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathrm{CA}(24;4,12,2)$	$(1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1)$	1	$\mathrm{CA}(24;4,12,2)$	[25]
$\mathrm{CA}(12;3,11,2)$	–	0	$\mathrm{CA}(12;3,11,2)$	[25]

- $N = 21,\ k = 6,\ t = 4,\ v = 2$

$$\mathsf{CA}(21;4,6,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathsf{CA}(42;5,7,2)$	(1 1 1 1 1 1)	1	$\mathsf{CA}(42;5,7,2)$	[25]
$\mathsf{CA}(21;4,6,2)$	–	0	$\mathsf{CA}(21;4,6,2)$	[25]

- $N = 11,\ k = 5,\ t = 2,\ v = 3$

$$\mathsf{CA}(11;2,5,3) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 & 2 \\ 0 & 2 & 2 & 1 & 2 \\ 1 & 0 & 2 & 2 & 1 \\ 1 & 1 & 1 & 0 & 2 \\ 1 & 1 & 2 & 1 & 0 \\ 1 & 2 & 0 & 0 & 1 \\ 2 & 0 & 2 & 0 & 2 \\ 2 & 1 & 0 & 1 & 1 \\ 2 & 2 & 1 & 2 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathsf{CA}(33;3,6,3)$	(1 2 1 1 1 2 1 2 2 2)	1	$\mathsf{CA}(33;3,6,3)$	[25]
$\mathsf{CA}(11;2,5,3)$	–	0	$\mathsf{CA}(11;2,5,3)$	[25]

- $N = 13,\ k = 4,\ t = 2,\ v = 3$

$$\mathsf{CA}(13;2,4,3) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 2 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 0 & 0 & 2 \\ 1 & 1 & 1 & 0 \\ 1 & 2 & 2 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
$\mathsf{CA}(351;5,7,3)$	(0 1 2 1 2 1 2 0 2 0 1 1)	3	$\mathsf{CA}(351;5,7,3)$	[25]
$\mathsf{CA}(117;4,6,3)$	(1 2 2 2 0 2 1 0 2 2)	2	$\mathsf{CA}(111;4,6,3)$	[25]
$\mathsf{CA}(39;3,5,3)$	(0 0 1 2 0 0 2 1)	1	$\mathsf{CA}(33;3,5,3)$	[25]
$\mathsf{CA}(13;2,4,3)$	–	0	$\mathsf{CA}(9;2,4,3)$	[25]

- $N = 15,\ k = 5,\ t = 2,\ v = 3$

$$\mathsf{CA}(15;2,5,3) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 1 & 2 & 0 & 1 \\ 0 & 2 & 1 & 1 & 2 \\ 0 & 2 & 2 & 2 & 0 \\ 1 & 0 & 1 & 0 & 2 \\ 1 & 1 & 2 & 2 & 1 \\ 1 & 2 & 0 & 1 & 0 \\ 2 & 0 & 2 & 1 & 2 \\ 2 & 1 & 0 & 2 & 0 \\ 2 & 2 & 1 & 0 & 1 \end{pmatrix}$$

TCA	Vector δ	Height	Best Known CA	Reference
CA(405; 5, 8, 3)	(0 0 0 0 1 2 2 2 2 2 0 0 2 2)	3	CA(405; 5, 8, 3)	[25]
CA(135; 4, 7, 3)	(0 1 1 2 0 2 0 2 2 1 0 1)	2	CA(123; 4, 7, 3)	[25]
CA(45; 3, 6, 3)	(1 2 2 2 2 2 1 1 1 1)	1	CA(33; 3, 6, 3)	[25]
CA(15; 2, 5, 3)	–	0	CA(11; 2, 5, 3)	[25]

The following are the covering arrays in the TCAs presented in this section that equal a current upper bound:

CA(5; 2, 4, 2),	CA(10; 3, 5, 3),	CA(6; 2, 5, 2),
CA(12; 3, 6, 2),	CA(24; 4, 7, 2),	CA(12; 3, 7, 2),
CA(24; 4, 8, 2),	CA(12; 3, 8, 2),	CA(24; 4, 9, 2),
CA(12; 3, 9, 2),	CA(24; 4, 10, 2),	CA(12; 3, 10, 2),
CA(24; 4, 11, 2),	CA(12; 3, 11, 2),	CA(24; 4, 12, 2),
CA(21; 4, 6, 2),	CA(42; 5, 7, 2),	CA(11; 2, 5, 3),
CA(33; 3, 6, 3),	CA(351; 5, 7, 3),	CA(405; 5, 8, 3).

5.3 Computational Results for the NonIsoCA Algorithm

The algorithm to generate the non-isomorphic bases solves the problem of generating all the non-isomorphic covering arrays of minimum rank that exist for a particular combination of the parameters N, t, k, and v, that define a covering array. This a very hard problem, because all the search space of the CACP problem (Section 2.3) must be explored in order to find all the non-isomorphic covering arrays.

For this reason, we perform a separate computational experimentation for the NonIsoCA algorithm. This experimentation only took into account combinations of the parameters N, t, k, and v in which either N is optimal with respect to t, k, and v, or it is unknown if a covering array with N rows exists for the parameters t, k, and v.

Table 5.14 shows the computational results of the NonIsoCA algorithm for covering arrays of strength two and order two with $N = 4, 5, 6, 7, 8$ rows. From left to right the first four columns of the table are the parameters N, k, t, and v; the fifth column is the approximate cardinality of search space given by Equation (2.3) of Section 2.3; the sixth column is the number of non-isomorphic covering arrays that exist for the parameters in columns 1-4; and the seventh column is the time required by the NonIsoCA algorithm to generate the non-isomorphic covering arrays. The CPU time is reported using the following abbreviations for time units: μs for microsecond, *s* for second, and *h* for hour.

For binary covering arrays of strength tree and strength four we compute the non-isomorphic arrays up to $k = 15$ columns for strength three, and up to $k = 12$ columns for strength four; the results obtained are shown in Table 5.15. For the orders $v = 3$ and $v = 4$ we compute the non-isomorphic covering arrays of strength two up to $k = 10$ columns for order three, and up to $k = 5$ columns for order four; the results are shown in Table 5.16.

The shaded rows in Tables 5.15 and 5.16 shows that none non-isomorphic covering array exists for the parameter combinations $\{N = 15, t = 3, k = 13, v = 2\}$, $\{N = 16, t = 3, k = 15, v = 2\}$, and

N	k	t	v	Search Space	Non-isomorphic CAs	CPU Time
4	2	2	2	1.00×10^{0}	1	50.00 μs
4	3	2	2	7.00×10^{1}	1	85.00 μs
5	4	2	2	4.36×10^{3}	1	197.00 μs
6	5	2	2	9.06×10^{5}	7	0.003 s
6	6	2	2	7.49×10^{7}	4	0.006 s
6	7	2	2	5.42×10^{9}	3	0.009 s
6	8	2	2	3.68×10^{11}	1	0.012 s
6	9	2	2	2.42×10^{13}	1	0.015 s
6	10	2	2	1.57×10^{15}	1	0.025 s
7	11	2	2	2.96×10^{19}	26	0.279 s
7	12	2	2	3.81×10^{21}	10	0.308 s
7	13	2	2	4.89×10^{23}	4	0.329 s
7	14	2	2	6.27×10^{25}	1	0.347 s
7	15	2	2	8.04×10^{27}	1	0.373 s
8	16	2	2	8.43×10^{33}	700,759	0.90 h
8	17	2	2	2.16×10^{36}	579,466	1.32 h
8	18	2	2	5.53×10^{38}	440,826	1.88 h
8	19	2	2	1.41×10^{41}	309,338	2.46 h
8	20	2	2	3.62×10^{43}	200,326	3.12 h
8	21	2	2	9.27×10^{45}	119,752	3.78 h
8	22	2	2	2.37×10^{48}	65,993	4.32 h
8	23	2	2	6.08×10^{50}	33,463	4.68 h
8	24	2	2	1.55×10^{53}	15,596	4.95 h
8	25	2	2	3.98×10^{55}	6,704	5.03 h
8	26	2	2	1.02×10^{58}	2,646	5.25 h
8	27	2	2	2.61×10^{60}	977	5.31 h
8	28	2	2	6.68×10^{62}	343	5.38 h
8	29	2	2	1.71×10^{65}	118	5.38 h
8	30	2	2	4.38×10^{67}	39	5.39 h
8	31	2	2	1.12×10^{70}	15	5.39 h
8	32	2	2	2.87×10^{72}	5	5.41 h
8	33	2	2	7.35×10^{74}	2	5.43 h
8	34	2	2	1.88×10^{77}	1	5.48 h
8	35	2	2	4.81×10^{79}	1	5.51 h

Table 5.14: Results for $k = 2, 3, \ldots, 35$ for binary covering arrays of strength two.

N	k	t	v	Search Space	Non-isomorphic CAs	CPU Time
8	3	3	2	1.00×10^{0}	1	358.00 μs
8	4	3	2	1.28×10^{4}	1	666.00 μs
10	5	3	2	6.45×10^{7}	1	0.012 s
12	6	3	2	3.28×10^{12}	9	0.337 s
12	7	3	2	2.37×10^{16}	2	0.353 s
12	8	3	2	1.27×10^{20}	2	0.372 s
12	9	3	2	5.94×10^{23}	1	0.436 s
12	10	3	2	2.60×10^{27}	1	0.539 s
12	11	3	2	1.10×10^{31}	1	0.851 s
15	12	3	2	1.14×10^{42}	2	1.26 h
15	13	3	2	3.79×10^{46}	0	1.39 h
16	13	3	2	1.93×10^{49}	89	937.68 h
16	14	3	2	1.27×10^{54}	8	978.42 h
16	15	3	2	8.41×10^{58}	0	1,052.65 h
16	4	4	2	1.00×10^{0}	1	0.127 s
16	5	4	2	6.01×10^{8}	1	0.158 s
21	6	4	2	4.11×10^{16}	1	694.60 s
24	7	4	2	6.03×10^{25}	1	51.86 h
24	8	4	2	3.32×10^{33}	1	52.07 h
24	9	4	2	9.81×10^{40}	1	52.88 h
24	10	4	2	2.17×10^{48}	1	53.09 h
24	11	4	2	4.17×10^{55}	1	53.19 h
24	12	4	2	7.49×10^{62}	1	53.39 h

Table 5.15: Results for binary covering arrays of strength three and strength four.

$\{N = 13, t = 2, k = 10, v = 3\}$. After exploring all the search space the NonIsoCA algorithm does not find any non-isomorphic covering array of minimum rank for the above parameter combinations; this implies that none covering array for these parameters exists.

The non-existence of a covering array for these parameter combinations had not been proved before. The previous results for these parameters combinations were the following ones:

- For $t = 3$, $k = 13$, and $v = 2$, Colbourn et al. [26] established that $15 \leq \text{CAN}(3, 13, 2) \leq 16$.

- For $t = 3$, $k = 15$, and $v = 2$, Choi et al. [17] established that $\text{CAN}(3, 15, 2) \geq 15$.

N	k	t	v	Search Space	Non-isomorphic CAs	CPU Time
9	2	2	3	1.00×10^{0}	1	0.002 s
9	3	2	3	4.68×10^{6}	1	0.003 s
9	4	2	3	2.60×10^{11}	1	0.006 s
11	5	2	3	3.47×10^{18}	3	1.78 s
12	6	2	3	4.29×10^{25}	13	240.15 s
12	7	2	3	2.42×10^{31}	1	252.38 s
13	8	2	3	6.62×10^{39}	5	4.01 h
13	9	2	3	1.06×10^{46}	4	4.02 h
13	10	2	3	1.70×10^{52}	0	4.14 h
16	2	2	4	1.00×10^{0}	1	145.97 s
16	3	2	4	4.88×10^{14}	2	215.04 s
16	4	2	4	1.00×10^{25}	1	347.52 s
16	5	2	4	6.20×10^{34}	1	629.66 s

Table 5.16: Results for covering arrays of orders three and four.

- For $t = 2$, $k = 10$, and $v = 3$, Colbourn et al. [26] established that $13 \leq \mathrm{CAN}(2, 10, 3) \leq 14$.

This way, we conclude that $\mathrm{CAN}(3, 13, 2) = 16$, $\mathrm{CAN}(3, 15, 2) = 17$, and $\mathrm{CAN}(2, 10, 3) = 14$. Moreover, from the tables of covering arrays at [25] we know the existence of the covering arrays $\mathrm{CA}(16; 3, 14, 2)$ and $\mathrm{CA}(17; 3, 16, 2)$. So, given the optimality of the covering arrays $\mathrm{CA}(16; 3, 13, 2)$ and $\mathrm{CA}(17; 3, 15, 2)$ we have that the covering arrays $\mathrm{CA}(16; 3, 14, 2)$ and $\mathrm{CA}(17; 3, 16, 2)$ are also optimal. Table 5.17 summarizes these important results obtained with the NonIsoCA algorithm.

The non-existence of the covering array $\mathrm{CA}(15; 3, 13, 2)$ was validated by running the order encoding and the mixed encoding of Bambara et al. [7] for this instance. The result of the two runs

Previous result	Reference	New result
$15 \leq \mathrm{CAN}(3, 13, 2) \leq 16$	[26]	$\mathrm{CAN}(3, 13, 2) = 16$
$15 \leq \mathrm{CAN}(3, 14, 2) \leq 16$	[26]	$\mathrm{CAN}(3, 14, 2) = 16$
$\mathrm{CAN}(3, 15, 2) \geq 15$	[17]	$\mathrm{CAN}(3, 15, 2) = 17$
$\mathrm{CAN}(3, 16, 2) \geq 15$	[17]	$\mathrm{CAN}(3, 16, 2) = 17$
$13 \leq \mathrm{CAN}(3, 10, 2) \leq 14$	[26]	$\mathrm{CAN}(2, 10, 3) = 14$

Table 5.17: Optimal covering array numbers determined with the NonIsoCA algorithm.

was UNSATISFIABLE, meaning that it was not possible to construct the covering array. The SAT solver used was GLUEMINISAT 2.2.5.

The results in Table 5.17 are a very important contribution of this thesis because the exact lower bound for the five cases in the table had not been determined before by any other means.

5.4 Computational Analysis of the Functions *apply_*$\mathcal{E}$() and *is_minimum*()

In Section 4.2.2 was proposed to generate the vectors δ in v-ary Gray code in order to apply efficiently the construction $\mathcal{E}$ to these vectors δ. The function *apply_*$\mathcal{E}$(), given in Algorithm 1 of Chapter 4, implements this operation; but its execution time could not determined analytically, since the number of times the verification of the matrix B produced by the construction $\mathcal{E}$ is omitted depends largely on the base covering array.

Something similar occurs for the function *is_minimum*() to verify if a covering arrays is of minimum rank, given in Algorithm 12 of Chapter 4, where the number of column permutations and column relabelings pruned of the search tree depends on the covering array being verified. So, the objective of this section is to analyze the computational work done by the functions *apply_*$\mathcal{E}$() and *is_minimum*() for a number of input covering arrays.

5.4.1 Analysis of the Function *apply_*$\mathcal{E}$()

Without taking advantage of the generation of the vectors δ in v-ary Gray code, the computational cost of applying the construction $\mathcal{E}$ to the $v^{k(v-1)}$ vectors δ that exist for a base covering array CA$(N; t, k, v)$ is the next one:

$$O\left(v^{k(v-1)}\left[Nv(k+1)+\binom{k}{t+1}Nv(t+1)\right]\right). \tag{5.2}$$

Which is the cost of creating the $v^{k(v-1)}$ matrices B of dimensions $Nv \times (k+1)$ and of verifying if they are covering arrays of strength $t+1$. By generating the vectors δ in v-ary Gray code the cost of creating the matrices B is reduced from $O(v^{k(v-1)}[Nv(k+1)])$ to $O(Nv(k+1)+N[v^{k(v-1)}-1])$, because only the first matrix B is generated in $O(Nv[k+1])$ time, while the remaining $v^{k(v-1)}-1$ matrices B are generated in $O(N)$ time.

In addition, if the vectors δ are generated in v-ary Gray code then the verification of some matrices B can be omitted. Let be α the number of matrices B verified when the construction $\mathcal{E}$ is applied to all vectors δ, and let be β the number of matrices B for which the verification is omitted. Then the execution time to verify if the $v^{k(v-1)}$ matrices B of dimensions $Nv \times (k+1)$ are covering arrays of strength $t+1$ is:

$$O\left(\alpha\binom{k}{t+1}Nv(t+1) + \beta(t+1)\right). \tag{5.3}$$

Where the second term $\beta(t+1)$ is the cost of checking if the vector D of size $t+1$, used in the Algorithm 6 of Chapter 4, contains the index of the column updated to produce the matrix B for the current vector δ.

Table 5.18 shows how many times the verification process is permored for a set of 20 covering arrays. The first column of the table is the base covering array CA$(N;t,k,v)$, the second columns is the number of vectors δ for the base covering array, the third columns is the number of matrices B for which the verification process was performed, the fourth column is the number of matrices B for which the verification was omitted, and the last column is the percentage of the verifications, i.e., $(\alpha/v^{k(v-1)}) * 100$.

In the table we can notice that the greater percentage of verifications occur for the matrices with a small number of columns, say $k \leq 8$, because there is more chance that the updated column of matrix B is one of the columns whose indices are in vector D. For greater values of k, say $k \geq 9$, the number of matrices B verified is much smaller than the number of matrices B whose verification is

CA$(N;t,k,v)$	$v^{k(v-1)}$	Verified	Omitted	Percentage of verifications
CA$(6;2,5,2)$	32	18	14	56.25%
CA$(6;2,7,2)$	128	19	109	14.84%
CA$(6;2,10,2)$	1,024	23	1,001	2.25%
CA$(7;2,11,2)$	2,048	31	2,017	1.51%
CA$(7;2,15,2)$	32,768	45	32,723	0.14%
CA$(8;2,11,2)$	2,048	30	2,018	1.46%
CA$(8;2,16,2)$	65,536	119	65,417	0.18%
CA$(8;2,20,2)$	1,048,576	122	1,048,454	0.01%
CA$(9;2,13,2)$	8,192	194	7,998	2.37%
CA$(10;3,5,2)$	32	24	8	75.00%
CA$(12;3,7,2)$	128	30	98	23.44%
CA$(12;3,11,2)$	2,048	48	2,000	2.34%
CA$(15;3,12,2)$	4,096	61	4,035	1.49%
CA$(24;4,10,2)$	1,024	68	956	6.64%
CA$(24;4,12,2)$	4,096	81	4,015	1.98%
CA$(9;2,4,3)$	6561	4637	1924	70.68%
CA$(11;2,5,3)$	59,049	12,522	46,527	21.21%
CA$(12;2,7,3)$	4,782,969	115,957	4,667,012	2.42%
CA$(13;2,9,3)$	387,420,489	1,514,159	385,906,330	0.39%
CA$(16;2,5,4)$	1,073,741,824	184,673,582	889,068,242	17.20%

Table 5.18: Percentage of verifications of the matrices B.

omitted. Therefore the generation of the vectors δ in v-ary Gray code allows important time savings when the construction $\mathcal{E}$ is applied to all the vectors δ for a base covering array, and specially for $k \geq 9$ the number of matrices B verified is a small percentage of the total number of the matrices B.

5.4.2 Analysis of the Function *is_minimum()*

The smarter verification of minimum rank implement given in Algorithm 12 has a worst case running time of $O(N^2\, k!\, (v!^k))$ to verify if a covering array CA$(N;t,k,v)$ is of minimum rank. However, it is very improbable that all the $(v!)^k$ relabelings are required for each one of the $k!$ column permutations. To estimate the execution time of this algorithm we count the number of times γ the function

sort_rows() is called in Algorithm 12. In the worst case γ will be equal to $k!\,(v!)^k$; but in general not all column permutations are of minimum rank, and also not all the relabelings for a column permutation are of minimum rank.

The verification of minimum rank requires more work when the covering array tested is one of minimum rank; so we only consider covering arrays of minimum rank to test the function *is_minimum*(). Table 5.19 shows the number of times the function *sort_rows*() of Algorithm 12 is called for 20 different covering arrays of minimum rank; the first column of the table is the covering array verified, the second column contains the product of the $k!$ permutations of columns by the $(v!)^k$ distinct columns relabelings that exist for each column permutation; and the third column shows the number of times the function *sort_rows*() is called.

CA$(N;t,k,v)$	$k!(v!)^k$	Calls to *sort_rows*()
CA$(6;2,5,2)$	3.84×10^3	119
CA$(6;2,7,2)$	6.45×10^5	3,028
CA$(6;2,10,2)$	3.71×10^9	38,740
CA$(7;2,11,2)$	8.17×10^{10}	5,629
CA$(7;2,15,2)$	4.28×10^{16}	80,865
CA$(8;2,11,2)$	8.17×10^{10}	514
CA$(8;2,16,2)$	1.37×10^{18}	1,715
CA$(8;2,20,2)$	2.55×10^{24}	80,334
CA$(9;2,13,2)$	5.10×10^{13}	676
CA$(10;3,5,2)$	3.84×10^3	1,700
CA$(12;3,7,2)$	6.45×10^5	17,644
CA$(12;3,11,2)$	8.17×10^{10}	575,124
CA$(15;3,12,2)$	1.96×10^{12}	5,574
CA$(24;4,10,2)$	3.71×10^9	1,985,080
CA$(24;4,12,2)$	1.96×10^{12}	13,803,024
CA$(9;2,4,3)$	3.11×10^4	9,120
CA$(11;2,5,3)$	9.33×10^5	1,304
CA$(12;2,7,3)$	1.41×10^9	9,942
CA$(13;2,9,3)$	3.65×10^{12}	7,632
CA$(16;2,5,4)$	9.55×10^8	1,278,960

Table 5.19: Number of times the function *sort_rows*() is called in Algorithm 12.

From the results of Table 5.19 we can say that the number of isomorphic covering arrays checked in order to determine if a covering array A is of minimum rank is very small with respect to the total number of covering arrays isomorphic to A; even if A is of minimum rank.

5.5 Chapter Summary

In this chapter were given the important computational results for the construction of TCAs. We found that for every order $v \geq 2$ there exists an infinite TCA of order v conformed by covering arrays that have an optimal number of rows with respect to the other three parameters; moreover, a number of the covering arrays in the infinite TCAs are also optimal or equal to the best known covering arrays with respect to the number of columns. An extensive computational experimentation was done in this thesis to construct TCAs other than the infinite ones; the instances executed in the computational experimentation included covering arrays of order $v = 2, 3, 4$. As a result we obtain seven TCAs that improves seven current bounds, and also we found another twelve TCAs that equal some current bounds. Other very important computational results of this thesis is the verification of the optimality of five covering arrays through the NonIsoCA algorithm; the optimality of these covering arrays had not been proved before by any other mean. Finally, we prove computationally that the optimizations discussed in Chapter 4 for the functions *apply_*$\mathcal{E}$() and *is_minimum*() allow substantial time savings in the application of the construction $\mathcal{E}$ to each vector δ and in the verification of minimum rank. The following chapter summarizes all the important results obtained from the construction of towers of covering arrays.

6

Conclusions

This chapter finalizes the thesis document. Section 6.1 summarizes the main contributions of the thesis; Section 6.2 provides some directions for further research; and Section 6.3 provides a final discussion of the work done in this thesis.

6.1 Main Contributions

In this thesis was explored a new way to construct the combinatorial designs called covering arrays. The proposed method consists in the construction of towers of covering arrays (TCAs); where a TCA of height h as a succession of $h+1$ covering arrays $C_0, C_1, \ldots, C_h$, in which the covering array C_0 is of strength t, and for $i = 1, 2, \ldots, h$, C_i is a covering array of strength $t+i$.

The TCAs are generated by means of the construction $\mathcal{E}$. From a base covering array CA($N; t, k, v$) of strength t, N rows, k columns, and order v, the construction $\mathcal{E}$ can sometimes generate a covering array CA($Nv; t+1, k+1, v$) of strength $t+1$, Nv rows, $k+1$ columns, and order v. The basic idea of the construction $\mathcal{E}$ is to juxtapose vertically v copies of the base covering

array translating the j-th column of the i-th copy by a value $\delta_{i,j} \in \{0, 1, \ldots, v-1\}$. The column $k+1$ of the covering array of strength $t+1$ is conformed by N zeroes, followed by N ones, and so on until finish with N elements equal to $v-1$. The construction $\mathcal{E}$ can be applied to the covering array of strength $t+1$ to generate a covering array of strength $t+3$, and so on, forming a TCA.

Next are summarized the main contributions derived from the construction of TCAs:

- A new way to construct covering arrays based on the construction of TCAs.
- A methodology to construct efficiently the TCAs.
- The identification of infinite TCAs for any order $v \geq 2$ conformed by optimal covering arrays. In addition, the following covering arrays equal the best known covering arrays with respect to the number of columns in the range $2 \leq t \leq 6$ and $2 \leq v \leq 25$:

CA(4; 2, 3, 2),	CA(8; 3, 4, 2),	CA(16; 4, 5, 2),
CA(32; 5, 6, 2),	CA(64; 6, 7, 2),	CA(27; 3, 4, 3),
CA(81; 4, 5, 3),	CA(243; 5, 6, 3),	CA(729; 6, 7, 3),
CA(256; 4, 5, 4),	CA(1024; 5, 6, 4),	CA(4096; 6, 7, 4),
CA(3125; 5, 6, 5),	CA(15625; 6, 7, 5),	CA(36; 2, 3, 6),
CA(216; 3, 4, 6),	CA(1296; 4, 5, 6),	CA(7776; 5, 6, 6),
CA(46656; 6, 7, 6),	CA(1000; 3, 4, 10),	CA(10000; 4, 5, 10),
CA(100000; 5, 6, 10),	CA(1000000; 6, 7, 10),	CA(20736; 4, 5, 12),
CA(248832; 5, 6, 12),	CA(2985984; 6, 7, 12),	CA(2744; 3, 4, 14),
CA(38416; 4, 5, 14),	CA(537824; 5, 6, 14),	CA(7529536; 6, 7, 14),
CA(50625; 4, 5, 15),	CA(759375; 5, 6, 15),	CA(11390625; 6, 7, 15),
CA(5832; 3, 4, 18),	CA(104976; 4, 5, 18),	CA(1889568; 5, 6, 18),
CA(34012224; 6, 7, 18),	CA(160000; 4, 5, 20),	CA(3200000; 5, 6, 20),
CA(64000000; 6, 7, 20),	CA(194481; 4, 5, 21),	CA(4084101; 5, 6, 21),
CA(85766121; 6, 7, 21),	CA(10648; 3, 4, 22),	CA(234256; 4, 5, 22),
CA(5153632; 5, 6, 22),	CA(113379904; 6, 7, 22),	CA(331776; 4, 5, 24),
CA(7962624; 5, 6, 24),	CA(191102976; 6, 7, 24).	

- A set of TCAs conformed by covering arrays of quality competitive with the best known ones. The following table shows the upper bounds improved by means of the construction of TCAs:

t	k	v	Previous N	New N
7	10	2	274	224
7	11	2	386	256
7	12	2	506	256
7	13	2	634	256
7	14	2	762	256
7	15	2	$\geq$762	256
7	16	2	$\geq$762	256
8	11	2	563	512
9	12	2	1230	1024
10	13	2	2491	2048
11	14	2	5190	4096

And these are the covering arrays that equal a current upper bound:

CA(64; 5, 11, 2),	CA(128; 6, 12, 2),	CA(64; 5, 12, 2),
CA(128; 6, 13, 2),	CA(64; 5, 13, 2),	CA(128; 6, 14, 2),
CA(32; 4, 13, 2),	CA(64; 5, 14, 2),	CA(128; 6, 15, 2),
CA(5; 2, 4, 2),	CA(10; 3, 5, 3),	CA(6; 2, 5, 2),
CA(12; 3, 6, 2),	CA(24; 4, 7, 2),	CA(12; 3, 7, 2),
CA(24; 4, 8, 2),	CA(12; 3, 8, 2),	CA(24; 4, 9, 2),
CA(12; 3, 9, 2),	CA(24; 4, 10, 2),	CA(12; 3, 10, 2),
CA(24; 4, 11, 2),	CA(12; 3, 11, 2),	CA(24; 4, 12, 2),
CA(21; 4, 6, 2),	CA(42; 5, 7, 2),	CA(11; 2, 5, 3),
CA(33; 3, 6, 3),	CA(351; 5, 7, 3),	CA(405; 5, 8, 3),

- An algorithm, called NonIsoCA, to solve the problem of generating all the non-isomorphic covering arrays of minimum rank that exist for a combination of the parameters N, t, k, v.
- The NonIsoCA algorithm can be used as an exact method to construct covering arrays by terminating the algorithm as soon as it finds the first non-isomorphic covering array of minimum rank for the input parameters N, t, k, and v.
- The determination of the exact value of the following covering arrays numbers, which had not been found before neither by computational search nor by algebraic analysis:

$$\begin{aligned}
\text{CAN}(3,13,2) &= 16 \\
\text{CAN}(3,14,2) &= 16 \\
\text{CAN}(3,15,2) &= 17 \\
\text{CAN}(3,16,2) &= 17 \\
\text{CAN}(2,10,3) &= 14
\end{aligned}$$

In [26] the lower bounds for CAN$(3,13,2)$, CAN$(3,14,2)$, and CAN$(2,10,2)$ were established to be grater than or equal 15, 15, and 13 respectively; but we determine that the exact lower bound for these covering array numbers is 16, 16, and 14 respectively. Similarly, in [17] the lower bounds for CAN$(3,15,2)$ and CAN$(3,16,2)$ were established to be greater than or equal to 15 and 15 respectively, but we found that the exact lower bound for these two cases is 17.

6.2 Future Work

The methodology proposed in this thesis is not the only approach to construct TCAs, nor the algorithms given here are the only way to implement that methodology. In the next list are given some directions for further research:

- To parallelize the algorithm to generate the non-isomorphic bases, and the algorithm to apply the construction $\mathcal{E}$ to the base covering arrays, in order to accelerate the construction of the TCAs. This would make possible to consider greater values of N, t, k, and v.

- When a TCA can not be extended to the next strength is because none of the matrices B produced by the application of the construction $\mathcal{E}$ was a covering array. However, some of these matrices B could by close to be a covering array. Whereby, in some cases, the best of the matrices B could be a very good initial solution for a metaheuristic method to construct covering arrays.

- The construction $\mathcal{E}$ defined in this thesis is based on translating the columns of the base covering array, but the operation of column translation is not the only operation that can be

applied to construct the next covering array of the TCA. For example, the copies of the base covering array could be constituted by permutations of the columns of the base covering array; in this case the vectors δ would contain a permutation of the indices of the columns of the base covering array, rather than constant values to translate the columns. Furthermore, the two approaches might be combined, that is, the columns of the base covering array might be permuted and translated by a constant value at the same time. Further operations and combinations of operations may be defined.

6.3 Final Discussion

One characteristic of the TCAs is that the number of rows of the covering array of strength $t+1$ is exactly v times the number of rows of the covering array of strength t (the previous covering array in the TCA), where v is the order of the covering arrays in the TCA. This characteristic is very important because the ratio between the number of rows of the covering arrays in the TCA is constant, and this does not occur in the best known covering arrays for the same values of t, k, and v, of the covering arrays in a TCA. This implies that it is necessary to extend the TCAs to their maximum height possible, because the possibility of obtaining a competitive covering array increases as the height of a TCA increases; this conclusion was derived analytically in Chapter 4 and proved experimentally in Chapter 5.

The construction of TCAs of maximum height requires to solve efficiently two problems:

1. To generate all the non-isomorphic covering arrays of minimum rank for a combination of the parameters N, t, k, and v, in order to check each of them as the base of the TCA.

2. To apply the construction $\mathcal{E}$ to all the $v^{k(v-1)}$ vectors δ that exist for a base covering array $\mathrm{CA}(N; t, k, v)$.

The solution to the first problem was the NonIsoCA algorithm, and for the second problem we

follow an efficient strategy consisting in generating the vectors δ in v-ary Gray code.

Each part of the methodology contributed to the viability of the TCA approach, because without generating and extending all the non-isomorphic bases it would not be possible to construct TCAs of maximum height, and without TCAs of maximum height it would not be possible to obtain competitive covering arrays. The same is true if not all vectors δ would have been checked for a base covering array.

Bibliography

[1] Aarts, E. and Lenstra, J. K., editors (2003). *Local Search in Combinatorial Optimization.* Princeton University Press, Princeton, NJ, USA.

[2] Avila-George, H. (2010). Verificación de covering arrays utilizando supercomputación y computación grid. Master's thesis, Universidad Politécnica de Valencia.

[3] Avila-George, H., Torres-Jimenez, J., and Hernandez, V. (Accepted on July 07, 2012a). New bounds for ternary covering arrays using a parallel simulated annealing. *Mathematical Problems in Engineering.*

[4] Avila-George, H., Torres-Jimenez, J., Hernández, V., and Rangel-Valdez, N. (2010). Verification of general and cyclic covering arrays using grid computing. In *Proceedings of the Third international conference on Data management in grid and peer-to-peer systems*, Globe'10, pages 112–123, Berlin, Heidelberg. Springer-Verlag.

[5] Avila-George, H., Torres-Jimenez, J., Hernández, V., and Rangel-Valdez, N. (2011). A parallel algorithm for the verification of covering arrays. In *Proceedings of The International Conference on Parallel and Distributed Techniques and Applications(PDPTA 2011)*, pages 879–885, Las Vegas Nevada, USA.

[6] Avila-George, H., Torres-Jimenez, J., and Hernández, V. (2012b). Parallel simulated annealing for the covering arrays construction problem. In *To appear in the 18th International Conference on Parallel and Distributed Processing Techniques and Applications.*

[7] Banbara, M., Matsunaka, H., Tamura, N., and Inoue, K. (2010). Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers. In *Proceedings of the 17th international*

conference on Logic for programming, artificial intelligence, and reasoning, LPAR'10, pages 112–126, Berlin, Heidelberg. Springer-Verlag.

[8] Barker, H. A. (1986). Sum and product tables for galois fields. *International Journal of Mathematical Education in Science and Technology*, 17(4):473–485.

[9] Bracho-Rios, J., Torres-Jimenez, J., and Rodriguez-Tello, E. (2009). A new backtracking algorithm for constructing binary covering arrays of variable strength. In *Proceedings of the 8th Mexican International Conference on Artificial Intelligence*, MICAI '09, pages 397–407, Berlin, Heidelberg. Springer-Verlag.

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

[11] Bush, K. A. (1952). Orthogonal arrays of index unity. *Annals of Mathematics Statistics*, 23(3):426–434.

[12] Calvagna, A. and Gargantini, A. (2012). T-wise combinatorial interaction test suites construction based on coverage inheritance. *Software Testing, Verification and Reliability*, 22(7):507–526.

[13] Carrizales-Turrubiates, O. A. (2011). Reducción Óptima de covering arrays. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

[14] Cawse, J. N. (2003). *Experimental design for combinatorial and high throughput materials development*. John Wiley & Sons, New York.

[15] Chateauneuf, M. and Kreher, D. L. (2002). On the state of strength-three covering arrays. *Journal of Combinatorial Designs*, 10(3):217–238.

[16] Chateauneuf, M. A., Colbourn, C. J., and Kreher, D. L. (1999). Covering arrays of strength three. *Des. Codes Cryptography*, 16(3):235–242.

[17] Choi, S., Kim, H. K., and Oh, D. Y. (2012). Structures and lower bounds for binary covering arrays. *Discrete Mathematics*, 312(19):2958–2968.

[18] Cohen, D., Dalal, S., Fredman, M., and Patton, G. (1997). The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444.

[19] Cohen, D., Dalal, S., Parelius, J., and Patton, G. (1996). The combinatorial design approach to automatic test generation. *Software, IEEE*, 13(5):83–88.

[20] Cohen, M. B., Colbourn, C. J., and Lin, A. C. (2008). Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics*, 308(13):2709–2722.

[21] Cohen, M. B., Gibbons, P. B., Mugridge, W. B., Colbourn, C. J., and Collofello, J. S. (2003). Variable strength interaction testing of components. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, COMPSAC '03, pages 413–418, Washington, DC, USA. IEEE Computer Society.

[22] Colbourn, C. J. (2004). Combinatorial aspects of covering arrays. *Le Matematiche*, 59:121–167.

[23] Colbourn, C. J. (2008). Strength two covering arrays: Existence tables and projection. *Discrete Mathematics*, 308(5):772–786.

[24] Colbourn, C. J. (2010). Covering arrays from cyclotomy. *Des. Codes Cryptography*, 55(2-3):201–219.

[25] Colbourn, C. J. (2013). Covering array tables for $t = 2, 3, 4, 5, 6$. last time accessed on may 23, 2013. http://www.public.asu.edu/ ccolbou/src/tabby/catable.html.

[26] Colbourn, C. J., Kéri, G., Soriano, P. P. R., and Schlage-Puchta, J. C. (2010). Covering and radius-covering arrays: Constructions and classification. *Discrete Appl. Math.*, 158(11):1158–1180.

[27] Colbourn, C. J., Martirosyan, S. S., Mullen, G. L., Shasha, D., Sherwood, G. B., and Yucas, J. L. (2006a). Products of mixed covering arrays of strength two. *Journal of Combinatorial Designs*, 14(2):124–138.

[28] Colbourn, C. J., Martirosyan, S. S., Trung, T., and Walker, II, R. A. (2006b). Roux-type constructions for covering arrays of strengths three and four. *Des. Codes Cryptography*, 41(1):33–57.

[29] Covarrubias-Flores, E. (2008). Cálculo de covering arrays binarios de fuerza variable, usando un algoritmo de recocido simulado. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

[30] F., M., Lawrence, J., Lei, Y., Kacker, R. N., and Kuhn, D. R. (2008). 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.

[31] Glover, F. (1989). Tabu search - part 1. *ORSA Journal on Computing*, 1(3):190–206.

[32] Gonzalez-Hernandez, L., Rangle-Valdez, N., and Torres-Jimenez, J. (2012). Construction of mixed covering arrays of strengths 2 through 6 using a tabu search approach. *Discrete Mathematics, Algorithms and Applications*, 04(03):1250033.

[33] Gonzalez-Hernandez, L. and Torres-Jimenez, J. (2010). Mits: A new approach of tabu search for constructing mixed covering arrays. In *MICAI 2010. LNAI Advances in Soft Computing, Vol. 6438, pp. 382-392.*

[34] Gonzalez-Hernandez, L., Torres-Jimenez, J., and Rangel-Valdez, N. (2011). An exact approach to maximize the number of wild cards in a covering array. *LNAI*, 7094:210–221.

[35] Grindal, M., Offutt, J., and Andler, S. F. (2005). Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199.

[36] Guan, D. (1998). Generalized gray codes with applications. In *Proc. Natl. Sci. Couc. ROC(A)*, volume 22, pages 841–848.

[37] Hartman, A. (2005). Software and hardware testing using combinatorial covering suites. In Golumbic, M. C. and Hartman, I. B.-A., editors, *Graph Theory, Combinatorics and Algorithms*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 237–266. Springer US.

[38] Hartman, A. and Raskin, L. (2004). Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–156.

[39] Hedayat, A. S., Sloane, N. J. A., and Stufken, J. (1999). *Orthogonal Arrays: Theory and Applications*. Springer-Verlag.

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

[41] Holland, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, USA.

[42] Katona, G. O. H. (1973). Two applications (for search theory and truth functions) of sperner type theorems. *Periodica Mathematica Hungarica*, 3(1–2):19–26.

[43] Kleitman, D. J. and Spencer, J. (1973). Families of k-independent sets. *Discrete Mathematics*, 6(3):255–262.

[44] Kuhn, D. R., Kacker, R. N., Lei, Y., and Hunter, J. (2009). Combinatorial software testing. *Computer*, 42(8):94–96.

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

[46] Kuhn, D. R. and Reilly, M. J. (2002). An investigation of the applicability of design of experiments to software testing. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 91–95, Washington, DC, USA. IEEE Computer Society.

[47] Kuhn, D. R., Wallace, D. R., and Gallo, Jr., A. M. (2004). Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421.

[48] Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., and Lawrence, J. (2007). Ipog: A general strategy for t-way software testing. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 549–556, Washington, DC, USA. IEEE Computer Society.

[49] Lei, Y. and Tai, K. (1998). In-parameter-order: a test generation strategy for pairwise testing. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 254–261.

[50] Lopez-Escogido, D., Torres-Jimenez, J., Rodriguez-Tello, E., and Rangel-Valdez, N. (2008). Strength two covering arrays construction using a sat representation. In Gelbukh, A. F. and Morales, E. F., editors, *MICAI*, volume 5317 of *Lecture Notes in Computer Science*, pages 44–53. Springer.

[51] Martinez-Pena, J. and Torres-Jimenez, J. (2010). A branch and bound algorithm for ternary covering arrays construction using trinomial coefficients. *Research in Computing Science*, 49:61–71.

[52] Martinez-Pena, J., Torres-Jimenez, J., Rangel-Valdez, N., and Avila-George, H. (2010). A heuristic approach for constructing ternary covering arrays using trinomial coefficients. In *Proceedings of the 12th Ibero-American conference on Advances in artificial intelligence*, IBERAMIA'10, pages 572–581, Berlin, Heidelberg. Springer-Verlag.

[53] Martirosyan, S. and Trung, T. v. (2004). On t-covering arrays. *Designs, Codes and Cryptography*, 32:323–339. 10.1023/B:DESI.0000029232.40302.6d.

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

[55] Meagher, K. and Stevens, B. (2005). Group construction of covering arrays. *Journal of Combinatorial Designs*, 13(1):70–77.

[56] Nayeri, P., Colbourn, C. J., and Konjevod, G. (2013). Randomized post-optimization of covering arrays. *European Journal of Combinatorics*, 34(1):91–103.

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

[58] Quiz-Ramos, P. (2010). Maximización de renglones constantes para covering arrays. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

[59] Rényi, A. (1970). *Foundations of probability.* Holden-Day.

[60] Rodríguez-Cristerna, A. (2012). Construcción de covering arrays mixtos usando una generalización del operador de fusión. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

[61] Ronneseth, A. H. and Colbourn, C. J. (2009). Merging covering arrays and compressing multiple sequence alignments. *Discrete Applied Mathematics*, 157(9):2177–2190. Optimal Discrete Structures and Algorithms ODSA 2006.

[62] Roux, G. (1987). *k-propriétés dans des tableaux de n colonnes; cas particulier de la k-surjectivité et de la k-permutivité.* PhD thesis, University of Paris.

[63] Shasha, D. E., Kouranov, A. Y., Lejay, L. V., Chou, M. F., and Coruzzi, G. M. (2001). Using combinatorial design to study regulation by multiple input signals. a tool for parsimony in the post-genomics era. *Plant Physiology*, 127:1590–1594.

[64] Shiba, T., Tsuchiya, T., and Kikuno, T. (2004). Using artificial life techniques to generate test cases for combinatorial testing. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 1, pages 72–77.

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

[66] Stardom, J. (2001). Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University.

[67] Stevens, B. (1998). *Transversal covers and packings*. PhD thesis, University of Toronto.

[68] Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation*. Wiley Publishing.

[69] Tang, D. T. and Woo, L. S. (1983). Exhaustive test pattern generation with constant weight vectors. *IEEE Trans. Comput.*, 32(12):1145–1150.

[70] Torres-Jimenez, J., Rangel-Valdez, N., Gonzalez-Hernandez, A. L., and Avila-George, H. (2011). Construction of logarithm tables for galois fields. *International Journal of Mathematical Education in Science and Technology*, 42(1):91–102.

[71] Torres-Jimenez, J. and Rodriguez-Tello, E. (2010). Simulated annealing for constructing binary covering arrays of variable strength. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8.

[72] Torres-Jimenez, J. and Rodriguez-Tello, E. (2012). New bounds for binary covering arrays using simulated annealing. *Inf. Sci.*, 185(1):137–152.

[73] Wallace, D. R. and Kuhn, D. R. (2001). Failure modes in medical device software: an analysis of 15 years of recall data. In *ACS/ IEEE International Conference on Computer Systems and Applications*, pages 301–311.

[74] Yan, J. and Zhang, J. (2006). Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '06, pages 385–394, Washington, DC, USA. IEEE Computer Society.

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL IPN

UNIDAD TAMAULIPAS

Cd. Victoria, Tamaulipas, a 19 de agosto de 2013.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará el C. Idelfonso Izquierdo Márquez, declaramos que hemos revisado la tesis titulada:

"Construcción de Torres de *Covering Arrays*"

Y consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias en Computación.

Atentamente,

Dr. Arturo Díaz Pérez

Dr. Wilfrido Gómez Flores

Dr. José Torres Jiménez

Km. 5.5 Carretera a Soto la Marina. Ciudad Victoria, Tamaulipas. CP 87130
Parque Científico y Tecnológico TECNOTAM
Tels.: (834) 1070220 y 316 6600
E-mail: admin@tamps.cinvestav.mx