Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional

Unidad Cinvestav Tamaulipas

Clasificación de Covering Arrays

Tesis que presenta:

Idelfonso Izquierdo Marquez

Para obtener el grado de:

Doctor en Ciencias en Ingeniería y Tecnologías Computacionales

Director de la Tesis:
Dr. José Torres Jiménez

Cd. Victoria, Tamaulipas, México Abril, 2019

Center for Research and Advanced Studies
of the National Polytechnic Institute

Cinvestav Tamaulipas

Classification of Covering Arrays

Thesis by:

Idelfonso Izquierdo Marquez

as the fulfillment of the
requirement for the degree of:

**Doctor of Science
in Engineering and Computing
Technologies**

Thesis Director:
Dr. José Torres Jiménez

Cd. Victoria, Tamaulipas, México April, 2019

© Copyright by
Idelfonso Izquierdo Marquez
2019

The thesis of Idelfonso Izquierdo Marquez is approved by:

Dr. Héctor Hugo Avilés Arriaga

Dr. José Juan García Hernández

Dr. Ricardo Landa Becerra

Dr. Said Polanco Martagón

Dr. José Torres Jiménez, Committe Chair

Cd. Victoria, Tamaulipas, México, April 5 2019

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

Acknowledgements

- I am very grateful to CINVESTAV-Tamaulipas for accepting me into its doctoral program, and for providing me with the necessary means to complete the degree.
- I express my gratitude to CONACYT for its doctoral scholarship, which allowed me to concentrate full-time on my research.
- I acknowledge to ABACUS-CINVESTAV and to CINVESTAV-CGSTIC for providing access to high performance computing.
- Finally, I thank to my advisor Dr. José Torres Jiménez for his invaluable guidance and help.

Contents

List of Figures

List of Tables

List of Algorithms

Publications

Idelfonso Izquierdo-Marquez and Jose Torres-Jimenez, *New covering array numbers*, Applied Mathematics and Computation, Vol 353, 2019, pp 134-146.

Idelfonso Izquierdo-Marquez and Jose Torres-Jimenez, *New optimal covering arrays using an orderly algorithm*, Discrete Mathematics, Algorithms and Applications, Vol 10, No 1, 2018, 16 pages.

Jose Torres-Jimenez, Idelfonso Izquierdo-Marquez, and Himer Avila-George, *Methods to construct uniform covering arrays*, IEEE Access, accepted for publication.

Idelfonso Izquierdo-Marquez, Jose Torres-Jimenez, Brenda Acevedo-Juárez, and Himer Avila-George, *A greedy-metaheuristic 3-stage approach to construct covering arrays*, Information Sciences, Vol 460-461, 2018, pp 172-189.

Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez, *Covering arrays of strength three from extended permutation vectors*, Designs, Codes and Cryptography, Vol 86, No 11, 2018, pp 2629-2643.

Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez, *A simulated annealing algorithm to construct covering perfect hash families*, Mathematical Problems in Engineering, Vol 2018, Article ID 1860673, 14 pages.

Himer Avila-George, Jose Torres-Jimenez, and Idelfonso Izquierdo-Marquez, *Improved pairwise test suites for non-prime-power orders*, IET Software, Vol 12, No 3, 2018, pp 215-224.

Jose Torres-Jimenez, Idelfonso Izquierdo-Marquez, and Himer Avila-George, *Search-based software engineering for constructing covering arrays*, IET Software, Vol 12, No 4, 2018, pp 324-332.

Resumen

Clasificación de Covering Arrays

por

Idelfonso Izquierdo Marquez
Unidad Cinvestav Tamaulipas
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2019
Dr. José Torres Jiménez, Director

Un covering array CA$(N; t, k, v)$ es un arreglo de tamaño $N \times k$ sobre $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$ tal que cada subarreglo de t columnas contiene como renglón al menos una vez cada una de las t-tuplas sobre $\mathbb{Z}_v$. El mínimo número de renglones para el cual existe un CA con fuerza t, k columnas, y orden v, es el *covering array number* de t, k, v, y se denota por CAN(t, k, v).

Hay tres isomorfismos en los CAs: permutaciones de renglones, permutaciones de columnas, y permutaciones de símbolos en una columna. Para valores particulares de N, t, k, v, el conjunto de todos los covering arrays CA$(N; t, k, v)$ se particiona en clases de CAs isomórficos. Todos los CAs de una clase son isomórficos entre sí, pero ningún CA de una clase es isomórfico a un CA de otra clase. La clasificación de CAs implica la generación de un elemento de cada clase.

En esta tesis se desarrollan dos nuevos algoritmos de clasificación; el primero es una versión mejorada de un algoritmo reportado previamente el cual sigue la estrategia de generación ordenada de subarreglos, y el segundo se basa en yuxtaponer verticalmente v CAs de fuerza t para generar CAs de fuerza $t+1$. Estos dos algoritmos pueden clasificar CAs más grandes que los clasificados por algoritmos del estado del arte. Para reducir el tiempo de ejecución de los algoritmos, desarrollamos versiones paralelas de ellos usando el modelo de paso de mensajes. Los resultados computacionales más importantes son la clasificación de 39 nuevos CAs, el hallazgo de 19 nuevos CANs, y la mejora de 13 cotas inferiores de CANs.

Abstract

Classification of Covering Arrays

by

Idelfonso Izquierdo Marquez
Cinvestav Tamaulipas
Center for Research and Advanced Studies of the National Polytechnic Institute, 2019
Dr. José Torres Jiménez, Advisor

A covering array CA$(N; t, k, v)$ is an array of size $N \times k$ over $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$ such that every subarray of t columns contains as a row each t-tuple over $\mathbb{Z}_v$ at least once. The minimum number of rows for which exists a CA with strength t, k columns, and order v, is the *covering array number* of t, k, v, and it is denoted by CAN(t, k, v).

There are three isomorphisms in CAs: row permutations, column permutations, and symbol permutations in a column. For particular values of N, t, k, v, the set of all covering arrays CA$(N; t, k, v)$ is partitioned in classes of isomorphic CAs. All CAs in a class are isomorphic among them, but no CA of one class is isomorphic to a CA of another class. The classification of CAs implies the generation of one element of each class.

In this thesis there are developed two new classification algorithms; the first one is an improved version of a previously reported algorithm which follows the strategy of orderly generation of subarrays, and the second one is based on juxtaposing vertically v CAs of strength t to generate CAs of strength $t + 1$. These two algorithms can classify CAs larger than those classified by state of the art algorithms. To reduce the execution time of the algorithms, we develop parallel versions of them using the message passing model. The main computational results are the classification of 39 new CAs, the finding of 19 new CANs, and the improvement of 13 lower bounds of CANs.

1

Introduction

A covering array CA$(N;t,k,v)$ is an array of size $N \times k$ over $\mathbb{Z}_v = \{0,1,\ldots,v-1\}$ such that every subarray of t columns contains as a row each t-tuple over $\mathbb{Z}_v$ at least once; the parameter t is called the strength of the CA. For given parameters N, t, k, v the set of all covering arrays CA$(N;t,k,v)$ is partitioned in classes of isomorphic CAs. The problem of classifying CAs consists in generating one element of each isomorphism class. In this thesis we develop two new algorithms for the classification of CAs; the first algorithm is an improved version of a previously reported algorithm which follows the strategy of orderly generation of subarrays, and the second algorithm is based on juxtaposing vertically v CAs of strength t to generate CAs of strength $t+1$. Section 1.1 introduces CAs and gives two examples of them. Section 1.2 provides useful background on CAs, specially the three isomorphisms of CAs. Section 1.3 describes the classification of CAs, presents some applications of it, and lists known classification results for optimal CAs. Section 1.4 states the research hypothesis, the main objective, and the particular objectives of the work. Finally, Section 1.5 provides an overview of the remaining chapters.

1.1 Covering arrays

Let N, t, k, v be positive integers. A *covering array* CA$(N; t, k, v)$ is an array with N rows, k columns, and entries from the set $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$, such that every subarray of t columns contains as a row each t-tuple over $\mathbb{Z}_v$ at least once. So, each of the $\binom{k}{t}$ subarrays of t columns contains at least once each element of $\mathbb{Z}_v^t$, which is the set of the v^t tuples of length t with symbols from $\mathbb{Z}_v$. The integers v and t are called respectively the *order* and the *strength* of the CA.

Figure 1.1 shows a covering array CA$(12; 3, 11, 2)$; this CA has $N = 12$ rows, $k = 11$ columns, order $v = 2$, and strength $t = 3$. Then, each of the $\binom{k}{t} = \binom{11}{3} = 165$ subarrays of $t = 3$ columns contains as a row the eight 3-tuples over $\mathbb{Z}_2 = \{0, 1\}$ at least once; these eight 3-tuples are the elements of the set $\mathbb{Z}_2^3 = \{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)\}$. Take for example the subarray formed by the first two columns and by the fourth column; these columns are marked with a down arrow in the figure. The first row of the subarray contains the tuple $(1, 1, 0)$, the second row contains the tuple $(1, 0, 0)$, and so on. The first occurrence of the eight 3-tuples over $\mathbb{Z}_2$ in the above subarray is colored in purple. Some 3-tuples over $\mathbb{Z}_2$ occur more than once in a subarray of 3 columns, but the requirement is that they appear at least once.

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

Figure 1.1: A covering array CA$(12; 3, 11, 2)$. In the columns marked with a down arrow the first occurrence of the eight 3-tuples over $\mathbb{Z}_2 = \{0, 1\}$ is colored in purple.

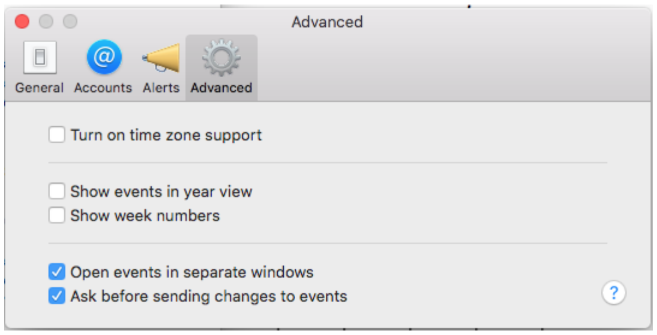
If a subarray of t columns contains as a row a t-tuple x over $\mathbb{Z}_v$ we say that the subarray *covers* the tuple x. If x is not covered in the subarray then x is a *missing tuple*. Figure 1.2 shows a CA$(11; 2, 5, 3)$; in this case, each of the $\binom{k}{t} = \binom{5}{2} = 10$ subarrays formed by $t = 2$ columns covers the nine 2-tuples over $\mathbb{Z}_3 = \{0, 1, 2\}$ at least once. For example, in the subarray formed by the second and by the last column, the first occurrence of the tuples $(0, 0)$, $(0, 1)$, $(0, 2)$, $(1, 0)$, $(1, 1)$, $(1, 2)$, $(2, 0)$, $(2, 1)$, $(2, 2)$ is colored in purple.

	↓			↓
0	2	0	1	0
2	0	2	1	2
1	2	0	2	2
0	2	2	0	1
2	1	0	0	0
2	0	0	2	1
0	1	2	2	2
0	0	1	0	2
1	0	2	0	0
1	1	1	1	1
2	2	1	2	0

Figure 1.2: A covering array CA$(11; 2, 5, 3)$. In the columns marked with a down arrow the first occurrence of the nine 2-tuples over $\mathbb{Z}_3$ is colored in purple.

The goal of the construction of CAs is to minimize the number of rows N. Then, for given number of columns k, strength t, and order v, the objective is to employ as few rows as possible to cover at least once all t-tuples over $\mathbb{Z}_v$ in every subarray of t columns. CAs have practical applications in several areas including GUI testing [54], fire accident reconstruction [53], testing the effects of multiple inputs in regulating a biological system [42], clustering business process models [38], and specially in combinatorial testing [26].

The objective of the combinatorial testing technique is to detect failures triggered by interactions among input parameters, that is, failures triggered when some input parameters take specific values. As an example consider the software component to set the advanced configurations of the Calendar application of the macOS operating system, shown in Figure 1.3. To perform an exhaustive testing



- [ ] Turn on time zone support
- [ ] Show events in year view
- [ ] Show week numbers
- [✓] Open events in separate windows
- [✓] Ask before sending changes to events

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

Figure 1.3: Example of the use of CAs as test suites. Every row of the CA is a test case for the five parameters of the software component.

of this software component we require a set of $2^5 = 32$ test cases, because there are two possible values (unselected and selected) for each of the five parameters. If only the interactions of size $t = 2$ are tested, then we require six test cases because we can use the CA$(6;2,5,2)$ of the same Figure 1.3 as test suite. Every row of this CA is a test case for the five input parameters; the value "unselected" is represented by 0 and "selected" is represented by 1. The six rows of this CA cover at least once the four possible configurations of values (unselected, unselected), (unselected, selected), (selected, unselected), and (selected, selected) between any two parameters. If for example a failure occurs when parameters p_0 and p_4 interact, then the failure will be detected because there is at least one test case for the four possible combinations of values between p_1 and p_4.

The value of the strength t modulates the coverage of interactions tested; if t is equal to the number of parameters k then we have full coverage. A series of studies conducted by the National Institute of Standards and Technology (NIST), in a wide range of domains, found that all failures in the software products under study were due to interactions involving at most six parameters [50, 28, 29, 27]. This result indicates that a failure is triggered by the interaction of a relatively small number of parameters, and therefore testing all configurations of size $t < k$ is an effective way to detect failures in software products.

The construction of CAs is a broad and active field, and several works have reviewed the different strategies to construct CAs. The works of Hartman [15] and Colbourn [8] review combinatorial and searching techniques. Lawrence *et al.* [31] present a survey for binary CAs, which are CAs of order two. Kuliamin and Petukhov [30] summarize a large number of methods to construct CAs, and a special characteristic of their work is the study of the complexity of the algorithms. Torres-Jimenez and Izquierdo-Marquez [45] describe briefly some construction methods. Nie and Leung [35] study construction methods in the context of test suite generation for combinatorial testing; also Khalsa and Labiche [23] present the methods in the context of combinatorial testing and review methods to construct CAs of various types, like CAs with variable strength and CAs with constraints. Finally, the work of Zhang *et al*. [55] explains in great detail several techniques for constructing CAs.

1.2 Background on covering arrays

This section presents some important concepts about CAs needed in the rest of the document. Subsection 1.2.1 introduces the concepts of minimum number of rows (CAN) and maximum number of columns (CAK) for CAs; and Subsection 1.2.2 presents the three isomorphisms of CAs.

1.2.1 Covering array number

Given the values of the strength t, the number of columns k, and the order v, the problem of constructing CAs consists in finding the smallest N such that there exists a CA$(N; t, k, v)$. This smallest N is the *covering array number* of t, k, and v, which is denoted by CAN(t, k, v). Then, CAN$(t, k, v) = \min\{N : \exists$ CA$(N; t, k, v)\}$.

Currently, there is no direct way to obtain CAN(t, k, v) for general values of t, k, and v. Some relevant cases with known values of CAN(t, k, v) are these:

- CAN$(t, t+1, 2) = 2^t$ for each $t \geq 1$.
- CAN$(2, k, 2) = N$, where N is the least positive integer for which $\binom{N-1}{\lceil \frac{N}{2} \rceil} \geq k$ [21, 24].
- CAN$(t, v+1, v) = v^t$ for v prime-power and $v > t$ [5].
- CAN$(t, t+1, v) = v^t$ for v prime-power and $v \leq t$ [10].

- CAN$(3, v+2, v) = v^3$ for $v = 2^n$ [5].
- CAN$(v-1, v+2, v) = v^{v-1}$ for $v = 2^n$ [16].
- CAN$(t, t+2, 2) = \lfloor \frac{4}{3} 2^t \rfloor$ for each $t \geq 1$ [19].

Apart from these cases, only a few CANs have been found by computational search. In Chapter 2 we will review computational methods to obtain CAN values.

A CA with strength t and order v must have at least v^t rows; so a trivial *lower bound* for the covering array number is CAN$(t, k, v) \geq v^t$. Some works where CAN lower bounds are studied are [11, 7]. Similarly, a trivial *upper bound* for CAN(t, k, v) is v^k, which is the number of vectors of length k over $\mathbb{Z}_v$. The study of theoretical upper bounds on the size of CAs focuses on determining the value of CAN(t, k, v) as function of k for fixed t and v. Recent works on this topic are [13, 39].

The improvement of upper bounds for CAN(t, k, v) is an active research topic, motivated in part by the reduction of the number of test cases when CAs are used as test suites. In the last years, the Covering Array Tables [9] have been used as the main source to report improvements in the upper bounds of covering array numbers.

Another optimality criteria for CAs is the maximum number of columns k that we can have for fixed values of N, t, and v. The maximum number of columns k for which a CA$(N; t, k, v)$ exists is denoted by CAK$(N; t, v) = \max\{k : \exists\,\text{CA}(N; t, k, v)\}$. Values CAN and CAK are related: CAN$(t, k, v) = \min\{N : \text{CAK}(N; t, v) \geq k\}$ and CAK$(N; t, v) = \max\{k : \text{CAN}(t, k, v) \leq N\}$. A CA can be optimal in the number of rows but be non-optimal in the number of columns; if CAN$(t, k, v) = N$ and CAK$(N; t, v) = k$, then CA$(N; t, k, v)$ is optimal in both the number of rows and the number of columns.

1.2.2 Isomorphisms of covering arrays

There are three isomorphisms of CAs:

1. Permutation of rows

2. Permutation of columns

3. Permutation of symbols in a column

Any combination of these three operations produces an isomorphic CA. Let $A = \text{CA}(4; 2, 3, 2)$ be the the following CA:

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

Now, let B be the array whose rows 0, 1, 2, 3 are respectively the rows 2, 0, 3, 1 of A. Similarly, let C be the array whose columns 0, 1, 2 are respectively the columns 1, 2, 0 of A. Finally, let D be the array derived from A by permuting symbols in the last column of A. These arrays B, C, D, shown next, are isomorphic to A:

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

We will use the symbol $\simeq$ to denote isomorphism between CAs; so, $B \simeq A$, $C \simeq A$, and $D \simeq A$.

For a covering array CA$(N; t, k, v)$ there are $N!$ row permutations, $k!$ column permutations, and $(v!)^k$ different combinations of symbol permutations in the columns. The number of different combinations of symbol permutation is $(v!)^k$ because there are $v!$ possible symbol permutations for each of the k columns. The operation of symbol permutation in a column is also called *relabeling*. In this way, the number of CAs isomorphic to one in particular is $N!\,k!\,(v!)^k$.

Isomorphic CAs 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 CA then the matrix after the operations is also a CA. Similarly, if the initial matrix has m missing tuples, then the matrix after the operations also has m missing tuples, although not necessarily the same missing tuples.

On the other hand, non-isomorphic CAs can not be transformed among them by permutations of rows, columns, and symbols. For particular values of N, t, k, v, the set of all covering arrays CA(N; t, k, v) is partitioned in classes $\mathcal{C}_0$, $\mathcal{C}_1$, ..., $\mathcal{C}_{n-1}$ of isomorphic CAs, where all CAs in a class are isomorphic among them, and no CA of one class is isomorphic to a CA of a distinct class. We will also use the term *distinct CAs* to mean non-isomorphic CAs. The classification problem is the generation of one element for each isomorphism class $\mathcal{C}_0$, $\mathcal{C}_1$, ..., $\mathcal{C}_{n-1}$.

In every isomorphism class $\mathcal{C}_0$, $\mathcal{C}_1$, ..., $\mathcal{C}_{n-1}$ we select one specific CA, the canonical one, to be the representative of the class. For $X = \text{CA}(N;t,k,v)$ let $\lambda(X)$ be the vector of length $N \cdot k$ obtained by arranging the elements of X in column-major order. The CA X is *canonical* if for all Y isomorphic to X the vector $\lambda(X)$ is smaller than or equal to $\lambda(Y)$ in lexicographic order. For example, the following CA $A = \text{CA}(6;2,7,2)$ is the canonical representative of its isomorphism class:

$$A = \text{CA}(6;2,7,2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

In this case $\lambda(A) = (0,0,0,0,1,1,0,0,1,1,0,1,0,0,1,1,1,0,0,1,0,1,0,1,0,1,0,1,1,0,0,1,1,0,0,1,0,1,1,0,1,0)$. For each of the $6!\,7!\,(2!)^7$ CAs $B = \text{CA}(6;2,7,2)$ isomorphic to A, the vector $\lambda(A)$ is smaller than or equal to $\lambda(B)$ in lexicographic order.

1.3 Classification of covering arrays

The classification problem for CAs is the problem of generating one element of each isomorphism class in the set of all CAs with particular values of N, t, k, and v. The book of Kaski and Östergård [20] presents general techniques for classifying combinatorial designs and error-correcting codes. One of these techniques is *orderly generation*, where only the canonical representatives of the isomorphism classes are considered in the search process, and all non-canonical elements are discarded; this is done to avoid the exploration of isomorphic objects. As an example of classification of CAs, Figure 1.4 shows the canonical CAs of the three isomorphism classes of CA$(6; 2, 7, 2)$.

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

Figure 1.4: The canonical representatives of the three isomorphism classes of CA$(6; 2, 7, 2)$.

Because the CAs obtained in the classification process are non-isomorphic among them, we use the terms *generation of non-isomorphic CAs* or *construction of non-isomorphic CAs* to be equivalent to the term *classification of CAs*; and if there are n isomorphism classes in the set of all CA$(N; t, k, v)$ we say there are n non-isomorphic covering arrays CA$(N; t, k, v)$.

The classification of CAs is relevant in itself because it enables the progress of the state of the art in the field of CAs. In addition, the classification of CAs is useful in the following practical situations:

- **Generation of all CAs**. Suppose $C_0, C_1, \ldots, C_{n-1}$ are the non-isomorphic CAs for the parameters N, t, k, v. Then, we can generate all possible CA$(N; t, k, v)$ by applying the operations of row permutation, column permutation, and symbol permutation to each $C_0, C_1, \ldots, C_{n-1}$.

- **Avoid redundant work**. In many tasks all isomorphic CAs give the same result. For example, all isomorphic CAs produce the same number of *wildcards*; being a wildcard an entry to which any value can be assigned without affecting the coverage properties of the CA. In these tasks it is enough to consider only the non-isomorphic CAs $C_0, C_1, \ldots, C_{n-1}$.

- **Find new CANs**. If CA$(N; t, k, v)$ is known to exist and we find zero isomorphism classes for CA$(N - 1; t, k, v)$, then we have determined CAN$(t, k, v) = N$.

Table 1.1: Families of optimal CAs with only one isomorphism class.

CA	Restrictions	# Classes	Reference
CA$(N; 2, \binom{N-1}{\lceil \frac{N}{2} \rceil}, 2)$	$N \geq 4$	1	[21]
CA$(\lfloor \frac{4}{3} 2^t \rfloor; t, t+2, 2)$	$t \geq 1$	1	[19]
CA$(2^t; t, t+1, 2)$	$t \geq 1$	1	[40]
CA$(3^t; t, t+1, 3)$	$t \geq 1$	1	[17]

There are combinations of the parameters N, t, k, v for which the number of isomorphism classes is known. When $k = t$ and $N = v^t$ the unique solution up to isomorphisms is obtained by listing the v^t tuples of length t over $\mathbb{Z}_v$. So, the number of isomorphism classes for CA$(v^t; t, t, v)$ is one. Other families of optimal CAs for which there is only one isomorphism class are given in Table 1.1. The first column of the table shows the resulting CA, the second column indicates the restrictions on the parameters of the CA, the third columns shows the number of isomorphism classes, and the last column contains the references.

In addition to these infinite families of algebraic results, a number of CAs have been classified by computation; these CAs are shown in Tables 1.2 and 1.3. The results in the tables do not include those CAs classified by algebraic techniques.

Table 1.2: Optimal CAs classified by computation, part I.

CA	# Classes	Reference
CA$(6; 2, 5, 2)$	7	[22]
CA$(6; 2, 6, 2)$	4	[22]
CA$(6; 2, 7, 2)$	3	[22]
CA$(6; 2, 8, 2)$	1	[22]
CA$(6; 2, 9, 2)$	1	[22]
CA$(7; 2, 11, 2)$	26	[22]
CA$(7; 2, 12, 2)$	10	[22]
CA$(7; 2, 13, 2)$	4	[22]
CA$(7; 2, 14, 2)$	1	[22]
CA$(8; 2, 16, 2)$	700 759	[22]
CA$(8; 2, 17, 2)$	579 466	[22]
CA$(8; 2, 18, 2)$	440 826	[22]
CA$(8; 2, 19, 2)$	309 338	[22]
CA$(8; 2, 20, 2)$	200 326	[22]
CA$(8; 2, 21, 2)$	119 752	[22]
CA$(8; 2, 22, 2)$	65 993	[22]
CA$(8; 2, 23, 2)$	33 463	[22]
CA$(8; 2, 24, 2)$	15 596	[22]
CA$(8; 2, 25, 2)$	6 704	[22]
CA$(8; 2, 26, 2)$	2 646	[22]
CA$(8; 2, 27, 2)$	977	[22]
CA$(8; 2, 28, 2)$	343	[22]
CA$(8; 2, 29, 2)$	118	[22]
CA$(8; 2, 30, 2)$	39	[22]
CA$(8; 2, 31, 2)$	15	[22]
CA$(8; 2, 32, 2)$	5	[22]
CA$(8; 2, 33, 2)$	2	[22]
CA$(8; 2, 34, 2)$	1	[22]

Table 1.3: Optimal CAs classified by computation, part II.

CA	# Classes	Reference
CA$(12; 3, 6, 2)$	9	[11]
CA$(12; 3, 7, 2)$	2	[11]
CA$(12; 3, 8, 2)$	2	[11]
CA$(12; 3, 9, 2)$	1	[11]
CA$(12; 3, 10, 2)$	1	[11]
CA$(12; 3, 11, 2)$	1	[11]
CA$(15; 3, 12, 2)$	2	[46]
CA$(16; 3, 13, 2)$	89	[46]
CA$(24; 4, 7, 2)$	1	[11]
CA$(24; 4, 8, 2)$	1	[11]
CA$(24; 4, 9, 2)$	1	[11]
CA$(24; 4, 10, 2)$	1	[11]
CA$(24; 4, 11, 2)$	1	[11]
CA$(24; 4, 12, 2)$	1	[11]
CA$(9; 2, 4, 3)$	1	[11]
CA$(11; 2, 5, 3)$	3	[11]
CA$(12; 2, 6, 3)$	13	[11]
CA$(12; 2, 7, 3)$	1	[11]
CA$(13; 2, 8, 3)$	5	[46]
CA$(13; 2, 9, 3)$	4	[46]
CA$(14; 2, 10, 3)$	4 490	[25]
CA$(16; 2, 3, 4)$	2	[33]
CA$(16; 2, 4, 4)$	1	[11]
CA$(16; 2, 5, 4)$	1	[11]
CA$(19; 2, 6, 4)$	4	[25]
CA$(25; 2, 3, 5)$	2	[33]
CA$(25; 2, 4, 5)$	1	[12]
CA$(25; 2, 5, 5)$	1	[12]
CA$(25; 2, 6, 5)$	1	[12]
CA$(29; 2, 7, 5)$	281	[25]
CA$(36; 2, 3, 6)$	12	[33]
CA$(37; 2, 4, 6)$	13	[25]
CA$(39; 2, 5, 6)$	289	[25]
CA$(49; 2, 3, 7)$	147	[33]
CA$(49; 2, 4, 7)$	7	[12]
CA$(49; 2, 5, 7)$	1	[12]
CA$(49; 2, 6, 7)$	1	[12]
CA$(49; 2, 7, 7)$	1	[12]
CA$(49; 2, 8, 7)$	1	[12]
CA$(64; 2, 3, 8)$	283 657	[33]
CA$(64; 2, 4, 8)$	2165	[12]
CA$(64; 2, 5, 8)$	39	[12]
CA$(64; 2, 6, 8)$	1	[12]
CA$(64; 2, 7, 8)$	1	[12]
CA$(64; 2, 8, 8)$	1	[12]
CA$(64; 2, 9, 8)$	1	[12]
CA$(81; 2, 3, 9)$	19 270 853 541	[33]
CA$(81; 2, 4, 9)$	91 846 374	[12]
CA$(81; 2, 5, 9)$	371	[12]
CA$(81; 2, 6, 9)$	96	[12]
CA$(81; 2, 7, 9)$	56	[12]
CA$(81; 2, 8, 9)$	15	[12]
CA$(81; 2, 9, 9)$	11	[12]
CA$(81; 2, 10, 9)$	7	[12]
CA$(100; 2, 3, 10)$	34 817 397 894 749 939	[33]

In the results of Tables 1.2 and 1.3 we make the following observations:

- The algorithms in the references [33, 22, 12, 25] only handle strength $t = 2$.
- In addition [12] and [33] only consider cases where $N = v^2$, because their purpose is to classify Latin squares and MOLS (mutually orthogonal Latin squares).
- The algorithms of [11] and [46] can handle general values of N, t, k, v.

The algorithms in [33, 22, 12] only handle strength two because they were developed to classify other combinatorial objects that are equivalent to CAs of strength two. The algorithm of [25] was developed to classify CAs but only strength two CAs are reported. On the other hand, the algorithms in [11, 46] can handle general values of N, t, k, v. However, the algorithm of [46], called NonIsoCA, can construct all the results obtained with the algorithm of [11], plus other additional results. Then, we use the algorithm of [11] as a basis to develop a more sophisticated classification algorithm in Chapter 3. In Chapter 4 we develop another classification algorithm that is not based on a previous one; this algorithm can also handle general values of N, t, k, v.

The algebraic and computational methods cited in Tables 1.1, 1.2, and 1.3 will be described in more detail in Chapter 2.

1.4 Research problem, hypothesis, and objectives

The algorithms of [11] and [46] construct the non-isomorphic CAs in a similar way. Starting from a CA with one column, more columns are added one at a time until the desired number of columns is reached. Each added column must form a CA of strength t with the previous columns, and this CA must be non-isomorphic to previously generated CAs with the same number of columns. The main limitation of these two algorithms is that they test all columns lexicographically greater than the last one in order to extend the current CA with one more column (as we will see in Chapter 2). This limitation makes the algorithms impractical for larger instances. Our motivation is that we can develop more efficient algorithms by using the coverage properties and the isomorphisms of CAs. So, we can skip a candidate columns if the resulting CA will be isomorphic to a previously explored CA.

The research problem and the research hypothesis are as follows:

Research problem:

To develop classification algorithms that use the coverage properties and the isomorphisms of CAs to make the search more efficient.

Research hypothesis:

It is possible to develop faster algorithms for the classification of CAs by taking advantage of the coverage properties and the isomorphisms of CAs.

The main purpose of developing new algorithms for the classification of CAs is to find new results, i.e., to find for the first time the non-isomorphic CAs that exist for some specific values of the parameters N, t, k, v.

General objective:

To develop new classification algorithms able to classify some CAs larger than those classified by state of the art algorithms. Specially CAs with strength $t > 2$.

The **particular objectives** are:

1. To develop an improved version of the NonIsoCA algorithm reported in [46]. The improved NonIsoCA algorithm should take into account the coverage properties and the isomorphisms of CAs to reduce the number of candidate columns to extend a CA.

2. To develop an algorithm that tests all possible juxtapositions of v CAs CA$(N_0; t, k, v)$, CA$(N_1; t, k, v)$, $\ldots$, CA$(N_{v-1}; t, k, v)$ to find all non-isomorphic CA$(N; t+1, k+1, v)$, where $N = \sum_{i=0}^{v-1} N_i$. This algorithm should be able to use the isomorphisms and the coverage properties of CAs to avoid testing juxtapositions that are isomorphic to previously tested juxtapositions, and to avoid testing juxtapositions with no possibilities of being a CA with strength $t+1$ and $k+1$ columns.

As we will see in Chapter 4, the central idea in the method based on juxtapositions is that we can construct any CA with strength $t+1$ and $k+1$ columns by juxtaposing v CAs of strength t and k columns, plus a column formed by v constant subcolumns. So, if we explore all possible juxtapositions of v CAs with strength t and k columns, then we will obtain all possible CAs with strength $t+1$ and $k+1$ columns.

1.5 Organization of the document

The thesis document has five more chapters; in the next list we give a brief summary for each of them:

- *Chapter 2 State of the Art*. This chapter reviews the current methods to classify CAs, and to classify objects equivalent to CAs of strength two. Exact methods to construct CAs are also reviewed, as well as methods based on juxtapositions of smaller objects.

- *Chapter 3 Improved NonIsoCA Algorithm*. In this chapter we develop an improved version of the NonIsoCA algorithm. The improved version is faster than the original algorithm in most cases. To obtain the results in less time, a parallel version of the improved NonIsoCA algorithm is developed.

- *Chapter 4 Juxtaposition of Covering Arrays*. This chapter describes the strategy to test all possible ways of constructing a CA of strength $t+1$ and $k+1$ columns from the juxtapositions of v CAs of strength t and k columns. In this case we develop one sequential version and two parallel versions of the algorithm.

- *Chapter 5 Computational Results*. This chapter presents the main computational results obtained. The chapter gives the results found by using the improved NonIsoCA algorithm, and the results of the algorithm based on juxtapositions of CAs. The complexity of the two algorithms is used to estimate which one will execute faster for an specific instance of the classification problem. In addition, the performance of the parallel implementations of the algorithms is studied.

- *Chapter 6 Conclusions*. The final chapter summarizes the contributions of the present work and provides pointers for future research.

1.6 Chapter summary

The classification problem for CAs implies the generation one element for each isomorphism class. This problem is relevant in itself because the classification problem is very important for all combinatorial designs, but the classification problem has also practical uses like determining new covering array numbers by computational search.

The problem addressed in this doctoral thesis is the development of two new classification algorithms that take advantage of the coverage properties and the isomorphisms of CAs to accelerate the search. The first algorithm is based on a previous algorithm called NonIsoCA, and the second algorithm is completely new. The main objective of this work is to classify some CAs larger than those classified by state of the art algorithms.

In the next chapter we review the methods that have been developed to classify CAs, and to construct CAs by exhaustive computational search.

2 State of the Art

This chapter reviews the methods that have been developed to classify CAs either directly or indirectly by classifying objects equivalent to CAs. In Section 2.1 four families of optimal CAs with a unique isomorphism class are reviewed. Section 2.2 describes computational methods to classify 2-surjective codes, Latin squares, and mutually orthogonal Latin squares (MOLS), which are objects equivalent to CAs of strength two; this section also reviews computational methods developed specifically to classify CAs. Computational classification methods are related to exact methods to construct CAs, however the objective of the construction methods is to find only one CA instead of one CA for each isomorphism class; some exact methods for constructing CAs are reviewed in Section 2.3. The construction of combinatorial objects by juxtapositions of smaller objects is reviewed in Section 2.4; the classification method of Chapter 4 was inspired by the techniques described in this section.

2.1 Algebraic methods

To the best of our knowledge there are four families of CAs whose members are optimal and unique; the term unique indicates that there is only one isomorphism class. These families are the ones given in Table 1.1 of the previous chapter; but now they are described in more detail. Subsection 2.1.1 describes the case $t = v = 2$ with the maximum number of columns; Subsection 2.1.2 describes the construction of Johnson and Entringer; Subsection 2.1.3 shows the classification of the optimal CA with $v = 2$ and $k = t + 1$; Subsection 2.1.4 presents the Zero-Sum construction for order $v = 3$; and Subsection 2.1.5 presents two families of optimal CAs whose elements are probably unique but the number of isomorphism classes is unknown.

2.1.1 Case $t = v = 2$

Given N, a CA$(N; 2, k, 2)$ with $k = \binom{N-1}{\lceil \frac{N}{2} \rceil}$ columns is constructed by placing as columns the distinct binary vectors of length N with a 0 as their first element and having $\lceil \frac{N}{2} \rceil$ 1's. This number of columns is the maximum possible. Katona [21] proved that this CA$(N; 2, k, 2)$ is optimal and unique.

Figure 2.1 shows an example of the construction for $N = 6$. In this case we have $k = \binom{6-1}{\lceil \frac{6}{2} \rceil} = \binom{5}{3} = 10$. The columns of CA$(6; 2, 10, 2)$ are the 10 binary vectors of length 6, having $\lceil \frac{6}{2} \rceil = 3$ ones, and with a zero in the first position.

$$\text{CA}(6; 2, 10, 2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 2.1: Example of the construction for $t = v = 2$.

Table 2.1 shows the first seven members of the family CA$(N; 2, \binom{N-1}{\lceil \frac{N}{2} \rceil}, 2)$. Since the CAs have strength two the smallest value of N is four.

Table 2.1: First seven optimal and unique CAs with $t = v = 2$ and maximum number of columns.

N	CA$(N; 2, \binom{N-1}{\lceil \frac{N}{2} \rceil}, 2)$
4	CA$(4; 2, 3, 2)$
5	CA$(5; 2, 4, 2)$
6	CA$(6; 2, 10, 2)$
7	CA$(7; 2, 15, 2)$
8	CA$(8; 2, 35, 2)$
9	CA$(9; 2, 56, 2)$
10	CA$(10; 2, 126, 2)$

From $k = \binom{(N-1)-1}{\lceil \frac{N-1}{2} \rceil} + 1$ to $k = \binom{N-1}{\lceil \frac{N}{2} \rceil} - 1$ the covering array CA$(N; 2, k, 2)$ is optimal, although it is not necessarily unique.

2.1.2 Johnson-Entringer construction

The construction of Johnson and Entringer [19] gives an optimal and unique CA$(\lfloor \frac{4}{3} 2^t \rfloor; t, t+2, 2)$ for each strength $t \geq 1$. Let $|u|$ be the weight of the binary vector u, and let $k = t + 2$. The 2^k binary vectors are partitioned in three sets V_k^0, V_k^1, V_k^2, where for $j = 0, 1, 2$ the set V_k^j contains the binary vectors u of length k such that $|u| \equiv j \pmod 3$. The three sets V_k^0, V_k^1, V_k^2 are CAs of strength t and $k = t + 2$ columns, and at least one of them has $\lfloor \frac{4}{3} 2^t \rfloor$ rows.

Consider for example $t = 4$; then $k = t+2 = 6$. Figure 2.2 shows the three CAs given respectively by the sets V_6^0, V_6^1, V_6^2. In this case there are 64 binary vectors of length 6, and the sets V_6^0, V_6^1, V_6^2 contain respectively 22, 21, and 21 vectors. The value $\lfloor \frac{4}{3} 2^t \rfloor$ is equal to $\lfloor \frac{4}{3} 2^4 \rfloor = \lfloor 21.3333 \rfloor = 21$, which is the number of rows of the CA$(21; 4, 6, 2)$ given by the sets V_6^1 and V_6^2. These two CAs are isomorphic because there is only one isomorphism class for CA$(\lfloor \frac{4}{3} 2^t \rfloor; t, t+2, 2)$. Table 2.2 shows the first seven members of the family CA$(\lfloor \frac{4}{3} 2^t \rfloor; t, t+2, 2)$.

The method of Torres-Jimenez *et al.* [47] based on juxtapositions of set of vectors represented by binomial coefficients also produces the CAs of the Johnson-Entringer construction.

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

Figure 2.2: CAs produced by the construction of Johnson-Entringer for $t = 4$.

Table 2.2: First seven optimal and unique CAs obtained with the Johnson-Entringer construction.

t	CA$(\lfloor\frac{4}{3}2^t\rfloor; t, t+2, 2)$
1	CA$(2; 1, 3, 2)$
2	CA$(5; 2, 4, 2)$
3	CA$(10; 3, 5, 2)$
4	CA$(21; 4, 6, 2)$
5	CA$(42; 5, 7, 2)$
6	CA$(85; 6, 8, 2)$
7	CA$(170; 7, 9, 2)$

2.1.3 Binary orthogonal arrays with $t + 1$ columns

An orthogonal array OA$_\lambda(N; t, k, v)$ is an $N \times k$ array over $\mathbb{Z}_v$ where every subarray of t columns covers exactly λ times each t-tuple with symbols from $\mathbb{Z}_v$. When the index λ is equal to 1 it is omitted from the notation and the OA$(N; t, k, v)$ has $N = v^t$ rows; so it is an optimal CA$(v^t; t, k, v)$.

For $t \geq 1$ there is only one orthogonal array OA$(2^t; t, t+1, 2)$ of index unity according to Seiden a Zemach [40]. They showed that for any $t \geq 1$ the set of all binary tuples of length $t+1$, which is denoted by $\mathbb{Z}_2^{t+1}$, can be split in a unique way into two orthogonal arrays OA$(2^t; t, t+1, 2)$. The partitioning of the set $\mathbb{Z}_2^{t+1}$ is done as follows: take any tuple $x \in \mathbb{Z}_2^{t+1}$; then, there are exactly $2^t - 1$ tuples in $\mathbb{Z}_2^{t+1}$ which differ from x in a even number of positions; these $2^t - 1$ tuples together with x form the first OA$(2^t; t, t+1, 2)$, and the other 2^t tuples of $\mathbb{Z}_2^{t+1}$ form the second OA$(2^t; t, t+1, 2)$.

These two OA$(2^t; t, t+1, 2)$ are the only ones that can be formed with tuples from $\mathbb{Z}_2^{t+1}$, because in any orthogonal array with strength t and $t+1$ columns any two rows differ in a even number of positions, according to the same work [40]. Finally, the two OAs are isomorphic because we can transform one into the other by permuting symbols in one of the $t+1$ columns and rearranging the 2^t rows.

Consider the case $t = 4$. The set $\mathbb{Z}_2^{t+1} = \mathbb{Z}_2^5$ contains the 32 binary 5-tuples $(0,0,0,0,0)$, $(0,0,0,0,1)$, ..., $(1,1,1,1,1)$. Take any tuple of $\mathbb{Z}_2^5$, say $x = (1,1,1,0,0)$; then, the 15 tuples of $\mathbb{Z}_2^5$ that differ from x in an even number of positions form the first OA$(16; 4, 5, 2)$, and the 16 tuples of $\mathbb{Z}_2^5$ that differ from x in an odd number of positions form the second OA$(16; 4, 5, 2)$; these two OAs are shown in Figure 2.3. These two OA$(16; 4, 5, 2)$ are isomorphic because we can transform the first one into the second one by permuting symbols in any of the five columns and then rearranging the rows of the resulting array.

Table 2.3 shows the first seven members of the family OA$(2^t; t, t+1, 2)$, or which it is the same the family CA$(2^t; t, t+1, 2)$.

Table 2.3: First seven optimal and unique CAs of the case $v = 2$ and $k = t + 1$.

t	CA$(2^t; t, t+1, 2)$
1	CA$(2; 1, 2, 2)$
2	CA$(4; 2, 3, 2)$
3	CA$(8; 3, 4, 2)$
4	CA$(16; 4, 5, 2)$
5	CA$(32; 5, 6, 2)$
6	CA$(64; 6, 7, 2)$
7	CA$(128; 7, 8, 2)$

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

Figure 2.3: The two OA$(16; 4, 5, 2)$ that make a partition of the 32 binary tuples of length 5. In each OA any two rows differ in an even number of positions.

2.1.4 Zero-Sum

The Zero-Sum technique [10] constructs an OA$(v^t; t, t+1, v)$ from the trivial OA$(v^t; t, t, v)$ that is generated by listing the v^t tuples of length t over $\mathbb{Z}_v$. Each element of the new column is the negative of the sum of the elements in the first t columns; so the sum of the $t+1$ elements in every row is zero. When the order is two or three, the OA given by the Zero-Sum construction is unique.

For order $v = 2$ the Zero-Sum technique adds a parity bit to each row of CA$(2^t; t, t, 2)$, as shown in the OA at the right in Figure 2.3 for the case $t = 4$. The CA$(16; 4, 4, 2)$ is formed by the 16 binary tuples of length 4, and a CA$(16; 4, 5, 2)$ is obtained by adding a parity bit to the 16 binary tuples of length 4. So, the two OA$(2^t; t, t+1, 2)$ given by the partitioning of Seiden a Zemach [40] are equivalent to the OA$(2^t; t, t+1, 2)$ given by the Zero-Sum construction.

For $v = 3$ the uniqueness of OA$(3^t; t, t+1, 3)$ was proved by Hedayat *et al.* [17] by showing that the element of $\mathbb{Z}_3$ appended to the row $(x_0, x_1, \ldots, x_{t-1})$ of $X =$ OA$(3^t; t, t+1, 3)$ is given by $\sum_{i=0}^{t-1} c_i x_i + c$, where $c_0, \ldots, c_{t-1} \in \{1, 2\}$ and $c \in \{0, 1, 2\}$. Then, without taking into account row permutations, there are $3(2^t)$ possible OA$(3^t; t, t+1, 3)$, and any of these OAs can be transformed into any other OA by permutations of symbols in the columns.

Figure 2.4 shows an example of the Zero-Sum construction for $v = 3$ and $t = 3$; the elements of the extra column of CA$(27; 3, 4, 3)$ make zero (in modulo $v = 3$) the sum of the four elements of each row of CA$(27; 3, 3, 3)$. Table 2.4 shows the first seven members of the family CA$(3^t; t, t+1, 3)$.

$$\text{CA}(27;3,3,3) = \begin{pmatrix} 0&0&0\\ 0&0&1\\ 0&0&2\\ 0&1&0\\ 0&1&1\\ 0&1&2\\ 0&2&0\\ 0&2&1\\ 0&2&2\\ 1&0&0\\ 1&0&1\\ 1&0&2\\ 1&1&0\\ 1&1&1\\ 1&1&2\\ 1&2&0\\ 1&2&1\\ 1&2&2\\ 2&0&0\\ 2&0&1\\ 2&0&2\\ 2&1&0\\ 2&1&1\\ 2&1&2\\ 2&2&0\\ 2&2&1\\ 2&2&2 \end{pmatrix} \qquad \text{CA}(27;3,4,3) = \begin{pmatrix} 0&0&0&0\\ 0&0&1&2\\ 0&0&2&1\\ 0&1&0&2\\ 0&1&1&1\\ 0&1&2&0\\ 0&2&0&1\\ 0&2&1&0\\ 0&2&2&2\\ 1&0&0&2\\ 1&0&1&1\\ 1&0&2&0\\ 1&1&0&1\\ 1&1&1&0\\ 1&1&2&2\\ 1&2&0&0\\ 1&2&1&2\\ 1&2&2&1\\ 2&0&0&1\\ 2&0&1&0\\ 2&0&2&2\\ 2&1&0&0\\ 2&1&1&2\\ 2&1&2&1\\ 2&2&0&2\\ 2&2&1&1\\ 2&2&2&0 \end{pmatrix}$$

Figure 2.4: Example of the Zero-Sum technique to add a column to a CA$(27; 3, 3, 3)$.

Table 2.4: First seven optimal and unique CAs obtained with the Zero-Sum construction for $v = 3$.

t	CA$(2^t; t, t+1, 2)$
1	CA$(3; 1, 2, 3)$
2	CA$(9; 2, 3, 3)$
3	CA$(27; 3, 4, 3)$
4	CA$(81; 4, 5, 3)$
5	CA$(243; 5, 6, 3)$
6	CA$(729; 6, 7, 3)$
7	CA$(2187; 7, 8, 3)$

2.1.5 Other families of orthogonal arrays

To the best of our knowledge it is unknown if the Zero-Sum construction gives unique CAs for orders $v \geq 4$. So, the uniqueness of OA$(v^t; t, t+1, v)$ is an open question when $v \geq 4$.

Another construction for OAs of index unity is the construction of Bush [5]. This construction gives an OA$(v^t; t, v+1, v)$ when v is prime-power and $v > t$; the constructed OA is not unique in general. However there is a special case: when v is a power of two $(v = 2^n)$ and the strength is three the construction produces an OA$(v^3; 3, v+2, v)$. This OA has one more column than the OA of the general case, and so it is more likely to be unique, but we do not know if the OA is unique.

Another family of OAs of index unity is the family OA$(v^{v-1}; v-1, v+2, v)$ where $v = 2^n$, which is given in [16]; in this case the number of columns is also $v+2$. Again, to the best of our knowledge the uniqueness of these OAs is unknown.

2.2 Computational methods

For general values of N, t, k, v, the only current way to classify CA$(N; t, k, v)$ is by computational methods. These methods explore the entire search space, although they use the isomorphisms of CAs to make cuts in the search tree. In this section we briefly review the known computational methods. The algorithms revised in Subsections 2.2.1, 2.2.2, and 2.2.3 were developed to classify other objects equivalent to CAs of strength two. The methods described in Subsections 2.2.4 and

2.2.5 can handle strengths greater than two; and the method of Subsection 2.2.6 is also general although it has been used only for CAs of strength two.

2.2.1 Classification of 2-surjective binary codes

A v-ary code of length k is a set of vectors or codewords of length k over $\mathbb{Z}_v$. A code C is t-surjective if for any t coordinates $j_0, j_1, \ldots, j_{t-1}$ and any t-tuple $(x_0, x_1, \ldots, x_{t-1}) \in \mathbb{Z}_v^t$ there is a codeword $c \in C$ such that $c_{j_i} = x_i$ for $0 \leq i \leq t-1$. According to this definition, t-surjective codes are CAs of strength t.

Kéri and Östergård [22] classified 2-surjective binary codes with $N = 6, 7, 8$ codewords and lengths from $k = 2$ to $k = \binom{N-1}{\lceil \frac{N}{2} \rceil}$. The classification was done by exhaustive computer search. The numbers of inequivalent codes (non-isomorphic CAs) they computed are shown in Table 2.5.

2-surjective binary codes are CAs of strength two and order two; so the value at row k and column N of Table 2.5 indicates the number of non-isomorphic CA$(N; 2, k, 2)$. For $N = 6$ the maximum length of the codewords is $k = \binom{6-1}{\lceil \frac{6}{2} \rceil} = \binom{5}{3} = 10$, and similarly for $N = 7, 8$ the maximum number of codewords is 15 and 36. The extremal values $k = \binom{N-1}{\lceil \frac{N}{2} \rceil}$ give the family of optimal and unique CAs of Subsection 2.1.1. Not all 2-surjective codes or CAs that were classified are optimal, but the classification of non-optimal CAs is also important.

2.2.2 Classification of Latin squares

McKay *et al.* [33] classified Latin squares of order 1 to 10. A Latin square L of order n is an $n \times n$ array such that each row and each column contains a permutation of $\{0, 1, \ldots, n-1\}$. A Latin square can be represented by an orthogonal array OA$(n^2; 2, 3, n)$ formed by the 3-tuples $\{(i, j, L[i, j]) \,|\, 0 \leq i, j \leq n-1\}$.

Two Latin squares are *paratopic* if their associated OAs are isomorphic. All paratopic Latin squares form a *paratopy class*; paratopy classes are also called *species* or *main classes*. So, a main class of Latin squares is an isomorphism class in the OA representation. In summary, classifying

Table 2.5: Inequivalent 2-surjective binary codes with cardinality $N = 6, 7, 8$ [22].

k	$N=6$	$N=7$	$N=8$
2	3	4	8
3	7	15	37
4	8	35	156
5	7	70	719
6	4	107	3 112
7	3	139	12 006
8	1	134	38 497
9	1	102	101 068
10	1	64	215 292
11		26	377 177
12		10	554 538
13		4	701 066
14		1	779 013
15		1	775 641
16			700 759
17			579 466
18			440 826
19			309 338
20			200 326
21			119 752
22			65 993
23			33 463
24			15 596
25			6 704
26			2 646
27			977
28			343
29			118
30			39
31			15
32			5
33			2
34			1
35			1

Latin squares of order v into main classes is the same as classifying OA$(v^2; 2, 3, v)$ or CA$(v^2; 2, 3, v)$. Table 2.6 shows the number of main classes for Latin squares of order up to 10.

Table 2.6: Main classes of Latin squares of order 1 to 10 [33].

Order	Main classes
1	1
2	1
3	1
4	2
5	2
6	12
7	147
8	283 657
9	19 270 853 541
10	34 817 397 894 749 939

2.2.3 Classification of MOLS

Two Latin squares $A = (a_{ij})$ and $B = (b_{ij})$ of order n are orthogonal if the n^2 pairs (a_{ij}, b_{ij}) are distinct. A set of Latin squares where each pair is orthogonal is called a set of *mutually orthogonal Latin squares* (MOLS). A set of k MOLS $L_0, L_1, \ldots, L_{k-1}$ of order n can be used to construct an orthogonal array OA$(n^2; 2, k+2, n)$ where the row $(i \cdot n) + j$ contains the $(k+2)$-tuple $(i, j, L_0[i,j], L_1[i,j], \ldots, L_{k-1}[i,j])$. Two set of MOLS are paratopic if they are represented by isomorphic OAs. Then, finding paratopy classes of k MOLS of order n, k-MOLS(n), is equivalent to classify OA$(n^2; 2, k+2, n)$. Sets of k-MOLS(n) for $2 \leq n \leq 9$ were classified by Egan and Wanless [12]. Table 2.7 shows the number of paratopy classes they found.

If n is prime-power, then the maximum number of MOLS is $n-1$. In Table 2.7 we can see that the maximum k for $n = 2, 3, 4, 5, 7, 8, 9$ is $k = 1, 2, 3, 4, 6, 7, 8$ respectively. The case $n = 9$ is the first one where the largest set of MOLS is not unique, since there are 7 paratopy classes of 8-MOLS(9).

2.2.4 Extension of CAs

The algorithm mentioned in Colbourn *et al.* [11] can handle general values of the parameters N, t, k, and v. This algorithm constructs the non-isomorphic CA$(N; t, k, v)$ as follows:

Table 2.7: Paratopy classes of k-MOLS(n) [12].

n	k	Paratopy classes
2	1	1
3	1	1
3	2	1
4	1	2
4	2	1
4	3	1
5	1	2
5	2	1
5	3	1
5	4	1
6	1	12
7	1	147
7	2	7
7	3	1
7	4	1
7	5	1
7	6	1
8	1	283 657
8	2	2 165
8	3	39
8	4	1
8	5	1
8	6	1
8	7	1
9	1	19 270 853 541
9	2	91 846 374
9	3	371
9	4	96
9	5	56
9	6	15
9	7	11
9	8	7

- The set of non-isomorphic CA($N; 1, 1, v$) is constructed by an independent computer program.
- For each non-isomorphic CA($N; 1, 1, v$) all v^N candidate columns are tested to see which of them form a CA($N; 2, 2, v$). An equivalence test is performed on the CA($N; 2, 2, v$) to discard isomorphic solutions. The process is repeated for the set of non-isomorphic CA($N; 2, 2, v$) to obtain the non-isomorphic CA($N; 3, 3, v$); and this continues until the non-isomorphic CA($N; t, t, v$) are obtained.
- For $j = t + 1, \ldots, k$ the set of non-isomorphic CA($N; t, j, v$) is constructed by extending (adding a new column) each of the non-isomorphic CA($N; t, j - 1, v$).

It is not indicated which CA is taken from an isomorphism class, and also it is not described the equivalence test that is applied to reject isomorphic solutions.

Table 2.8 lists the classified CAs. The result of zero isomorphism classes for CA($14; 3, 12, 2$) and the already known existence of CA($15; 3, 12, 2$) [37], proves CAN($3, 12, 2$) $= 15$. Similarly, the result of zero CA($11; 2, 6, 3$) implies CAN($2, 6, 3$) $=$ CAN($2, 7, 3$) $= 12$. Finally, the nonexistence of

CA$(12;2,8,3)$ and the existence of CA$(13;2,9,3)$, which was constructed in [44] as an equivalent object called transversal cover, gives CAN$(2,8,3)=$ CAN$(2,9,3)=13$.

Table 2.8: CAs classified by Colbourn *et al.* [11].

CA	# Classes	CA	# Classes
CA$(12;3,3,2)$	19	CA$(14;3,9,2)$	1336
CA$(12;3,4,2)$	79	CA$(14;3,10,2)$	989
CA$(12;3,5,2)$	33	CA$(14;3,11,2)$	533
CA$(12;3,6,2)$	9	CA$(14;3,12,2)$	0
CA$(12;3,7,2)$	2	CA$(11;2,2,3)$	3
CA$(12;3,8,2)$	2	CA$(11;2,3,3)$	20
CA$(12;3,9,2)$	1	CA$(11;2,4,3)$	27
CA$(12;3,10,2)$	1	CA$(11;2,5,3)$	3
CA$(12;3,11,2)$	1	CA$(11;2,6,3)$	0
CA$(12;3,12,2)$	0	CA$(12;2,2,3)$	7
CA$(14;3,3,2)$	68	CA$(12;2,3,3)$	134
CA$(14;3,4,2)$	657	CA$(12;2,4,3)$	987
CA$(14;3,5,2)$	1714	CA$(12;2,5,3)$	891
CA$(14;3,6,2)$	3376	CA$(12;2,6,3)$	13
CA$(14;3,7,2)$	3585	CA$(12;2,7,3)$	1
CA$(14;3,9,2)$	2395	CA$(12;2,8,3)$	0

2.2.5 NonIsoCA algorithm

The NonIsoCA algorithm of Torres-Jimenez and Izquierdo-Marquez [46] improved the algorithm of Colbourn *et al.* [11] described in Subsection 2.2.4. The NonIsoCA algorithm is able to produce all computational results of [11] plus other additional results.

The NonIsoCA algorithm extends a subarray $A=$ CA$(N;t,r,v)$ with $r<k$ columns $a_0,a_1,\ldots,a_{r-1}$ by testing all columns lexicographically greater than column a_{r-1} until finding a column l for which: (a) the columns $a_0,a_1,\ldots,a_{r-1},l$ form a CA$(N;t,r+1,v)$ of strength t, and (b) this CA is the canonical representative of its isomorphism class. The canonical arrays with one column are created by a special procedure that requires a small number of computations (see Algorithm 1 of [46]).

For example, suppose the algorithm wants to construct all non-isomorphic CA$(6; 2, 6, 2)$, and suppose the algorithm has constructed the canonical CA with three columns shown in Figure 2.5(a). For each column l greater than $a_2 = (0\ 0\ 1\ 1\ 0\ 1)^T$ the algorithm first verifies that $G = (a_0\ a_1\ a_2\ l)$ is a CA of strength $t = 2$, and then the algorithm tests if G is the canonical representative of its class. If G satisfies these two conditions then the algorithm has constructed a canonical CA with $r + 1$ columns, and the next step is to try to extend the array to $r + 2$ columns. The CA of Figure 2.5(b) shows the new column added to the CA of Figure 2.5(a).

$$\begin{pmatrix} 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 1 & 1 & * & * & * \\ 1 & 0 & 1 & * & * & * \\ 1 & 1 & 0 & * & * & * \\ 1 & 1 & 1 & * & * & * \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * & * \\ 0 & 1 & 1 & 1 & * & * \\ 1 & 0 & 1 & 1 & * & * \\ 1 & 1 & 0 & 1 & * & * \\ 1 & 1 & 1 & 0 & * & * \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 1 & * & * \\ 0 & 1 & 1 & 0 & * & * \\ 1 & 0 & 1 & 0 & * & * \\ 1 & 1 & 0 & 1 & * & * \\ 1 & 1 & 1 & 1 & * & * \end{pmatrix}$$
$$(a) \qquad\qquad\qquad (b) \qquad\qquad\qquad (c)$$

Figure 2.5: Some steps in the construction of the non-isomorphic CA$(6; 2, 6, 2)$: (a) a canonical CA with three columns; (b) the first CA with four columns that is generated when the algorithm extends the CA with three columns; and (c) the next constructed CA with four columns.

When no column l forms a canonical CA, the algorithm replaces the last column of the CA with the next column in lexicographic order, and starts the search of the next canonical CA with r columns. Suppose the CA of Figure 2.5(b) can not be extended to five columns; then, the algorithm replaces column a_3 with the next column greater than a_3 in lexicographic order to try to find another canonical CA with four columns; in this case the algorithms finds the CA of Figure 2.5(c).

At some point all canonical CAs with four columns will be generated and extended. At this moment the algorithm backtracks to the third column (a_2) and replaces it with the next column greater than a_2 in lexicographic order. This is done with the objective of generating the next canonical CA with three columns, from which the search of another canonical CA with four columns will restart. The backtracking process ends when all CAs of one column have been generated and expanded. Since the new columns added to the CA are generated in lexicographic order, every constructed canonical CA is different from all previously constructed canonical CAs.

Table 2.9 lists all classified CAs. The nonexistence of the covering arrays CA$(15; 3, 13, 2)$, CA$(16; 3, 15, 2)$, and CA$(13; 2, 10, 3)$ leads to the following results: CAN$(3, 13, 2)$ = CAN$(3, 14, 2) = 16$, CAN$(3, 15, 2)$ = CAN$(3, 16, 2) = 17$, and CAN$(2, 10, 3) = 14$.

Table 2.9: CAs classified by using the NonIsoCA algorithm [46].

CA	# Classes
CA$(4; 2, 2, 2)$	1
CA$(4; 2, 3, 2)$	1
CA$(5; 2, 4, 2)$	1
CA$(6; 2, 5, 2)$	7
CA$(6; 2, 6, 2)$	4
CA$(6; 2, 7, 2)$	3
CA$(6; 2, 8, 2)$	1
CA$(6; 2, 9, 2)$	1
CA$(6; 2, 10, 2)$	1
CA$(7; 2, 11, 2)$	26
CA$(7; 2, 12, 2)$	10
CA$(7; 2, 13, 2)$	4
CA$(7; 2, 14, 2)$	1
CA$(7; 2, 15, 2)$	1
CA$(8; 2, 16, 2)$	700 759
CA$(8; 2, 17, 2)$	579 466
CA$(8; 2, 18, 2)$	440 826
CA$(8; 2, 19, 2)$	309 338
CA$(8; 2, 20, 2)$	200 326
CA$(8; 2, 21, 2)$	119 752
CA$(8; 2, 22, 2)$	65 993
CA$(8; 2, 23, 2)$	33 463
CA$(8; 2, 24, 2)$	15 596
CA$(8; 2, 25, 2)$	6 704
CA$(8; 2, 26, 2)$	2 646
CA$(8; 2, 27, 2)$	977
CA$(8; 2, 28, 2)$	343
CA$(8; 2, 29, 2)$	118
CA$(8; 2, 30, 2)$	39
CA$(8; 2, 31, 2)$	15
CA$(8; 2, 32, 2)$	5
CA$(8; 2, 33, 2)$	2
CA$(8; 2, 34, 2)$	1
CA$(8; 2, 35, 2)$	1
CA$(8; 3, 3, 2)$	1
CA$(8; 3, 4, 2)$	1
CA$(10; 3, 5, 2)$	1
CA$(12; 3, 6, 2)$	9
CA$(12; 3, 7, 2)$	2
CA$(12; 3, 8, 2)$	2
CA$(12; 3, 9, 2)$	1
CA$(12; 3, 10, 2)$	1
CA$(12; 3, 11, 2)$	1
CA$(15; 3, 12, 2)$	2
CA$(15; 3, 13, 2)$	0
CA$(16; 3, 13, 2)$	89
CA$(16; 3, 14, 2)$	8
CA$(16; 3, 15, 2)$	0
CA$(16; 4, 4, 2)$	1
CA$(16; 4, 5, 2)$	1
CA$(21; 4, 6, 2)$	1
CA$(24; 4, 7, 2)$	1
CA$(24; 4, 8, 2)$	1
CA$(24; 4, 9, 2)$	1
CA$(24; 4, 10, 2)$	1
CA$(24; 4, 11, 2)$	1
CA$(24; 4, 12, 2)$	1
CA$(9; 2, 2, 3)$	1
CA$(9; 2, 3, 3)$	1
CA$(9; 2, 4, 3)$	1
CA$(11; 2, 5, 3)$	3
CA$(12; 2, 6, 3)$	13
CA$(12; 2, 7, 3)$	1
CA$(13; 2, 8, 3)$	5
CA$(13; 2, 9, 3)$	4
CA$(13; 2, 10, 3)$	0
CA$(16; 2, 2, 4)$	1
CA$(16; 2, 3, 4)$	2
CA$(16; 2, 4, 4)$	1
CA$(16; 2, 5, 4)$	1

2.2.6 Canonical augmentation

The canonical augmentation technique consists in extending a CA with $r < k$ columns in a canonical way, rather than searching a canonical CA with $r + 1$ columns. In the canonical augmentation technique the children of isomorphic parents must be isomorphic; in this case the parents have r

columns and the children have $r + 1$ columns. Then, if two CAs with k columns are isomorphic then their sequence of ancestors are isomorphic. In the orderly generation technique it is possible to have two isomorphic CAs with k columns that are not isomorphic if only the first $r < k$ columns are considered.

Children not satisfying the rule of canonical augmentation are rejected, i.e., they are not extended. For children satisfying the rule of canonical augmentation an isomorph test is performed to reject all but one of the isomorphic children. The canonical augmentation technique ensures the generation of one CA for each isomorphism class. In the doctoral thesis of Kokkala [25] canonical augmentation was used to classify CAs of strength two. Table 2.10 shows the classified CAs. The new classification results established 12 new CANs: CAN$(2, 7, 4) = 2$, CAN$(2, 5, 6) = 39$, and CAN$(2, k, 3) = 15$ for $k = 11, \ldots, 20$.

Table 2.10: CAs classified by canonical augmentation [25].

CA	# Classes
CA$(10; 2, 4, 3)$	2
CA$(10; 2, 5, 3)$	0
CA$(11; 2, 5, 3)$	3
CA$(11; 2, 6, 3)$	0
CA$(12; 2, 6, 3)$	13
CA$(12; 2, 7, 3)$	1
CA$(12; 2, 8, 3)$	0
CA$(13; 2, 8, 3)$	5
CA$(13; 2, 9, 3)$	4
CA$(13; 2, 10, 3)$	0
CA$(14; 2, 10, 3)$	4 490
CA$(14; 2, 11, 3)$	0
CA$(17; 2, 5, 4)$	4
CA$(17; 2, 6, 4)$	0
CA$(18; 2, 5, 4)$	201
CA$(18; 2, 6, 4)$	0
CA$(19; 2, 6, 4)$	4
CA$(19; 2, 7, 4)$	0
CA$(20; 2, 6, 4)$	25 760
CA$(20; 2, 7, 4)$	0
CA$(26; 2, 6, 5)$	6
CA$(26; 2, 7, 5)$	0
CA$(27; 2, 6, 5)$	11 603
CA$(27; 2, 7, 5)$	0
CA$(28; 2, 6, 5)$	75 720 344
CA$(28; 2, 7, 5)$	0
CA$(29; 2, 7, 5)$	281
CA$(29; 2, 8, 5)$	0
CA$(37; 2, 4, 6)$	13
CA$(37; 2, 5, 6)$	0
CA$(38; 2, 4, 6)$	8 865
CA$(38; 2, 5, 6)$	0
CA$(39; 2, 5, 6)$	289
CA$(39; 2, 6, 6)$	0

The improved NonIsoCA algorithm that will be studied in Chapter 3 also determined these covering arrays numbers. The work [25] and the improved NonIsoCA algorithm were developed independently and published at about the same time. Because both algorithms obtained the same results we have more confidence in the accuracy of the results we have found.

2.3 Exact methods to construct covering arrays

Exact methods construct CAs by exhaustive search in a similar way to the computational classification methods. Exact methods also use the isomorphisms of CAs for pruning the search space. However, most methods only consider the row and column symmetries, and omit symbol permutations. In this section we review some exact methods to construct CAs.

2.3.1 The automatic generator EXACT

Yan and Zhang [51] introduced an exhaustive search technique to construct CAs. The algorithm assigns each cell of an $N \times k$ matrix until all covering conditions are satisfied. After assigning a variable, a constraint propagation function is executed to determine if this assignment implies a value for another variable or a contradiction. The search is accelerated by symmetry breaking techniques and by two heuristics called respectively LNH and SCEH. The heuristic LNH uses a variable *mdn* to store the largest value present in the assigned cells. The candidate values for assigning a new cell are $\{0, 1, \ldots, \mathit{mdn} + 1\}$; so, values greater than $\mathit{mdn} + 1$ are not considered. The heuristic SCEH assumes that it is always possible to find a CA where each sub-combination (or sub-tuple) of size s occurs almost the same number of times in a subset of s columns. The authors integrated all these techniques in a program called EXACT (EXhaustive seArch of Combinatorial Test suites). The EXACT tool was further improved in [52].

2.3.2 New backtracking algorithm

Bracho-Rios *et al.* [4] introduced a searching algorithm to construct binary CAs of strength t and dimensions $N \times k$. The algorithm constructs the CAs column by column imposing a lexicographic ordering of the columns to break the column and row symmetries. The columns to construct the CAs are balanced in symbols, so the candidate columns have $\lfloor \frac{N}{2} \rfloor$ zeros and $N - \lfloor \frac{N}{2} \rfloor$ ones. Before starting the search, a block of t columns is fixed, the first $N - 2^t$ rows of the block are filled with zeros and the last 2^t rows are filled with the 2^t tuples of size t over the symbol set $\{0, 1\}$. Suppose we have a partial solution with r columns ($t \leq r < k$), and let l be the last column of the partial solution. To construct a CA of strength t and $r + 1$ columns, the algorithm checks all columns l' greater than l in lexicographic order until it finds one which makes a CA of strength t with the r columns of the partial solution, and such that the rows and columns of the new partial solution are sorted lexicographically. If no such column is found the algorithm backtracks to column $r - 1$.

2.3.3 NonIsoCA algorithm

The NonIsoCA algorithm of Torres-Jimenez and Izquierdo-Marquez [46] can also be used as an exact method for constructing CAs. As said before, this algorithm constructs the CAs column by column rejecting the non-canonical CAs. The objective of the algorithm is to construct all non-isomorphic CA($N; t, k, v$), but it can be updated to finish when the first CA($N; t, k, v$) is constructed. The action of expanding only the canonical CAs with $r < k$ columns acts as a powerful pruning criteria.

2.3.4 Constraint programming

Hnich *et al.* [18] developed constraint programming models for the construction of CAs. In the first model, called *naive matrix model*, a variable x_{ri} is set to m, $x_{ri} = m$, if the entry (r, i) of the matrix is equal to m. Consider $t = 3$; to express that the tuple in row r and in columns (i, j, l) is equal to (m, n, p) the constraint is $x_{rijlmnp} = (x_{ri} = m \; \& \; x_{rj} = n \; \& \; x_{rl} = p)$. The constraint that each t-tuple must occur at least once in every subset of t columns is expressed by $\sum_r x_{rijlmnp} \geq 1$.

In the *alternative matrix model* a tuple of t variables of the first model is represented by a *compound* variable. With $t = 3$, for example, the compound variable $y_{r(i,j,l)}$ represents the tuple of variables (x_{ri}, x_{rj}, x_{rl}). The domain of a compound variable is $\{0, 1, \ldots, v^t - 1\}$; and the coverage constraints require that in every subset of t columns there must be at least one compound variable for each number from 0 to $v^t - 1$.

The *integrated model* combines the variables of the two previous models. By assigning a value to a compound variable, a value is designated to each variable of the naive model. Similarly, assigning a value to a variable of the naive model reduces the domain of the compound variable.

2.3.5 SAT encodings

Lopez-Escogido *et al.* [32] introduced a SAT encoding for strength-two CAs. The model uses v variables for each entry of the matrix $M = (m_{ij})$ used to contain the CA; if M has size $N \times k$ the total number of variables is Nkv. Element m_{ij} gets the value $0 \leq x < v$ if and only if the variable $m_{i,j,x}$ is true. The clauses of the model guarantee that: (a) each element of M takes at least one value from the set $\{0, 1, \ldots, v-1\}$, (b) each element of M takes only one value, and (c) the matrix M satisfy the coverage properties to be a CA.

The work of Ansótegui *et al.* [1] proposes a MAXSAT encoding for the construction of optimal CAs. Banbara et al. [2] also developed two SAT encodings to construct CA; these encodings are called *order encoding* and *mixed encoding* respectively.

2.4 Juxtaposition of smaller objects

There are some constructions which juxtapose smaller objects to construct larger ones. These methods inspired the classification method we develop in Chapter 4.

In the book of Hedayat, Sloane, and Stufken [16] 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. Thus, an orthogonal array $A =$ OA$(N; 2u, k, 2)$ can be used to construct $B =$ 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:

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

The column vectors $\mathbf{0}$ and $\mathbf{1}$ are formed by N zeros and N ones respectively. This construction is limited to the case $v = 2$ and t even.

Other objects that can be constructed by juxtapositions of smaller objects are error-correcting codes. An (n, M, d) binary code is a set of M vectors of length n over the finite filed of order 2, $\mathbb{F}_2$, called codewords, whose minimum Hamming distance is d, that is, d is the minimum distance among any two codewords. If any linear combination of codewords is also a codeword then the code is *linear*, otherwise the code is *nonlinear*. A code with minimum distance d can correct $\lfloor (d-1)/2 \rfloor$ or fewer errors. An (n, M, d) code is optimal if M is the maximum number of codewords with length n having minimum distance d.

Nordstrom and Robinson [36] constructed a $(13, 64, 5)$ code by juxtaposing two Nadler codes $(12, 32, 5)$ [34], and by adding to this juxtaposition a column vector formed by 32 zeros and 32 ones. After that, they repeated the process and juxtaposed two $(13, 64, 5)$ codes to construct a $(14, 128, 5)$ code, and finally they took two copies of this last code to form a $(15, 256, 5)$ code. In every case the appropriate column of a block of zeros and a block of ones was added to the juxtaposition of the two codes. The last constructed code, $(15, 256, 5)$, is known as the Nordstrom-Robinson code, and it is an optimal nonlinear code because 256 is the maximum number of codewords of length 15 having minimum mutual distance 5. This code was also discovered independently by Semakov and Zinoviev [41].

Finally, the Tower of Covering Arrays method of [48] takes a covering array $A = \text{CA}(N; t, k, v)$, called the *base CA*, and tries to construct a covering array $B = \text{CA}(Nv; t+1, k+1, v)$ by juxtaposing vertically v copies of the base CA:

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

The copies A' of the base CA A are obtained by translating the columns of A. Translating a column of a CA with order v means to add a value $a \in \mathbb{Z}_v$ to every element of the column and take the modulo v of the sum in each entry. Column translation is a special case of symbol permutation because for each column of a CA$(N; t, k, v)$ there are v possible column translations, but there are $v!$ possible permutations of symbols, which include the v possible column translations. The last column of B is formed by v subcolumns, where for $0 \leq i \leq v-1$ the elements of the i-th subcolumn are equal to i.

2.5 Chapter summary

This chapter presented a brief summary of the different methods to classify CAs or to classify objects equivalent to CAs. The revised methods were grouped into algebraic and computational methods. The algebraic method for the case CA$(N; 2, k, 2)$, the Johnson-Entringer construction, and the Zero-Sum construction for $v = 2, 3$ provide infinite families of CAs which are both optimal and unique. The first three computational methods reviewed in this chapter were developed to classify 2-surjective codes, Latin squares, and MOLS respectively; but these objects are equivalent to CAs of strength two. The last three computational methods which were analyzed can handle general values of N, t, k, and v. In the next chapter we develop a new computational method that improves the NonIsoCA algorithm revised in this chapter.

We also reviewed exact methods to construct CAs because they are similar to the computational classification methods, since both kind of methods traverse the entire search space. Finally, we reviewed some methods based on juxtaposing smaller objects; these methods are related to the second classification method (developed in Chapter 4) that tests all possible juxtapositions of v CAs of strength t to construct the non-isomorphic CAs of strength $t+1$.

3 Improved NonIsoCA Algorithm

This chapter describes an improved version of the NonIsoCA algorithm to construct non-isomorphic CAs. The new algorithm, as the original algorithm, constructs CAs column by column rejecting the non-canonical CAs. However, the new algorithm constructs a new column cell by cell instead of testing all columns greater than the last one in lexicographic order. As a new column is constructed, the improved algorithm rejects the current subarray if it does not have possibilities of being a canonical CA of strength t. Section 3.1 presents the improved algorithm to classify CAs; Section 3.2 analyzes the complexity of the algorithm; and Section 3.3 describes a parallel implementation of the new algorithm.

3.1 Improved algorithm

The NonIsoCA algorithm introduced in [46] is an orderly algorithm in which the canonical representative of an isomorphism class is the CA with the smallest lexicographic order when its $N \cdot k$ elements are arranged in column-major order. The algorithm constructs the non-isomorphic

CAs one column at a time rejecting the non-canonical CAs. The result produced by this algorithm is a list of the canonical representatives of each isomorphism class. A detailed description of this algorithm is given in Subsection 2.2.5.

The improved algorithm is also an orderly algorithm. The improvement with regard to the original algorithm consists in generating the new column of a CA by filling the cells of the new column from top to bottom, instead of testing all columns greater than the last one in lexicographic order. The allowed values for a cell in the new column depend on the values assigned to the previous cells.

3.1.1 Rules for valid symbols

Suppose the algorithm has constructed a CA with r columns $A = \mathrm{CA}(N; t, r, v)$, and let c_0, c_1, …, c_{N-1} be the N cells of the new column. The algorithm fills the cells of the new column from top to bottom, i.e., from c_0 to c_{N-1}. Cells without an assigned symbol are called *free cells*. Before assigning a symbol to the next free cell the algorithm first determines which symbols are valid for the cell. A symbol is valid for the next free cell if after assigning the symbol to the cell all of the following rules are satisfied:

R_1: The array is sorted by rows and by columns.

R_2: The remaining free cells in the new column are enough to satisfy the minimum number of times that each symbol must occur in the new column.

R_3: The number of occurrences of each symbol in the new column is not greater than the number of zeros in the first column.

R_4: In every combination of t columns containing the new column the number of repeated tuples is less than or equal to the maximum allowed value.

R_5: In every combination of t columns containing the new column there is at least one candidate row to cover every tuple not yet covered.

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

$$(a) \qquad\qquad\qquad\qquad (b) \qquad\qquad\qquad\qquad (c)$$

Figure 3.1: Examples of the applications of rules R_1, R_2, and R_3 for filling the next free cell (first cell with $\ast$): (a) Symbols 0 and 1 do not satisfy R_1; (b) Symbol 2 does not satisfy R_2; and (c) Symbol 1 does not satisfy R_3.

R_1 is used because one characteristic of canonical CAs is that their rows and columns are sorted in lexicographic order. Figure 3.1(a) shows a CA$(11; 2, 3, 3)$ and the new column being constructed; symbols 0 and 1 are not valid for the next free cell because they will produce an array that is unordered by columns.

R_2 guarantees the occurrence of each symbol at least v^{t-1} times in every column of a CA, because this is the required number of times for a CA of strength t and order v. In this way, a symbol is not valid for the next cell to fill if the remaining number of free cells in the column is not sufficient for each symbol to occur at least v^{t-1} times. Figure 3.1(b) shows another CA$(11; 2, 3, 3)$; in this case every symbol must occur at least $v^{t-1} = 3^{2-1} = 3$ times in the new column. Then, symbol 2 is not valid for the next free cell because after assigning the cell there will only be one free cell and symbols 0 and 1 will miss one occurrence.

The purpose of R_3 is to avoid the generation of non-canonical CAs. Due to the isomorphisms of CAs, any symbol in the new column can not occur more times than symbol 0 in the first column. Suppose that the first column of a CA has n zeros and that symbol $e \in \mathbb{Z}_v$ occurs $n + 1$ times in the new column. By a permutation of columns it is possible to move the last column of the CA to the first column; next, the symbols of the new first column can be permuted to transform symbol

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 2 & 2 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & * \\ 1 & 2 & 2 & * \\ 2 & 0 & 2 & * \\ 2 & 1 & 0 & * \\ 2 & 2 & 1 & * \end{pmatrix} \qquad \textit{times\_covered} = \begin{bmatrix} 1 & 2 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 2 \end{bmatrix}$$

Figure 3.2: In the left matrix, symbol 1 is not valid for the next free cell because symbol 1 does not satisfy R_4. In the right part, the current status of the *times_covered* matrix.

e into symbol 0; and finally, a sorting of rows will produce a CA lexicographically smaller than the original CA. Figure 3.1(c) shows a CA$(11; 2, 3, 3)$ extended to four columns; in the next free cell, the symbol 1 is not valid because it has already occurred four times, which is the number of occurrences of symbol 0 in the first column.

Rules R_4 and R_5 are more sophisticated. To be part of a CA of strength t and order v, the new column plus any other $t-1$ columns must cover at least once each tuple of the set $\mathbb{Z}_v^t$. If the array has N rows then v^t of these rows will contain the first occurrence of the tuples of $\mathbb{Z}_v^t$, and the remaining $N - v^t$ rows will contain repeated tuples. Therefore, every combination of t columns can have at most $N - v^t$ repeated tuples. If the assignment of symbol $e \in \mathbb{Z}_v$ to the next free cell makes the number of repeated tuples greater than $N - v^t$, then symbol e is not valid for the next free cell. In the example of Figure 3.2 the maximum number of repeated tuples in every combination of t columns of CA$(11; 2, 3, 3)$ is 2, because $N - v^t = 11 - 3^2 = 2$, and so every symbol that produces three repeated tuples is invalid for the next free cell. In this way, symbol 1 is not valid because in the combination of columns $\{2, 3\}$ there are already two repeated tuples, the tuple $(2, 2)$ and the tuple $(0, 1)$; if symbol 1 is assigned to the next free cell then the tuple $(1, 1)$ will also be repeated, making the total number of repeated tuples greater than the maximum allowed.

Let $A =$ CA$(N; t, r, v)$ be the CA that is being extended to $r + 1$ columns. To validate R_4

efficiently the algorithm uses a matrix called *times_ covered* of dimensions $\binom{r}{t-1} \times v^t$. The i-th row of the *times_ covered* matrix is associated to the i-th combination of columns containing column r, where the combinations are listed in lexicographic order. Also, the j-th column of the matrix is associated to the j-th tuple of $\mathbb{Z}_v^t$ in lexicographic order. Therefore, the entry (i, j) *times_ covered* contains the number of times that the j-th tuple has been covered in the i-th combination of t columns containing column r. The algorithm uses this matrix to determine efficiently when a symbol causes more repeated tuples than the maximum allowed in a combination of t columns.

Figure 3.2 shows the status of the *times_ covered* matrix for the CA at the left of the figure. In this example $r = 3$, $t = 2$, and $v = 3$. The $\binom{r}{t-1}$ combinations of $t - 1$ columns are $\{0\}, \{1\}, \{2\}$. Adding the column r to these combinations gives the combinations of t columns $\{0, 3\}, \{1, 3\}, \{2, 3\}$; which are associated in order with the three rows of *times_ covered*. Similarly, the $v^t = 3^2 = 9$ tuples of the set $\mathbb{Z}_3^2 = \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}$ are associated in order with the nine columns of *times_ covered*.

The rule R_5 ensures the existence of at least one free cell to cover each non-covered tuple. Suppose that we are trying to extend $A = \text{CA}(N; t, r, v)$ to $r+1$ columns. Row i of A is a candidate row to cover a t-tuple $(x_0, \ldots, x_{t-1})$ in a combination of t columns $\{j_0, \ldots, j_{t-1} = r\}$ containing the column under construction r if $x_l = A_{i,j_l}$ for $0 \leq l \leq t - 2$. Figure 3.3 shows a CA$(11; 2, 3, 3)$ being extended to four columns. At the left of the figure we have the *times_ covered* matrix, and a matrix called *total_ candidates* which stores at entry (i, j) the number of rows that can cover the j-th tuple in the i-th combination of t columns involving column r. For example, the first three elements of the first row of the *total_ candidates* matrix are 1 because there is only one candidate row to cover the tuples $(0, 0)$, $(0, 1)$, and $(0, 2)$ in the combination of columns $\{0, 3\}$ (this candidate row is the fourth row of the CA). Therefore, symbols 0 and 1 are not valid for the next free cell because if one of them is assigned to the cell, then there will be zero candidate rows to cover the tuple $(0, 2)$ in the combination of columns $\{0, 3\}$.

By rules R_4 and R_5 every complete new column makes a CA of strength t with the previous column; so, it is not necessary to test if the new column forms a CA with the previous columns.

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

$$\textit{times\_covered} = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\textit{total\_candidates} = \begin{bmatrix} 1 & 1 & 1 & 4 & 4 & 4 & 3 & 3 & 3 \\ 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 2 & 2 & 4 & 4 & 4 \end{bmatrix}$$

Figure 3.3: In the left matrix, symbols 0 and 1 are not valid for the next free cell because they do not satisfy R_5. At the right it is shown the corresponding status of the matrices *times_covered* and *total_candidates*.

3.1.2 Construction of covering arrays

Before filling a free cell c_i the algorithm firstly determines which symbols satisfy the rules R_1, R_2, R_3, R_4, and R_5. After that, the algorithm assigns to c_i its first valid symbol and advances to the next free cell c_{i+1}; the other valid symbols for c_i will be tested later on. When all valid symbols have been assigned to a cell, the algorithm backtracks to the previous cell and assigns to it its next valid symbol. So, the method to construct the next column also uses backtracking. The backtracking in the new column ends when all valid symbols have been tested in the first cell of the column.

Algorithm 1 shows the new algorithm to construct the non-isomorphic CAs for given parameters N, t, k, v. These values are supposed to be globally accessible to the recursive function *search_column*(A, r). The covering array A received by the function has $r \geq 1$ columns because there is an efficient way to determine the possible first columns of canonical CAs (see Algorithm 1 of [46]); r is also the index of the new column to construct.

The first sentence of Algorithm 1 is a call to *set_symbol*(*symbol*, A, *row*, *column*) function. This function assigns the symbol *symbol* to the entry (*row*, *column*) of A, and updates the auxiliary data structures used to compute efficiently which symbols are valid for the next cell. Similarly, the *remove_symbol*(A, i, r) function deletes the current symbol in the entry (i, r) of A and undoes the

Algorithm 1: search_column(A, r)

```
1  set_symbol($0, A, 0, r$);
2  $i \leftarrow 1$;
3  determine_valid_symbols($A, i, r$);
4  while $i \geq 1$ do
5      symbol $\leftarrow$ next_valid_symbol($A, i, r$);
6      if symbol $\neq -1$ then
7          set_symbol(symbol, $A, i, r$);
8          if $i = N - 1$ then
9              if is_canonical($A, r$) then
10                 if $r = k - 1$ then
11                     write($A$);
12                 else
13                     search_column($A, r + 1$);
14             remove_symbol($A, i, r$);
15         else
16             $i \leftarrow i + 1$;
17             determine_valid_symbols($A, i, r$);
18     else
19         $i \leftarrow i - 1$;
20         remove_symbol($A, i, r$);
```

changes done by *set_symbol*() in the auxiliary data structures. At line 4 it begins a *while* loop that is executed while $i \geq 1$. Inside the loop, only the cells c_i for $i \geq 1$ are modified; the cell c_0 is set in line 1 and it never changes its value. This is because every column $c = (c_0\ c_1\ \cdots\ c_{N-1})^T$ of a canonical CA satisfies $c_i \leq i$ for $i = 0, 1, \ldots, v - 1$. In line 5 the *next_valid_symbol*() function stores in the variable *symbol* the next valid symbol for c_i. On every call this function returns the next valid symbol for c_i, and when there are no more valid symbols the function returns -1.

In line 9 the *is_canonical*() function tests if the constructed array is the canonical representative of its class. If the array is canonical then the algorithm checks if r is equal to $k-1$; when $r = k-1$ the array A has k columns and therefore a new non-isomorphic CA($N; t, k, v$) is reported in line 11. If $r < k-1$ then the array A is not yet complete and the *search_column*() function is called recursively to construct the next column of the CA. The *is_canonical*() function used by the improved algorithm is a recursive version of the function with the same name reported in [46]; this recursive version is presented in Subsection 3.1.3.

3.1.3 Simplified canonical test

The *is_canonical*() function of [46] uses several auxiliary structures to interleave the generation of column permutations with the generation of symbol permutations. Because it is an iterative function, some iterations of the main loop are for generating permutations of columns, and some are for checking symbol permutations. To determine if a given covering array A with r columns is the canonical representative of its class the function uses r lists, $L_0, L_1, \ldots, L_{r-1}$, where for $0 \leq j \leq r-1$ list L_j stores the symbol permutations that produce an array equal to A up to j columns; the operations of adding and removing elements from the lists can affect the running time of the program because the *is_canonical*() function is called constantly.

For the improved algorithm we developed a recursive version of the *is_canonical*() function. In Algorithm 2 an array called *assigned* of length r is initialized with *false*, and the helper function *test_column*() of Algorithm 3 is called. To construct the arrays isomorphic to A, a global array D is used; the columns of A are copied to the columns of D ($d_0, d_1, \ldots, d_{r-1}$) in such a way that D gets all column and symbol permutations of A that have sense to explore.

Algorithm 2: is_canonical(A, r)

```
1 for $i = 0, 1, \ldots, r-1$ do
2     assigned[$i$] ← false;
3 return test_column($A, r, 0$);
```

The *assigned* vector is used to mark which columns of A are currently copied to a column of D. The *test_column*() function is called with three parameters A, r, and s, which are respectively the array A, the number of columns of A, and the index of the next column of D that will be filled with a column of A. The canonical test is based on the fact that if an array with r columns is canonical then every subarray with $s \leq r$ columns is also canonical.

Algorithm 3: test_column(A, r, s)

```
if $s = r$ then
    return true;
for $j = 0, 1, \ldots, r-1$ do
    if assigned[$j$] = false then
        assigned[$j$] ← true;
        foreach permutation of symbols $\varepsilon$ do
            $d_s \leftarrow a_j$ relabeled with $\varepsilon$;
            sort_rows($D, s+1$);
            if $V(D, s+1)$ is lexicographically smaller than $V(A, s+1)$ then
                return false;
            else if $V(D, s+1) = V(A, s+1)$ then
                if test_column($A, r, s+1$) = false then
                    return false;
        assigned[$j$] ← false;
return true;
```

On every call, the *for* loop of the *test_column*() function iterates over the r columns of A, but the body of the loop is executed only for the columns of A not currently assigned to a column of D. All non-assigned columns of A are copied to column s of D to ensure the verification of all column permutations. However, to test the $v!$ symbol permutations in each column, every non-assigned column of A is copied $v!$ times, one with a distinct permutation of symbols. After a column of A is copied to column s of D, the *sort_rows*() function sorts the rows of D considering only the first $s+1$ columns of D. The $V(X, j)$ function returns the vector formed by the first j columns of the array X arranged in column-major order. If D up to $s+1$ columns is lexicographically smaller than A up to $s+1$ columns, then the function returns *false*, because by adding to D the non-copied columns of A we can obtain an array isomorphic to A and smaller than it. If D and A are equal up to $s+1$ columns, then *test_column*() is called recursively to fill the column with index $s+1$ of D with the non-assigned columns of A.

When the first $s+1$ columns of D are lexicographically greater than the first $s+1$ columns of A, the *test_column*() function is not called recursively. All permutations of columns starting with the current one are skipped because arrays D with $s+2,\dots,k$ columns will also be lexicographically greater than the corresponding subarrays of A.

In the worst case, determining if $A = \text{CA}(N;t,k,v)$ is canonical takes time $O(N \log_2 N \cdot k! \cdot (v!)^k)$ because the $N!$ row permutations are reduced to a row sorting done in $O(N \log_2 N)$.

3.1.4 ExtendNonIsoCA algorithm

We adapted the improved NonIsoCA algorithm to receive as input the set of all non-isomorphic CAs with k columns to produce the non-isomorphic CAs with $k+1$ columns. We call ExtendNonIsoCA to this adaptation of the NonIsoCA algorithm. The algorithm ExtendNonIsoCA works in the same way as the NonIsoCA algorithm: to extend a partial subarray with k columns new column is constructed following the rules R_1 to R_5 for valid symbols, and using the canonical test to discard non-canonical CAs. The objective of the ExtendNonIsoCA algorithm is to take advantage of previous work to find the canonical CAs with k columns.

3.2 Complexity of the improved algorithm

In this section we analyze the running time of the improved NonIsoCA algorithm to classify CA$(N;t,k,v)$. To make the analysis in an easier way we will do the following simplification: in every column each symbol of $\mathbb{Z}_v = \{0,1,\dots,v-1\}$ occurs N/v times, i.e., the symbols are balanced in every column.

Since the rows of canonical CAs are sorted, the first column of CA$(N;t,k,v)$ is formed by a block of zeros, followed by a block of ones, and so on until ending with a block of elements equal to $v-1$. Then, by our simplification there is only one possible first column, and it is the one formed by N/v zeros, followed by N/v ones, and so on.

For the second column onwards, the improved NonIsoCA algorithm constructs the candidate columns cell by cell. We estimate the number of constructed candidate columns as follows: for each $j = 1,\dots,v-1$ partition column j of the CA into v contiguous blocks $B_0, B_1, \dots, B_{v-1}$ of N/v

rows each; so block B_i is formed by the rows with indices $(N/v) \cdot i, \dots, (N/v) \cdot (i+1) - 1$. To be a CA of strength t, every block B_i in the columns $j = 1, \dots, k-1$ must contain at least v^{t-2} occurrences of each symbol of $\mathbb{Z}_v$. Taking our simplification one step further, we suppose that each symbol of $\mathbb{Z}_v$ occurs N/v^2 times in each block B_i. Since $N \geq v^t$, the inequalities $N/v \geq v^{t-1}$ and $N/v^2 \geq v^{t-2}$ hold, and so we do not underestimate the number of occurrences of each symbol.

The number of ways to assign each block B_i is the number of permutations of N/v objects where there are v distinct objects each repeated N/v^2 times; this number is given by $\frac{(N/v)!}{[(N/v^2)!]^v}$. Therefore, the number of candidate columns, or the number of ways to assigns the v blocks B_i, is $\beta = \left[\frac{(N/v)!}{[(N/v^2)!]^v}\right]^v$.

In general v is not a divisor of N, so we use the gamma $\Gamma(x)$ function instead of the factorial function to compute β. The gamma function is an extension of the factorial function to real numbers and complex numbers; if n is a positive integer then $\Gamma(n) = (n-1)!$; so the argument of the gamma function should be one unit more than the argument of the factorial function. In this way, the number of candidate columns is $O(\beta)$ where $\beta = \left[\frac{\Gamma(N/v+1)}{[\Gamma(N/v^2+1)]^v}\right]^v$.

The improved NonIsoCA algorithm uses backtracking to construct the non-isomorphic CAs with k columns. So, when a non-isomorphic CA with $r < k$ columns is found, the algorithm constructs all $O(\beta)$ candidate columns to see which of them form a canonical CA with $r+1$ columns. The candidate columns are constructed cell by cell, and for each assigned cell the five rules R_1, R_2, R_3, R_4, R_5 of Subsection 3.1.1 are verified. By rules R_4 and R_5 a candidate column makes a CA with the previous columns when the N cells of the candidate column are assigned; so the CA test is not made explicitly when the candidate column is complete, but the test is done part by part with each new assigned cell. For simplicity in our analysis, we compute the cost of the CA test for each candidate column as if the test is performed after filling all cells of the new column. The computational cost of the CA test for a partial CA with $r \leq k$ columns is $O\big(\binom{r-1}{t-1} \cdot Nt\big)$, where $\binom{r-1}{t-1}$ is the number of subarrays of t columns involving the last column, and Nt is the cost of validating that each of these subarrays covers the v^t t-tuples over $\mathbb{Z}_v$ at least once. For the canonical test the computational cost is $O(N \log_2 N \cdot k! \cdot (v!)^k)$, as we saw in Subsection 3.1.3.

If each candidate column passes both the CA test and the canonical test, then the number of constructed candidate columns is 1 for the first column, and it is $O(\beta^j)$ for each column $j = 1, \ldots, k-1$. However, by the isomorphisms of CAs the number of canonical CAs with $r = 2$ columns is $O(\beta/[(2-1)!(v!)^{2-1}]) = O(\beta/[1!(v!)^1]$ because excluding the first column the number of isomorphic subarrays with 1 column is $1!(v!)^1$, and only one of these isomorphic arrays is canonical. For $r = 3$ the number of candidate columns is again β, so the algorithm constructs $O(\frac{\beta}{1!(v!)^1} \cdot \beta)$ candidate arrays of which $O(\frac{\beta^2}{1!(v!)^1 2!(v!)^2})$ are canonical; in this case the $2!$ permutation of columns and the $(v!)^2$ distinct symbol relabelings are done over the last two columns of the candidate arrays with three columns. In general for $r+1$ columns we have $O\big(\frac{\beta^{r-1}}{\prod_{i=1}^{r-1} i!(v!)^i} \cdot \beta\big)$ candidate arrays of which $O\big(\frac{\beta^r}{\prod_{i=1}^{r} i!(v!)^i}\big)$ are canonical.

Thus, for the last column, the number of candidate arrays is $O\big(\frac{\beta^{k-2}}{\prod_{i=1}^{k-2} i!(v!)^i} \cdot \beta\big)$. This term dominates the terms corresponding to $j = 2, \ldots, k-1$ columns; so we remove the lower-order terms and take the number of candidate arrays for the last column as the total number of candidate arrays for the instance CA$(N; t, k, v)$.

Therefore, the approximate computational cost of the improved NonIsoCA algorithm is:

$$O\Big(\frac{\beta^{k-1}}{\prod_{i=1}^{k-2} i!(v!)^i} \cdot \binom{k-1}{t-1} Nt \cdot (N \log_2 N) k! (v!)^k\Big), \text{ where } \beta = \left[\frac{\Gamma(N/v+1)}{[\Gamma(N/v^2+1)]^v}\right]^v.$$

3.3 Parallelization of the improved algorithm

This section describes a parallel version of the improved NonIsoCA algorithm. Subsection 3.3.1 explains the parallelization strategy, and Subsection 3.3.2 gives an implementation in MPI.

3.3.1 Parallelization strategy

The basic strategy to parallelize the construction of the non-isomorphic CAs is to create a new execution flow every time the algorithm constructs a canonical CA with $r < k$ columns. When the sequential algorithm constructs a canonical CA with r columns, the next step is to search the column $r+1$ of the CA; after some time the backtracking process reaches column r again and the algorithm searches for the next canonical CA with r columns.

In the parallel algorithm a new thread is created when a canonical CA with r columns is constructed. The parent thread, i.e., the thread that constructed the canonical CA with r columns, launches a child thread whose work is to search the column $r+1$ of the CA, while the work of the parent thread is to search the next canonical CA with r columns. In the child thread the first r columns are fixed; the algorithm never backtracks to a column index smaller than r because its parent thread is searching the next CA with r columns.

Suppose the algorithm wants to construct all non-isomorphic covering arrays CA$(6; 2, 6, 2)$, and suppose the algorithm has constructed the following partial CA with three columns:

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

Then, the following execution flows can be done in parallel:

- The child thread takes the CA constructed by its parent thread, and tries to complete the CA without modifying the first three columns of the CA. In this case the child thread finds the canonical CA with four columns shown next at the right:

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

- Meanwhile the parent thread searches the next canonical CA with three columns. In this case the parent thread finds the CA shown next at the right:

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

A child process in turn can launch another thread when a canonical CA is found. The number of threads that can be created is the accumulated sum of all canonical partial CAs with $1 \leq j < k$ columns; for the last column of the CA no child thread is created. This number of threads can exceed the number of available processors; and therefore sometimes it is not possible to create a new thread when a canonical CA is found. In these cases the parent thread first searches the next column of the CA, and later the same thread searches for the next canonical CA with r columns.

3.3.2 Implementation in MPI

The implementation in MPI of the improved algorithm to construct non-isomorphic CAs uses the master/slaves model. Let P be the number of processes in the MPI job; the master has rank 0 and the slaves have ranks from 1 to $P - 1$.

The first task done by the master process is to generate all first columns that can be part of a canonical CA for the given parameters, and to send those first columns to the slaves. The slaves receive the partial arrays with $r = 1$ columns and they start the construction of the second column of the CA. After sending the arrays with one column to the slaves, the work of the master process is to keep track of which slave processes are working and which ones are idle, in order to authorize or reject the creation of a new thread when a slave finds a canonical CA.

The master stores the ranks of the idle process in a stack. Initially this stack contains the ranks of all slaves $1, 2, \ldots, P-1$. When a new thread can be created, the master pops one element from the stack and the associated slave executes the thread; when a slave process becomes idle (the thread finalizes) its rank is pushed onto the stack.

For the communication among the processes two vectors are used, the first one is called *message* and the other one is called *data*. The *message* vector can store three integers and it is used to send a message identified by a global constant; on the other hand, *data* is a vector of characters of length $N \cdot k$ used to send a CA with $r \leq k$ columns.

The three positions of *message* are used as follows: position 0 contains the rank of the process sending the message; position 1 contains a global constant to identify the message; and position 2 contains an argument for the constant in position 1. The six possible values for position 1 of *message* are the following ones:

- SEARCH_NEXT_COLUMN. This constant indicates to a slave process that it will receive a partial CA, and therefore its next task is to search the next column of the partial CA. The number of columns r of the partial CA to be received is sent in position 2 of the vector.

- REQUEST_IDLE_PROCESS. Slave processes use this constant to request to the master the rank of an idle process.

- PROCESS_AVAILABLE. The master uses this constant to notify to a slave process the existence of an idle process who can execute a new thread. The rank of the idle process is sent in position 2 of *message*.

- PROCESS_NOT_AVAILABLE. The master uses this constant to notify to a slave process that no process is available to create a new thread.

- THREAD_FINALIZED. A slave process uses this constant to notify the finalization of its current task, and so it is free to receive another partial CA.

- EXIT_SLAVE. The master uses this constant to tell the slave processes that they should finalize because the job is complete.

Algorithm 4 shows the work done by the master process. The parameters N, t, k, v are supposed to be globally accessible to the *master*() function, as well as the matrix A to store the CA. The *first_column*() function uses the algorithm given in [46] to generate on every call the next column that can be the first column of a canonical CA$(N; t, k, v)$. If a first column was successfully generated the function stores it in the first column of A and returns *true*; when all first columns have been generated the function returns *false*. In the MPI functions the variable *comm* is the default communicator MPI_COMM_WORLD.

The *while* loop of lines 3-5 is to wait for a slave process to become idle in case of the stack of idle processes is empty. Line 4 blocks the master until some slave sends a message; when the message is received, the *handle_message*() function takes the appropriate action according to the type of the received message. Algorithm 5 implements the *handle_message*() function. The master can only receive messages with THREAD_FINALIZED or with REQUEST_IDLE_PROCESS. The action for THREAD_FINALIZED is to add the rank of the slave who sent the message to the stack of idle processes. For REQUEST_IDLE_PROCESS the master checks if the stack of idle processes is empty; in the positive case the master responses to the slave with PROCESS_NOT_AVAILABLE; but in the other case the master puts in *message*[2] the rank of the idle process and sends to the slave a message with PROCESS_AVAILABLE.

In lines 6-12 the *master*() function sends two messages to an idle slave. The first one to indicate that an array with one column will be sent, and the second one to send the partial CA with one column. The tags T0 and T1 are used to identify the messages.

In lines 13 to 15 the master waits for messages sent by the slaves and performs the appropriate action for the received message. The master is in the loop of these lines while there is at least one

Algorithm 4: master()

```
1  stack ← $\{1, 2, \ldots, P-1\}$;
2  while first_column($A$) do
3      while stack_empty(stack) do
4          MPI_Recv(message, 3, MPI_INT, MPI_ANY_SOURCE, T0, comm, &stat);
5          handle_message(message);
6      slave ← pop(stack);
7      message[0] ← $0$;
8      message[1] ← SEARCH_NEXT_COLUMN;
9      message[2] ← $1$;
10     MPI_Send(message, 3, MPI_INT, slave, T0, comm);
11     data ← first column of $A$;
12     MPI_Send(data, $N$, MPI_CHAR, slave, T1, comm);
13 while stack_size(stack) $< P-1$ do
14     MPI_Recv(message, 3, MPI_INT, MPI_ANY_SOURCE, T0, comm, &stat);
15     handle_message(message);
16 message[0] ← $0$;
17 message[1] ← EXIT_SLAVE;
18 for $j \leftarrow 1$ to $P-1$ do
19     MPI_Isend(message, 3, MPI_INT, j, T0, comm, &req);
20 MPI_Finalize();
```

Algorithm 5: handle_message(*message*)

```
1  source ← message[0];
2  if message[1] = THREAD_FINALIZED then
3      push(stack, source);
4  else if message[1] = REQUEST_IDLE_PROCESS then
5      message[0] ← $0$;
6      if stack_empty(stack) then
7          message[1] ← PROCESS_NOT_AVAILABLE;
8      else
9          message[1] ← PROCESS_AVAILABLE;
10         message[2] ← pop(stack);
11     MPI_Send(message, 3, MPI_INT, source, source, comm);
```

active slave. When all slaves become idle the master sends to them a message with EXIT_SLAVE, in lines 16-19, in order to the slaves finalize their execution.

Slave processes search the next columns of the partial CAs they receive from the master or from other slaves. Algorithm 6 shows the *slave*() function executed by the slave processes. When this function starts its execution, the *MPI_Recv*() function blocks the slave until another process sends a message to it. When the received message contains SEARCH_NEXT_COLUMN the slave gets the number of columns r of the partial CA that will be received in the next message; after that, the second message is received and *copy_columns*() copies the columns stored in *data* to the matrix A; finally, the *search_column*() function is called to start the search of the next column. If the message that is received in line 2 contains the constant EXIT_SLAVE then the slave calls *MPI_Finalize*() to finalize its execution. In lines 10-12 the slave notifies the master that it has finished the search of the next column and it is ready to accept a new partial CA.

Algorithm 6: slave()

```
 1 while true do
 2     MPI_Recv(message, 3, MPI_INT, MPI_ANY_SOURCE, T0, comm, &stat);
 3     if message[1] = SEARCH_NEXT_COLUMN then
 4         $r \leftarrow$ message[2];
 5         MPI_Recv(data, $N * r$, MPI_CHAR, MPI_ANY_SOURCE, T1, comm, &stat);
 6         copy_columns($A$, data, $r$);
 7         search_column($A, r$);
 8     else if message[1] = EXIT_SLAVE then
 9         MPI_Finalize();
10     message[0] $\leftarrow$ rank;
11     message[1] $\leftarrow$ THREAD_FINALIZED;
12     MPI_Send(message, 3, MPI_INT, 0, T0, comm);
```

Algorithm 7 shows the *search_column*() function used in the parallel algorithm. This function is almost the same as the *search_column*() function of Algorithm 1; however when a canonical CA is constructed and $r < k - 1$ the slave sends a message to the master requesting the rank of an idle process. If there is an idle process the slave sends to the idle process the current CA; the idle process will try to extend the received CA to $r + 2$ columns and the slave who sent the message will search the next canonical CAs with $r + 1$ columns. If there is not an idle process, then the slave calls *search_column*() recursively to search the next column of the CA.

Algorithm 7: search_column(A, r)

```
set_symbol($0, A, 0, r$);
$i \leftarrow 1$;
determine_valid_symbols($A, i, r$);
while $i \geq 1$ do
    symbol ← next_valid_symbol($A, i, r$);
    if symbol $\neq -1$ then
        set_symbol(symbol$, A, i, r$);
        if $i = N - 1$ then
            if is_canonical($A, r$) then
                if $r = k - 1$ then
                    write($A$);
                else
                    message[0] ← rank;
                    message[1] ← REQUEST_IDLE_PROCESS;;
                    MPI_Send(message, 3, MPI_INT, 0, T0, comm);
                    MPI_Recv(message, 3, MPI_INT, 0, rank, comm, &stat);
                    if message[1] = PROCESS_AVAILABLE then
                        destination ← message[2];
                        message[0] ← rank;
                        message[1] ← SEARCH_NEXT_COLUMN;
                        message[2] ← $r + 1$;
                        MPI_Send(message, 3, MPI_INT, destination, T0, comm);
                        data ← $(a_0\ a_1\ \cdots\ a_r)$;
                        MPI_Send(data, $N * (r + 1)$, MPI_CHAR, destination, T1, comm);
                    else
                        search_column($A, r + 1$)
            remove_symbol($A, i, r$);
        else
            $i \leftarrow i + 1$;
            determine_valid_symbols($A, i, r$);
    else
        $i \leftarrow i - 1$;
        remove_symbol($A, i, r$);
```

3.4 Chapter summary

An improved version of the NonIsoCA algorithm to construct non-isomorphic CAs was developed in this chapter. As we will see in Chapter 5 the new algorithm is faster than the previous one when

$t > 2$ of $v > 2$, and so it can handle larger instances. The improved algorithm constructs the non-isomorphic CAs column by column, and fills the cells of the new column from top to bottom. Before assigning a symbol to a cell, the algorithm firstly determines which symbols are valid for the cell by using a set of five rules. These rules have the purpose of avoiding the exploration of arrays that can not be canonical CAs. A parallel implementation of the improved NonIsoCA algorithm was developed to reduce even further the execution time of the algorithm.

The next chapter introduces the second algorithm proposed in this thesis to classify CAs. This algorithm performs the classification of a CA of strength $t+1$ by generating all possible juxtapositions of v CAs of strength t.

4

Juxtaposition of Covering Arrays

This chapter develops a new classification algorithm that is not based on a previous algorithm. This new algorithm constructs the non-isomorphic CA$(N; t+1, k+1, v)$ by generating all possible juxtapositions of v CAs of strength t, k columns, and respective number of rows $N_0, N_1, \ldots, N_{v-1}$, where $N = \sum_{i=0}^{v-1} N_i$. When the juxtaposition of v CAs forms a CA$(N; t+1, k, v)$, a column formed by N_i elements equal to i for $0 \leq i \leq v-1$ is added to the CA to obtain a CA$(N; t+1, k+1, v)$. Since the algorithm explores all possible ways of constructing CA$(N; t+1, k+1, v)$, the obtained CAs cover all isomorphism classes. None of the constructed CA$(N; t+1, k+1, v)$ is canonical, and there may be isomorphic CAs in the results. To obtain only the canonical CA of each isomorphism class, the solutions are canonized and duplicate arrays are removed. The new classification algorithm is called *JuxtaposeCA*. Section 4.1 studies the structure and the existence of CAs, which provide the basic ideas for the JuxtaposeCA algorithm; Section 4.2 describes the JuxtaposeCA algorithm in detail; Section 4.3 analyzes the complexity of the algorithm; and Section 4.4 provides two parallelizations of the algorithm.

4.1 Structure and existence of covering arrays

Colbourn *et al*. [11] used a technique called *derivation* to reduce in one unit the strength and the number of columns of a CA. In a given CA$(N; t, k, v)$ take any column j and construct v arrays as follows: for $0 \leq i \leq v-1$ delete column j and all rows where the element at column j is not i; the v resulting arrays are CAs with $k-1$ columns and strength $t-1$; finally, take the CA with the smallest number of rows, which has at most $\lfloor N/v \rfloor$ rows.

Let $C =$ CA$(N; t+1, k+1, v)$ be a CA with strength $t+1 \geq 2$, and construct C' isomorphic to C by reordering the rows of C so that the elements of the last column of C' are sorted in increasing order. Then C' has the following structure, were blocks $A_0, A_1, \ldots, A_{v-1}$ are CAs of strength t and k columns, and $\mathbf{0}$, $\mathbf{1}$, ..., $\mathbf{v-1}$ are subcolumns of elements equal to 0, 1, ..., $v-1$ respectively:

$$C' = \begin{pmatrix} A_0 & \mathbf{0} \\ A_1 & \mathbf{1} \\ \vdots & \vdots \\ A_{v-1} & \mathbf{v-1} \end{pmatrix}$$

Figure 4.1 shows an example where $C =$ CA$(11; 3, 5, 2)$. The array C' isomorphic to C is obtained by permuting the rows of C in such a way that the elements of the last column are sorted. The first four columns and the first six rows of C' form $A_0 =$ CA$(6; 2, 4, 2)$, and the first four columns and the last five rows of C' form $A_1 =$ CA$(5; 2, 4, 2)$. The subcolumn $\mathbf{0}$ is formed by six zeros, and the subcolumn $\mathbf{1}$ is formed by five ones.

We can reverse the process and construct a CA$(N; t+1, k+1, v)$ by juxtaposing vertically v CAs of strength t and k columns $A_0 =$ CA$(N_0; t, k, v)$, $A_1 =$ CA$(N_1; t, k, v)$, ..., $A_{v-1} =$ CA$(N_{v-1}; t, k, v)$, where $N = \sum_{i=0}^{v-1} N_i$, and by adding to this juxtaposition a column $(\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$ formed by N_i elements equal to i for $0 \leq i \leq v-1$. For example, if we want to construct a CA$(11; 3, 5, 2)$, then one possible solution is to juxtapose the CAs $A_0 =$ CA$(6; 2, 4, 2)$ and $A_1 =$ CA$(5; 2, 4, 2)$ shown in Figure 4.1. This juxtaposition forms a CA$(11; 3, 4, 2)$, and by adding to this CA the column $(\mathbf{0}\ \mathbf{1})^T$

$$C = \begin{pmatrix} 0&0&0&1&1\\ 0&0&0&0&0\\ 0&1&0&1&0\\ 0&1&1&1&1\\ 1&0&0&1&0\\ 1&1&1&1&0\\ 0&0&1&0&1\\ 0&1&1&0&0\\ 1&0&1&1&1\\ 1&0&1&0&0\\ 1&1&0&0&1 \end{pmatrix} \qquad C' = \left(\begin{array}{cccc|c} 0&0&0&0&0\\ 0&1&0&1&0\\ 0&1&1&0&0\\ 1&0&0&1&0\\ 1&0&1&0&0\\ 1&1&1&1&0\\ \hline 0&0&0&1&1\\ 0&0&1&0&1\\ 0&1&1&1&1\\ 1&0&1&1&1\\ 1&1&0&0&1 \end{array}\right) = \begin{pmatrix} A_0 & \mathbf{0}\\ A_1 & \mathbf{1} \end{pmatrix} \qquad A_0 = \begin{pmatrix} 0&0&0&0\\ 0&1&0&1\\ 0&1&1&0\\ 1&0&0&1\\ 1&0&1&0\\ 1&1&1&1 \end{pmatrix} \quad A_1 = \begin{pmatrix} 0&0&0&1\\ 0&0&1&0\\ 0&1&1&1\\ 1&0&1&1\\ 1&1&0&0 \end{pmatrix}$$

Figure 4.1: Structure of the covering array $C = \text{CA}(11; 3, 5, 2)$.

formed by 6 zeros and 5 ones we obtain a CA$(11; 3, 5, 2)$.

Not all juxtapositions of a CA$(6; 2, 4, 2)$ and a CA$(5; 2, 4, 2)$, plus a column formed by 6 zeros and 5 ones, form a CA$(11; 3, 5, 2)$. So, to construct CA$(11; 3, 5, 2)$ by juxtaposing a CA$(6; 2, 4, 2)$ and a CA$(5; 2, 4, 2)$ we need to test all possible juxtapositions of two CAs with these sizes.

However, if CA$(N; t + 1, k + 1, v)$ exists then there is at least one juxtaposition of v CAs $A_0 =$ CA$(N_0; t, k, v)$, $A_1 =$ CA$(N_1; t, k, v)$, $\ldots$, $A_{v-1} =$ CA$(N_{v-1}; t, k, v)$, where $N = \sum_{i=0}^{v-1} N_i$, that forms a CA$(N; t + 1, k, v)$, as stated in the following Theorem 4.1.1. From CA$(N; t + 1, k, v)$ a CA$(N; t + 1, k + 1, v)$ is obtained by appending to the first CA the column $(\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$.

Theorem 4.1.1. *CA$(N; t + 1, k + 1, v)$ exists if and only if there exist v covering arrays CA$(N_0; t, k, v)$, CA$(N_1; t, k, v)$, $\ldots$, CA$(N_{v-1}; t, k, v)$, where $N = \sum_{i=0}^{v-1} N_i$, that juxtaposed vertically form a CA$(N; t + 1, k, v)$.*

Proof. Assume $C =$ CA$(N; t + 1, k + 1, v)$ exists. For $0 \leq i \leq v - 1$ let N_i be the number of elements equal to i in the last column of C. Construct C' isomorphic to C by reordering the rows of C in such a way that the elements of the last column of C' are sorted in increasing order. For $0 \leq i \leq v - 1$ let B_i be the block of the N_i rows of C' where the symbol in the last column is i. Divide B_i into two blocks: A_i containing the first k columns, and $\mathbf{i}$ containing the last column; then

C' has the following structure:

$$C' = \begin{pmatrix} A_0 & \mathbf{0} \\ A_1 & \mathbf{1} \\ \vdots & \vdots \\ A_{v-1} & \mathbf{v-1} \end{pmatrix}$$

The juxtaposition of blocks $A_0, A_1, \ldots, A_{v-1}$ form a CA$(N; t+1, k, v)$ because C' has strength $t+1$; then, to complete the first part of the proof we need to show that blocks $A_0, A_1, \ldots, A_{v-1}$ are CAs of strength t. Index columns of C' starting from 0, so the last column of $C' =$ CA$(N; t+1, k+1, v)$ has index k. Any combination $(c_0, c_1, \ldots, c_t = k)$ of $t+1$ columns containing the last column of C' covers in block B_i all $(t+1)$-tuples of the form $(x_0, x_1, \ldots, x_{t-1}, x_t = i)$ over $\mathbb{Z}_v$. Thus, every combination of t columns $(c_0, c_1, \ldots, c_{t-1})$ from the first k columns of C' covers all t-tuples $(x_0, x_1, \ldots, x_{t-1})$ over $\mathbb{Z}_v$ in every block A_i, and therefore $A_i =$ CA$(N_i; t, k, v)$.

Now, suppose there are v covering arrays $A_0 =$ CA$(N_0; t, k, v)$, $A_1 =$ CA$(N_1; t, k, v)$, …, $A_{v-1} =$ CA$(N_{v-1}; t, k, v)$, whose vertical juxtaposition forms $G =$ CA$(N; t+1, k, v)$ of strength $t+1$, where $N = \sum_{i=0}^{v-1} N_i$. Let $E = (\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$ be the column formed by concatenating vertically N_i elements equal to i for $0 \le i \le v-1$. Because every A_i is a CA of strength t, for $0 \le i \le v-1$ any subarray formed by t columns of A_i joined with $\mathbf{i}$ covers all $(t+1)$-tuples of the form $(x_0, x_1, \ldots, x_{t-1}, x_t = i)$. Then, any subarray formed by t columns of G and by column E covers all $(t+1)$-tuples over $\mathbb{Z}_v$. Therefore, the horizontal concatenation of G and E is a CA$(N; t+1, k+1, v)$. □

From Theorem 4.1.1 if CA$(N; t+1, k+1, v)$ exists then it can be constructed by juxtaposing vertically v CAs with strength t and k columns. Also, if CA$(N; t+1, k+1, v)$ does not exist then there are no v CAs with strength t and k columns that juxtaposed vertically form a CA$(N; t+1, k, v)$.

Based on this theorem we developed an algorithm to classify CA$(N; t+1, k+1, v)$ by generating all possible juxtapositions of v CAs with strength t and k columns. To each CA$(N; t+1, k, v)$ the column $E = (\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$ is added to form a CA$(N; t+1, k+1, v)$. The algorithm constructs all CA$(N; t+1, k+1, v)$ whose elements of the last column are sorted. The constructed CAs cover all isomorphism classes because for any CA there is at least one isomorphic CA whose last column is sorted. To obtain only the canonical representatives of the isomorphism classes, the CAs are canonized and duplicates are removed. The classification algorithm based on juxtapositions of v CAs of a smaller strength is called *JuxtaposeCA algorithm*.

4.2 Classification of CAs by juxtapositions

The number of possible juxtapositions of v CAs with strength t and k columns may be very large, so the classification algorithm should be able to:

- Use the isomorphisms of CAs to avoid testing juxtapositions that will produce a result isomorphic to the result of another juxtaposition.
- Use the coverage properties of CAs to discard juxtapositions with no possibilities of being a CA with strength $t + 1$.

Subsection 4.2.1 describes the strategies of the classification algorithm to ensure the coverage of all possible juxtapositions of v CAs of strength t and k columns, and to skip juxtapositions that produce isomorphic arrays. The most important step of the classification algorithm is the generation of all possible juxtapositions that can be derived from a given tuple of v non-isomorphic CAs of strength t and k columns. Subsection 4.2.2 presents an algorithm to perform this step; this algorithm is able to skip a number of juxtapositions with no possibilities of being a CA of strength $t + 1$.

4.2.1 Strategy to generate all possible juxtapositions

The algorithm developed in this work to construct the non-isomorphic CA(N; $t+1$, $k+1$, v) verifies all possible juxtapositions of v CAs with strength t and k columns to see which of them produce a CA($N; t + 1, k, v$). To each constructed CA($N; t + 1, k, v$) the column $E = (\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$ is added to form a CA($N; t+1, k+1, v$). None of the resulting CA($N; t+1, k+1, v$) is canonical, and there may be isomorphic CAs in the results. To obtain only the canonical CA of each isomorphism class the CA$(N; t+1, k+1, v)$ are canonized and duplicate elements are removed.

The first step of the JuxtaposeCA algorithm is to determine the multisets S_j =$\{N_0$, N_1, ..., $N_{v-1}\}$ of v elements such that $N_i \geq$ CAN(t, k, v) and $N = \sum_{i=0}^{v-1} N_i$. These multisets will be called *valid multisets*. To classify for example CA$(27; 3, 5, 3)$ we need to check all juxtapositions of three CAs with strength two, four columns, and number of rows given by $S_0 = \{9, 9, 9\}$. In this case $S_0 = \{9, 9, 9\}$ is the unique valid multiset because CAN$(2, 4, 3) = 9$; therefore, if CA$(27; 3, 5, 3)$ exists, it is necessarily composed by three CA$(9; 2, 4, 3)$. On the other hand, to classify CA$(29; 3, 5, 3)$ the multisets to consider are $S_0 = \{9, 9, 11\}$ and $S_1 = \{9, 10, 10\}$, because CA$(29; 3, 5, 3)$ can be composed by two CAs of nine rows and one CA of eleven rows, or by one CA of nine rows and two CAs of ten rows.

The second step of the algorithm generates the non-isomorphic CAs with strength t, k columns, and number of rows given by a valid multiset $S_j = \{N_0, N_1, \dots, N_{v-1}\}$ where $N = \sum_{i=0}^{v-1} N_i$. From each non-isomorphic CA the other members of its isomorphism class will be derived by permutations of rows, columns, and symbols. To construct the non-isomorphic CAs we can use the improved NonIsoCA algorithm developed in Chapter 3, or any other algorithm for the same purpose.

Given a valid multiset $S_j = \{N_0, N_1, \dots, N_{v-1}\}$, let D_i $(0 \leq i \leq v-1)$ be the set of all non-isomorphic CA$(N_i; t, k, v)$. From the CAs in the sets D_i all juxtapositions of v CAs whose number of rows are given by S_j will be generated. In the example with $S_0 = \{9, 9, 11\}$, the sets D_0 and D_1 contain the non-isomorphic CA$(9; 2, 4, 3)$, and the set D_2 contains the non-isomorphic CA$(11; 2, 4, 3)$. Now, let $P_j = \{(A_0, A_1, \dots, A_{v-1}) : A_i \in D_i \text{ for } 0 \leq i \leq v-1)\}$ be the Cartesian product of the sets D_i; then, P_j contains all possible ways of combining the non-isomorphic CAs with number of rows given by S_j.

The next step of the algorithm is to check all juxtapositions that are derived from tuple of P_j. For a tuple $T = (A_0, A_1, \dots, A_{v-1}) \in P_j$ let $[A_0; A_1; \cdots; A_{v-1}]$ denote the juxtaposition of the v CAs in T. From $[A_0; A_1; \cdots; A_{v-1}]$ we will generate all arrays $J = [A'_{r_0}; A'_{r_1}; \cdots; A'_{r_{v-1}}]$ where each A'_{r_s} is derived from exactly one $A_i \in T$ by permutations of rows, columns, and symbols in the columns; in other words, the list of indices $(r_0, r_1, \ldots, r_{v-1})$ is a permutation π of $(0, 1, \ldots, v-1)$, and $A'_{\pi(i)}$ is isomorphic to A_i.

The number of arrays J that are derived from one tuple $T \in P_j$ is $v! \prod_{i=0}^{v-1} N_i! k! (v!)^k$. Each array J is checked to see if it is a CA($N; t+1, k, v$). If this is the case, then CA($N; t+1, k+1, v$) exists by Theorem 4.1.1, and the CA is obtained by adding to J the column $E = (\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$.

For each tuple of P_j all possible arrays J are generated; then, all possible juxtapositions of v CAs with strength t, k columns, and number of rows given by a valid multiset S_j are explored. Since this is done for every valid multiset S_j, all possible juxtapositions of v CAs with strength t and k columns are explored.

The number of generated juxtapositions may be very large, however some juxtapositions produce isomorphic arrays, and to accelerate the search we need to skip as many isomorphic arrays as possible. Fortunately, the number of arrays J created from a tuple $T = (A_0, A_1, \ldots, A_{v-1})$ of P_j can be reduced considerably. Consider the horizontal concatenation of an array J and the column $E = (\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$, denoted as (JE):

$$(JE) = \begin{pmatrix} A'_{r_0} & \mathbf{0} \\ A'_{r_1} & \mathbf{1} \\ \vdots & \vdots \\ A'_{r_{v-1}} & \mathbf{v-1} \end{pmatrix}$$

We can reorder the rows of (JE) so that the array derived from A_0 is placed in the first rows of (JE), the array derived from A_1 is placed next, and so on. This permutation of rows produces an array $(JE)'$ isomorphic to (JE), and by a permutation of symbols in the last column of $(JE)'$ it is possible to transform the last column of $(JE)'$ in $(\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$:

$$(JE) = \begin{pmatrix} A'_{r_0} & \mathbf{0} \\ A'_{r_1} & \mathbf{1} \\ \vdots & \vdots \\ A'_{r_{v-1}} & \mathbf{v-1} \end{pmatrix} \simeq \begin{pmatrix} A'_0 & \mathbf{l_0} \\ A'_1 & \mathbf{l_1} \\ \vdots & \vdots \\ A'_{v-1} & \mathbf{l_{v-1}} \end{pmatrix} \simeq \begin{pmatrix} A'_0 & \mathbf{0} \\ A'_1 & \mathbf{1} \\ \vdots & \vdots \\ A'_{v-1} & \mathbf{v-1} \end{pmatrix}$$

Therefore, the arrays J to be generated from a tuple $T = (A_0,\ A_1,\ \ldots,\ A_{v-1})$ of P_j are those arrays $J = [A'_0; A'_1; \cdots; A'_{v-1}]$ where for $0 \leq i \leq v-1$ the array A'_i is derived from A_i by permutations of rows, columns, and symbols. So, the number of arrays J is reduced from $v! \prod_{i=0}^{v-1} N_i!k!(v!)^k$ to $\prod_{i=0}^{v-1} N_i!k!(v!)^k$.

Another reduction in the number of arrays $J = [A'_0; A'_1; \cdots; A'_{v-1}]$ is possible: we can permute the first N_0 rows of J, and permute the columns and symbols in the entire array J to get an array $J' = [A''_0; A''_1; \cdots; A''_{v-1}]$, where $A''_0 = A_0$ and $A''_1, \ldots, A''_{v-1}$ are other CAs isomorphic to the original arrays $A_1, \ldots, A_{v-1}$. Thus, the following arrays are isomorphic:

$$\begin{pmatrix} A'_0 & \mathbf{0} \\ A'_1 & \mathbf{1} \\ \vdots & \vdots \\ A'_{v-1} & \mathbf{v-1} \end{pmatrix} \simeq \begin{pmatrix} A_0 & \mathbf{0} \\ A''_1 & \mathbf{1} \\ \vdots & \vdots \\ A''_{v-1} & \mathbf{v-1} \end{pmatrix}$$

In this way, the arrays J to be generated from a tuple $T = (A_0, A_1, \ldots, A_{v-1})$ of P_j are those arrays $J = [A_0; A'_1; \cdots; A'_{v-1}]$ where A_0 is fixed, and for $1 \leq i \leq v-1$ the array A'_i is derived from A_i by permutations of rows, columns, and symbols. Since A_0 is fixed the number of arrays J is now $\prod_{i=1}^{v-1} N_i!k!(v!)^k$.

Finally, note that block A'_i of J is complemented with a column vector $\mathbf{i}$ formed by N_i elements equal to i. Then, any row permutation of A'_i produces an array isomorphic to J. On the contrary, column and symbol permutations in A'_i do not produce in general arrays isomorphic to J. Therefore, the only arrays A'_i necessary to explore are those derived from A_i by permutations of columns and symbols. In this way, the number of arrays J to be generated from a tuple $T \in P_j$ is $[k!(v!)^k]^{v-1}$.

Algorithm 8 presents the JuxtaposeCA algorithm to classify CAs. The number of columns $k = k' - 1$ and the strength $t = t' - 1$ of the CAs to be juxtaposed are obtained from the input parameters k' and t'. The generation of the valid multisets $\{N_0, N_1, \ldots, N_{v-1}\}$ can be accomplished without difficulty, but it requires to know the value of CAN(t, k, v). The construction of the sets D_i requires the computation of the non-isomorphic CA$(N_i; t, k, v)$, which as mentioned before can be done with any classification algorithm. The key function is *generate_juxtapositions*(T), where arrays $J = [A_0; A'_1; \cdots; A'_{v-1}]$ are generated from a tuple T of the set P; this function will be

Algorithm 8: juxtapose_algorithm(N, k', t', v)

```
1  $k \leftarrow k' - 1$;
2  $t \leftarrow t' - 1$;
3  $\mathcal{S} \leftarrow$ all multisets $\{N_0, N_1, \ldots, N_{v-1}\}$ such that $N_i \geq$ CAN$(t, k, v)$ and $N = \sum_{i=0}^{v-1} N_i$;
4  $R \leftarrow \emptyset$;
5  foreach $S \in \mathcal{S}$ do
6      for $i = 0, \ldots, v-1$ do
7          $D_i \leftarrow$ all non-isomorphic CA$(N_i; t, k, v)$;
8      $P = D_0 \times D_1 \times \cdots \times D_{v-1} = \{(A_0, A_1, \ldots, A_{v-1}) : A_i \in D_i$ for $0 \leq i \leq v-1)\}$;
9      foreach $T = (A_0, A_1, \ldots, A_{v-1}) \in P$ do
10         generate_juxtapositions$(T)$;
11 $F \leftarrow \emptyset$;
12 foreach $B \in R$ do
13     $B' \leftarrow$ canonize$(B)$;
14     if $B' \notin F$ then $F \leftarrow F \cup \{B'\}$;
15 write$(F)$;
```

described in Subsection 4.2.2.

The CA$(N; t+1, k+1, v)$ that are constructed by *generate_juxtapositions*(T) are stored in a set R, which is initialized at line 4 with the empty set. There may be isomorphic CAs in R; then to obtain only the non-isomorphic CA$(N; t+1, k+1, v)$ the CAs in R are canonized and added to the final result set F. The *canonize*() function is implemented in Subsection 4.2.3.

4.2.2 Generating all juxtapositions of v non-isomorphic CAs

The crucial step of the JuxtaposeCA algorithm is to generate all arrays $J = [A_0; A_1'; \cdots; A_{v-1}']$ that can be derived from a given v-tuple of CAs $T = (A_0, A_1, \ldots, A_{v-1})$. Recall that in each J, the array A_0 is fixed, and for $1 \leq i \leq v-1$ the array A_i' is derived from A_i by permutations of columns and symbols. After generating an array J, the algorithm checks if J is a CA$(N; t+1, k, v)$.

This section presents an algorithm to perform this crucial step. The algorithm constructs J one column at a time validating that each new column forms a CA of strength $t+1$ with the columns previously added to J. This is done to avoid the exploration of arrays J with no possibilities of being a CA$(N; t+1, k, v)$. The algorithm starts by constructing the following array J, where block A_0 is fixed, and blocks $F_1, \dots, F_{v-1}$ are unassigned or free; later on these arrays will be filled with arrays derived from $A_1, \dots, A_{v-1}$ by permutations of columns and symbols:

$$J = \begin{pmatrix} A_0 \\ F_1 \\ \vdots \\ F_{v-1} \end{pmatrix}$$

For $1 \leq i \leq v-1$ let $f_{i_0}, f_{i_1}, \dots, f_{i_{k-1}}$ be the k columns of F_i, and let a_{i_0}, a_{i_1}, ..., $a_{i_{k-1}}$ be the k columns of A_i. Then, the previous array J is equivalent to this one:

$$J = \begin{pmatrix} a_{0_0} & a_{0_1} & \cdots & a_{0_{k-1}} \\ f_{1_0} & f_{1_1} & \cdots & f_{1_{k-1}} \\ f_{2_0} & f_{2_1} & \cdots & f_{2_{k-1}} \\ \vdots & \vdots & \ddots & \vdots \\ f_{v-1_0} & f_{v-1_1} & \cdots & f_{v-1_{k-1}} \end{pmatrix}$$

The algorithm fills column 0 in all free blocks, then it fills column 1 in all free blocks, and so on. In this way, columns $f_{1_0}, f_{2_0}, \dots, f_{v-1_0}$ are filled first, then columns $f_{1_1}, f_{2_1}, \dots, f_{v-1_1}$ are filled, and so on. When the first $t+1$ columns of all free blocks have been filled or assigned, the algorithm checks if they form a CA of strength $t+1$. Columns are indexed from 0, so the first $t+1$ columns of J are formed by columns $a_{0_0}, a_{0_1}, \dots, a_{0_t}$, and by columns $f_{i_0}, f_{i_1}, \dots, f_{i_t}$ for $1 \leq i \leq v-1$.

If the first $t+1$ columns of J form a CA of strength $t+1$, then the algorithm advances to the next column of the free blocks, and column $f_{1_{t+1}}$ is assigned, then column $f_{2_{t+1}}$ is assigned, and so on until column $f_{v-1_{t+1}}$ is assigned. At this point the algorithm verifies if the current $t+2$ columns of J form a CA of strength $t+1$. In the negative case the current value of $f_{v-1_{t+1}}$ is replaced by its next available value to see if the first $t+2$ columns of J form a CA of strength $t+1$. This is done for all available values of $f_{v-1_{t+1}}$, and when all values are checked the algorithm backtracks to $f_{v-2_{t+1}}$ and assigns to it its next available value; in the next step the algorithm advances to $f_{v-1_{t+1}}$

to check again all its available values.

To construct all possible arrays J, the algorithm fills the free block F_i with all isomorphic CAs derived from A_i by permutations of columns and symbols. Thus, the possible values for a column of F_i are the columns obtained by permuting symbols in the columns of A_i; so the number of available values for a column of F_i is $(v!)^k$. When the first r columns of F_i have been assigned the number of available values for f_{i_r} is $(v!)^{k-r}$, which are the $v!$ relabelings of the columns of A_i not currently assigned to one of the first r columns of F_i.

In every free block F_i the algorithm works as follows: columns of A_i are added to F_i in such a way that f_{i_0} gets all columns $a_{i_0}, \ldots, a_{i_{k-1}}$ in order; then for a fixed value of f_{i_0}, column f_{i_1} gets in order all columns of F_i distinct from the one assigned to f_{i_0}; and for fixed values of f_{i_0} and f_{i_1}, column f_{i_2} gets in order all columns of A_i not currently assigned to f_{i_0} or f_{i_1}; the same applies for the other columns of F_i. In this way F_i gets all CAs derived from A_i by permutations of columns.

However, for each column permutation of A_i the JuxtaposeCA algorithm requires to test all possible symbol permutations in the columns of A_i. Symbol permutations are integrated in the following way: suppose that the first r columns of F_i have been assigned, and that the next free column f_{i_r} of F_i gets assigned column a_{i_j} of A_i; we can consider that the current value of f_{i_r} is the identity relabeling of a_{i_j}; the next $v! - 1$ values to assign to f_{i_r} are the other $v! - 1$ relabelings of a_{i_j}. When all relabelings of a_{i_j} are assigned to f_{i_r}, the next value for f_{i_r} is the identity relabeling of the next column of A_i that has not been assigned to f_{i_r}.

The *generate_juxtapositions*() function of Algorithm 9 receives as parameter a v-tuple $T = (A_0, A_1, \ldots, A_{v-1})$ of CAs. This function initializes the fixed block A_0 of J, and initializes with *false* the elements of a $v \times k$ matrix called *assigned*; this matrix is used to record which columns of A_i are currently assigned to a column of F_i. The last sentence of the function is a call to the *add_column*() function, which fills the free blocks F_i with CAs derived from A_i by permutations of columns and symbols. The *add_column*() function is called from *generate_juxtapositions*() with arguments 1 and 0, because the first column to fill in the array J is column 0 of the free block F_1, or f_{1_0}.

Algorithm 9: generate_juxtapositions($T = (A_0, A_1, \ldots, A_{v-1})$)

```
1 $J \leftarrow$ array($N, k$);
2 for $i = 0, \ldots, N_0 - 1$ do
3     for $j = 0, \ldots, k - 1$ do
4         $J[i][j] \leftarrow A_0[i][j]$;
5 for $i = 0, \ldots, v - 1$ do
6     for $j = 0, \ldots, k - 1$ do
7         assigned$[i][j] \leftarrow$ false;
8 add_column($1, 0$);
```

The *add_column*() function of Algorithm 10 receives an index i of a free block and an index r of a column of the free block as parameters; the work of the function is to set column f_{i_r}. The variable F_i is used as an alias of the block of J where a CA derived from A_i will be placed. The function relies on recursion to assign column 0 of every free block, then to assign column 1 of every free block, and so on. In addition, in every free block F_i recursion allows to test in column r all columns of A_i not currently assigned to a column of F_i; the main *for* loop iterates over all columns j of A_i, but the body of the loop is executed only for those columns j for which *assigned*$[i][j]$ is equal to *false*. Recursion also allows to check in order the $v!$ symbol permutations $\epsilon_0, \epsilon_1, \ldots, \epsilon_{v!-1}$ of the column of A_i assigned to column r of F_i. If a CA$(N; t+1, k, v)$ is constructed, then column E is appended to it to form a covering array $B =$ CA$(N; t+1, k+1, v)$.

The *sort_rows*() function sorts the rows of B to obtain an isomorphic CA B'; finally B' is added to the set R if B' is not currently in the set. The sorting of the rows of B is done because we found experimentally that many of the arrays $B = (JE)$ are equal after a row sorting. So, the row sorting helps to reduce the number of isomorphic solutions in the set R. Instead of sorting the rows of B, we could have canonized B to obtain the canonical representative of the class to which B belongs; however, the canonization is much more costly than the row sorting. In the worst case, the canonization of $B =$ CA$(N; t+1, k+1, v)$ takes time $O(N \log_2 N \cdot (k+1)! \cdot (v!)^{k+1})$, and the row sorting takes time proportional to $O(N \log_2 N)$.

Algorithm 10: add_column((i, r))

```
for $j = 0, \dots, k-1$ do
    if assigned$[i][j]$ = false then
        assigned$[i][j] \leftarrow$ true;
        foreach permutation $\epsilon$ of the symbols $\{0, 1, \dots, v-1\}$ do
            copy column $j$ of $A_i$ to column $r$ of $F_i$ and permute its symbols using $\epsilon$;
            if $i = v - 1$ then
                if $r < t$ or is_covering_array$(J, r)$ = true then
                    if $r = k - 1$ then
                        $B \leftarrow (JE)$;
                        $B' \leftarrow$ sort_rows$(B)$;
                        if $B' \notin R$ then
                            $R \leftarrow R \cup \{B'\}$;
                    else
                        add_column$(1, r + 1)$;
            else
                add_column$(i + 1, r)$;
        assigned$[i][j] \leftarrow$ false;
```

For a v-tuple of CAs $T = (A_0, A_1, \dots, A_{v-1})$, Algorithm 9 and its helper function Algorithm 10 generate in the worst case $[k!(v!)^k]^{v-1}$ arrays $J = [A_0; A'_1; \cdots; A'_{v-1}]$, since array A_0 is fixed and $A'_1, \dots, A'_{v-1}$ are derived respectively from $A_1, \dots, A_{v-1}$ by permutations of columns and symbols. However, the condition that every new column added to the partial array J must form a CA of strength $t + 1$ with the previous columns of J reduces the number of arrays J that are explored. For example if the condition fails at the column with index j, then in each free block F_i we skip the remaining $(k - j - 1)!$ permutations of columns for the free columns $f_{i_{j+1}}, \dots, f_{i_{k-1}}$, plus the $(v!)^{k-j-1}$ associated column relabelings.

We can see in Algorithm 10 what makes the JuxtaposeCA algorithm significantly distinct from previous ones. The target covering array CA$(N; t+1, k+1, v)$ is not constructed element by element, but subcolumn by subcolumn, where a subcolumn is a column of a CA of strength t. Nevertheless, our algorithm requires the construction of the non-isomorphic CAs of strength t and k columns, which could have been constructed element by element, or by any other procedure. However, the cost of constructing the non-isomorphic CA$(N_i; t, k, v)$, plus the cost of exploring the juxtapositions of v CAs derived from them by permutations of columns and symbols, is in general smaller than the cost of constructing CA$(N; t+1, k+1, v)$ element by element (at least with the implemented algorithms) if the number of distinct CA$(N_i; t, k, v)$ is not very large, as we will see in Section 5.3.

4.2.3 Canonization of covering arrays

The *canonize*(A) function is used in Algorithm 8 to obtain the canonical CA isomorphic to A. This function can be derived from a slight modification of the *is_canonical*(A, r) function of Algorithm 2. Algorithms 11 and 12 implement the canonization of a CA with size $N \times k$ and order v.

Algorithm 11: canonize(A)

```
1 $M \leftarrow A$;
2 for $i = 0, 1, \ldots, k-1$ do
3     assigned$[i] \leftarrow$ false;
4 check_column$(A, 0)$;
5 return $M$;
```

The objective is to transform A into its isomorphic CA which is canonical. In Algorithm 11 the array M is initialized with A, but in Algorithm 12 M is updated when it is found an array D smaller than M. Here smaller means smaller in lexicographic order when the arrays are linearized by columns. The $V(X, r)$ function returns a vector of length $N \cdot r$ containing the elements of the first r columns of X in column-major order.

The k elements of the *assigned* vector are initialized with *false* in Algorithm 11 to indicate that none column of A has been copied to a column of the auxiliary array D. The k columns of the array D are $d_0, d_1, \ldots, d_{k-1}$. Algorithm 11 calls the *check_column*() function with parameters A and 0 because the k columns of A will be copied to the column 0 of D one at a time. After the call to *check_column*() returns, the function returns M to its caller.

Algorithm 12: check_column(A, s)

```
1  if $s = k$ then
2      return;
3  for $j = 0, 1, \ldots, k-1$ do
4      if assigned[$j$] = false then
5          assigned[$j$] ← true;
6          foreach permutation of symbols $\varepsilon$ do
7              $d_s \leftarrow a_j$ relabeled with $\varepsilon$;
8              sort_rows($D, s+1$);
9              if $V(D, s+1)$ is lexicographically smaller than $V(M, s+1)$ then
10                 copy the $k-s+1$ non-used columns of $A$ to the last $k-s+1$ columns of $D$;
11                 $M \leftarrow D$;
12                 check_column($A, s+1$);
13             else if $V(D, s+1) = V(M, s+1)$ then
14                 check_column($A, s+1$);
15         assigned[$j$] ← false;
```

When the algorithm finds an array D smaller than M up to $s+1$ columns the action is to copy the non-assigned columns of A to D, and copy D to M. If D is smaller than or equal to M up to $s+1$ columns, then the *check_column*() function is called recursively to fill the next column of the auxiliary array D. If D is greater than M, then the function is not called recursively because the current array D can not lead to the canonical representative of the isomorphism class of A.

The basic difference between the *is_canonical*(A) function of Algorithm 2 and the canonize(A) function is the following one: the first function returns *false* as soon as it constructs an array D smaller than A, but the second function copies D to M and calls itself recursively to search for another array D smaller than the current M. The complexity of both functions is $O(N \log_2 N \cdot k! \cdot (v!)^k)$.

4.3 Complexity of the JuxtaposeCA algorithm

In an execution of the JuxtaposeCA algorithm to classify CA($N; t+1, k+1, v$) we need to known in advance: (a) the value of CAN(t, k, v); (b) the valid multisets $\{N_0, N_1, \ldots, N_{v-1}\}$ such that each

N_i is greater than or equal to CAN(t,k,v) and $N = \sum_{i=0}^{v-1} N_i$; and (c) the non-isomorphic CAs that exist for every possible number of rows N_i in a valid multiset $\{N_0, N_1, \dots, N_{v-1}\}$.

Let $m =$ CAN(t,k,v); so mv is the number of rows that are obtained by juxtaposing v optimal CA$(m;t,k,v)$. If $N = mv$ then the unique valid multiset is $\{N_0 = m, N_1 = m, \dots, N_{v-1} = m\}$. Otherwise, the number of valid multisets is equal to the number of partitions of the positive integer $N - mv$ into at most v parts, which is denoted by $p(N - mv, v)$. The number of partitions of $N - mv$ into at most v parts is the number of ways in which we can distribute the extra $N - mv$ rows among the v CAs of size m. Each partition $\{x_0, x_1, \dots, x_{r-1}\}$ of $N - mv$, where $x_i > 0$, $r \le v$, and $N - mv = \sum_{i=0}^{r-1} x_i$, generates a valid multiset $\{N_0, N_1, \dots, N_{v-1}\}$ in the following way: $N_i = m + x_i$ if $i < r$, and $N_i = m$ if $i \ge r$.

For a multiset $S_j = \{N_{0_j}, N_{1_j}, \dots, N_{v-1_j}\}$ the non-isomorphic CA$(N_{0_j};t,k,v)$, CA$(N_{1_j};t,k,v)$, …, CA$(N_{v-1_j};t,k,v)$ to be juxtaposed are stored respectively in sets $D_{0_j}, D_{1_j}, \dots, D_{v-1_j}$. So, the number of v-tuples of non-isomorphic CAs for S_j is $|D_{0_j} \times D_{1_j} \times \cdots \times D_{v-1_j}|$. Thus, the total number of v-tuples of non-isomorphic CAs to be processed by the JuxtaposeCA algorithm is:

$$\sum_{j=0}^{p(N-mv,v)-1} |D_{0_j} \times D_{1_j} \times \cdots \times D_{v-1_j}|.$$

Each of these v-tuples of non-isomorphic CAs is processed by the *generate_juxtapositions*() function of Algorithm 9, and in the worst case each v-tuple generates $[k!(v!)^k]^{v-1}$ arrays J of size $N \times k$. For each J, a CA test is performed to determine if J is a CA$(N;t+1,k,v)$; this test requires $O(\binom{k}{t+1}N(t+1))$ time. The column $E = (\mathbf{0}\ \mathbf{1}\ \cdots\ \mathbf{v-1})^T$ is added to the arrays J that pass the CA test. In the worst case each array (JE) is sorted by rows in $O(N \log_2 N)$ time and then canonized in $O(N \log_2 N \cdot (k+1)! \cdot (v!)^{k+1})$ time, but the cost of the row sorting is absorbed by the cost of the canonization. Therefore, the computational cost of the JuxtaposeCA algorithm is:

$$O\Bigg(\sum_{j=0}^{p(N-v\,\mathsf{CAN}(t,k,v),\,v)-1} |D_{0_j} \times \cdots \times D_{v-1_j}| \cdot [k!(v!)^k]^{v-1} \cdot \binom{k}{t+1} N(t+1) \cdot (N \log_2 N)(k+1)!(v!)^{k+1}\Bigg).$$

4.4 Parallelization of the JuxtaposeCA algorithm

In this section we develop two parallel implementations of the JuxtaposeCA algorithm using MPI. The first implementation studied in Subsection 4.4.1 parallelizes the *for-each* loop located at line 9 of Algorithm 8. The second implementation exposed in Subsection 4.4.2 parallelizes the calls to the *add_column*() function of Algorithm 10.

4.4.1 Parallelization of the calls to *generate_juxtapositions*(T)

The first approach to parallelize the JuxtaposeCA algorithm is to parallelize the *for-each* loop of line 9 of Algorithm 8. For each valid multiset $\{N_0, N_1, \ldots, N_{v-1}\}$, the body of this loop is executed $\prod_{i=0}^{v-1} |D_i|$ times, where $|D_i|$ is the number of non-isomorphic CA($N_i; t, k, v$). If the number of valid multisets or the number of non-isomorphic CA($N_i; t, k, v$) is large, then the body of the *for-each* loop, which consists in a call to *generate_juxtapositions*(T), is executed many times.

Algorithm 13 shows the pseudocode of the master process. In the MPI functions the variable *comm* is the default communicator MPI_COMM_WORLD. The first four sentences are the same as in Algorithm 8 to obtain the valid multisets and to initialize the set R. After that in line 5 the stack with the ranks of idle processes is initialized. In the *for-each* loop of lines 10-18 the *while* loop waits until there is at least one idle process; after the *while* loop ends, the rank of the idle process on top of the stack is assigned to the variable *slave*. The *message* vector is used to indicate to the slave the type of action to do; the options are TEST_JUXTAPOSITIONS, CANONIZE_CA, and EXIT_SLAVE.

The master sends the tuple $T = (A_0,\ A_1,\ \ldots,\ A_{v-1})$ to a slave in two steps; in the first step the master sends a message with TEST_JUXTAPOSITIONS to indicate to the slave that the next message will be an array of size $N \cdot k$ containing the tuple T. After all tuples T for each valid multiset have been sent to a slave, the work of the master process is to wait in the *while* loop of lines 19-21 until all slaves finish the processing of the last tuple T they received.

Algorithm 13: master(N, k', t', v)

```
1  $k \leftarrow k' - 1$;
2  $t \leftarrow t' - 1$;
3  $\mathcal{S} \leftarrow$ all multisets $\{N_0, N_1, \ldots, N_{v-1}\}$ such that $N_i \geq$ CAN$(t, k, v)$ and $N = \sum_{i=0}^{v-1} N_i$;
4  $R \leftarrow \emptyset$;
5  *stack* $\leftarrow \{1, 2, \ldots, P-1\}$;
6  **foreach** $S \in \mathcal{S}$ **do**
7      **for** $i = 0, \ldots, v-1$ **do**
8          $D_i \leftarrow$ all non-isomorphic CA$(N_i; t, k, v)$;
9      $P = D_0 \times D_1 \times \cdots \times D_{v-1} = \{(A_0, A_1, \ldots, A_{v-1}) : A_i \in D_i$ for $0 \leq i \leq v-1)\}$;
10     **foreach** $T = (A_0, A_1, \ldots, A_{v-1}) \in P$ **do**
11         **while** *stack_empty(stack)* **do**
12             MPI_Recv(*message*, 2, MPI_INT, MPI_ANY_SOURCE, T0, *comm*, &*stat*);
13             handle_message(*message*);
14         *slave* $\leftarrow$ pop(*stack*);
15         *message*[0] $\leftarrow$ TEST_JUXTAPOSITIONS;
16         copy_tuple(*data*, $T$);
17         MPI_Send(*message*, 1, MPI_INT, *slave*, T0, *comm*);
18         MPI_Send(*data*, $N * k$, MPI_CHAR, *slave*, T1, *comm*);
19 **while** *stack_size(stack)* $< P - 1$ **do**
20     MPI_Recv(*message*, 2, MPI_INT, MPI_ANY_SOURCE, T0, *comm*, &*stat*);
21     handle_message(*message*);
22 $F \leftarrow \emptyset$;
23 canonize_CAs($R$);
24 write($F$);
25 *message*[0] $\leftarrow$ EXIT_SLAVE;
26 **for** $j \leftarrow 1$ **to** $P - 1$ **do**
27     MPI_Isend(*message*, 1, MPI_INT, *j*, T0, *comm*, &*req*);
28 MPI_Finalize();
```

After processing all tuples T, the next step is to canonize the constructed CA$(N; t+1, k+1, v)$. The *canonize_CAs*() function of Algorithm 14 divides the CAs in R among the slave processes; each CA $B \in R$ is sent to a slave, the slave canonizes the CA and returns the canonical CA to the master;

Algorithm 14: canonize_CAs(R)

```
1  foreach $B \in R$ do
2      while stack_empty(stack) do
3          MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE, T0, comm, &stat);
4          handle_message(message);
5      slave ← pop(stack);
6      message[0] ← CANONIZE_CA;
7      copy_CA(data, $B$);
8      MPI_Send(message, 1, MPI_INT, slave, T0, comm);
9      MPI_Send(data, $N * (k + 1)$, MPI_CHAR, slave, T1, comm);
10 while stack_size(stack) $< P - 1$ do
11     MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE, T0, comm, &stat);
12     handle_message(message);
```

finally, the master stores the canonical CAs without repetitions in the final result set F. When the CAs have been canonized the master sends the slaves a message with EXIT_SLAVE.

The *handle_message*() function invoked in lines 13 and 21 of Algorithm 13, and in lines 4 and 12 of Algorithm 14, is given in Algorithm 15. This function processes the messages received from the slaves. These messages can be of three types: THREAD_FINALIZED, SOLUTION_FOUND, or CA_CANONIZED. The action for THREAD_FINALIZED is to push onto the stack the rank of the process who sent the message; the action for SOLUTION_FOUND is to add the received CA($N; t+1, k+1, v$) to the set R if the CA is not in the set; and the action of CA_CANONIZED is to push onto the stack the rank of the process who sent the message, and to add the received canonical CA to the set F.

Algorithm 16 contains the function executed by the slaves. The first sentence initializes a set R with the empty set; this set R will store the non-repeated CA($N; t+1, k+1, v$) after a row sorting. Then, both the master and the slaves have a set R to store the non-repeated CAs. Since there is no communication among the slaves, they are unaware of the CA($N; t+1, k+1, v$) that are generated in other slaves, so the master needs to filter the duplicate CAs that were generated by different

Algorithm 15: handle_message(*message*)

```
1  source ← message[0];
2  if message[1] = THREAD_FINALIZED then
3      push(stack, source);
4  else if message[1] = SOLUTION_FOUND then
5      MPI_Recv(data, $N * (k+1)$, MPI_CHAR, source, T1 , comm, &stat);
6      $B$ ← get_CA(data);
7      if $B \notin R$ then
8          $R \leftarrow R \cup \{B\}$;
9  else if message[1] = CA_CANONIZED then
10     push(stack, source);
11     MPI_Recv(data, $N * (k+1)$, MPI_CHAR, source, T1 , comm, &stat);
12     $B$ ← get_CA(data);
13     if $B \notin F$ then
14         $F \leftarrow F \cup \{B\}$;
```

slaves.

If the message type is TEST_JUXTAPOSITIONS, then the slave regenerates the tuple $T = (A_0, A_1, \ldots, A_{v-1})$ from the received data in the *data* vector, and calls the *generate_juxtapositions*(T) function to test all possible arrays J than can be derived from the CAs in T by permutations of columns and symbols in the columns. The *generate_juxtapositions*() function in the slaves is identical to the function in Algorithm 9 of the sequential algorithm. However, the *add_column*() function, invoked from *generate_juxtapositions*(), is slightly different from the function given in Algorithm 10. The only difference is that after adding the array B' to the set R this array is sent to the master to be recorded in its own set R. Algorithm 17 shows the *add_column*() function that is executed by the slaves.

If the message type is CANONIZE_CA, then the slave canonizes the received CA and returns the canonical CA to the master. If the message type is EXIT_SLAVE, then the slave calls the *MPI_finalize*() function to finalize.

Algorithm 16: slave()

```
1  $R \leftarrow \emptyset$;
2  while true do
3      MPI_Recv(message, 1, MPI_INT, 0, T0, comm, &stat);
4      if message[0] = TEST_JUXTAPOSITIONS then
5          MPI_Recv(data, $N * k$, MPI_CHAR, 0, T1, comm, &stat);
6          get_tuple($T$,data);
7          generate_juxtapositions($T$);
8          message[0] $\leftarrow$ rank;
9          message[1] $\leftarrow$ THREAD_FINALIZED;
10         MPI_Send(message, 2, MPI_INT, 0, T0, comm);
11     else if message[0] = CANONIZE_CA then
12         MPI_Recv(data, $N * (k + 1)$, MPI_CHAR, 0, T1, comm, &stat);
13         get_CA($B$,data);
14         $B' \leftarrow$ canonize($B$);
15         copy_CA(data, $B'$);
16         message[0] $\leftarrow$ rank;
17         message[1] $\leftarrow$ CA_CANONIZED;
18         MPI_Send(message, 2, MPI_INT, slave, T0, comm);
19         MPI_Send(data, $N * (k + 1)$, MPI_CHAR, slave, T1, comm);
20     else if message[0] = EXIT_SLAVE then
21         MPI_Finalize();
```

4.4.2 Parallelization of the permutations of columns and symbols

The second approach to parallelize the JuxtaposeCA algorithm is intended for cases where the *generate_juxtapositions*(T) function is called a small number of times, but the CAs in the tuple $T = (A_0, A_1, \ldots, A_{v-1})$ are of considerable size. There may be cases where *generate_juxtapositions*(T) is called only once from Algorithm 8, but the time required by the function is long. So, in these scenarios we want to parallelize the work done inside the function *generate_juxtapositions*(), i.e., to parallelize the calls to the helper function *add_column*().

As mentioned before, the arrays J that are generated from a tuple $T = (A_0, A_1, \ldots, A_{v-1})$ are

Algorithm 17: add_column(i, r)

```
1  for $j = 0, \dots, k-1$ do
2      if assigned$[i][j]$ = false then
3          assigned$[i][j] \leftarrow$ true;
4          foreach permutation $\epsilon$ of the symbols $\{0, 1, \dots, v-1\}$ do
5              copy column $j$ of $A_i$ to column $r$ of $F_i$ and permute its symbols using $\epsilon$;
6              if $i = v - 1$ then
7                  if $r < t$ or is_covering_array$(J, r)$ = true then
8                      if $r = k - 1$ then
9                          $B \leftarrow (JE)$;
10                         $B' \leftarrow$ sort_rows($B$);
11                         if $B' \notin R$ then
12                             $R \leftarrow R \cup \{B'\}$;
13                             copy_CA(data, $B'$);
14                             MPI_Send(data, $N * (k+1)$, MPI_CHAR, 0, T1, comm);
15                     else
16                         add_column($1, r+1$);
17             else
18                 add_column($i+1, r$);
19         assigned$[i][j] \leftarrow$ false;
```

those arrays $J = [A_0; A'_1; \cdots; A'_{v-1}]$ where A_0 is fixed, and for $1 \leq i \leq v-1$ the array A'_i is derived from A_i by permutations of columns and symbols. Thus, for the array A_1 we need to generate all $k! \cdot (v!)^k$ permutations of columns and symbols in the worst case, although many of them are skipped when the algorithm finds that they do not have possibilities of being a CA of strength $t+1$.

The strategy we follow is to assign, or to make fixed, the first FIXED $< k$ columns of the first free block F_1 of the arrays J; this free block is used to contain the isomorphic copies of A_1. Let $P(n, r)$ be the number of permutations of size r from n objects. We can partition the $k!$ permutations of columns of A_1 in $P(k, \text{FIXED})$ chunks, where the chunks correspond to the $P(k, \text{FIXED})$ possible ways to assign the first FIXED columns of F_1 with FIXED columns of A_1. In addition, each of these chunks is relabeled using the $(v!)^{\mathsf{FIXED}}$ possible combinations of the $v!$ symbol relabelings for each of the FIXED columns. Thus, the number of partitions for a given value of FIXED is $P(k, \mathsf{FIXED}) \cdot (v!)^{\mathsf{FIXED}}$.

Algorithm 18 shows the function that is executed by the master process. At line 11, the *generate_partitions*(T) function is invoked to generate the $P(k, \mathsf{FIXED}) \cdot (v!)^{\mathsf{FIXED}}$ partitions of the arrays J that can be generated from a tuple $T = (A_0, A_1, \ldots, A_{v-1})$. The rest of the *master*() function is equal to the function with the same name in Algorithm 13; also the functions *handle_message*() and *canonize_CAs*() are the same as in the first parallel approach (previous subsection).

Algorithm 18: master(N, k', t', v)

```
1  $k \leftarrow k' - 1$;
2  $t \leftarrow t' - 1$;
3  $\mathcal{S} \leftarrow$ all multisets $\{N_0, N_1, \ldots, N_{v-1}\}$ such that $N_i \geq \mathsf{CAN}(t, k, v)$ and $N = \sum_{i=0}^{v-1} N_i$;
4  $R \leftarrow \emptyset$;
5  stack $\leftarrow \{1, 2, \ldots, P-1\}$;
6  foreach $S \in \mathcal{S}$ do
7      for $i = 0, \ldots, v-1$ do
8          $D_i \leftarrow$ all non-isomorphic CA$(N_i; t, k, v)$;
9      $P = D_0 \times D_1 \times \cdots \times D_{v-1} = \{(A_0, A_1, \ldots, A_{v-1}) : A_i \in D_i$ for $0 \leq i \leq v-1)\}$;
10     foreach $T = (A_0, A_1, \ldots, A_{v-1}) \in P$ do
11         generate_partitions($T$);
12 while stack_size(stack) $< P - 1$ do
13     MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE, T0, comm, &stat);
14     handle_message(message);
15 $F \leftarrow \emptyset$;
16 canonize_CAs($R$);
17 write($F$);
18 message[0] $\leftarrow$ EXIT_SLAVE;
19 for $j \leftarrow 1$ to $P - 1$ do
20     MPI_Isend(message, 1, MPI_INT, j, T0, comm, &req);
21 MPI_Finalize();
```

The *generate_partitions*(T) function is given in Algorithm 19; the work of this function is the creation of the partitions to process the tuple T. The *PC* vector contains a permutation of size FIXED from the k column indices of A_1. For each permutation of columns *PC* the $v!^{\mathsf{FIXED}}$ possible relabelings *PS* for the FIXED columns are generated. Every position of *PS* stores the index of a permutation of $\mathbb{Z}_v$ stored in a table called PERM_SYMB_TABLE; this table contains the $v!$ permutations of $\mathbb{Z}_v$. The permutation ε at index i of PERM_SYMB_TABLE represents the permutation $\left(\begin{smallmatrix} 0 & 1 & \cdots & v-1 \\ \varepsilon[0] & \varepsilon[1] & \cdots & \varepsilon[v-1] \end{smallmatrix}\right)$. The vectors *PC* and *PS* are sent in the *message* vector of size $2 * \mathsf{FIXED} + 1$.

Algorithm 19: generate_partitions(T)

```
foreach permutation PC of size FIXED from $\{0, 1, \ldots, k\}$ do
    foreach of the $v!^{FIXED}$ possible independent relabeling PS of the columns in PC do
        while stack_empty(stack) do
            MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE, T0, comm, &stat);
            handle_message(message);
        slave ← pop(stack);
        message[0] ← TEST_JUXTAPOSITIONS;
        copy PC into message from position 1 to position FIXED;
        copy PS into message from position FIXED + 1 to position 2 * FIXED;
        copy_tuple(data, $T$);
        MPI_Send(message, $2 * \mathsf{FIXED} + 1$, MPI_INT, slave, T0, comm);
        MPI_Send(data, $N * k$, MPI_CHAR, slave, T1, comm);
```

The *slave*() function is shown in Algorithm 20. If the message type is TEST_JUXTAPOSITIONS, then the function retrieves in its local vectors *PC* and *PS* the corresponding vectors sent by the master process. After that, the slave receives the tuple T of non-isomorphic CAs and calls *generate_juxtapositions*() with arguments T, PC, and PS. If the message type is CANONIZE_CA, the slave canonizes the received CA and returns the canonical CA to the master. If the message type is EXIT_SLAVE, the slave calls MPI_Finalize() to finish.

In the sequential algorithm and in the first parallel approach, the *add_column*(i, r) function is

Algorithm 20: slave()

```
1  $R \leftarrow \emptyset$;
2  while true do
3      MPI_Recv(message, $2 * \text{FIXED} + 1$, MPI_INT, 0, T0, comm, &stat);
4      if message[0] = TEST_JUXTAPOSITIONS then
5          PC $\leftarrow$ values in message from position 1 to position FIXED;
6          PS $\leftarrow$ values in message from position FIXED $+ 1$ to position $2 *$ FIXED;
7          MPI_Recv(data, $N * k$, MPI_CHAR, 0, T1, comm, &stat);
8          get_tuple($T$,data);
9          generate_juxtapositions($T, PC, PS$);
10         message[0] $\leftarrow$ rank;
11         message[1] $\leftarrow$ THREAD_FINALIZED;
12         MPI_Send(message, 2, MPI_INT, 0, T0, comm);
13     else if message[0] = CANONIZE_CA then
14         MPI_Recv(data, $N * (k + 1)$, MPI_CHAR, 0, T1, comm, &stat);
15         get_CA($B$,data);
16         $B' \leftarrow$ canonize($B$);
17         copy_CA(data, $B'$);
18         message[0] $\leftarrow$ rank;
19         message[1] $\leftarrow$ CA_CANONIZED;
20         MPI_Send(message, 2, MPI_INT, slave, T0, comm);
21         MPI_Send(data, $N * (k + 1)$, MPI_CHAR, slave, T1, comm);
22     else if message[0] = EXIT_SLAVE then
23         MPI_Finalize();
```

called by *generate_juxtapositions*() with arguments $i = 1$ and $r = 0$ because the next column to fill is the column 0 of the first free block. However, in the second parallel approach the arguments are:

- $i = 1$ and $r =$ FIXED, if $v = 2$
- $i = 2$ and $r = 0$, if $v > 2$

The *generate_juxtapositions*(T) function is given in Algorithm 21. Besides the tuple T, this function receives the partial permutation of columns *PC*, and the permutation of symbols *PS* for the

Algorithm 21: generate_juxtapositions($T = (A_0, A_1, \ldots, A_{v-1}), PC, PS$)

```
1  $J \leftarrow$ array$(N, k)$;
2  for $i = 0, \ldots, N_0 - 1$ do
3      for $j = 0, \ldots, k - 1$ do
4          $J[i][j] \leftarrow A_0[i][j]$;
5  for $i = 0, \ldots, v - 1$ do
6      for $j = 0, \ldots, k - 1$ do
7          assigned$[i][j] \leftarrow$ false;
8  $p \leftarrow N_0$;
9  for $j = 0, \ldots,$ FIXED $- 1$ do
10     $c \leftarrow PC[j]$;
11     $\varepsilon \leftarrow$ PERM_SYMB_TABLE[ PS$[j]$ ];
12     for $i = 0, \ldots, N_1 - 1$ do
13         $J[p + i][j] \leftarrow \varepsilon[\, A_1[i][c]\, ]$;
14     assigned$[1][c] \leftarrow$ true;
15 if $v = 2$ then
16     add_column$(1,$ FIXED$)$;
17 else
18     add_column_case2$(2, 0)$;
```

first FIXED columns. The first 7 lines of this function are equal to the first 7 lines of the function with the same name in Algorithm 9. In line 8 the variable p gets the index of the first row of the block F_1, and the *for* loop at lines 9-14 copies the FIXED columns of A_1 whose indices are in PC to the first FIXED columns of the free block F_1; the columns copied are relabeled using the relabeling stored in the corresponding position of the *PS* vector.

If $v = 2$ the function *add_column*() of Algorithm 17 is called with parameters 1 and FIXED, because the next subcolumn to fill in the array J is the subcolumn at index FIXED of block 1. If $v > 2$ the *add_column_case2*() function of Algorithm 22 is called. This function is divided into two parts; the first part is the *if* block from lines 1 to 11, and the second part is the *else* block from lines 12 to 31. The second part is exactly the same as the whole *add_column*() function of Algorithm 17. The first part handles the case $i = 1$, which is the case when the next column to fill r is in the first free block F_1; this part is subdivided into two subcases: (a) $r <$ FIXED and (b) $r \geq$ FIXED. When $r <$ FIXED the action is to advance to the next free block because the column r in the first free block is fixed. If $r \geq$ FIXED then the function tests in column r all columns of A_1 not currently assigned to a column of the block F_1, together with its associated symbol relabelings. Notice that because $v > 2$ the first part of the function does not handle the case $i = v - 1$.

The performance of the two parallel versions are compared against the performance of the sequential version in Section 5.4.

4.5 Chapter summary

This chapter presented a new algorithm called JuxtaposeCA to classify CA$(N; t{+}1, k{+}1, v)$ by testing all possible juxtapositions of v CAs of strength t. The JuxtaposeCA algorithm is able to discard a number of juxtapositions because either they will produce arrays isomorphic to other juxtapositions or because they can not be a CA of strength $t{+}1$. The algorithm generates CAs from all isomorphism classes because all possible ways of constructing CA$(N; t + 1, k + 1, v)$ are explored. However, the CA$(N; t + 1, k + 1, v)$ are not canonical and there may be isomorphic CAs in the results. To obtain only the canonical CA of each isomorphism class, the constructed CA$(N; t{+}1, k{+}1, v)$ are canonized and duplicate elements are removed.

We provided two parallel versions of the JuxtaposeCA algorithm using MPI. The first version is intended for cases where the number of v-tuples of non-isomorphic CAs to be juxtaposed is large; and the second version is intended for cases where the number of v-tuples of non-isomorphic CAs is small but the time required to process each v-tuple of CAs is long.

In the next chapter, the computational results that were obtained with the algorithms developed in this chapter and in Chapter 3 are given.

Algorithm 22: add_column_case2(i, r)

```
if $i = 1$ then
    if $r <$ FIXED then
        add_column_case2($i + 1, r$);
    else
        for $j = 0, \dots, k - 1$ do
            if assigned$[i][j]$ = false then
                assigned$[i][j] \leftarrow$ true;
                foreach permutation $\epsilon$ of the symbols $\{0, 1, \dots, v - 1\}$ do
                    copy column $j$ of $A_i$ to column $r$ of $F_i$ and permute its symbols using $\epsilon$;
                    add_column_case2($i + 1, r$);
                assigned$[i][j] \leftarrow$ false;
else
    for $j = 0, \dots, k - 1$ do
        if assigned$[i][j]$ = false then
            assigned$[i][j] \leftarrow$ true;
            foreach permutation $\epsilon$ of the symbols $\{0, 1, \dots, v - 1\}$ do
                copy column $j$ of $A_i$ to column $r$ of $F_i$ and permute its symbols using $\epsilon$;
                if $i = v - 1$ then
                    if $r < t$ or is_covering_array$(J, r)$ = true then
                        if $r = k - 1$ then
                            $B \leftarrow (JE)$;
                            $B' \leftarrow$ sort_rows($B$);
                            if $B' \notin R$ then
                                $R \leftarrow R \cup \{B'\}$;
                                copy_CA(data, $B'$);
                                MPI_Send(data, $N * (k + 1)$, MPI_CHAR, 0, T1, comm);
                        else
                            add_column_case2($1, r + 1$);
                else
                    add_column_case2($i + 1, r$);
            assigned$[i][j] \leftarrow$ false;
```

5 Computational Results

This chapter presents the main computational results that were obtained by using the two new classification algorithms. The computational results of interest are new CAs classified, new CANs, and improvements in the lower bounds of CANs. Section 5.1 shows the results obtained with the improved NonIsoCA algorithm; and Section 5.2 presents the results obtained with the JuxtaposeCA algorithm. Section 5.3 analyzes when it is more appropriate to use the improved NonIsoCA algorithm or to use the JuxtaposeCA algorithm. Finally, Section 5.4 studies the performance of the parallel implementations of the NonIsoCA algorithm and of the JuxtaposeCA algorithm.

5.1 Results of the improved NonIsoCA algorithm

This section presents the results that were obtained with the improved NonIsoCA algorithm. Subsection 5.1.1 starts by comparing the execution time of the improved sequential algorithm against the execution time of the previous algorithm of [46] for some CAs. Subsection 5.1.2 shows the new results resulting from the executions of the sequential and the parallel NonIsoCA algorithms.

5.1.1 Comparisons with the previous algorithm

We compare the performance of the improved sequential algorithm to construct non-isomorphic CAs against the performance of the original algorithm developed in [46]. The execution times of the original algorithm were taken from tables 2, 3, and 4 of [46]. Both algorithms were implemented in C language, compiled with GCC using the optimization flag -O3, and executed in processors AMD Opteron™ 6274 at 2.2 GHz.

In general, the improved sequential algorithm outperforms the original algorithm, but there are some cases in which the original algorithm performs better. The most remarkable of these cases is $v = t = 2$. Table 5.1 shows a comparison between the execution times of both algorithms for $v = t = 2$ and $2 \leq k \leq 20$. When $v = 2$ and $t = 2$ there are few invalid symbols in the new column being constructed because of the following two reasons: (1) $v = 2$ implies that the probability of passing rules R_1, R_2, and R_3 is high, because there are only two symbols 0 and 1; (2) $t = 2$ implies that there are few combinations of columns involving the current column under construction, and so rules R_4 and R_5 are satisfied most of the times. Therefore, the cost of verifying the five rules for valid symbols is greater than the cost of processing all candidate columns, as the original algorithm does.

However, when $v > 2$ or $t > 2$ the improved sequential algorithm is faster than the original one, as shown in Table 5.2, except in the first two instances of the table. The bigger improvements were for the cases $N = 16$, $13 \leq k \leq 15$, $t = 3$, $v = 2$, where the execution time was reduced from about one thousand hours to approximately one hundred and thirty hours; and for the cases $N = 24$, $7 \leq k \leq 12$, $t = 4$, $v = 2$, where the execution time was improved from more than fifty hours to less than 73 seconds.

It is worth to mention that with regard to the number of non-isomorphic CAs the improved sequential algorithm produced exactly the same results as the original algorithm; and the results of both algorithms are consistent with the results of the algorithms studied in Chapter 2. Also the results for the instances CA$(N; 2, k, 2)$ where $N = 6$ and $6 \leq k \leq 10$ (shown in Table

Table 5.1: Comparison between the original and the improved sequential algorithm when $v = 2$ and $t = 2$. The abbreviations for time units are μs for microseconds, *s* for seconds, and *h* for hours.

CA	# Classes	Original Algorithm	Improved Algorithm
CA$(4; 2, 2, 2)$	1	50.00 μs	0.001 s
CA$(4; 2, 3, 2)$	1	85.00 μs	0.002 s
CA$(5; 2, 4, 2)$	1	197.00 μs	0.003 s
CA$(6; 2, 5, 2)$	7	0.003 s	0.004 s
CA$(6; 2, 6, 2)$	4	0.006 s	0.008 s
CA$(6; 2, 7, 2)$	3	0.009 s	0.013 s
CA$(6; 2, 8, 2)$	1	0.012 s	0.018 s
CA$(6; 2, 9, 2)$	1	0.015 s	0.020 s
CA$(6; 2, 10, 2)$	1	0.025 s	0.034 s
CA$(7; 2, 11, 2)$	26	0.279 s	0.420 s
CA$(7; 2, 12, 2)$	10	0.308 s	0.472 s
CA$(7; 2, 13, 2)$	4	0.329 s	0.500 s
CA$(7; 2, 14, 2)$	1	0.347 s	0.530 s
CA$(7; 2, 15, 2)$	1	0.373 s	0.570 s
CA$(8; 2, 16, 2)$	700 759	0.90 h	2.17 h
CA$(8; 2, 17, 2)$	579 466	1.32 s	4.84 h
CA$(8; 2, 18, 2)$	440 826	1.88 h	7.10 h
CA$(8; 2, 19, 2)$	309 338	2.46 h	9.73 h
CA$(8; 2, 20, 2)$	200 326	3.12 h	12.52 h

5.1), and for the instances CA$(12; 3, k, 2)$ where $6 \leq k \leq 11$ (shown in Table 5.2), match the number of non-equivalent CAs found by Choi *et al.* [7] for the same instances. So, the results of the improved sequential NonIsoCA algorithm are consistent with the already known results. The instances presented in Table 5.2 are the ones with $v > 2$ or $t > 2$ for which the original algorithm was executed due to time constraints.

5.1.2 New classification results and new CANs

The improved sequential algorithm is able to process instances which are larger than the ones processed by the original algorithm. The main objective is to classify CAs whose optimality is known, or to classify CAs to determine if they exist or not. A secondary objective is to classify non-optimal CAs with the purpose of using the obtained non-isomorphic CAs in the JuxtaposeCA

Table 5.2: Comparison between the original and the improved sequential algorithm for $v = 2$ and $t = 3, 4$; and for $v = 3, 4$ and $t = 2$.

CA	# Classes	Original Algorithm	Improved Algorithm
CA$(8; 3, 3, 2)$	1	358.000 μs	0.001 s
CA$(8; 3, 4, 2)$	1	666.000 μs	0.002 s
CA$(10; 3, 5, 2)$	1	0.012 s	0.005 s
CA$(12; 3, 6, 2)$	9	0.337 s	0.062 s
CA$(12; 3, 7, 2)$	2	0.353 s	0.078 s
CA$(12; 3, 8, 2)$	2	0.372 s	0.133 s
CA$(12; 3, 9, 2)$	1	0.436 s	0.226 s
CA$(12; 3, 10, 2)$	1	0.539 s	0.482 s
CA$(12; 3, 11, 2)$	1	0.851 s	1.088 s
CA$(15; 3, 12, 2)$	2	1.260 h	0.443 h
CA$(15; 3, 13, 2)$	0	1.390 h	0.443 h
CA$(16; 3, 13, 2)$	89	937.680 h	129.879 h
CA$(16; 3, 14, 2)$	8	978.420 h	130.512 h
CA$(16; 3, 15, 2)$	0	1 052.650 h	130.573 h
CA$(16; 4, 4, 2)$	1	0.127 s	0.004 s
CA$(16; 4, 5, 2)$	1	0.158 s	0.189 s
CA$(21; 4, 6, 2)$	1	694.600 s	0.346 s
CA$(24; 4, 7, 2)$	1	51.860 h	19.269 s
CA$(24; 4, 8, 2)$	1	52.070 h	19.746 s
CA$(24; 4, 9, 2)$	1	52.880 h	21.890 s
CA$(24; 4, 10, 2)$	1	53.090 h	28.810 s
CA$(24; 4, 11, 2)$	1	53.190 h	39.045 s
CA$(24; 4, 12, 2)$	1	53.390 h	72.846 s
CA$(9; 2, 2, 3)$	1	0.002 s	0.001 s
CA$(9; 2, 3, 3)$	1	0.003 s	0.002 s
CA$(9; 2, 4, 3)$	1	0.006 s	0.011 s
CA$(11; 2, 5, 3)$	3	1.780 s	0.028 s
CA$(12; 2, 6, 3)$	13	240.150 s	1.535 s
CA$(12; 2, 7, 3)$	1	252.380 s	1.520 s
CA$(13; 2, 8, 3)$	5	4.010 h	0.364 h
CA$(13; 2, 9, 3)$	4	4.020 h	0.368 h
CA$(13; 2, 10, 3)$	0	4.140 h	0.368 h
CA$(16; 2, 2, 4)$	1	145.970 s	0.003 s
CA$(16; 2, 3, 4)$	2	215.040 s	0.228 s
CA$(16; 2, 4, 4)$	1	347.520 s	0.699 s
CA$(16; 2, 5, 4)$	1	629.660 s	2.216 s

Table 5.3: Optimal or unknown CAs classified by the improved sequential NonIsoCA algorithm.

CA	# Classes
CA$(14; 2, 10, 3)$	4 490
CA$(14; 2, 11, 3)$	0
CA$(33; 3, 5, 3)$	1
CA$(33; 3, 6, 3)$	1
CA$(19; 2, 6, 4)$	4
CA$(19; 2, 7, 4)$	0
CA$(20; 2, 7, 4)$	0
CA$(37; 2, 4, 6)$	13
CA$(37; 2, 5, 6)$	0
CA$(38; 2, 5, 6)$	0

algorithm. The results that were obtained for the main objective are shown in Table 5.3. Except for the three instances with $v = t = 3$, all these classification results were also obtained independently by Kokkala [25]. Based on the values of v and t the results of Table 5.3 are divided into four subsets:

- $v = 3$ and $t = 2$. In this case there are two instances: $N = 14$ and $k = 10$, and $N = 14$ and $k = 11$. The CA$(14; 2, 10, 3)$ is known to exist [37], but the number of non-isomorphic CAs for this instance was unknown. The improved sequential algorithm found 4,990 non-isomorphic CA$(14; 2, 10, 3)$. For the second instance the number of isomorphism classes was zero, so CA$(14; 2, 11, 3)$ does not exist. The nonexistence of CA$(14; 2, 11, 3)$ and the existence of CA$(15; 2, 20, 3)$ [37] implies CAN$(2, k, 3) = 15$ for $11 \leq k \leq 20$. Finally, the optimality of CA$(15; 2, 13, 3)$ and the existence of CA$(45; 3, 14, 3)$ [9] implies CAN$(3, 14, 3) = 45$ due to the inequality CAN$(t, k, v) \leq \lfloor$CAN$(t + 1, k + 1, v)/v\rfloor$ [11]. All these new CANs are listed in Table 5.4.

- $v = 3$ and $t = 3$. The two CAs of this case are known to be optimal, but the number of isomorphism classes was unknown.

- $v = 4$ and $t = 2$. The CA$(19; 2, 6, 4)$ is known to be optimal [11]. However, for $k = 7$ the current upper bound is 21 [44] and the current lower bound is 19 [11]. The improved

Table 5.4: New covering array numbers obtained with the improved sequential algorithm.

Previous result	New result
$14 \leq \text{CAN}(2, 11, 3) \leq 15$	$\text{CAN}(2, 11, 3) = 15$
$14 \leq \text{CAN}(2, 12, 3) \leq 15$	$\text{CAN}(2, 12, 3) = 15$
$14 \leq \text{CAN}(2, 13, 3) \leq 15$	$\text{CAN}(2, 13, 3) = 15$
$14 \leq \text{CAN}(2, 14, 3) \leq 15$	$\text{CAN}(2, 14, 3) = 15$
$14 \leq \text{CAN}(2, 15, 3) \leq 15$	$\text{CAN}(2, 15, 3) = 15$
$14 \leq \text{CAN}(2, 16, 3) \leq 15$	$\text{CAN}(2, 16, 3) = 15$
$14 \leq \text{CAN}(2, 17, 3) \leq 15$	$\text{CAN}(2, 17, 3) = 15$
$14 \leq \text{CAN}(2, 18, 3) \leq 15$	$\text{CAN}(2, 18, 3) = 15$
$14 \leq \text{CAN}(2, 19, 3) \leq 15$	$\text{CAN}(2, 19, 3) = 15$
$14 \leq \text{CAN}(2, 20, 3) \leq 15$	$\text{CAN}(2, 20, 3) = 15$
$42 \leq \text{CAN}(3, 14, 3) \leq 45$	$\text{CAN}(3, 14, 3) = 45$
$19 \leq \text{CAN}(2, 7, 4) \leq 21$	$\text{CAN}(2, 7, 4) = 21$
$37 \leq \text{CAN}(2, 5, 6) \leq 39$	$\text{CAN}(2, 5, 6) = 39$

sequential algorithm found zero non-isomorphic CAs for $k = 7$ and $N \in \{19, 20\}$; therefore, CAN$(2, 7, 4) = 21$.

- $v = 6$ and $t = 2$. The instance CA$(37; 2, 4, 6)$ is optimal [11]. For $k = 5$ the current upper bound is 39 [37] and the current lower bound is 37 [11]. The result of zero non-isomorphic CAs for $N \in \{37, 38\}$ implies CAN$(2, 5, 6) = 39$.

Table 5.5 shows the non-optimal CAs which were classified using the improved parallel NonIsoCA algorithm. The objectives of classifying these CAs is to use the obtained non-isomorphic CAs in the executions of the JuxtaposeCA algorithm, and to have a record of the number of isomorphism classes to validate future algorithms. The results in the table are used in the next section to obtain the computational results of the JuxtaposeCA algorithm.

5.2 Results of the JuxtaposeCA algorithm

This section presents the main computational results which were obtained with the JuxtaposeCA algorithm. Subsection 5.2.1 describes the process to classify the covering arrays CA$(32; 4, 13, 2)$,

Table 5.5: Non-optimal CAs classified by the improved parallel NonIsoCA algorithm.

CA	# Classes
CA$(17; 3, 12, 2)$	3 238 165 485
CA$(25; 4, 7, 2)$	6
CA$(26; 4, 7, 2)$	228
CA$(27; 4, 7, 2)$	13 012
CA$(28; 4, 7, 2)$	919 874
CA$(29; 4, 7, 2)$	58 488 647
CA$(30; 4, 7, 2)$	3 177 398 378
CA$(25; 4, 8, 2)$	7
CA$(26; 4, 8, 2)$	195
CA$(27; 4, 8, 2)$	9 045
CA$(28; 4, 8, 2)$	522 573
CA$(29; 4, 8, 2)$	27 826 894
CA$(30; 4, 8, 2)$	1 374 716 212

CA$(64; 5, 14, 2)$, CA$(128; 6, 15, 2)$, and CA$(256; 7, 16, 2)$. Subsection 5.2.2 shows the steps to classify CA$(52; 5, 8, 2)$, and Subsection 5.2.3 gives the computations to obtain the non-isomorphic CA$(54; 5, 9, 2)$. Subsection 5.2.4 presents the experimentation done to improve the lower bound of CAN$(6, 9, 2)$ from 96 to 107; and Subsection 5.2.5 shows the computational experimentation to improve the lower bounds of CAN$(3, 7, 3)$, CAN$(3, 9, 3)$, and CAN$(4, 7, 3)$. Subsection 5.2.6 summarizes all new results that were found. Finally, Subsection 5.2.7 performs a consistency check for the JuxtaposeCA algorithm.

The parallel versions of the JuxtaposeCA algorithm are used only in Subsection 5.2.1, in all other subsections the sequential version is used.

In some subsections we use the improved parallel NonIsoCA algorithm to generate the non-isomorphic CAs used by the JuxtaposeCA algorithm. In these subsections we refer the improved parallel NonIsoCA algorithm simply as the NonIsoCA algorithm, so we omit the words “improved” and “parallel”.

5.2.1 Classification of CA$(32;4,13,2)$, CA$(64;5,14,2)$, CA$(128;6,15,2)$, and CA$(256;7,16,2)$

The current lower bound of CAN$(4,13,2)$ is 30 [11], and its current upper bound is 32 [49]. In this section we prove the nonexistence of CA$(30;4,13,2)$ and CA$(31;4,13,2)$, and therefore CAN$(4,13,2) = 32$. In addition we found that there is only one CA$(32;4,13,2)$ up to isomorphisms; so this CA is both optimal and unique.

From Theorem 4.1.1 if CA$(30;4,13,2)$ exists, then there exist two covering arrays CA$(N_0;3,12,2)$ and CA$(N_1;3,12,2)$ such that $N_0 + N_1 = 30$, and their vertical juxtaposition forms a CA of strength four. Now, the only possibility for the values of N_0 and N_1 is $N_0 = N_1 = 15$ because CAN$(3,12,2) = 15$ [11]; so the unique valid multiset in this case is $\{15,15\}$. The NonIsoCA algorithm gives two distinct CA$(15;3,12,2)$, and when using these CAs the sequential JuxtaposeCA algorithm did not find a CA$(30;4,13,2)$.

Similarly, to construct CA$(31;4,13,2)$ the unique valid multiset is $\{15,16\}$. The sequential JuxtaposeCA algorithm tested the juxtapositions of the two non-isomorphic CA$(15;3,12,2)$ with the 44,291 non-isomorphic CA$(16;3,12,2)$ reported by the NonIsoCA algorithm. Also in this case no CA$(31;4,13,2)$ was found. Therefore, CA$(32;4,13,2)$ is optimal, and CAN$(4,13,2) = 32$.

The covering array CA$(32;4,13,2)$ is also optimal in the number of columns. The value CAN$(3,13,2) = 16$ was proved in [46]; so, the only valid multiset to construct a CA$(32;4,14,2)$ is $\{16,16\}$. The NonIsoCA reported 89 distinct CA$(16;3,13,2)$, and when using these CAs the sequential JuxtaposeCA algorithm did not find a CA$(32;4,14,2)$, which implies CAK$(32;4,2) = 13$, where CAK$(N;t,v)$ denotes the maximum value of k for which exists a CA$(N;t,k,v)$.

The new covering array number CAN$(4,13,2) = 32$ has important consequences. In [48] it was reported a Tower of Covering Arrays (TCA) beginning with CA$(8;2,11,2)$ and ending at CA$(256;7,16,2)$. A TCA is a succession of CAs where the first one is CA$(N;t,k,v)$ and the i-th CA ($i > 0$) has Nv^i rows, $k+i$ columns, strength $t+i$, and order v. The complete TCA of [48] is this:

CA$(8; 2, 11, 2)$, CA$(16; 3, 12, 2)$, CA$(32; 4, 13, 2)$, CA$(64; 5, 14, 2)$, CA$(128; 6, 15, 2)$, CA$(256; 7, 16, 2)$.

The first two CAs of the tower are not optimal because CAN$(2, 11, 2) = 7$ and CAN$(3, 12, 2)$ $= 15$. However, from CAN$(4, 13, 2) = 32$ we have CAN(5, 14, 2) $= 64$, CAN$(6, 15, 2) = 128$, and CAN$(7, 16, 2) = 256$, due to the inequality CAN$(t+1, k+1, 2) \geq 2\,$CAN$(t, k, 2)$ [31], which says the optimal CA with $k+1$ columns and strength $t+1$ has at least two times the number of rows of the optimal CA with k columns and strength t. In a TCA with $v = 2$ every CA, other than the first one, has exactly two times the number of rows of the previous CA, and so if the i-th CA is optimal then the j-th CAs, $j > i$, are also optimal.

Now, we construct the non-isomorphic CA$(32; 4, 13, 2)$. In this case there are two valid multisets $\{15, 17\}$ and $\{16, 16\}$. For $\{15, 17\}$ we juxtaposed the 2 non-isomorphic CA$(15; 3, 12, 2)$ with the 3,238,165,485 non-isomorphic CA$(17; 3, 12, 2)$ reported by the NonIsoCA algorithm and none CA$(32; 4, 13, 2)$ was constructed. For $\{16, 16\}$ the juxtaposition of the 44,291 non-isomorphic CA$(16; 3, 12, 2)$ with themselves produced only one distinct CA$(32; 4, 13, 2)$. Then, all CA$(32; 4, 13, 2)$ are isomorphic among them. In this case we used the first parallel version of the JuxtaposeCA algorithm (Subsection 4.4.1) since the number of non-isomorphic CAs CA$(N_0; 3, 12, 2)$ and CA$(N_1; 3, 12, 2)$ that were juxtaposed is large.

Taking advantage of the uniqueness of CA$(32; 4, 13, 2)$ we executed the second parallel version of the JuxtaposeCA algorithm (Subsection 4.4.2) to find the number of distinct CA$(64; 5, 14, 2)$, and the result was only one non-isomorphic CA$(64; 5, 14, 2)$. In a similar way, the second parallel version of the JuxtaposeCA algorithm found that CA$(128; 6, 15, 2)$ and CA$(256; 7, 16, 2)$ are also unique.

In Section 2.4 we mentioned the construction of a $(13, 64, 5)$ code by juxtaposing two Nadler codes $(12, 32, 5)$ done by Nordstrom and Robinson [36]. After that, they repeated the process and juxtaposed two $(13, 64, 5)$ codes to construct a $(14, 128, 5)$ code, and finally they took two copies of this last code to form a $(15, 256, 5)$ code. The extended versions of these four codes are the $(13, 32, 6)$, $(14, 64, 6)$, $(15, 128, 6)$, $(16, 256, 6)$ codes. These four codes are optimal and unique; their optimality was proven in [3], the uniqueness of the first three codes was proven in [14], and the uniqueness of the last one was shown in [43]. We found that these four codes are equivalent respectively to the covering arrays CA$(32; 4, 13, 2)$, CA$(64; 5, 14, 2)$, CA$(128; 6, 15, 2)$, CA$(256; 7, 16, 2)$. In each case we validate that the code is a CA of the alleged strength, and that the rows of the CA have minimum mutual distance 6.

5.2.2 Classification of CA$(52; 5, 8, 2)$

CAN$(5, 8, 2)$ is the first element of the class CAN$(t, t+3, 2)$ whose exact value is unknown; its current status is $48 \leq$ CAN$(5, 8, 2) \leq 52$ [11, 49]. To find CAN$(5, 8, 2)$ we need to check the juxtapositions of the non-isomorphic CA$(N_0; 4, 7, 2)$ with the non-isomorphic CA$(N_1; 4, 7, 2)$ for $N_0 + N_1 \in \{48, 49, 50, 51, 52\}$. Since CAN$(4, 7, 2) = 24$ we have $N_0, N_1 \geq 24$. The first step is to search a CA with 48 rows, if it does not exist the next step is to search a CA with 49 rows, and so on.

As shown in Subtable 5.6(a) there is only one non-isomorphic CA$(24; 4, 7, 2)$. Subtable 5.6(b) shows that no CA$(48; 5, 8, 2)$ was constructed from the juxtaposition of the unique CA$(24; 4, 7, 2)$ with itself. This result is consistent with the demonstration of the nonexistence of CA$(48; 5, 13, 2)$ done in [7].

Now, to search if CA$(49; 5, 8, 2)$ exists, we need to juxtapose the non-isomorphic CA$(24; 4, 7, 2)$ with the non-isomorphic CA$(25; 4, 7, 2)$. There is only one CA$(24; 4, 7, 2)$, and for CA$(25; 4, 7, 2)$ the NonIsoCA algorithm reported 6 distinct CAs. When using these CAs the sequential JuxtaposeCA

Table 5.6: Computations to find the value of CAN$(5, 8, 2)$. (a) Number of non-isomorphic CA$(M; 4, 7, 2)$ for $M = 24, 25, 26, 27, 28$. (b) Number of non-isomorphic CA$(N; 5, 8, 2)$ constructed by juxtaposing CA$(N_0; 4, 7, 2)$ and CA$(N_1; 4, 7, 2)$, where $N = N_0 + N_1$ and $48 \leq N \leq 52$.

(a) Non-iso CA$(M; 4, 7, 2)$

M	# Classes
24	1
25	6
26	228
27	13 012
28	919 874

(b) Non-iso CA$(N; 5, 8, 2)$

N	Multisets $\{N_0, N_1\}$	# Classes
48	$\{24, 24\}$	0
49	$\{24, 25\}$	0
50	$\{24, 26\}, \{25, 25\}$	0
51	$\{24, 27\}, \{25, 26\}$	0
52	$\{24, 28\}, \{25, 27\}, \{26, 26\}$	8

algorithm did not find a CA$(49;$ 5, 8, $2)$. The same strategy is repeated to determine the existence of CA$(50;5,8,2)$, CA$(51;5,8,2)$, and CA$(52;5,8,2)$. From the results in Subtable 5.6(b) we have CAN$(5,8,2)=52$, and there are eight non-isomorphic CA$(52;5,8,2)$.

5.2.3 Classification of CA$(54;5,9,2)$

For CAN$(5,9,2)$ the current lower bound is 52 (Subsection 5.2.2) and the current upper bound is 54 [49]. Then, to determine the exact value of CAN$(5,9,2)$ we need to check if there is a CA with 52 or 53 rows. Subtable 5.7(a) shows the number of non-isomorphic CA$(M;4,8,2)$ that were generated by the NonIsoCA algorithm for $M=24,\dots,30$. These CAs are used to search for the non-isomorphic CA$(N;5,9,2)$ with $N=52,53,54$. Subtable 5.7(b) shows the valid multisets $\{N_0,N_1\}$ and the number of non-isomorphic CAs constructed for each $N\in\{52,53,54\}$. From the results we have CAN$(5,9,2)=54$, and there is only one distinct CA$(54;5,9,2)$, which is shown in Figure 5.1 (transposed).

The execution of the ExtendNonIsoCA algorithm of Subsection 3.1.4 with the unique CA$(54;5,9,2)$ produced zero CA$(54;5,10,2)$; then, CAK$(54;5,2)=9$. This result improves from 52 to 55 the lower bounds of CAN$(5,10,2)$, CAN$(5,11,2)$, CAN$(5,12,2)$, and CAN$(5,13,2)$, taken from [11].

Table 5.7: Computations to find the value of CAN$(5,9,2)$. (a) Number of non-isomorphic CA$(M;4,8,2)$ for $M=24,\dots,30$. (b) Number of non-isomorphic CA$(N;5,9,2)$ constructed by juxtaposing CA$(N_0;4,8,2)$ and CA$(N_1;4,8,2)$, where $N=N_0+N_1$ and $52\le N\le 54$.

(a) Non-iso CA$(M;4,8,2)$

M	# Classes
24	1
25	7
26	195
27	9 045
28	522 573
29	27 826 894
30	1 374 716 212

(b) Non-iso CA$(N;5,9,2)$

N	Multisets $\{N_0,N_1\}$	# Classes
52	$\{24,28\},\{25,27\},\{26,26\}$	0
53	$\{24,29\},\{25,28\},\{26,27\}$	0
54	$\{24,30\},\{25,29\},\{26,28\},\{27,27\}$	1

$$\begin{pmatrix}
000000000000000000000000000111111111111111111111111111 \\
000000000000000111111111111000000000000111111111111111 \\
000000000111111000000111111000000111111000000111111111 \\
000000111000111000111000111000111000111000111000111111 \\
000001011011001011001001011011001001011001011011001111 \\
000110001001011001011110001001110011001011001001110111 \\
001010001010110011100010011010011101001010101101011011 \\
010010010010101001110101001011010010101100011101011101 \\
011101100110011101101100010101011001100010010110100001
\end{pmatrix}$$

Figure 5.1: The canonical representative of the unique isomorphism class of CA$(54; 5, 9, 2)$.

5.2.4 Improving the lower bound of CAN$(6, 9, 2)$

The next CAN of the class CAN$(t, t + 3, 2)$ to be determined is CAN$(6, 9, 2)$. Its current status is $96 \leq$ CAN$(6, 9, 2) \leq 108$ [11, 9]. However, from CAN$(5, 8, 2) = 52$ (Subsection 5.2.2) and from the inequality CAN$(t + 1, k + 1, 2) \geq 2\,CAN(t, k, 2)$ we have CAN$(6, 9, 2) \geq 104$. Therefore, the new lower bound of CAN$(6, 9, 2)$ is 104, but we can further improve this lower bound by using the JuxtaposeCA algorithm.

Firstly, the juxtapositions of the 8 non-isomorphic CA$(52; 5, 8, 2)$ that were found in Subsection 5.2.2 with themselves do not produce a CA$(104; 6, 9, 2)$; therefore CAN$(6, 9, 2) \geq 105$.

To determine the existence of CA$(105; 6, 9, 2)$ we need to test the juxtaposition of the non-isomorphic CA$(52; 5, 8, 2)$ with the non-isomorphic CA$(53; 5, 8, 2)$. But to obtain the non-isomorphic CA$(53; 5, 8, 2)$ we need to juxtapose CA$(N_0; 4, 7, 2)$ and CA$(N_1; 4, 7, 2)$ where $N_0 + N_1 = 53$. Previously, in Subsection 5.2.2, the non-isomorphic CA$(M; 4, 7, 2)$ for $M = 24, \ldots, 28$ were listed; and in addition the NonIsoCA algorithm reported 58,488,647 distinct CA$(29; 4, 7, 2)$ and 3,177,398,378 distinct CA$(30; 4, 7, 2)$, as shown in Subtable 5.8(a). Subtable 5.8(b) shows the multisets for $N = 53$ and the number of non-isomorphic CA$(53; 5, 8, 2)$ constructed by the sequential JuxtaposeCA algorithm; in this case there are 213 distinct CA$(53; 5, 8, 2)$. Subtable 5.8(c) shows the result of juxtaposing the non-isomorphic CA$(52; 5, 8, 2)$ with the non-isomorphic CA$(53; 5, 8, 2)$

Table 5.8: Computations to improve the lower bound of CAN$(6,9,2)$. (a) Number of non-isomorphic CA$(M;4,7,2)$ for $M = 29,30$. (b) Number of non-isomorphic CA$(N;5,8,2)$ constructed by juxtaposing CA$(N_0;4,7,2)$ and CA$(N_1;4,7,2)$, where $N = N_0+N_1$ and $53 \le N \le 54$. (c) Number of non-isomorphic CA$(L;6,9,2)$ constructed by juxtaposing CA$(L_0;5,8,2)$ and CA$(L_1;5,8,2)$, with $L = L_0 + L_1$ and $104 \le L \le 106$.

(a) Non-iso CA$(M;4,7,2)$

M	# Classes
29	58 488 647
30	3 177 398 378

(b) Non-iso CA$(N;5,8,2)$

N	Multisets $\{N_0,N_1\}$	# Classes
53	$\{24,29\},\{25,28\},\{26,27\}$	213
54	$\{24,30\},\{25,29\},\{26,28\},\{27,27\}$	20 450

(c) Non-iso CA$(L;6,9,2)$

L	Multisets $\{L_0,L_1\}$	# Classes
104	$\{52,52\}$	0
105	$\{52,53\}$	0
106	$\{52,54\},\{53,53\}$	0

to try to construct CA$(105;6,9,2)$. No CA$(105;6,9,2)$ was generated, then CAN$(6,9,2) \ge 106$.

Note that we are using the non-isomorphic CAs that were generated by the JuxtaposeCA algorithm in another execution of it, because from the non-isomorphic CAs with $t = 4$ and $k = 7$ the non-isomorphic CAs with $t = 5$ and $k = 8$ are constructed, and these last CAs are used to search for the non-isomorphic CAs with $t = 6$ and $k = 9$.

Now, to determine if CA$(106;6,9,2)$ exists we first compute the valid multisets $\{L_0,L_1\}$ such that the juxtaposition of CA$(L_0;5,8,2)$ and CA$(L_1;5,8,2)$ might produce CA$(106;6,9,2)$. In this case there are two possibilities: $\{52,54\}$ and $\{53,53\}$. The non-isomorphic CA$(52;5,8,2)$ and CA$(53;5,8,2)$ have been constructed previously, but it remains to construct the distinct CA$(54;5,8,2)$. To do this, we juxtapose the non-isomorphic CA$(N_0;4,7,2)$ with the non-isomorphic CA$(N_1;4,7,2)$ where $N_0 + N_1 = 54$. Subtable 5.8(a) shows that there are 3,177,398,378 distinct CA$(30;4,7,2)$. Subtable 5.8(b) shows the results of juxtaposing CA$(N_0;4,7,2)$ and CA$(N_1;4,7,2)$ where $N_0 + N_1 = 54$; in total there are 20,450 distinct CA$(54;5,8,2)$. Subtable 5.8(c) contains the result of juxtaposing the distinct CA$(52;5,8,2)$ with the distinct CA$(54;5,8,2)$, and the distinct CA$(53;5,8,2)$ with themselves. No CA$(106;6,9,2)$ was generated, thus CAN$(6,9,2) \ge 107$.

It was not possible to determine the existence of CA$(107;6,9,2)$ due to the huge computational time required to construct the non-isomorphic CA$(55; 5, 8, 2)$. However, the result CAN$(6, 9, 2) \geq 107$ improves the lower bounds of CAN$(t, t + 3, 2)$ for $7 \leq t \leq 11$; their old and new values are shown next, old values were taken from [11]:

CAN(t, k, v)	Previous value	New value
CAN$(7, 10, 2)$	192	214
CAN$(8, 11, 2)$	385	428
CAN$(9, 12, 2)$	770	856
CAN$(10, 13, 2)$	1540	1712
CAN$(11, 14, 2)$	3080	3424

5.2.5 Results for $v = 3$

This section presents the computational results that were obtained for CAs with order $v = 3$. The results are given in a list format. Lower and upper bounds were taken respectively from [11] and [9]:

- *There is a unique CA*$(33; 3, 6, 3)$. This CA is known to be optimal [6], but we prove its uniqueness. Since CAN$(2, 5, 3) = 11$, the only valid multiset to construct CA$(33; 3, 6, 3)$ is $\{11, 11, 11\}$. The NonIsoCA algorithm reported 3 non-isomorphic CA$(11; 2, 5, 3)$, and when using these CAs the sequential JuxtaposeCA algorithm constructed only one CA$(33; 3, 6, 3)$, which is shown next (transposed):

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

- *Nonexistence of CA*$(99; 4, 7, 3)$. The current status of CAN$(4, 7, 3)$ is $99 \leq$ CAN$(4, 7, 3) \leq 123$. Using the unique CA$(33; 3, 6, 3)$ the sequential JuxtaposeCA algorithm determined the nonexistence of CA$(99; 4, 7, 3)$. Therefore, $100 \leq$ CAN$(4, 7, 3) \leq 123$.

- *Nonexistence of CA*$(36;3,7,3)$. Currently $36 \leq$ CAN$(3,7,3) \leq 39$. Since CAN$(2,6,3) = 12$, the only way to form a CA$(36;3,7,3)$ is by juxtaposing three CA$(12;2,6,3)$. There are 13 non-isomorphic CA$(12;2,6,3)$ [11], and by using them the sequential JuxtaposeCA algorithm did not find a CA$(36;3,7,3)$. Thus, $37 \leq$ CAN$(3,7,3) \leq 39$.

- *Nonexistence of CA*$(39;3,9,3)$. The current lower bound of CAN$(3,9,3)$ is 39 and its current upper bound is 45. Because CAN$(2,8,3) = 13$ the only possibility to form a CA$(39;3,9,3)$ is to juxtapose three CA$(13;2,8,3)$. The number of non-isomorphic CA$(13;2,8,3)$ is five [46], and by using these CAs the sequential JuxtaposeCA algorithm searched for CA$(39;3,9,3)$ but no such CA was found. Therefore, $40 \leq$ CAN$(3,9,3) \leq 45$.

5.2.6 Summary of the new results

Now we present a summary of the results achieved by the JuxtaposeCA algorithm. Among the results there are new classification results, new CANs, and improvements in the lower bounds of some CANs. The new classification results are listed in Table 5.9, and the new CANs and the improved lower bounds of CAN are given in Table 5.10.

The first two columns of Table 5.9 show the CAs that were classified and the number of isomorphism classes; the columns with headers A_0 and A_1 contain the CAs which were juxtaposed to try to construct the CA in the first column of the table; finally, the fourth and the sixth column contains respectively the number of non-isomorphic CAs that exist for the CAs in the columns A_0 and A_1.

Other classification results are the classification of the non-optimal CAs of Subtable 5.8(b). These CAs are CA$(53;5,8,2)$ for which there are 213 classes, and CA$(54;5,8,2)$ for which there are 20,450 classes.

Table 5.9: New classification results obtained with the JuxtaposeCA algorithm.

CA classified	# Classes	A_0	# Classes	A_1	# Classes
CA$(30;4,13,2)$	0	CA$(15;3,12,2)$	2	CA$(15;3,12,2)$	2
CA$(31;4,13,2)$	0	CA$(15;3,12,2)$	2	CA$(16;3,12,2)$	44 291
CA$(32;4,13,2)$	1	CA$(15;3,12,2)$	2	CA$(17;3,12,2)$	3 238 165 485
		CA$(16;3,12,2)$	44 291	CA$(16;3,12,2)$	44 291
CA$(64;5,14,2)$	1	CA$(32;4,13,2)$	1	CA$(32;4,13,2)$	1
CA$(128;6,15,2)$	1	CA$(64;5,14,2)$	1	CA$(64;5,14,2)$	1
CA$(256;7,16,2)$	1	CA$(128;6,15,2)$	1	CA$(128;6,15,2)$	1
CA$(48;5,8,2)$	0	CA$(24;4,7,2)$	1	CA$(24;4,7,2)$	1
CA$(49;5,8,2)$	0	CA$(24;4,7,2)$	1	CA$(25;4,7,2)$	6
CA$(50;5,8,2)$	0	CA$(24;4,7,2)$	1	CA$(26;4,7,2)$	228
		CA$(25;4,7,2)$	6	CA$(25;4,7,2)$	6
CA$(51;5,8,2)$	0	CA$(24;4,7,2)$	1	CA$(27;4,7,2)$	13 012
		CA$(25;4,7,2)$	6	CA$(26;4,7,2)$	228
CA$(52;5,8,2)$	8	CA$(24;4,7,2)$	1	CA$(28;4,7,2)$	919 874
		CA$(25;4,7,2)$	6	CA$(27;4,7,2)$	13 012
		CA$(26;4,7,2)$	228	CA$(26;4,7,2)$	228
CA$(52;5,9,2)$	0	CA$(24;4,8,2)$	1	CA$(28;4,8,2)$	522 573
		CA$(25;4,8,2)$	7	CA$(27;4,8,2)$	9 045
		CA$(26;4,8,2)$	195	CA$(26;4,8,2)$	195
CA$(53;5,9,2)$	0	CA$(24;4,8,2)$	1	CA$(29;4,8,2)$	27 826 894
		CA$(25;4,8,2)$	7	CA$(28;4,8,2)$	522 573
		CA$(26;4,8,2)$	195	CA$(27;4,8,2)$	9 045
CA$(54;5,9,2)$	1	CA$(24;4,8,2)$	1	CA$(30;4,8,2)$	1 374 716 212
		CA$(25;4,8,2)$	7	CA$(29;4,8,2)$	27 826 894
		CA$(26;4,8,2)$	195	CA$(28;4,8,2)$	522 573
		CA$(27;4,8,2)$	9 045	CA$(27;4,8,2)$	9 045

Table 5.10: Summary of the new CANs and of the improved lower bounds of CAN. (a) The new covering array numbers. (b) The improved lower bounds of CAN.

(a)

CAN$(t,k,v) = N$
CAN$(4,13,2) = 32$
CAN$(5,14,2) = 64$
CAN$(6,15,2) = 128$
CAN$(7,16,2) = 256$
CAN$(5,8,2) = 52$
CAN$(5,9,2) = 54$

(b)

CAN$(t,k,v) \geq LB$
CAN$(5,10,2) \geq 55$
CAN$(5,11,2) \geq 55$
CAN$(5,12,2) \geq 55$
CAN$(5,13,2) \geq 55$
CAN$(6,9,2) \geq 107$
CAN$(7,10,2) \geq 214$
CAN$(8,11,2) \geq 428$
CAN$(9,12,2) \geq 856$
CAN$(10,13,2) \geq 1712$
CAN$(11,14,2) \geq 3424$
CAN$(4,7,3) \geq 100$
CAN$(3,7,3) \geq 37$
CAN$(3,9,3) \geq 40$

5.2.7 Consistency check

As a way to doubly validate the results of the JuxtaposeCA algorithm, we constructed the non-isomorphic CAs for some known optimal cases of order $v = 2$ and strengths $t = 3, 4$. The idea is to compare the results of the JuxtaposeCA algorithm with the results of the NonIsoCA algorithm of Chapter 3 to see if they are equal. For both strengths we constructed the non-isomorphic CAs up to $k = 12$ columns. The obtained results are shown in Table 5.11; A_0 and A_1 are the two CAs which were juxtaposed to construct the CA in the first column. We can verify that the results in the first two columns of the table match the results of the NonIsoCA algorithm given in Chapter 3.

5.3 When to use NonIsoCA or JuxtaposeCA

In the computational results shown in the previous sections we can see that the JuxtaposeCA algorithm was used to classify the larger instances. The largest instance which was processed by the JuxtaposeCA algorithm is CA$(256; 7, 16, 2)$, while the larger instances processed by the improved

Table 5.11: Construction of known results using the JuxtaposeCA algorithm.

Constructed CA	# Classes	A_0	# Classes	A_1	# Classes
CA$(8;3,3,2)$	1	CA$(4;2,2,2)$	1	CA$(4;2,2,2)$	1
CA$(8;3,4,2)$	1	CA$(4;2,3,2)$	1	CA$(4;2,3,2)$	1
CA$(10;3,5,2)$	1	CA$(5;2,4,2)$	1	CA$(5;2,4,2)$	1
CA$(12;3,6,2)$	9	CA$(6;2,5,2)$	7	CA$(6;2,5,2)$	7
CA$(12;3,7,2)$	2	CA$(6;2,6,2)$	4	CA$(6;2,6,2)$	4
CA$(12;3,8,2)$	2	CA$(6;2,7,2)$	3	CA$(6;2,7,2)$	3
CA$(12;3,9,2)$	1	CA$(6;2,8,2)$	1	CA$(6;2,8,2)$	1
CA$(12;3,10,2)$	1	CA$(6;2,9,2)$	1	CA$(6;2,9,2)$	1
CA$(12;3,11,2)$	1	CA$(6;2,10,2)$	1	CA$(6;2,10,2)$	1
CA$(15;3,12,2)$	2	CA$(7;2,11,2)$	26	CA$(8;2,11,2)$	377 177
CA$(16;4,4,2)$	1	CA$(8;3,3,2)$	1	CA$(8;3,3,2)$	1
CA$(16;4,5,2)$	1	CA$(8;3,4,2)$	1	CA$(8;3,4,2)$	1
CA$(21;4,6,2)$	1	CA$(10;3,5,2)$	1	CA$(11;3,5,2)$	4
CA$(24;4,7,2)$	1	CA$(12;3,6,2)$	9	CA$(12;3,6,2)$	9
CA$(24;4,8,2)$	1	CA$(12;3,7,2)$	2	CA$(12;3,7,2)$	2
CA$(24;4,9,2)$	1	CA$(12;3,8,2)$	2	CA$(12;3,8,2)$	2
CA$(24;4,10,2)$	1	CA$(12;3,9,2)$	1	CA$(12;3,9,2)$	1
CA$(24;4,11,2)$	1	CA$(12;3,10,2)$	1	CA$(12;3,10,2)$	1
CA$(24;4,12,2)$	1	CA$(12;3,11,2)$	1	CA$(12;3,11,2)$	1

NonIsoCA algorithm were CA$(30;4,8,2)$ and CA$(38;2,5,6)$; the difference in size is clear. The advantage of the JuxtaposeCA algorithm over the NonIsoCA algorithm is due to the first algorithm works with subcolumns of non-isomorphic CAs of a smaller strength, while the second algorithm constructs the CAs basically cell by cell. However, many of the new results of the JuxtaposeCA algorithm were obtained by using the non-isomorphic CAs found by the NonIsoCA algorithm.

In this section we compare the execution times of the sequential versions of the NonIsoCA algorithm and the JuxtaposeCA algorithm. Firstly, we describe in detail the execution times of both algorithms for the instances CA$(30;4,13,2)$, CA$(31;4,13,2)$, and CA$(16;3,12,2)$, and then we employ the formulas obtained in the complexity analysis of the algorithms to compute the approximate number of operations done by them for the instances CA$(31;4,13,2)$ and CA$(16;3,12,2)$.

The instance CA$(30;4,13,2)$ classified in Subsection 5.2.1 required the juxtaposition of the two CA$(15;3,12,2)$ with themselves. The improved NonIsoCA algorithm found these two CAs in 0.44

hours; but the JuxtaposeCA algorithm required only 3 seconds to process the four tuples $T = (A_0, A_1)$ where both A_0 and A_1 are any of the two non-isomorphic CA$(15; 3, 12, 2)$. Thus, the total time to determine the nonexistence of CA$(30; 4, 13, 2)$ was about 0.44 hours. On the other hand, we attempted to classify CA$(30; 4, 13, 2)$ using the NonIsoCA algorithm, but we aborted the search after 20 days because based on the partial results we estimated that the execution would not end soon.

Similarly for the instance CA$(31; 4, 13, 2)$ we could not execute the NonIsoCA algorithm due to time constraints; so the solution was to use the JuxtaposeCA algorithm. To classify this instance the JuxtaposeCA algorithm juxtaposed the two distinct CA$(15; 3, 12, 2)$ with the 44,291 non-isomorphic CA$(16; 3, 12, 2)$. The time required to process the $2 \cdot 44{,}291 = 88{,}582$ tuples $T = (A_0, A_1)$, where A_0 is a non-isomorphic CA$(15; 3, 12, 2)$ and A_1 is a non-isomorphic CA$(16; 3, 12, 2)$, was about 16 hours. Now, the time required by the NonIsoCA algorithm to construct the non-isomorphic CA$(16; 3, 12, 2)$ was approximately 128 hours; so the total time to classify CA$(31; 4, 13, 2)$ was $0.44 + 128 + 16 = 144.44$ hours. We do not attempt to execute the NonIsoCA algorithm to classify CA$(31; 4, 13, 2)$ because it will take too much time.

In general the execution time of the NonIsoCA algorithm for an instance CA$(N; t, k, v)$ is much larger than the execution time for the instance with one less row CA$(N-1; t, k, v)$; an example of this situation is given by the times required to classify CA$(15; 3, 12, 2)$ and CA$(16; 3, 12, 2)$. In addition the number of non-isomorphic CAs increases considerably from CA$(N-1; t, k, v)$ to CA$(N; t, k, v)$; examples of this fact are the classification results given in Table 5.5.

In the process of classifying CA$(31; 4, 13, 2)$ almost all execution time was consumed in constructing the 44,291 non-isomorphic CA$(16; 3, 12, 2)$. We could use the JuxtaposeCA algorithm to construct these CAs; however, in this case the JuxtaposeCA algorithm is not the best option because there are too many non-isomorphic CAs with strength $t = 2$ and $k = 11$ columns to be juxtaposed. Since CAN$(2, 11, 2) = 7$, we can construct a CA$(16; 3, 12, 2)$ by juxtaposing a CA$(7; 2, 11, 2)$ and a CA$(9; 2, 11, 2)$, or by juxtaposing two CA$(8; 2, 11, 2)$. The number of non-isomorphic CA$(7; 2, 11, 2)$ is only 26, but there are 377,177 non-isomorphic CA$(8; 2, 11, 2)$, and $2{,}148{,}812{,}219$ non-isomorphic CA$(9; 2, 11, 2)$. Thus, the number of tuples $T = (A_0, A_1)$ to be processed by the JuxtaposeCA algorithm is $(26)(2{,}148{,}812{,}219) + 377{,}177^2$.

We make an estimation of the time to process the $377{,}177^2$ tuples $T = (A_0, A_1)$ where both A_0 and A_1 are a non-isomorphic CA$(8; 2, 11, 2)$ as follows: we take the first 1,000 distinct CA$(8; 2, 11, 2)$ and call the *generate_juxtapositions*(T) function of Algorithm 8 $1{,}000^2$ times to process the tuples $T = (A_0, A_1)$ obtained from these 1,000 non-isomorphic CAs. The execution time for the $1{,}000^2$ tuples was 860 seconds; then, an approximation of the time needed to juxtapose the 377,177 non-isomorphic CA$(8; 2, 11, 2)$ with themselves is $(377{,}177^2/1{,}000^2) \cdot 860 = 142{,}262.48 \cdot 860 = 122{,}345{,}740.82$ seconds, which is a little more than 1,416 days.

In general, if the number of tuples $T = (A_0, A_1, \ldots, A_{v-1})$ to be processed is very large, then the JuxtaposeCA algorithm would not be the best option to classify CA$(N; t, k, v)$, and we should use the NonIsoCA algorithm instead. On the other hand, if the number of tuples T is small, then with high probability the JuxtaposeCA algorithm will perform better than the NonIsoCA algorithm. For example, the classification of CA$(64; 5, 14, 2)$ was possible due to the fact that the number of tuples T to be juxtaposed is just 1; in this case the only way to construct CA$(64; 5, 14, 2)$ is by juxtaposing two CA$(32; 4, 13, 2)$, and there is a unique CA$(32; 4, 13, 2)$. The JuxtaposeCA algorithm took only 189 seconds to classify CA$(64; 5, 14, 2)$, of which 53 seconds were consumed by the *generate_juxtapositions*() function, and 136 seconds were required to canonize the three CAs generated by the above function. For this instance the NonIsoCA algorithm would take an impractical amount of time.

We can also have an idea of the execution times of the algorithms by using the formulas of their computational complexity. From Section 3.2 the computational cost of the improved NonIsoCA algorithm for CA$(N; t, k, v)$ is:

$$O\left(\frac{\beta^{k-1}}{\prod_{i=1}^{k-2} i!(v!)^i} \cdot \binom{k-1}{t-1} Nt \cdot (N \log_2 N) k! (v!)^k\right), \text{ where } \beta = \left[\frac{\Gamma(N/v+1)}{[\Gamma(N/v^2+1)]^v}\right]^v.$$

And from Section 4.3 the computational cost of the JuxtaposeCA algorithm for CA$(N; t+1, k+1, v)$ is:

$$O\Big(\sum_{j=0}^{p(N-v\,\mathsf{CAN}(t,k,v),\,v)-1} |D_{0_j}\times\cdots\times D_{v-1_j}|\cdot[k!(v!)^k]^{v-1}\cdot\binom{k}{t+1}N(t+1)\cdot(N\log_2 N)(k+1)!(v!)^{k+1}\Big).$$

For the instance $\mathsf{CA}(31;4,13,2)$ the estimated number of operations performed by the algorithms is computed as follows:

- NonIsoCA:

$$\begin{aligned}
\beta &= \left[\frac{\Gamma(N/v+1)}{[\Gamma(N/v^2+1)]^v}\right]^v = \left[\frac{\Gamma(16.5)}{[\Gamma(8.75)]^2}\right]^2 = (9241.43)^2 = 85404049.13\\
\frac{\beta^{k-1}}{\prod_{i=1}^{k-2} i!(v!)^i} &= \frac{(85404049.13)^{12}}{\prod_{i=1}^{11} i!(v!)^i} = \frac{1.50571\times10^{95}}{1.96119\times10^{55}} = 7.67755\times10^{39}\\
\binom{k-1}{t-1}Nt &= \binom{12}{3}(31)(4) = 27280\\
(N\log_2 N)\,k!\,(v!)^k &= (31\log_2 31)\,13!\,(2!)^{13} = 7.83439\times10^{15}
\end{aligned}$$

  The product of the last three results is 1.64086×10^{60}.

- JuxtaposeCA: In this case $k=12$ and $t=3$ in the following formulas:

$$\begin{aligned}
\sum_{j=0}^{p(N-v\,\mathsf{CAN}(t,k,v),\,v)-1} |D_{0_j}\times\cdots\times D_{v-1_j}| &= 2(44291) = 88582\\
[k!(v!)^k]^{v-1} &= [12!(2!)^{12}]^1 = 12!(2^{12}) = 1.96199\times10^{12}\\
\binom{k}{t+1}N(t+1) &= \binom{12}{4}(31)(4) = 61380\\
(N\log_2 N)\,(k+1)!\,(v!)^{k+1} &= (31\log_2 31)\,13!\,(2!)^{13} = 7.83439\times10^{15}
\end{aligned}$$

  The product of the four results is 8.35746×10^{37}.

Then, the estimated number of operations for the NonIsoCA algorithm is greater than the estimated number of operations for the JuxtaposeCA algorithm. Of course, the complexity analysis is approximate and based on certain simplifications which do not take into account the pruning criteria to avoid the generation of all possible columns and the testing of all possible column permutations and symbol relabelings in the case of the NonIsoCA algorithm, and the pruning criteria to not generate all arrays J in the JuxtaposeCA algorithm. For this particular instance, CA$(31;4,13,2)$, the estimated number of operations for the algorithms is congruent with the computational experimentation, because the JuxtaposeCA algorithm is faster than the NonIsoCA algorithm in this case.

Now we repeat the exercise for the instance CA$(16;3,12,2)$. In the computational experimentation the NonIsoCA algorithm was faster than the JuxtaposeCA algorithm. The results using the formulas for the complexity of the algorithms are:

- For the NonIsoCA algorithm the estimated number of operations is 5.40262×10^{13}.
- For the JuxtaposeCA algorithm the estimated number of operations is 1.6108×10^{40}.

Also in this case the theoretical analysis matches the result of the computational experimentation with regard to which algorithm performs less operations. Thus, we can use the formulas derived from our complexity analysis to estimate approximately which algorithm is more appropriate for a particular instance of the classification problem.

5.4 Performance of the parallel implementations

This section studies the performance of the parallel implementations of the NonIsoCA algorithm and of the JuxtaposeCA algorithm. Subsection 5.4.1 analyzes the performance of the parallel version of the NonIsoCA algorithm developed in Section 3.3; and Subsection 5.4.2 analyzes the performance of the two parallel implementations of the JuxtaposeCA algorithm developed in Section 4.4.

5.4.1 Parallel version of the improved NonIsoCA algorithm

In the parallel version of the NonIsoCA algorithm a new execution flow can be created when a slave process constructs a canonical CA with $r < k$ columns. The new execution flow is created only if there is an available slave. In this case the slave that creates the CA sends it to the available slave, and the next two actions are executed in parallel: (a) the slave that constructed the canonical CA with $r < k$ columns searches the next canonical CA with r columns, and (b) the available slave searches canonical CAs with $r+1$ columns based on the received CA with r columns.

Depending on the covering array CA$(N; t, k, v)$ being classified, the number of requests for an available slave may be very large. In these situations the master process becomes a bottleneck because the slaves are constantly asking for an available slave. When a slave process requests an available slave, the first slave remains blocked until the master responds to it. The total number of requests sent to the master by all slaves is the number of non-isomorphic CAs with $r = 2, 3, \ldots, k-1$ columns, because a request is sent every time a canonical CA with $2 \leq r \leq k-1$ columns is found.

To evaluate the performance of the parallel implementation of the NonIsoCA algorithm we execute it with two instances: CA$(15; 3, 12, 2)$ and CA$(27; 4, 12, 2)$. The execution time of the sequential algorithm for these two instances was 1,594 and 3,627 seconds respectively. We executed the parallel algorithm using $P = 4, 8, 12, 16, 20, 24, 28$ processors. Figure 5.2 shows the execution times in seconds for the instance CA$(15; 3, 12, 2)$, and Figure 5.3 shows the results for CA$(27; 4, 12, 2)$.

The speedup $S = T_s/T_p$ is the ratio between the execution time of the sequential algorithm T_s

Processors	Time	Speedup	Efficiency
4	634	2.44	0.61
8	298	5.19	0.64
12	185	8.37	0.69
16	140	11.06	0.69
20	98	15.80	0.79
24	89	17.40	0.72
28	77	20.11	0.71

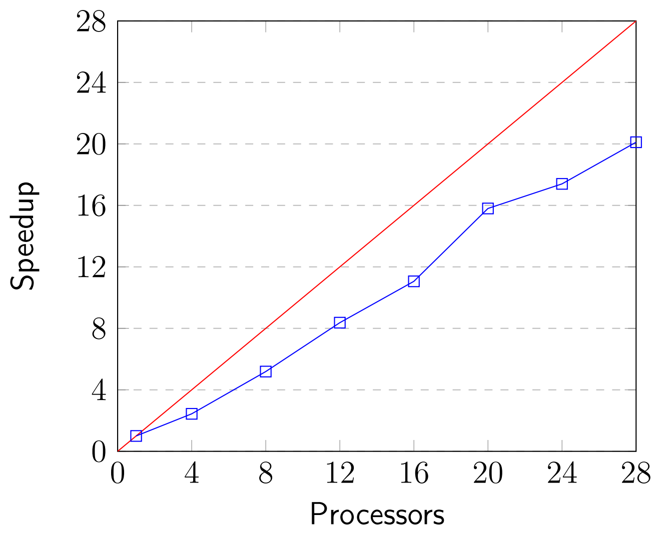


Figure 5.2: Execution times of the parallel NonIsoCA algorithm for CA$(15; 3, 12, 2)$.

Processors	Time	Speedup	Efficiency
4	1 586	2.28	0.57
8	652	5.56	0.69
12	427	8.49	0.70
16	335	10.82	0.67
20	263	13.79	0.68
24	226	16.04	0.66
28	174	20.84	0.74

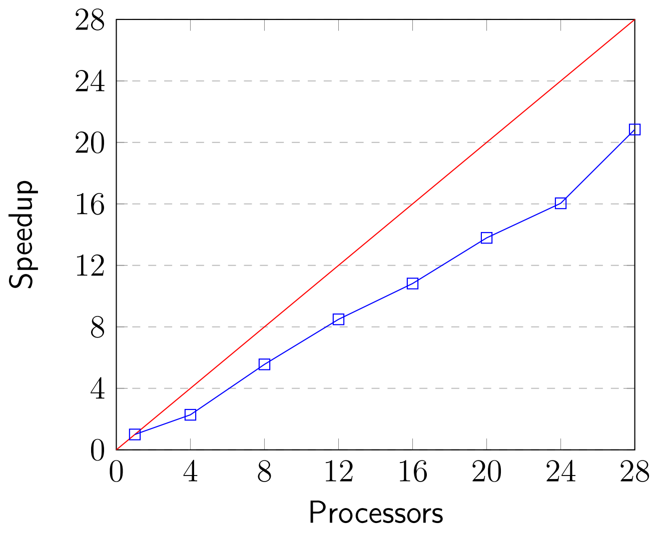


Figure 5.3: Execution times of the parallel NonIsoCA algorithm for CA$(27; 4, 12, 2)$.

and the execution time of the parallel algorithm with P processors T_p; if the program scales linearly then the speedup is equal to the number of processors ($S = P$). The speedup in the two instances is sublinear due to the cost of the communications, but it increases as the number of processors increases; so, for these instances and up to 28 processors the performance of the parallel algorithm is good. We also need to take into account that the master process does not participate actively (only at the beginning) in the construction of the non-isomorphic CAs; and therefore the maximum speedup we can obtain is $P - 1$.

To quantify in a better way the performance of the parallel algorithm we compute the parallel efficiency $E = S/P$, which is the ratio of speedup to the number of processors. The efficiency measures the percentage of utilization of a processor; if the program scales linearly then $S = P$ and the efficiency is 1. For the instance CA$(15; 3, 12, 2)$ the efficiency of the parallel algorithm lies between 0.61 and 0.79; and for CA$(27; 4, 12, 2)$ the efficiency lies between 0.57 and 0.74.

5.4.2 Parallel versions of the JuxtaposeCA algorithm

For the JuxtaposeCA algorithm we developed two parallel implementations. The first implementation divides the tuples $T = (A_0, A_1, \dots, A_{v-1})$ to be processed among the available processors. The partitioning of the tuples T is not static because the master sends a tuple to a slave as soon as the slave becomes idle.

The second implementation partitions the permutations of columns and symbols for the first free block in the arrays $J = [A_0, A'_1, \ldots, A'_{v-1}]$ to be generated from a tuple T into $P(k, \text{FIXED}) \cdot (v!)^{\text{FIXED}} = \frac{k!}{(k-\text{FIXED})!} \cdot (v!)^{\text{FIXED}}$ partitions, where $\text{FIXED} \in \{1, \ldots, k-1\}$ is the number of columns assigned directly; the other $k - \text{FIXED}$ columns of the first free block (the block that will contain a copy of A_1) are assigned in the normal way by testing all non-assigned columns of A_1 relabeled with the distinct $v!$ relabelings.

We tested the first parallel implementation of the JuxtaposeCA algorithm with the instance CA$(31; 4, 13, 2)$. In Subsection 5.2.1 we found that this CA does not exist. The process to prove this fact was to juxtapose the 2 non-isomorphic CA$(15; 3, 12, 2)$ with the 44,291 non-isomorphic CA$(16; 3, 12, 2)$; so the total number of tuples $T = (A_0, A_1)$, where $A_0 =$ CA$(15; 3, 12, 2)$ and $A_1 =$ CA$(16; 3, 12, 2)$, is $2 \cdot 44{,}291 = 88{,}582$. This number of tuples is much larger than the number of processors we have at our disposal; so we classify CA$(31; 4, 13, 2)$ using the first parallel version of the JuxtaposeCA algorithm. The obtained results for $P = 4, 8, 12, 16, 20, 24, 28$ processors are shown in Figure 5.4. The sequential algorithm took about 74,716 seconds, which is 20.75 hours.

The speedup is almost linear if we consider that only $P-1$ processors do the job of generating the arrays J derived from a tuple of non-isomorphic CAs $T = (A_0, A_1)$. The smaller efficiency is when $P = 4$, and the larger efficiency is 0.89 for $P = 12, 16$. For this parallel algorithm the speedup and the efficiency are better than the parallel NonIsoCA algorithm because there is no communication among the slave processes. In addition, for this particular instance the master process is not saturated with messages from the slaves because the only messages that are received by the master are those sent by the slaves when they have finished the processing of a tuple T. For other instances, the slaves can also send to the master the CAs they constructed.

The performance of the second parallel implementation of the JuxtaposeCA algorithm was analyzed by processing the instance CA$(340; 8, 10, 2)$. Since CAN$(7, 9, 2) = 170$ the only way to construct CA$(340; 8, 10, 2)$ is by juxtaposing two CA$(170; 7, 9, 2)$. However, this last CA is unique

Processors	Time	Speedup	Efficiency
4	25 024	2.98	0.74
8	10 714	6.97	0.87
12	6 920	10.79	0.89
16	5 221	14.31	0.89
20	4 215	17.72	0.88
24	3 505	21.31	0.88
28	3 044	24.54	0.87

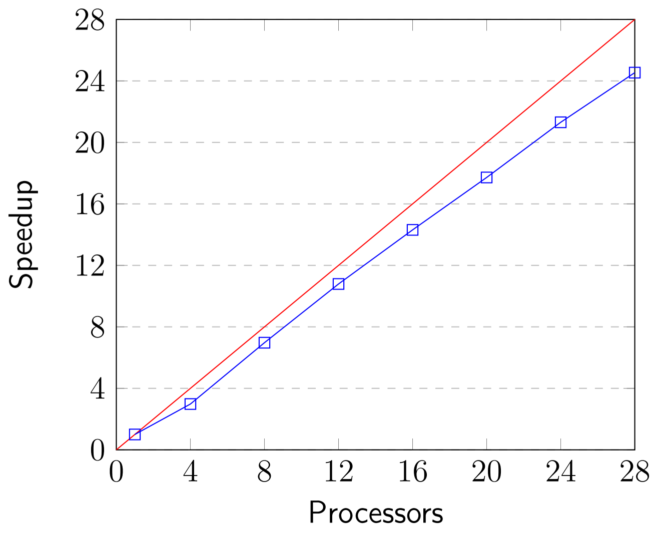


Figure 5.4: Execution times of the first parallel version of the JuxtaposeCA algorithm for CA$(31; 4, 13, 2)$.

up to isomorphisms [19]; then, the number of tuples $T = (A_0, A_1)$ to be juxtaposed is just 1. In this tuple both A_0 and A_1 are the canonical representative of the unique class for CA$(170; 7, 9, 2)$. Therefore, in this case the first parallel version of Juxtapose CA is not helpful because the number of tuples T is not large; in fact, the execution time of the first parallel version will be longer than the execution time of the sequential version due to the communication overhead. Thus, the second parallel implementation is the most suitable version for the instance CA$(340; 8, 10, 2)$.

From [19] we know that CA$(340; 8, 10, 2)$ does not exist because CAN$(8, 10, 2) = 341$. Figure 5.5 shows the results for the second parallel version of the JuxtaposeCA algorithm when $P = 4, 8, 12, 16, 20, 24, 28$ processors are used. The execution time of the sequential algorithm was 335 seconds. For this instance we used FIXED $= 3$, and so the number of partitions is $\frac{9!}{(9-3)!} \cdot 2^3 = 504 \cdot 8 = 4{,}032$. The efficiency for this instance lies between 0.67 and 0.79.

5.5 Chapter summary

This chapter presented the computational results that were obtained with the improved NonIsoCA algorithm and with the JuxtaposeCA algorithm. We classified optimal CAs, non-optimal CAs, and

Processors	Time	Speedup	Efficiency
4	123	2.72	0.68
8	62	5.40	0.67
12	40	8.37	0.69
16	28	11.96	0.74
20	21	15.95	0.79
24	18	18.61	0.77
28	15	22.33	0.79

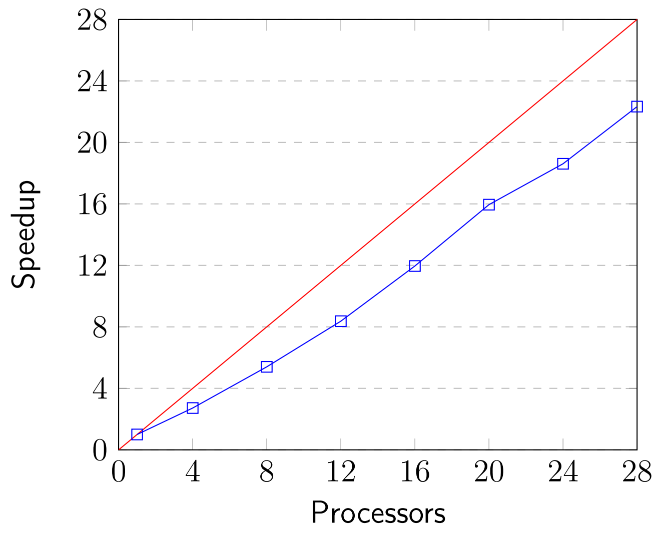


Figure 5.5: Execution times of the second parallel version of the JuxtaposeCA algorithm for CA$(340; 8, 10, 2)$.

CAs whose existence or nonexistence was unknown. Some nonexistence results lead to new covering array numbers: if we know the existence of CA$(N; t, k, v)$, and if we find zero isomorphism classes for CA$(N-1; t, k, v)$, then we conclude that CA$(N; t, k, v)$ is optimal and so CAN$(t, k, v) = N$. Other classification results imply improvements on the lower bound of some CANs. In addition, this chapter presented the results of evaluating the performance of the parallel implementations of the algorithms.

The next chapter summarizes the computational results of the two new classification algorithms, gives some conclusions of the work, and provides pointers for futre research.

6 Conclusions

This chapter ends the thesis document. Section 6.1 summarizes the main computational results; Section 6.2 compares the results of our two algorithms against the results of the state of the art algorithms; Section 6.3 gives some final remarks on the new classification algorithms; Section 6.4 states some ideas worth to be investigated in the future to improve the performance of the algorithms; and Section 6.5 lists the written journal papers.

6.1 Summary of the computational results

In this thesis we developed two new algorithms to classify CAs: the improved NonIsoCA algorithm and the JuxtaposeCA algorithm. For the improved NonIsoCA algorithm we developed one sequential version and one parallel version, and for the JuxtaposeCA algorithm we developed one sequential version and two parallel versions. The computational results that were obtained by using these algorithms were presented in Chapter 5. The relevant results are the classification of new CAs, the finding of exact values of CAN(t, k, v), and the improvement of lower bounds of CAN(t, k, v).

For the improved NonIsoCA algorithm the main results are:

- the classification of 23 CAs
- the finding of 13 exact values of CAN

These results are given in the following two tables:

CA	# Classes
CA$(14; 2, 10, 3)$	4 490
CA$(14; 2, 11, 3)$	0
CA$(33; 3, 5, 3)$	1
CA$(33; 3, 6, 3)$	1
CA$(19; 2, 6, 4)$	4
CA$(19; 2, 7, 4)$	0
CA$(20; 2, 7, 4)$	0
CA$(37; 2, 4, 6)$	13
CA$(37; 2, 5, 6)$	0
CA$(38; 2, 5, 6)$	0
CA$(17; 3, 12, 2)$	3 238 165 485
CA$(25; 4, 7, 2)$	6
CA$(26; 4, 7, 2)$	228
CA$(27; 4, 7, 2)$	13 012
CA$(28; 4, 7, 2)$	919 874
CA$(29; 4, 7, 2)$	58 488 647
CA$(30; 4, 7, 2)$	3 177 398 378
CA$(25; 4, 8, 2)$	7
CA$(26; 4, 8, 2)$	195
CA$(27; 4, 8, 2)$	9 045
CA$(28; 4, 8, 2)$	522 573
CA$(29; 4, 8, 2)$	27 826 894
CA$(30; 4, 8, 2)$	1 374 716 212

CAN$(t, k, v) = N$
CAN$(2, 11, 3) = 15$
CAN$(2, 12, 3) = 15$
CAN$(2, 13, 3) = 15$
CAN$(2, 14, 3) = 15$
CAN$(2, 15, 3) = 15$
CAN$(2, 16, 3) = 15$
CAN$(2, 17, 3) = 15$
CAN$(2, 18, 3) = 15$
CAN$(2, 19, 3) = 15$
CAN$(2, 20, 3) = 15$
CAN$(3, 14, 3) = 45$
CAN$(2, 7, 4) = 21$
CAN$(2, 5, 6) = 39$

The larger instances for the distinct combinations of v and t are CA$(14; 2, 11, 3)$, CA$(33; 3, 6, 3)$, CA$(20; 2, 7, 4)$, CA$(38; 2, 5, 6)$, CA$(17; 3, 12, 2)$, and CA$(30; 4, 8, 2)$.

For the JuxtaposeCA algorithm the new results are:

- the classification of 16 CAs
- the finding of 6 exact values of CAN

- the improvement of 13 lower bounds of CAN

These results are given in the following three tables:

CA	# Classes
CA$(30; 4, 13, 2)$	0
CA$(31; 4, 13, 2)$	0
CA$(32; 4, 13, 2)$	1
CA$(64; 5, 14, 2)$	1
CA$(128; 6, 15, 2)$	1
CA$(256; 7, 16, 2)$	1
CA$(48; 5, 8, 2)$	0
CA$(49; 5, 8, 2)$	0
CA$(50; 5, 8, 2)$	0
CA$(51; 5, 8, 2)$	0
CA$(52; 5, 8, 2)$	8
CA$(52; 5, 9, 2)$	0
CA$(53; 5, 9, 2)$	0
CA$(54; 5, 9, 2)$	1
CA$(53; 5, 8, 2)$	213
CA$(54; 5, 8, 2)$	20 450

CAN$(t, k, v) = N$
CAN$(4, 13, 2) = 32$
CAN$(5, 14, 2) = 64$
CAN$(6, 15, 2) = 128$
CAN$(7, 16, 2) = 256$
CAN$(5, 8, 2) = 52$
CAN$(5, 9, 2) = 54$

CAN$(t, k, v) \geq LB$
CAN$(5, 10, 2) \geq 55$
CAN$(5, 11, 2) \geq 55$
CAN$(5, 12, 2) \geq 55$
CAN$(5, 13, 2) \geq 55$
CAN$(6, 9, 2) \geq 107$
CAN$(7, 10, 2) \geq 214$
CAN$(8, 11, 2) \geq 428$
CAN$(9, 12, 2) \geq 856$
CAN$(10, 13, 2) \geq 1712$
CAN$(11, 14, 2) \geq 3424$
CAN$(4, 7, 3) \geq 100$
CAN$(3, 7, 3) \geq 37$
CAN$(3, 9, 3) \geq 40$

In this case, the largest processed CA was by far CA$(256; 7, 16, 2)$.

In total, we obtained 71 new results by using the two classification algorithms that were developed in this work.

6.2 Comparison with state of the art algorithms

To the best of our knowledge we have included in this thesis all known classification results for CAs. We do not reproduced all of them in this work because of time constraints, but the cases we reproduced matched the already known results. In Subsection 5.2.7 we performed a consistency check for the JuxtaposeCA algorithm; the objective was to reproduce some results obtained previously with the improved NonIsoCA algorithm; in all test cases the results of the JuxtaposeCA algorithm matched the results of the NonIsoCA algorithm. These facts give us confidence in the validity of the new results that are reported in this work.

In Chapter 2 we described six computational methods for the classification of CAs, or for the classification of objects equivalent to CAs of strength two. These methods are the following ones:

1. Classification of 2-surjective binary codes [22].
2. Classification of Latin squares [33].
3. Classification of MOLS [12].
4. Extension of CAs [11].
5. NonIsoCA algorithm [46].
6. Canonical augmentation [25].

By using the improved NonIsoCA algorithm we reproduced successfully all results of the works 1, 4, and 5. For the works 2 and 3 we only reproduced the results up to order $v = 6$. The work 2 reports the classification of Latin squares up to order 10, and the work 3 reports the classification of MOLS up to order 9. Finally, for the work 6 we reproduced almost all results, except for the instances CA$(29; 2, 7, 5)$, CA$(29; 2, 8, 5)$, CA$(39; 2, 5, 6)$, and CA$(39; 2, 6, 6)$.

For order $v = 2$ and strengths $t = 3, 4, 5, 6, 7$ our algorithms improved the results of the state of the art, with regard to the size of the classified CAs. The next table shows a comparison between the larger binary CAs that have been classified by state of the art algorithms and by our algorithms:

t	Largest CA in the state of the art	Largest CA in our algorithms
3	CA$(16; 3, 15, 2)$	CA$(17; 3, 12, 2)$
4	CA$(24; 4, 12, 2)$	CA$(32; 4, 13, 2)$
5	–	CA$(64; 5, 14, 2)$
6	–	CA$(128; 6, 15, 2)$
7	–	CA$(256; 7, 16, 2)$

To the best of our knowledge, no CA of strength $t \in \{5, 6, 7\}$ had been classified by computation until now. By using the JuxtaposeCA algorithm we classified ten CAs of strength 5, one CA of strength 6, and one CA of strength 7. So, these twelve CAs are the first ones with strength $t \in \{5, 6, 7\}$ whose classification was done through computation.

We can summarize our contributions to the state of the art as follows: for strengths $t > 2$ our new algorithms reproduced correctly all known results and found new ones; then, our new algorithms currently hold the record for CAs with strength $t > 2$.

6.3 Final remarks on the new algorithms

The improved NonIsoCA algorithm outperforms the original NonIsoCA algorithm when $t > 2$ or $v > 2$. The construction of the new column by means of backtracking allows to skip a number of candidate columns with no possibilities of making a CA of strength t with the current columns. Due to this, the execution time of the algorithm is reduced; in some cases, the execution time was reduced from several hours to a few seconds. When $v = 2$ and $t = 2$ the probability of passing the rules R1 to R5 is high because there are few symbols and there are few subarrays of t columns that contain the column being constructed; so the cost of checking the five rules is greater than the cost of processing all candidate columns, as the original NonIsoCA algorithm does.

The JuxtaposeCA algorithm works very differently than previous algorithms, because the non-isomorphic CAs are constructed subcolumn by subcolumn instead of cell by cell. The subcolumns are not arbitrary but columns of CAs with one less unit of strength, and this reduces considerably the search space. However, the search space is still large and we need to use the isomorphisms and the coverage properties of CAs to make cuts in the search space. The two key reductions are the following ones:

- By the isomorphisms of CAs the arrays J that we need to construct from a tuple $T = (A_0, A_1, \ldots, A_{v-1})$ are only those arrays $J = [A_0, A'_1, \ldots, A'_{v-1}]$ where A_0 is fixed and $A'_1, \ldots, A'_{v-1}$ are derived from $A_1, \ldots, A_{v-1}$ by column permutations and by symbol permutations. So, we do not consider row permutations to derive the arrays A'_i.

- By the coverage properties of CAs we can skip some column and symbol permutations in the construction of the arrays A'_i. The array J is constructed one column at a time and we can stop its construction if the current subarray is not a CA of strength $t+1$. Thus, not all column and symbol permutations are explored to construct the arrays A'_i.

Without these pruning criteria the JuxtaposeCA algorithm would take an impractical amount of time for the new CAs that were classified in this thesis.

In general, the JuxtaposeCA algorithm is faster than the NonIsoCA algorithm. However, if the number of tuples of non-isomorphic CAs $T = (A_0, A_1, \ldots, A_{v-1})$ to be processed is very large, then we should use the NonIsoCA algorithm instead. We can take a subset of the tuples T and construct all possible juxtapositions from these tuples to estimate the execution time of the JuxtaposeCA algorithm. If the number of tuples T is small, then with high probability the JuxtaposeCA algorithm will perform better than the NonIsoCA algorithm. We can also estimate the execution times of these two algorithms by using the formulas of their computational complexity.

6.4 Future work

In many instances the main limitation of the JuxtaposeCA algorithm is the huge number of tuples $T = (A_0, A_1, \ldots, A_{v-1})$ to be processed. For example, in Subsection 5.2.4 we could not classify CA$(107; 6, 9, 2)$ because we could not construct all non-isomorphic CA$(55; 5, 8, 2)$. To classify this last CA it would be necessary to test all juxtapositions of $A_0 =$ CA$(N_0; 4, 7, 2)$ and $A_1 =$ CA$(N_0; 4, 7, 2)$, where $N_0, N_1 \geq 24$ and $N_0 + N_1 = 55$. The most unbalanced multiset is $\{24, 31\}$. The CA$(24; 4, 7, 2)$ is unique, but for CA$(31; 4, 7, 2)$ there are a lot of non-isomorphic CAs, certainly more than 3,177,398,378, which is the number of distinct CA$(30; 4, 7, 2)$.

In the computational experimentation we noticed that the most unbalanced multisets produce less CAs of strength $t + 1$ than the more balanced multisets; so in the above example we expect more solutions for the multiset $\{27, 28\}$ than for the multiset $\{26, 29\}$, and more solutions for this last multiset than for the multisets $\{25, 30\}$ and $\{24, 31\}$.

If we can prove that the number of solutions for a multiset $\{a, b\}$ is always smaller than or equal to the number of solutions for a multiset $\{c, d\}$ whenever $|b - a| > |c - d|$, then we can skip the juxtapositions for $\{a, b\}$ if the number of solutions for $\{c, d\}$ is zero. This will reduce the execution time of the JuxtaposeCA algorithm because the most unbalanced multisets are commonly the most costly to process. Thus, a future line of investigation is to determine formally if this empirical observation is always true.

With regard to the improved NonIsoCA algorithm we think there are more rules to constraint the valid values for the unassigned cells of the new column. So, a future work is to investigate if we can reduce even more the number of candidate columns to extend the current CA.

Another possible improvement is to find a way to parallelize the *canonize*(A) function developed in Subsection 4.2.3. This function computes the canonical CA isomorphic to A. The complexity of this function is $O(N \log_2 N \cdot k! \cdot (v!)^k)$ for a CA$(N; t, k, v)$ because the $N!$ row permutations are reduced to a row sorting done in $O(N \log_2 N)$; so the execution time grows faster when the number of columns increases. For the instance CA$(256; 7, 16, 2)$ the sequential algorithm takes more than 24 hours. However, the canonization algorithm we have developed seems inherently sequential. Then, we need to design a smart strategy to parallelize the algorithm, or to design another canonization algorithm easier to parallelize.

6.5 Journal papers

We have written eight journal papers related to the work done in this thesis. In the first paper of the following list we report the sequential version of the JuxtaposeCA algorithm; and in the second paper we report the sequential version of the improved NonIsoCA algorithm. The remaining six papers are not directly related to the thesis problem, but we have included them because they were written during the doctoral program, at the time the objective of the thesis was to improve current upper bounds of CANs.

1. Idelfonso Izquierdo-Marquez and Jose Torres-Jimenez, *New covering array numbers*, Applied

Mathematics and Computation, Vol 353, 2019, pp 134-146.

2. Idelfonso Izquierdo-Marquez and Jose Torres-Jimenez, *New optimal covering arrays using an orderly algorithm*, Discrete Mathematics, Algorithms and Applications, Vol 10, No 1, 2018, 16 pages.

3. Jose Torres-Jimenez, Idelfonso Izquierdo-Marquez, and Himer Avila-George, *Methods to construct uniform covering arrays*, IEEE Access, accepted for publication.

4. Idelfonso Izquierdo-Marquez, Jose Torres-Jimenez, Brenda Acevedo-Juárez, and Himer Avila-George, *A greedy-metaheuristic 3-stage approach to construct covering arrays*, Information Sciences, Vol 460-461, 2018, pp 172-189.

5. Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez, *Covering arrays of strength three from extended permutation vectors*, Designs, Codes and Cryptography, Vol 86, No 11, 2018, pp 2629-2643.

6. Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez, *A simulated annealing algorithm to construct covering perfect hash families*, Mathematical Problems in Engineering, Vol 2018, Article ID 1860673, 14 pages.

7. Himer Avila-George, Jose Torres-Jimenez, and Idelfonso Izquierdo-Marquez, *Improved pairwise test suites for non-prime-power orders*, IET Software, Vol 12, No 3, 2018, pp 215-224.

8. Jose Torres-Jimenez, Idelfonso Izquierdo-Marquez, and Himer Avila-George, *Search-based software engineering for constructing covering arrays*, IET Software, Vol 12, No 4, 2018, pp 324-332.

Bibliography

[1] Ansótegui, C., Izquierdo, I., Manyà, F., and Torres-Jiménez, J. (2013). A max-sat-based approach to constructing optimal covering arrays. In Gibert, K., Botti, V., and Reig-Bolaño, R., editors, *Artificial Intelligence Research and Development*, volume 256 of *Frontiers in Artificial Intelligence and Applications*, pages 51–59. IOS Press BV, Amsterdam, Netherlands.

[2] 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. Springer-Verlag, Berlin, Heidelberg.

[3] Best, M., Brouwer, A., MacWilliams, F., Odlyzko, A., and Sloane, N. (1978). Bounds for binary codes of length less than 25. *IEEE Transactions on Information Theory*, 24(1):81–93.

[4] Bracho-Rios, J., Torres-Jimenez, J., and Rodriguez-Tello, E. (2009). A new backtracking algorithm for constructing binary covering arrays of variable strength. In Aguirre, A. H., Borja, R. M., and García, C. A. R., editors, *MICAI 2009: Advances in Artificial Intelligence*, volume 5845 of *Lecture Notes in Computer Science*, pages 397–407, Berlin, Heidelberg. Springer-Verlang.

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

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

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

[8] Colbourn, C. J. (2004). Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, 58:121–167.

[9] Colbourn, C. J. (2017). Covering array tables for $t = 2, 3, 4, 5, 6$. last time accessed on december 6, 2017.

[10] Colbourn, C. J. and Dinitz, J. H. (1996). *The CRC Handbook of Combinatorial Designs*. CRC press. ISBN 1-58488-506-8.

[11] Colbourn, C. J., Kéri, G., Soriano, P. P. R., and Schlage-Puchta, J. C. (2010). Covering and radius-covering arrays: Constructions and classification. *Discrete Applied Mathematics*, 158(11):1158–1180.

[12] Egan, J. and Wanless, I. M. (2016). Enumeration of mols of small order. *Mathematics of Computation*, 85(298):799–824.

[13] Francetić, N. and Stevens, B. (2017). Asymptotic size of covering arrays: An application of entropy compression. *Journal of Combinatorial Designs*, 25(6):243–257.

[14] Goethals, J.-M. (1977). The extended nadler code is unique (corresp.). *IEEE Transactions on Information Theory*, 23(1):132–135.

[15] 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.

[16] Hedayat, A. S., Sloane, N. J. A., and Stufken, J. (1999). *Orthogonal Arrays*. Springer-Verlag New York. ISBN 978-1-4612-7158-1.

[17] Hedayat, S., Stufken, J., and Su, G. (1997). On the construction and existence of orthogonal arrays with three levels and indexes 1 and 2. *The Annals of Statistics*, 25(5):2044–2053.

[18] 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.

[19] Johnson, K. A. and Entringer, R. (1989). Largest induced subgraphs of the n-cube that contain no 4-cycles. *Journal of Combinatorial Theory, Series B*, 46(3):346–355.

[20] Kaski, P. and Östergård, P. R. J. (2006). *Classification Algorithms for Codes and Designs*. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-28990-6.

[21] 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.

[22] Kéri, G. and Östergård, P. R. (2007). Further results on the covering radius of small codes. *Discrete Mathematics*, 307(1):69–77.

[23] Khalsa, S. K. and Labiche, Y. (2014). An orchestrated survey of available algorithms and tools for combinatorial testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 323–334.

[24] Kleitman, D. J. and Spencer, J. (1973). Families of k-independent sets. *Discrete Mathematics*, 6(3):255–262.

[25] Kokkala, J. I. (2017). *Computational Methods for Classification of Codes*. PhD thesis, Aalto University, Department of Communications and Networking, Espoo, Finland.

[26] Kuhn, D. R., Kacker, R. N., and Lei, Y. (2010). Practical combinatorial testing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States.

[27] 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.

[28] 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.

[29] 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.

[30] Kuliamin, V. V. and Petukhov, A. A. (2011). A survey of methods for constructing covering arrays. *Programming and Computer Software*, 37(3):121–146.

[31] Lawrence, J., Kacker, R. N., Lei, Y., Kuhn, D. R., and Forbes, M. (2011). A survey of binary covering arrays. *Electron. J. Combin.*, 18(1):P84.

[32] Lopez-Escogido, D., Torres-Jimenez, J., Rodriguez-Tello, E., and Rangel-Valdez, N. (2008). Strength two covering arrays construction using a sat representation. In *MICAI 2008: Advances in Artificial Intelligence*, volume 5317 of *Lecture Notes in Computer Science*, pages 44–53. Springer Berlin / Heidelberg.

[33] McKay, B. D., Meynert, A., and Myrvold, W. (2006). Small latin squares, quasigroups, and loops. *Journal of Combinatorial Designs*, 15(2):98–119.

[34] Nadler, M. (1962). A 32-point n=12, d=5 code (corresp.). *IRE Transactions on Information Theory*, 8(1):58–58.

[35] Nie, C. and Leung, H. (2011). A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29.

[36] Nordstrom, A. W. and Robinson, J. P. (1967). An optimum nonlinear code. *Information and Control*, 11(5):613–616.

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

[38] Ordoñez, H., Torres-Jimenez, J., Ordoñez, A., and Cobos, C. (2017). Clustering business process models based on multimodal search and covering arrays. In Pichardo-Lagunas, O. and Miranda-Jiménez, S., editors, *Advances in Soft Computing: 15th Mexican International Conference on Artificial Intelligence, MICAI 2016, Cancún, Mexico, October 23–28, 2016, Proceedings, Part II*, pages 317–328. Springer International Publishing, Cham.

[39] Sarkar, K. and Colbourn, C. J. (2017). Upper bounds on the size of covering arrays. *SIAM Journal on Discrete Mathematics*, 31(2):1277–1293.

[40] Seiden, E. and Zemach, R. (1966). On orthogonal arrays. *The Annals of Mathematical Statistics*, 37(5):1355–1370.

[41] Semakov, N. V. and Zinoviev, V. A. (1969). Complete and quasi-complete balanced codes. *Problems of Information Transmission*, 5(2):14–18.

[42] 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(4):1590–1594.

[43] Snover, S. L. (1973). *The uniqueness of the Nordstrom-Robinson and the Golay binary codes*. PhD thesis, Michigan State University, Department of Mathematics, Michigan, United States.

[44] Stevens, B. (1998). *Transversal Covers and Packings*. PhD thesis, University of Toronto, Department of Mathematics, Toronto, Ontario, Canada.

[45] Torres-Jimenez, J. and Izquierdo-Marquez, I. (2013). Survey of covering arrays. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 20–27.

[46] Torres-Jimenez, J. and Izquierdo-Marquez, I. (2016). Construction of non-isomorphic covering arrays. *Discrete Mathematics, Algorithms and Applications*, 08(02):1650033.

[47] Torres-Jimenez, J., Izquierdo-Marquez, I., Gonzalez-Gomez, A., and Avila-George, H. (2015a). A branch & bound algorithm to derive a direct construction for binary covering arrays. In Sidorov, G. and Galicia-Haro, N. S., editors, *Advances in Artificial Intelligence and Soft Computing: 14th Mexican International Conference on Artificial Intelligence, MICAI 2015, Cuernavaca, Morelos, Mexico, October 25-31, 2015, Proceedings, Part I*, pages 158–177. Springer International Publishing, Cham.

[48] Torres-Jimenez, J., Izquierdo-Marquez, I., Kacker, R. N., and Kuhn, D. R. (2015b). Tower of covering arrays. *Discrete Applied Mathematics*, 190-191:141–146.

[49] Torres-Jimenez, J. and Rodriguez-Tello, E. (2012). New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185(1):137 – 152.

[50] 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.

[51] 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.

[52] Yan, J. and Zhang, J. (2008). A backtracking search tool for constructing combinatorial test suites. *Journal of Systems and Software*, 81(10):1681–1693.

[53] Yang, P., Tan, X., Sun, H., Chen, D., and Li, C. (2011). Fire accident reconstruction based

on les field model by using orthogonal experimental design method. *Advances in Engineering Software*, 42(11):954 – 962.

[54] Yuan, X., Cohen, M. B., and Memon, A. M. (2011). Gui interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559–574.

[55] Zhang, J., Zhang, Z., and Ma, F. (2014). *Automatic Generation of Combinatorial Test Data*. Springer Publishing Company, Incorporated. ISBN 978-3-662-43428-4.