UNIVERSITAT POLITÈCNICA DE VALÈNCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
PROGRAMA DE DOCTORADO EN INFORMÁTICA

Constructing Covering Arrays using Parallel Computing and Grid Computing

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Author:
Himer Avila George
May 2012, Valencia, Spain.

PhD advisors:
Dr. Vicente Hernández García
Dr. José Torres Jiménez

Abstract

A good strategy to test a software component involves the generation of the whole set of cases that participate in its operation. While testing only individual values may not be enough, exhaustive testing of all possible combinations is not always feasible. An alternative technique to accomplish this goal is called combinatorial testing. Combinatorial testing is a method that can reduce cost and increase the effectiveness of software testing for many applications. It is based on constructing functional test-suites of economical size, which provide coverage of the most prevalent configurations. *Covering arrays* are combinatorial objects, that have been applied to do functional tests of software components. The use of covering arrays allows to test all the interactions, of a given size, among the input parameters using the minimum number of test cases.

For software testing, the fundamental problem is finding a covering array with the minimum possible number of rows, thus reducing the number of tests, the cost, and the time expended on the software testing process. Because of the importance of the construction of (near) optimal covering arrays, much research has been carried out in developing effective methods for constructing them. There are several reported methods for constructing these combinatorial models, among them are: (1) algebraic methods, recursive methods, (3) greedy methods, and (4) metaheuristics methods.

Metaheuristic methods, particularly through the application of *simulated annealing* has provided the most accurate results in several instances to date. Simulated annealing algorithm is a general-purpose stochastic optimization method that has proved to be an effective tool for approximating globally optimal solutions to many optimization problems. However, one of the major drawbacks of the simulated annealing is the time it requires to obtain good solutions.

In this thesis, we propose the development of an improved simulated annealing algorithm for constructing covering arrays of strength $t >= 2$ for their use in software interaction testing. In addition, we propose the use of Grid computing and Supercomputing to address the large amount of computing time necessary to obtain near-optimal covering arrays.

May 15, 2012

Himer Avila George
hiavgeo@i3m.upv.es

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

Notation	Description	Page List
$CA(N;t,k,v)$	Covering array with N runs, strength t, k factors and v levels	18
$CAN(t,k,v)$	The covering array number is the size of a covering array	20
$COD(N;t,k,v)$	Covering ordered design N rows, strength t, k columns and v symbols	38
$CODN(t,k,v)$	The covering design ordered number is the size of a covering ordered design	38
$DCA(N,\Gamma;2,k,v)$	Difference covering array with N runs, strength 2, k factors and v levels	37
$DCAN(2,k,v)$	The difference covering array number is the size of a difference covering array	37
$GF(v)$	Galois field	26
$MCA(N;t,k,v_1^{q_1}v_2^{q_2}\ldots v_g^{q_w})$	Mixed covering array with N runs, strength t, k factors and $v_1^{q_1}v_2^{q_2}\ldots v_g^{q_w})$ levels	22
$MOLS(v,w)$	A collection of w mutually orthogonal Latin squares of order v	15
N	Number of runs in an experiment	17
$OA_\lambda(N;t,k,v)$	Orthogonal array with N runs, strength t, k factors and v levels	17
$QCA(N;k,\ell,v)$	QCA is an $N\times k$ array with columns indexed by ordered pairs from $\{1,\ldots,k\}\times\{1,\ldots,\ell\}$	39
$QCAN(k,\ell,v)$	$QCAN$ denotes the minimum number of rows in QCA	39
V	Set of symbols or levels	13
λ	Index of array	17
k	Number of factors (variables) in an experiment	17
t	Strength of array	17
v	Number of levels	17

Acronyms

Notation	Description	Page List
ACO	Ant Colony Optimization	51, 52
ACTS	Advanced Combinatorial Testing System	45
AETG	Automatic Efficient Test Generator	41
ATBBA	Assigning Tasks by Blocks Approach	68, 74
ATBCBA	Assigning Tasks by Cyclic Blocks Approach	69
BBA	Building-Block Algorithm	45
BDII	Berkley Database Information System	72
CA	Covering Array	18
CAC	Covering Array Construction	20, 25, 26, 52, 54, 75
CAN	Covering Array Number	20
CAR	Covering Array Repository	92
CCA	Cyclic Covering Array	21
CE	Computing Elements	71
CLI	Command Line Interface Tools	71
CM	Cyclic Matrix	21, 28
COD	Covering Ordered Design	38
CPHF	Covering Perfect Hash Families	50
CSA	Cooperative Search Approach	92, 103
DATWA	Dynamic Assignment of Tasks to Workers Approach	67
DCA	Difference Covering Array	37
DDA	Deterministic Density Algorithm	42
DGSA	Developed Grid Simulated Annealing	87, 88, 93, 102, 116

Notation	Description	Page List
DPSA	Developed Parallel Simulated Annealing	93
DSSA	Developed Sequential Simulated Annealing	79, 84, 85, 92, 93, 98–100, 110, 112
EGI	European Grid Infrastructure	71, 86, 87
GA	Genetic Algorithms	52
GAVCA	Grid Algorithm to Verify Covering Arrays	74
GOED	Group of Optimal Experimental Design	140
IPO	In-Parameter-Order	43
IPOG	In-Parameter-Order-General	44, 51, 54
IRPS	Intersection Residual Pair Set Strategy	46, 48
ISA	Independent Search Approach	91, 103
LFC	Logic File Catalog	72
LRMS	Local Resource Management System	71
MA	Memetic Algorithm	54
MCA	Mixed Covering Array	22, 50
NGI	National Grid Initiatives	71, 86
OA	Orthogonal Array	16, 54
PSA	Parallel Simulated Annealing	90
R-GMA	Relational Grid Monitoring Architecture	72
SA	Simulated Annealing	52, 53
SATWA	Static Assignment of Tasks to Workers Approach	67, 68
SAVCA	Sequential Algorithm to Verify Covering Arrays	64, 65
SE	Storage Element	71
SSA	Semi-Independent Search Approach	91, 92, 103

Notation	**Description**	**Page List**
TCG	Test Case Generation	41
TS	Tabu Search	49, 50, 52
UI	User Interface	71
VO	Virtual Organisation	72
VOMS	Virtual Organisation Management System	72
WMS/RB	Workload Management System / Resource Broker	71
WN	Working Nodes	71

List of Publications

Journal papers

Avila-George, Himer et al. (2012d). “Supercomputing and Grid Computing on the verification of Covering Arrays”. In: *The Journal of supercomputing* (2012). Published online: 18 April 2012, pp. 1–30. DOI: 10.1007/s11227-012-0763-0 (cit. on pp. XV, XVI, 62, 68–70, 74).

Avila-George, Himer et al. (April 2012). “A metaheuristic approach for constructing functional test-suites”. In: *IET Software (submitted to a second round of revisions)* (April 2012).

Avila-George, Himer et al. (February 2012). “New bounds for ternary covering arrays using a parallel simulated annealing”. In: *Submitted to: Mathematical Problems in Engineering* (February 2012).

Torres-Jimenez, Jose et al. (2011a). “Construction of logarithm tables for Galois Fields”. In: *International Journal of Mathematical Education in Science and Technology* 42.1 (2011), pp. 91–102. DOI: 10.1080/0020739X.2010.510215 (cit. on pp. XV, 54, 56–59).

Torres-Jimenez, Jose et al. (September 2011). “CINVESTAV Covering Arrays Repository”. In: *submitted to: IET Software* (September 2011).

Books and books chapters

Avila-George, Himer et al. (2010a). *Verificación de Covering Arrays: Aplicando la Supercomputación y la Computación Grid*. LAP Lambert Academic Publishing, 2010. ISBN: 978-3-8433-5142-3.

Avila-George, Himer et al. (2012a). “Grid Computing - Technology and Applications, Widespread Coverage and New Horizons”. In: InTech, 2012. Chap. Using Grid Computing for the construction of ternary covering arrays. ISBN: 979-953-307-540-1 (cit. on pp. 139, 141).

Torres-Jimenez, Jose et al. (2012). "Cryptography and Security in Computing". In: InTech, 2012. Chap. Construction of Orthogonal Arrays of Index Unity using Logarithm Tables for Galois Fields, pp. 71–90. ISBN: 978-953-51-0179-6. URL: http://www.intechopen.com/download/pdf/29702 (cit. on pp. XV, 59, 61, 62).

International conference papers

Avila-George, Himer et al. (2010b). "Verification of General and Cyclic Covering Arrays Using Grid Computing". In: *Proceedings of the 3rd International Conference on Data Management in Grid and Peer-to-Peer Systems - GLOBE*. Vol. 6265. Lecture Notes in Computer Science. Bilbao, Spain, 30 August - 3 September: Springer-Verlag, 2010, pp. 112–123. ISBN: 978-3-642-15107-1. DOI: 10.1007/978-3-642-15108-8_10 (cit. on pp. XV, 61, 62, 65, 74).

Avila-George, Himer et al. (2011). "A parallel algorithm for the verification of Covering Arrays". In: *Proceedings of the 17th International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA*. Las Vegas, EEUU, July 18-21, 2011. ISBN: 1-60132-193-7. URL: http://world-comp.org/p2011/PDP8061.pdf (cit. on pp. XV, 62, 66, 74).

Avila-George, Himer et al. (2012b). "Parallel Simulated Annealing for the Covering Arrays Construction Problem". In: *(to appear) Proceedings of the 18th International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA*. Las Vegas, EEUU, July 16-19, 2012 (cit. on pp. 92, 139, 141).

Avila-George, Himer et al. (2012c). "Simulated Annealing for Constructing Mixed Covering Arrays". In: *Proceedings of the 9th International Symposium on Distributed Computing and Artificial Intelligence - DCAI*. Vol. 151. Advances in Intelligent and Soft Computing. Salamanca, Spain, from 28th to 30th March: Springer Berlin / Heidelberg, 2012, pp. 657–664. ISBN: 978-3-642-28764-0. DOI: 10.1007/978-3-642-28765-7_79 (cit. on pp. 9, 139, 141, 144).

Martinez-Pena, Jorge et al. (2010). "A Heuristic Approach for Constructing Ternary Covering Arrays Using Trinomial Coefficients". In: *Proceedings of the 12th Ibero-American conference on Advances in artificial intelligence - IBERAMIA*. Vol. 6433. Lecture Notes in Computer Science. Bahía Blanca, Argentina, November 1-5: Springer-Verlag, 2010, pp. 572–581. ISBN: 978-3-642-16951-9. DOI: 10.1007/978-3-642-16952-6_58 (cit. on pp. 53, 139, 141).

Torres-Jimenez, Jose et al. (2010). “Optimization of investment options using SQL”. In: *Proceedings of the 12th Ibero-American conference on Advances in artificial intelligence - IBERAMIA*. Vol. 6433. Lecture Notes in Computer Science. Bahía Blanca, Argentina, November 1-5: Springer-Verlag, 2010, pp. 30–39. ISBN: 978-3-642-16951-9. DOI: 10.1007/978-3-642-16952-6_4.

Torres-Jimenez, Jose et al. (2011b). “MAXCLIQUE Problem Solved Using SQL”. In: *Proceedings of the third International Conference on Advances in Databases, Knowledge, and Data Applications - DBKDA*. St. Maarten, The Netherlands Antilles, January 23-28: IARIA, 2011, pp. 83–88. ISBN: 978-1-61208-115-1. URL: http://www.thinkmind.org/download.php?articleid=dbkda_2011_4_40_30097.

Chapter 1

Introduction

1.1 Software testing overview

Software systems are heavily used in critical fields like medical diagnosis, air traffic control, space shuttle missions and stock market reporting. The presence of bugs in the software application can cause irreparable losses. In 2003 the National Institute of Standards and Technology (NIST) published a widely cited report which estimated that inadequate software testing costs the US economy $59.5 billion per year, even though 50% to 80% of development budgets go toward testing. This study highlights the need for more effective methods of software testing. According to Hartman (2005), the quality of the software relies strongly on the use of software testing.

> **Definition 1** (Software testing)**.**
>
> Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended. Software should be predictable and consistent, offering no surprises to users.

Software testing is commonly described in terms of a series of testing stages. A software testing stage is a process for ensuring that some aspect of a software product, system, or unit functions properly. General testing stages are basic to software testing and occur for all software. The following three stages are considered general software testing stages (Weyuker, 1998):

1. *Unit testing*, in which individual components are tested. Unit testing frequently uses test cases selected using the component's actual source code. Unit testing is generally done by *code developers* who have access to the source code and are familiar with its details, and therefore can constructively use this information. Also, the relatively small size of the individual modules or units being tested makes it feasible to consider the code details when determining appropriate unit test cases.

2. *Integration testing*, in which the subsystems formed by integrating the individually tested components are tested as an entity. The *code developers* themselves or an independent test organization may perform integration testing. Integration testing frequently emphasizes the interface code since the individual modules being integrated have already been tested. People other than the code developers usually do system testing; they may therefore be unfamiliar with the level of detail necessary to perform code-based testing and generally do not have access to the source code. They are only responsible for testing the fully integrated system; when they find symptoms of faults (that is, when failures occur in response to test cases), they simply transmit the information to the development organization for fault isolation and repair.

3. *System testing*, in which the system formed from the tested subsystems is tested as an entity. System testing typically uses test cases selected without reference to the code details, because at this level, there is generally far too much code to rely on such details.

Besides these stages of testing, there are many different methods of testing such as *structural testing* and *functional testing*. These methods had been developed in order to improve the quality of the software systems.

In structural testing (or white-box testing) test conditions are designed by examining paths of logic. Structural testing is typically used during unit testing, where the tester (usually the code developer) knows the internal structure and tries to exercise it based on detailed knowledge of the code. The tester examines the internal structure of the program or system. Test data is driven by examining the logic of the program or system, without concern for the program or system requirements. The tester knows the internal program structure and logic, just as a car mechanic knows the inner workings of an automobile. Specific examples in this category include:

- ▷ Data-flow testing: The data flow criteria are based on analysis that is similar to that done by an optimizing compiler, classifying occurrences of variables in a program as being either definitions or uses. Programs will be represented by flow graphs, consisting of nodes that represent blocks or sequences of statements that are always exercised as a unit, and edges that represent the flow of control between blocks (Frankl and Weyuker, 1988)

- ▷ Path testing: Each and every independent path within the code is executed at least once to find out any error (Yan and Zhang, 2008).
- ▷ Branch testing: Branch testing helps in the validation of all the branches in the code and making sure that no branching leads to abnormal behavior of the application (Frankl and Weiss, 1993).
- ▷ Condition Testing: Each and every condition is executed by making it true and false in test cases, in each of the ways at least once (McMinn, 2004).
- ▷ Mutation Testing: In this testing, the application is tested for the code that was modified after fixing a particular defect (Offutt et al., 1996).

An advantage of structural testing is that it is thorough and focuses on the produced code. Because there is knowledge of the internal structure or logic, errors or deliberate mischief on the part of a code developer have a higher probability of being detected.

One disadvantage of structural testing is that it does not verify that the specifications are correct; that is, it focuses only on the internal logic and does not verify the logic to the specification. Another disadvantage is that there is no way to detect missing paths and data-sensitive errors.

Functional testing (or black-box testing) is one in which test conditions are developed based on the functionality of the software system; that is, the tester requires information about the input data and observed output, but does not know how the program or system works. Just as one does not have to know how a car works internally to drive it, it is not necessary to know the internal structure of a program to execute it. The tester focuses on testing the functionality of the program against the specification. With functional testing, the tester views the program as a black-box and is completely unconcerned with the internal structure of the program or system. Functional testing is used during integration and system tests, where the emphasis is on the perspective of the user and not on the internal workings of the software. Functional testing tries to test the functionality of the software as it is perceived by the end users (based on user manuals) and the requirements writers. Thus, functional testing consists of subjecting the system under test to various user controlled inputs, and watching its performance and behavior. Some examples in this category include:

- ▷ Decision tables testing: Decision tables represent logical relationships between conditions (for example, inputs) and actions (for example, outputs). Derive test cases systematically by considering every possible combination of conditions and actions (Beizer, 1990).
- ▷ Equivalence partitioning testing: It divides the input domain into a collection of subsets, or equivalence classes, which are deemed equivalent according to the specification. Pick representative tests (sometimes only one)

from within each class. Can also be done with output, path, and program structure equivalence classes (Reid, 1997).

- ▷ Boundary value testing: It chooses test cases on or near the boundaries of the input domain of variables, with the rationale that many defects tend to concentrate near the extreme values of inputs (Reid, 1997).
- ▷ Requirements-based testing: Given a set of requirements, it devises tests so that each requirement has an associated test set. Trace test cases back to requirements to ensure that all requirements are covered (Mogyorodi, 2001).
- ▷ Combinatorial interaction testing: Combinatorial Interaction Testing is a black box sampling technique derived from the statistical field of design of experiments. It has been used extensively to sample inputs to software, and more recently to test highly configurable software systems and GUI event sequences.

A major advantage of functional testing is that the tests are aimed to what the program or system is supposed to do, and it is natural and understood by everyone. A limitation is that exhaustive input testing is not achievable, because this requires that every possible input condition or combination be tested. In addition, because there is no knowledge of the internal structure or logic, there could be errors or deliberate mischief on the part of a code developer that may not be detectable with functional testing.

Since the number of possible inputs is typically very large, testers need to select a subset, commonly called a *suite*, of test cases, based on effectiveness and adequacy. Below we discuss some of the popular testing methods that have been adopted by the testing community. This thesis is mainly related with functional testing and more specifically with combinatorial interaction testing. In the next section the general aspects of the combinatorial interaction testing are described.

1.2 Combinatorial interaction testing (CIT)

Software systems today are complex and have many possible configurations. Many software systems are built using reusable components of software. Interaction among components are often complex and abundant. Components may not be designed with the final product in mind which leaves them susceptible to unexpected *interaction faults*. Although in theory, tests could be run under all possible configurations in order to detect interaction faults, in practice this is infeasible either time-wise or cost-wise. Therefore, it is of widespread interest generating test suites that provide coverage of as many interactions as possible.

Combinatorial testing selects input values for individual parameters and combines these values to create tests. One strategy for combinatorial testing, called t-way testing, requires every possible combination of values of any t parameters to be included in some test case, where t is typically less than the total number of parameters. The key observation behind t-way testing is that not every parameter contributes to every fault, and many faults can be exposed by considering interactions among a small number of parameters.

Consider a hypothetical holiday-reservation system that has four components of interest shown in Table 1.1. There are three log-in types, three customer types, three reservation types, and three credit cards types. Different end users may use different combinations of components. To exhaustively test all combinations of the four parameters that have 3 options each from Table 1.1 would require $3^4 =$ 81 tests. The four components are *factors*, and the three values for each factor are their *levels*.

Table 1.1: The four parameters, and their three possible values, of a hypothetical holiday-reservation system.

Log-in type	Customer type	Reservation type	Credit card type
New customer - not logged in	New customer	Cars	Visa
New customer - logged in	Frequent customer	Hotels	Mastercard
Frequent customer - logged in	Employee	Flights	American Express

It is possible to reduce the 81 tests required for exhaustive testing by employing 2-way (or pairwise) interaction testing. Instead of testing every combination, all individual pairs of interactions are tested. Table 1.2 shows the resulting test suite, it contains only 9 tests. The entire test suite covers every possible pairwise combination between components.

Table 1.2: A small interaction test suite for the hypothetical holiday-reservation system showed in Table 1.1.

Test No.	Log-in Type	Customer type	Reservation type	Credit card type
1	New member - not logged in	New customer	Cars	Visa
2	New customer - not logged in	Frequent customer	Hotels	Mastercard
3	New customer - not logged in	Employee	Flights	American Express
4	New-customer - logged in	New customer	Flights	Mastercard
5	New-customer - logged in	Frequent customer	Cars	American Express
6	New-customer - logged in	Employee	Hotels	Visa
7	Customer - logged in	New customer	Hotels	American Express
8	Customer - logged in	Frequent customer	Flights	Visa
9	Customer - logged in	Employee	Cars	Mastercard

A suite with 81 test cases may sound reasonable, but the number of necessary tests grows exponentially as the number of components increases. Suppose we had a system with 12 possible factors and four levels each. We then need $4^{12} =$ 16,777,216 test cases, if we extrapolate these tests in terms of time, considering we could finish a test in one second, we would require 279,620 minutes (or 6 months) to finish all the tests. Pairwise combinatorial testing for 4^{12} can be achieved in as few as 24 tests.

Based on the example above, we note that it is easy to apply the combinatorial interaction testing. Combinatorial testing is a specification-based technique which requires no knowledge about the implementation under test. Note that the specification required by some forms of combinatorial testing is lightweight, as it only needs to identify a set of parameters and their possible values. This is in contrast with other testing techniques that require a complex operational model of the system under test. Finally, assuming that the parameters and values are properly identified, the actual combination generation process can be fully automated.

Combinatorial interaction testing is based on the premise that many errors in software can only arise from the interaction of two or more parameters (Bryce et al., 2010). The application of combinatorial testing to software applications has been studied extensively in recent years. Burr and Young (1998) showed that pairwise testing can achieve a higher block and decision coverage than traditional methods for a commercial email system. Dalal et al. (1999) reported a case study in which combinatorial testing was applied to a telephone system. Kuhn et al. (2004) studied the actual faults in several open source software projects. Colbourn et al. (2005) applied combinatorial testing to progressive ranking and composition of Web services. Yilmaz et al. (2006) applied combinatorial testing to an open-source CORBA middleware implementation ACE+TAO. Kuhn et al. (2010) presented the use of combinatorial testing in a smart phone application.

Many studies demonstrated the effectiveness of pairwise testing in a variety of applications. But, there is a possibility that some of the failures in a system are present when an interaction of more than 2 parameters occurs. An appropriate value for t to provide adequate coverage depends on the complexity of the system under test, and in general the value of t is not known. Studies were made in order to show the percentage of failures a real system will present when distinct levels of interaction were used The results of these tests are summarized in Table 1.3 and Figure 1.1.

Table 1.3: Percent fault detection at interaction levels 1 through 6 according to the type of application.

	Applications				
Interaction level	**Med services**	**Browser**	**Server**	**NASA database**	**Network security**
1	66%	29%	42%	68%	17%
2	97%	76%	70%	93%	62%
3	99%	95%	89%	98%	87%
4	100%	97%	96%	100%	98%
5	100%	99%	96%	100%	100%
6	100%	100%	100%	100%	100%

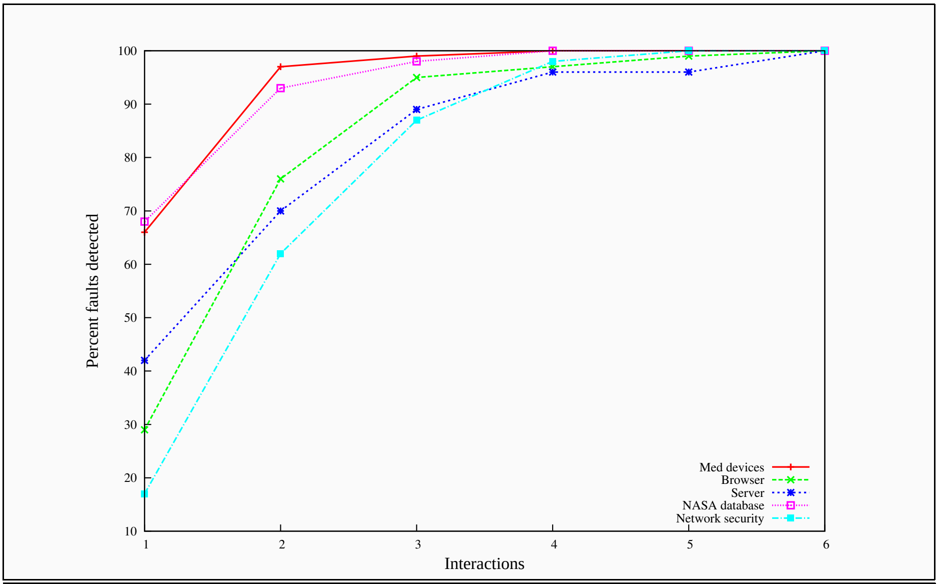

Figure 1.1: Fault detection at interaction levels 1 through 6 according to the type of application (Kuhn et al., 2008).

In Figure 1.1 we can clearly see how the failure detection rate increases rapidly with the interaction level. With the browser application, for example, 29% of the failures were triggered by only a single parameter value, 76% by pairwise combinations, and 95% by 3-way combinations. The detection rate curves for the other applications behaves in a similar way, reaching in some cases 100% of detection with 4 to 6-way interactions. This means that the interaction of size six or less parameters in these systems were causing 100% percent of the faults on the systems.

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

1.3 Research problem

To continue at the forefront in this fast paced and competitive world, companies have to be highly adaptable and to suit such transforming needs customized software solutions play a key role. To support this customization, software systems must provide numerous configurable options. While this flexibility promotes customizations, it creates many potential system configurations, which may need extensive quality assurance.

A good strategy to test a software component involves the generation of the whole set of cases that participate in its operation. While testing only individual values may not be enough, exhaustive testing of all possible combinations is not always feasible (Mala et al., 2010; Cohen et al., 2003). An alternative technique to accomplish this goal is called combinatorial testing. Combinatorial testing is a method that can reduce cost and increase the effectiveness of software testing for many applications. It is based on constructing functional test-suites of economical size, which provide coverage of the most prevalent configurations. Covering Arrays (CAs) are combinatorial objects, that have been applied to do functional tests of software components. The use of covering arrays allows to test all the interactions, of a given size, among the input parameters using the minimum number of test cases.

Definition 2.

A Covering Array (CA) is a combinatorial object, denoted by $CA(N; t, k, v)$ which can be described like a matrix with $N \times k$ elements, such that every $N \times t$ subarray contains all possible combinations of v^t symbols at least once. N represents the rows of the matrix, k is the number of parameters, which have v possible values, and t represents the strength or the degree of controlled interaction. When a covering array contains the minimum possible number of rows, it is optimal and its size is called the Covering Array Number (CAN).

For software testing, the fundamental problem is to determine $CAN(t, k, v)$. Because, it reduces the number of tests, the cost and the time expended on the software testing process.

A covering array has the same cardinality in all its parameters. However, software systems are generally composed with parameters that have different cardinalities; in this situation a mixed covering array (MCA) can be used.

Definition 3.

A Mixed Covering Array, denoted by $MCA(N; t, k, v_1 v_2 \ldots v_k)$, is an $N \times k$ array where $v_1 v_2 \ldots v_k$ is a cardinality vector that indicates the values for every column. The mixed covering arrays has the following two properties:

1. Each column $i (1 \leq i \leq k)$ contains only elements from a set S_i with $|S_i| = v_i$.
2. The rows of each $N \times t$ subarray cover all t-tuples of values from the t columns at least once.

Because of the importance of the construction of (near) optimal MCAs, much research has been carried out in developing effective methods for constructing them. There are several reported methods for constructing these combinatorial models, among them are:

1. Algebraic methods (Bush, 1952; Sherwood, 2008)
2. Recursive methods (Williams, 2000; Moura et al., 2003; Colbourn and Torres-Jimenez, 2010)
3. Greedy methods (Cohen et al., 1996; Tung and Aldiwan, 2000; Lei et al., 2007; Bryce and Colbourn, 2007; McDowell, 2011)
4. Metaheuristics methods (Cohen et al., 2003; Shiba et al., 2004; Gonzalez-Hernandez et al., 2010; Avila-George et al., 2012c)

Metaheuristic methods, particularly through the application of Simulated Annealing (SA), has provided the most accurate results in several instances until now. This local search method has provided many of the smallest covering arrays for different system configurations (Bryce et al., 2010). Simulated annealing algorithm is a general-purpose stochastic optimization method that has proved to be an effective tool for approximating globally optimal solutions to many optimization problems. However, one of the major drawbacks of the method is the time it requires to obtain good solutions (which increases when the evaluation function requires too much time).

In this thesis, we propose the development of an improved simulated annealing algorithm for constructing uniform and mixed covering arrays of strength $t >= 2$ for their use in software interaction testing. In addition, we propose the use of Grid computing and Supercomputing to address the large amount of computing time necessary to obtain near-optimal covering arrays.

1.3.1 Hypothesis

It is possible to develop a simulated annealing algorithm to construct covering arrays that use Parallel computing and Grid computing in order to address the slow convergence of the simulated annealing technique.

1.3.2 Objective

Develop and implement a simulated annealing algorithm for constructing optimal or near-optimal covering arrays using Parallel computing and *Grid computing* to address the slow convergence of the simulated annealing technique.

1.4 Contributions

The expected contributions of the present thesis were:

- ▷ An improved implementation of a simulated annealing algorithm for constructing uniform and mixed covering arrays of strength $t \geq 2$.
- ▷ A Grid implementation of simulated annealing algorithm.
- ▷ A Parallel simulated annealing algorithm for constructing covering arrays.
- ▷ An algorithm to verify covering arrays.

The constructed covering arrays have been published in the repository described in Appendix A, in order that others can study the actual covering arrays, build new covering arrays from them, and also use these covering arrays without having to spend the computational resources.

1.5 Thesis organization

The remaining of this thesis is structured as follows:

- ▷ Chapter 2 presents some basic definitions and terminology about combinatorial interaction testing objects.
- ▷ Chapter 3 describes the relevant related work to construct covering arrays. There are several reported methods for constructing these combinatorial models. Among them are: (1) Algebraic methods, (2) Recursive methods, (3) Greedy methods, and (4) Metaheuristics methods.
- ▷ Chapter 4 presents the specific details that were involved in the development of the simulated annealing proposed for constructing covering arrays.

- ▷ Chapter 5 analyzes the global performance of the developed simulated annealing algorithm and the influences that some of its key features have on it; A methodology for fine-tuning the developed algorithm is presented; Moreover, it shows the results obtained by the implementation of simulated annealing algorithm; Finally, it illustrates the development of test configurations for two real software applications.
- ▷ Chapter 6 contains the conclusions derived from this thesis, also it presents some possible directions for future research.

Chapter 2

Theoretical Framework

Combinatorial approaches to testing are used in several fields, and have recently gained momentum in the field of software testing through software interaction testing.

In this chapter is presented some basic definitions and terminology about combinatorial designs. Let V be a set of v symbols or levels. The term "level" is used because in the design of experiments, the symbols typically indicate the levels or settings of the factors or variables whose effects on a response of interest are to be studied. Usually it will denote the possible levels by $0, 1, \ldots, v-1$. Through this work, by an $N \times k$ array (or matrix) with entries from V it shall mean a collection of Nk elements of V arranged in N rows and k columns with one element per row-column pair.

2.1 Latin Square and Orthogonal Latin Squares

Latin squares are combinatorial designs most easily described as a $v \times v$ array. It is believed that Euler by 1782 was the first one to study them. Fisher (1926) used them in the design of statistical experiments. Mandl (1985) applied them in software testing, specifically in designing some of the tests in the Ada Compiler Validation Capability test suite.

Definition 4 (Latin Square)**.**

A *Latin square* of order v is a $v \times v$ array with entries from a set V of the cardinality v such that each element of V appears once in every row and every column (Hedayat et al., 1999).

It is easily seen that a *Latin square* of order v exists for every positive integer v.

Proposition 1.

For every positive integer v, there exists a $v \times v$ *Latin square* with V as the set of objects.

Proof. Set $L_{ij} = i + j$ module v. Thus,

$$L = \begin{pmatrix} 0 & 1 & \dots & v-1 \\ 1 & 2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ v-1 & 0 & \dots & v-2 \end{pmatrix}$$

Clearly the array L is a *Latin square*. □

Example 1.

For $v = 4$, the construction of *Proposition* 1 yields the *Latin square* shown in Figure 2.1.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

Figure 2.1: A *Latin square* of order 4, every symbol 0,1,2,3 appears once in every row and column.

Two *Latin squares* are called *orthogonal* if when one is superimposed upon the other every ordered pair of symbols occurs once in the resulting square.

Definition 5 (Orthogonal Latin Squares)**.**

Let $\mathcal{A} = (a_{ij})$ and $\mathcal{B} = (b_{ij})$ be *Latin squares* of order v. They are said to be orthogonal if every ordered pair of symbols occurs exactly once among the v^2 pairs (a_{ij}, b_{ij}), $i = 0, 1, \ldots, v-1$; $j = 0, 1, \ldots, v-1$.

Example 2.

The arrays in Figure 2.2 are *orthogonal Latin squares* of order 4. It can observe that none of these arrays is *orthogonal* with to the array in Figure 2.1.

(a)

$$\begin{pmatrix} 2 & 0 & 1 & 3 \\ 0 & 2 & 3 & 1 \\ 3 & 1 & 0 & 2 \\ 1 & 3 & 2 & 0 \end{pmatrix}$$

(b)

$$\begin{pmatrix} 3 & 1 & 0 & 2 \\ 0 & 2 & 3 & 1 \\ 1 & 3 & 2 & 0 \\ 2 & 0 & 1 & 3 \end{pmatrix}$$

(c)

$$\begin{pmatrix} 1 & 3 & 2 & 0 \\ 0 & 2 & 3 & 1 \\ 2 & 0 & 1 & 3 \\ 3 & 1 & 0 & 2 \end{pmatrix}$$

Figure 2.2: Three *orthogonal Latin squares* of order 4.

A *Latin square* is *orthogonal isolated* if there is no *Latin square* orthogonal to it. The *Latin square* in Figure 2.1 is *orthogonal isolated*.

A collection of w *Latin squares* of order v, any pair of which are orthogonal, is called a set of *Mutually Orthogonal Latin Squares*. Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ be *Latin squares* of order v. They are called *mutually orthogonal* if $\mathcal{A}_r$ and $\mathcal{A}_s$ are orthogonal for all r and s with $1 \leq r < s \leq k$. A set of *Mutually Orthogonal Latin Squares* is called a $MOLS(v, w)$. The arrays in Figure 2.2 form a $MOLS(4, 3)$.

Example 3.

The three arrays in Figure 2.3 shows that the *orthogonal Latin squares* shown in Figure 2.2 form a MOLS(4,3). Notice that all sixteen pairs of symbols (x, y) occurs once in the same cell of the squares.

There are certain basic operations which transform one *Latin square* into another. Any permutation of the rows of the array, or the columns of the array, or the elements of V gives another *Latin square*. Let's say two *Latin squares* are isomorphic

(a)

$$\begin{pmatrix} (2,3) & (0,1) & (1,0) & (3,2) \\ (0,0) & (2,2) & (3,3) & (1,1) \\ (3,1) & (1,3) & (0,2) & (2,0) \\ (1,2) & (3,0) & (2,1) & (0,3) \end{pmatrix}$$

(b)

$$\begin{pmatrix} (2,1) & (0,3) & (1,2) & (3,0) \\ (0,0) & (2,2) & (3,3) & (1,1) \\ (3,2) & (1,0) & (0,1) & (2,3) \\ (1,3) & (3,1) & (2,0) & (0,2) \end{pmatrix}$$

(c)

$$\begin{pmatrix} (3,1) & (1,3) & (0,2) & (2,0) \\ (0,0) & (2,2) & (3,3) & (1,1) \\ (1,2) & (3,0) & (2,1) & (0,3) \\ (2,3) & (0,1) & (1,0) & (3,2) \end{pmatrix}$$

Figure 2.3: Example of a MOLS(4,3). (a) A juxtaposed array corresponding to the pair of *orthogonal Latin squares* (Figure 2.2(a) and (b)); (b) A juxtaposed array corresponding to the pair of *orthogonal Latin squares* (Figure 2.2(a) and (c)); (c) A juxtaposed array corresponding to the pair of *orthogonal Latin squares* (Figure 2.2(b) and (c)); Notice that all sixteen pairs of symbols (x, y) occurs once in the same cell of the squares.

if and only if one can be transformed into the other by a combination of these three operations. Any two of the *Latin squares* in Figure 2.2 are isomorphic, but none of them is isomorphic to the *Latin square* in Figure 2.1.

In a $MOLS(v, w)$, a permutation of the element of V in one or more of the *Latin squares* will no affect their orthogonality. A permutation of the rows or columns that is performed simultaneously on all *Latin squares* of the $MOLS(v, w)$ also preserves the orthogonality.

2.2 Orthogonal Arrays

The Orthogonal Arrays (OAs) were introduced by Rao (1946) under the name of *hypercubes*. Besides being used for construction of various other combinatorial configurations, they are popular among statisticians for their properties in fractional factorial experiments. The first works where *orthogonal arrays* were applied to the designs of experiments, were made in disciplines like agriculture and medicine (Hedayat et al., 1999). The use of OAs for testing software was suggested by Mandl (1985), he described using orthogonal arrays in testing of a compiler. Tatsumi (1987) in his work on Test Case Design Support System used in Fujitsu Ltd, talks about two standards for creating test arrays: (1) with all combinations covered exactly the same number of times (orthogonal arrays).

Definition 6 (Orthogonal Array)**.**

An *orthogonal array* denoted by $OA_\lambda(N;t,k,v)$, can be defined as an $N \times k$ array on v symbols such that every $N \times t$ subarray contains all the ordered subsets of size t from v symbols exactly λ times.

Orthogonal arrays have the property that $\lambda = \frac{N}{v^t}$. In the case when $\lambda = 1$ it is customary to say that the OA has *index unity*, it is optimal. The integers N, t, k, v and λ may be referred to as the parameters of the OA. The number of rows N is also known as the size of the array, the number of runs (observations), the number of assemblies or the number of level or treatment combinations; the parameter t is the strength; the number of columns k is also called the number of constrains, or the number of factors or variables; and v is the number of symbols or number of levels associated with each factor, the order.

Example 4.

The array in Figure 2.4 is an *orthogonal array* based on three levels, with strength two, of index unity, with nine runs and with four factors. It is an $OA(9;2,4,3)$.

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

Figure 2.4: Example of an $OA(9;2,4,3)$, where the *strength* is $t = 2$, the *alphabet* is $v = 3$ and all nine combinations of symbols appear only once in each pair of columns of the *orthogonal array*.

The *orthogonal arrays* has some interesting properties, among them are the following ones:

1. The parameters of the OA satisfy $\lambda = N/v^t$;

2. An *orthogonal array* of strength t is also an *orthogonal array* of strength t', where $1 \geq t' \geq t$. The index λ' of an *orthogonal array* of strength t' is $\lambda' = \lambda \cdot m^{t-t'}$;

3. Let $A_i = \{0, 1, \ldots, r\}$ be a set of $OA(N_i; t_i, k, v)$, the juxtaposed array $A = \begin{bmatrix} A_0 \\ \ldots \\ A_r \end{bmatrix}$ is an $OA(N; t, k, v)$ where $N = N_1 + N_2 + \cdots + N_r$ and $t \geq \min\{t_0, t_1, \ldots, t_r\}$;

4. Any permutation of rows or columns in an *orthogonal array*, results in another *orthogonal array* with the same parameters;

5. Any subarray of size $N \times k'$ of an $OA(N; t, k, v)$, is an $OA(N; t', k', v)$ of strength $t' = \min\{k', t\}$;

6. Select the rows of an $OA(N; t, k, v)$ that starts with the symbol 0, and eliminate the first column; the resulting matrix is an $OA(N/v; t-1, k-1, v)$.

The requirement that every t-tuple arises exactly λ times can be too restrictive in applications that require only that every t-tuple be covered at least once. It can be relax the definition to introduce the *covering arrays* and *mixed covering arrays*.

2.3 Covering Arrays

The Covering Arrays (CAs) have been object of study and application in different research areas. Cawse (2003) used covering arrays for design of materials, Hedayat et al. (1999) used them in medicine and agriculture; in biology and industrial processes have also been used by Shasha et al. (2001) and Phadke (1995). Covering arrays have been used in hardware testing Vadde and Syrotiuk (2004) but significantly the area with the major application of these objects is in software testing Burr and Young (1998); Yilmaz et al. (2006).

Definition 7 (Covering Array)**.**

A *covering array* denoted by $CA(N; t, k, v)$, is a matrix $\mathcal{M}$ of size $N \times k$ which takes values from the set of symbols $\{0, 1, 2, \ldots, v-1\}$ (called the alphabet), and every submatrix of size $N \times t$ contains each tuple of symbols of size t (t-tuple), at least once.

The value N is the number of rows of $\mathcal{M}$, k is the number of parameters, where each parameter can take v values and the interaction degree between parameters is described by the strength t. Each combination of t columns must cover all the v^t t-tuples. Given that there are $\binom{k}{t}$ sets of t columns in $\mathcal{M}$, the total number of t-tuples in $\mathcal{M}$ must be $v^t\binom{k}{t}$. When a t-tuple is missing in a specific set of t columns we will refer to it as a missing t-wise combination. Then, $\mathcal{M}$ is a *covering array* if the number of missing t-wise combinations is zero.

Example 5.

The array in Figure 2.5 is a $CA(7;2,7,2)$. The strength of this covering array is $t = 2$ and the alphabet is $v = 2$, hence the combinations $\{0,0\}$, $\{0,1\}$, $\{1,0\}$, $\{1,1\}$ appear in each subset of size $N \times 2$ of the covering array.

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

Figure 2.5: Example of a $CA(7;2,7,2)$.

To illustrate the covering array approach applied to the design of software testing, consider the Web-based system example shown in Table 2.1, the example involves four parameters each with three possible values. A full experimental design ($t = 4$) should cover $3^4 = 81$ possibilities, however, if the interaction is relaxed to $t = 2$ (pair-wise), then the number of possible combinations is reduced to 9 test cases.

Table 2.1: Combinatorial testing using covering arrays, a Web-based system example (parameters).

	Browser	**OS**	**DBMS**	**Connections**
0	Firefox	Windows 7	MySQL	ISDN
1	Chromium	Ubuntu 10.10	PostgreSQL	ADSL
2	Netscape	Red Hat 5	MaxDB	Cable

Figure 2.6(a) shows the covering array corresponding to $CA(9;2,4,3)$; given that its strength and alphabet are $t = 2$ and $v = 3$, respectively, the combinations that must appear at least once in each subset of size $N \times 2$ are $\{0,0\}$, $\{0,1\}$, $\{0,2\}$, $\{1,0\}$, $\{1,1\}$, $\{1,2\}$, $\{2,0\}$, $\{2,1\}$, $\{2,2\}$. Finally, to make the mapping between the covering array and the Web-based system, every possible value of each parameter in Table 2.1 is labeled by the row number. Figure 2.6(b) shows the corresponding pair-wise test suite; each of its nine experiments is analogous to one row of the covering array shown in Figure 2.6(a).

(a)

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

(b)

1	Firefox	Windows 7	MySQL	ISDN
2	Firefox	Ubuntu 10.10	PostgreSQL	ADSL
3	Firefox	Red Hat 5	MaxDB	Cable
4	Chromium	Windows 7	PostgreSQL	Cable
5	Chromium	Ubuntu 10.10	MaxDB	ISDN
6	Chromium	Red Hat 5	MySQL	ADSL
7	Netscape	Windows 7	MaxDB	ADSL
8	Netscape	Ubuntu 10.10	MySQL	Cable
9	Netscape	Red Hat 5	PostgreSQL	ISDN

Figure 2.6: Combinatorial testing using covering arrays, a Web-based system example. (a) A combinatorial design, $CA(9;2,4,3)$. (b) Test-suite covering all 2-way interactions, $CA(9;2,4,3)$.

When a matrix has the minimum possible value of N to be a $CA(N;t,k,v)$, the value N is known as the Covering Array Number (CAN). The CAN is formally defined in (2.1) and is denoted by $CAN(t,k,v)$.

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

The trivial mathematical *lower bound* for a covering array is $v^t \leq CAN(t,k,v)$, however, this number is rarely achieved. Therefore determining achievable lower bounds is one of the main research lines for covering arrays. Finding the covering array number is also known in the literature as the Covering Array Construction (CAC). It is equivalent to the problem of maximizing the degree k of a covering array given the values N, t, and v (Sloane, 1993).

Given a $CA(N;t,k,v)$ permuting the rows and/or columns produces an equivalent covering array (Hnich et al., 2006). The rows represent a set of test vectors, and their order is irrelevant. Permuting the columns does not affect since every subset of t columns contains all the combinations of v^t symbols.

Definition 8 (Isomorphic Covering Arrays)**.**

Two *covering arrays* are said to be *isomorphic* if one can be obtained from the other by a sequence of permutations of the rows, the columns, and the symbols.

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

Example 6.

The two covering arrays of $CA(4; 2, 3, 2)$ in Figure 2.7 are isomorphic since we can get one from the other by swapping the first two columns of the matrix.

(a)

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

(b)

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

Figure 2.7: Example of isomorphic covering arrays.

Definition 9 (Cyclic covering arrays)**.**

According to Colbourn and Torres-Jimenez (2010), a Cyclic Matrix (CM) is an array $\mathcal{O}$ of size $k \times k$ that is formed by k rotations of a vector of size k (called starter vector s). The covering arrays derived from a cyclic matrix will be referred as Cyclic Covering Array (CCA).

Figure 2.8 gives an example of a cyclic matrix formed from the starter vector $\{0, 0, 0, 1, 0, 1, 1\}$. This matrix is a cyclic covering array $CA(7; 2, 7, 2)$.

This combinatorial object (covering array) is fundamental in developing interaction tests when all factors have an equal number of levels. However, software systems are generally composed with parameters that have different cardinalities. To remove this limitation of covering arrays, the mixed-level covering array can be used.

(a) Starter vector s of size 7.

0	0	0	1	0	1	1

(b) $CA(7;2,7,2)$

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

Figure 2.8: Example of a cyclic covering array $CA(7;2,7,2)$.

A Mixed Covering Array (MCA) is a generalization of a covering array that allows for different alphabets in different columns. This was introduced to remove the limitation that all parameters had to have the same number of possible values since different parameters in the system will often take on a different number of possible values. This is a more realistic approach in a software application context.

Definition 10 (Mixed Covering Array)**.**

A *mixed level covering array* denoted by $MCA(N;t,k,v_1\ v_2 \dots v_k)$, is an $N \times k$ array where $v_1 v_2 \dots v_k$ is a cardinality vector that indicates the values for every column. The mixed covering arrays have the following two properties: (1) Each column $i(1 \leq i \leq k)$ contains only elements from a set S_i with $|S_i| = v_i$. (2) The rows of each $N \times t$ subarray cover all t-tuples of values from the t columns at least once. The minimum N for which there exists a mixed covering array is called mixed covering array number $MCAN(t,k,v_1 v_2 \dots v_k)$. A short notation for the mixed covering array can be given using the exponential notation $MCA(N;t,k,v_1^{q_1} v_2^{q_2} \dots v_g^{q_w})$; the notation describes, that there are q_r parameters from the set $\{v_1, v_2, \dots, v_k\}$ that takes v_s values (Cohen et al., 2003).

To illustrate the mixed covering array approach applied to the design of software testing, consider the e-commerce system example shown in Table 2.2, the example involves four parameters, the first two parameters have 3 possible values and the last two parameters have only 2 possible values; to test in an exhaustive way the software is required a set of $3 \times 3 \times 2 \times 2 = 36$ test cases.

Table 2.2: Combinatorial testing using mixed covering arrays, a Web-based system example (parameters).

	Browser	**WebServer**	**Database**	**Payment**
0	Firefox	Apache	MySQL	Visa
1	Chromium	IIS	MaxDB	MasterCard
2	IE	WebSphere		

Using a mixed covering array (with interaction size $t = 2$) will require only 9 cases. Figure 2.9(a) shows a mixed covering array corresponding to $MCA(9; 2, 4, 3^2 2^2)$. Finally, to make the mapping between the mixed covering array and the *e*-commerce system, every possible value of each parameter in Table 2.2 is labeled by the row number. Figure 2.9(b) shows the corresponding pair-wise test suite; each of its nine experiments is analogous to one row of the mixed covering array shown in Figure 2.9(a).

(a)

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

(b)

1	Firefox	Apache	MySQL	Visa
2	IE	IIS	MySQL	MasterCard
3	Chromium	WebSphere	MySQL	MasterCard
4	Firefox	WebSphere	MaxDB	Visa
5	IE	Apache	MaxDB	MasterCard
6	Chromium	IIS	MaxDB	Visa
7	Firefox	IIS	MySQL	MasterCard
8	IE	WebSphere	MaxDB	Visa
9	Chromium	Apache	MaxDB	MasterCard

Figure 2.9: Combinatorial testing using mixed covering arrays, a Web-based system example. (a) It shows a $MCA(9; 2, 4, 3^2 2^2)$ for the *e*-commerce system. (b) Test-suite covering all 2-way interactions, $MCA(9; 2, 4, 3^2 2^2)$.

2.4 Summary

In this chapter we have presented in detail the evolution of combinatorial objects, starting from the Latin squares, orthogonal Latin squares, Mutually orthogonal Latin squares, orthogonal arrays, until reaching the Covering arrays. The primary combinatorial object that we will examine from now on is the covering array. We will discuss both uniform and mixed level arrays because our goal is to build real test suites. In the next chapter we will describe the relevant related work to the construction of covering arrays.

Chapter 3

State of the Art

Because of the importance of the construction of (near) optimal covering arrays, much research has been carried out in developing effective methods for constructing them. In this chapter we describe the relevant related work to the construction of covering arrays. Section 3.1 presents the computational complexity of the CAC problem. There are several reported methods for constructing these combinatorial models. Among them are:

1. Algebraic methods, see Section 3.2
2. Recursive methods, see Section 3.3
3. Greedy methods, see Section 3.4
4. Metaheuristic methods, see Section 3.5

Some of the algorithms used to solve the CAC problem are approximated, meaning that rather than constructing optimal covering arrays, they construct matrices of size close to that value. Section 3.7 presents a methodology to verify a given matrix as a covering array.

3.1 Computational complexity

There are a variety of computational problems that can be solved in polynomial time, others can only be solved in exponential time; algorithms that do not run in polynomial time are considered infeasible. The difficulty of problems are classified into complexity classes. P is the class of all problems that can be solved by a deterministic Turing machine in polynomial time, where the polynomial time bound is a function of the input size. NP is the class of (decision) problems for which solutions to the given input instance can be guessed and verified in polynomial time. In other words, problems in NP can be solved by a nondeterministic Turing machine in polynomial time.

Concerning the complexity of the CAC problem, Lei and Tai (1998) showed that the construction of optimal covering arrays is an NP-complete problem. Recently Lawrence et al. (2011) showed the proof presented by Lei and Tai (1998) is erroneous; since the "pair-cover problem" as described in that paper fails to match up correctly with the problem of finding strength $t = 2$ covering arrays. Therefore, the problem of determining the NP-completeness of the covering arrays construction problem in the general case, is still open. However, there exist certain closely related problems which are NP-complete (Seroussi and Bshouty, 1988; Colbourn, 2004; Cheng, 2007), suggesting that the covering arrays construction problem is a hard combinatorial optimization problem.

Due to the complexity of the problem, most of the algorithms are approximate, as meaning that they find a solution in a reasonable time, but not necessarily the optimal solution.

3.2 Algebraic methods

Algebraic methods construct covering arrays in polynomial time using predefined rules. Some algebraic approaches compute covering arrays directly using some mathematical functions or other algebraic procedures. There exist only some special cases where it is possible to find the covering array number using polynomial order algorithms.

3.2.1 Constructing optimal covering arrays within polynomial time

Bush construction

Bush (1952) presented a generalization of the concept of a set of orthogonal Latin squares, called *orthogonal arrays of index unity*. In his paper Bush introduced a very ingenious procedure for constructed these arrays. It employs a special class of polynomials which have coefficients in the finite $GF(v)$, where $v = p^{\alpha}$ is a prime or a power of prime and $v > t$. Theorem 1 ensures the existence of orthogonal arrays with $v + 1$ columns when v is a prime power.

Theorem 1 (Bush (1952)).

Let $v = p^\alpha$ be a prime power with $v > t$. Then $OAN(t, k, v) = v^t$ for all $k \leq v + 1$.

The resulting orthogonal arrays of index unity are equivalent to covering arrays of size $N = v^t$, strength $t \geq 2$ and degree $k = v + 1$. Section 3.6 will present an efficient implementation for this construction, based on the use of logarithm tables for Galois Fields.

Case: $t = v = 2$

Rényi (1971) determined sizes of covering arrays for the case $t = v = 2$ when N is even. Kleitman and Spencer (1973) and Katona (1973) independently determined covering array numbers for all N when $t = v = 2$. The construction is straightforward. It specifies that given N, in order to build a $CA(N; t, k, v)$ with maximum k, it forms a matrix in which the columns consist of all distinct binary N-tuples of weight $\lceil \frac{N}{2} \rceil$ that have a 0 in the first position. Theorem 2 guarantees that this is a covering array, and gives a maximum k.

Theorem 2 (Kleitman and Spencer (1973); Katona (1973)).

Let k be a positive integer, then

$$CAN(t, k, v) = \min \left\{ N : \binom{N-1}{\lceil \frac{N}{2} \rceil} \right\} \geq k.$$

3.2.2 Group construction of covering arrays

Chateauneuf and Kreher (2002) introduced a new method to construct covering arrays of strength three. This construction uses the structure of covering arrays and the repetition in covering arrays. The idea is to construct a covering array from a small array, a *starter vector* and a group. This construction builds the covering array column by column by considering the group acting on the columns of the starter vector. In some cases a small array will be appended to complete the covering array. However, they do not show a concrete strategy on how to obtain both the starter vector or the group acting. Meagher and Stevens (2005) extended the idea of Chateauneuf and Kreher (2002), presenting a strategy for obtaining the starter vector by local search and the selection of a group action.

This construction involves selecting a subgroup of the symmetric group on k elements, $G < Sym_v$, and finding a starter vector, $\alpha \in \mathbb{Z}_v^k$, the starter vector depends on the group G. The vector is used to form a Cyclic Matrix $\mathbb{Z}$. The group acting on the matrix $\mathbb{Z}$ produces several arrays which are concatenated to form a covering array Z. Often it will be necessary to add a small matrix, S, to complete the covering conditions.

Using this construction, if there is an initial vector $\mathbb{Z}$, with respect to the action of the group G, then it can construct a $CA(k(v-1)+1; 2, k, v)$.

Figure 3.1 shows an example of how this construction works. Let $G = \{e, (12)\} < Sym_3$ and $\alpha = \{0, 1, 1, 1, 2\} \in \mathbb{Z}_3^5$. Construct the cyclic matrix X (Figure 3.1(a)) taking α as the first row. The elements of G acting on the matrix X produce the arrays shown in Figure 3.1(b). The small vector $S = \{0, 0, 0, 0, 0\}$ is needed to ensure the coverage of all pairs. From this, a $CA(11; 2, 5, 3)$ is construct by juxtaposing the arrays S, X and $X_{(1,2)}$, see Figure 3.1(c).

(a)

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

(b)

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

$$X_{(12)} = \begin{pmatrix} 0 & 2 & 2 & 2 & 1 \\ 1 & 0 & 2 & 2 & 2 \\ 2 & 1 & 0 & 2 & 2 \\ 2 & 2 & 1 & 0 & 2 \\ 2 & 2 & 2 & 1 & 0 \end{pmatrix}$$

(c)

$$Z = \left(\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 2 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 2 & 0 & 1 & 1 \\ 1 & 1 & 2 & 0 & 1 \\ 1 & 1 & 1 & 2 & 0 \\ \hline 0 & 2 & 2 & 2 & 1 \\ 1 & 0 & 2 & 2 & 2 \\ 2 & 1 & 0 & 2 & 2 \\ 2 & 2 & 1 & 0 & 2 \\ 2 & 2 & 2 & 1 & 0 \end{array}\right)$$

Figure 3.1: Example of the construction of covering arrays using the *group construction*. (a) Circular matrix X constructed from the initial vector $\alpha = \{0, 1, 1, 1, 2\}$ as the first row; (b) The elements of G acting on X produce X_e and $X_{(12)}$; (c) Z is the $CA(11; 2, 5, 3)$ constructed using the *group construction*.

Finally, Lobb et al. (2012) presented a generalization of this method to permit any number of fixed points, permit an arbitrary group acting on the symbols, and permit an arbitrary group acting on the columns. With all these generalizations were obtained new bounds for covering arrays of strength two.

3.2.3 Constant weight vectors

Tang and Woo (1983) used *constant weight vectors* to construct test suites to be applied to logic circuit testing. These test sets are equivalent to covering arrays. A covering array can be formed by vectors of a particular set of weights.

Definition 11.

Let a be a vector of size k, with entries from $\{0, 1, \ldots, v-1\}$. The weight of a, denoted by w, is the sum of the values in the vector, see (3.1).

$$w = \sum_{i=0}^{k-1} a_i \tag{3.1}$$

Table 3.1 shows the set of all vectors where $v = 2$, $k = 5$, $w = 2$.

Table 3.1: Binary vectors of size $k = 5$ and weight $w = 2$.

Cardinality	Vectors
	1 1 0 0 0
	1 0 1 0 0
	1 0 0 1 0
	1 0 0 0 1
$\binom{5}{2}$	0 1 1 0 0
	0 1 0 1 0
	0 1 0 0 1
	0 0 1 1 0
	0 0 1 0 1
	0 0 0 1 1

Tang and Woo derive that to construct a $CA(N; t, k, v)$, it requires a set of vectors T that contains all vectors v space, size k, weights w such that $w \equiv c \bmod s$, for a constant c, where $s = (n-k)(v-1)+1$, $0 \leq c \leq k-t$, and $0 \leq w \leq k(v-1)$.

Thus, there are a total of $k - t + 1$ solutions. Then, we must find which of these solutions gives the set of vectors with lower cardinality, i.e., the lower value of N.

Table 3.2 gives an explicit example of this construction. In this example, to construct a $CA(N, 2, 4, 3)$, it obtains that $s = (k-t)(v-1)+1 = 5$. There are five ternary covering arrays, one for each constant c for $0 \leq c \leq s-1$. The $|T|_{\text{min}}$ is obtained when $w \equiv 1 \bmod 5$, then w takes values in $\{1, 6\}$. Therefore $N = 14$.

Table 3.2 shows the two subsets of vectors for the weights 1 and 6 that construct the $CA(14, 2, 4, 3)$. N is optimal in the domain of the sets of constant weight vectors, and is composed as follows: the number of vectors where $w = 1$ is 4, while the number of vectors where $w = 6$ is 10.

Table 3.2: A $CA(14; 2, 4, 3)$.

w	Cardinality	Vectors
1	$\binom{4}{1}$	1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
6	$\binom{4}{1} + \binom{4}{2}$	2 2 2 0 2 2 0 2 2 0 2 2 0 2 2 2 2 2 1 1 2 1 1 2 1 1 2 2 1 2 2 1 2 1 2 1 1 2 1 2

This construction for the binary case gives the following upper bound (3.2):

$$|T|_{\min} = \frac{2^k}{k - t + 1}. \tag{3.2}$$

Additionally, it is obtained that when $t \leq n/2$, the smallest possible cardinality of T is obtained directly with the sets of all vectors where $w = \lfloor t/2 \rfloor$ and $t - \lfloor t/2 \rfloor - 1$, that is (3.3):

$$|T|_{\min} = \binom{k}{\lfloor t/2 \rfloor} + \binom{k}{t - \lfloor t/2 \rfloor - 1}. \tag{3.3}$$

However, the values of N obtained from (3.3) are impractical in software testing for large values of k and t, because the number of tests needed is very large.

3.2.4 Using trinomial coefficients

Martinez-Pena and Torres-Jimenez (2010) introduced a new method for constructing covering arrays using trinomial coefficients, they improves the results presented by Tang and Woo (1983). They used the trinomial coefficients for the representation of the search space in the construction of ternary covering arrays. It is clear that any covering array is formed by a row set. In this sense, a trinomial coefficient represents a particular subset of rows which may belong to a ternary covering array.

Definition 12.

Let $0 \leq a,\ b,\ c \leq k,\ k = a + b + c$ and $k \geq 2$, where k is the ternary covering array degree. A candidate row subset $\Re^k_{a,b,c}$ is a collection of rows obtained by the trinomial coefficient $\binom{k}{a,b,c}$ and its cardinality is equal to that coefficient. The candidate row subset is generated by evaluating all combinations using $0^a\ 1^b\ 2^c$ symbols, i.e., symbol 0 is used a times, symbol 1 is used b times and symbol 2 is used c times over a k-column row.

The previous definition leads to the next theorem.

Theorem 3.

Let $\mathcal{A}$ be a set of k-th degree trinomial coefficients. For any strength $2 \leq t \leq k$, a vertical concatenation of the row subsets generated by each trinomial coefficient in $\mathcal{A}$ may construct any ternary covering array.

Let the strength and the degree of a required covering array equal q. Let Z be a $3^q \times q$ array. Adjoin the $\binom{q+2}{2}$ trinomial coefficients of q-th degree in the set $\mathcal{A}$. For each element in $\mathcal{A}$ generate its candidate row subset and append it vertically to Z. Then Z is an optimal $CA(3^q; q, q, 3)$.

We verify the result is a ternary covering array by looking into the definitions of the trinomial theorem and of the candidate row subsets. The trinomial theorem generates all possible trinomial coefficients of q-th degree. Remark that the sum of all q-th degree trinomial coefficients is 3^q. Hence if we transform each trinomial coefficient of q-th degree into candidate row subsets we will be producing 3^q different rows. These rows represent all the possible combinations that any $N \times q$ subarray must contain. By definition, any $CA(q, q, v)$ has only one subarray of size q. Therefore, we have constructed an optimal ternary CA of strength q and degree q.

Now, consider the constructed $CA(3^q; q, q, 3)$. The strength value in a covering array is upper bounded by degree value. For any $2 \leq s < q$ we automatically derive a $CA(3^q; s, q, 3)$. Then we can construct a ternary CA of any strength and any degree.

Figure 3.2 gives an explicit example of construction of a ternary covering arrays. In this example, it constructs a $CA(9; 2, 3, 3)$ by using a set $\mathcal{A}$ with 4 trinomial coefficients. Figure 3.2(a) shows a table that describes each trinomial coefficient in $\mathcal{A}$ and their corresponding candidate row subsets. Figure 3.2(b) displays the array Z (the ternary covering array), which is composed of every row generated by $\mathcal{A}$.

(a)

$\mathcal{A}$	$\Re^k_{a,b,c}$		
$\binom{3}{3,0,0}$	0	0	0
$\binom{3}{0,3,0}$	1	1	1
$\binom{3}{0,0,3}$	2	2	2
$\binom{3}{3,0,0}$	0	1	2
	1	2	0
	2	0	1
	0	2	1
	1	0	2
	2	1	0

(b)

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

Figure 3.2: Representation of a $CA(9; 2, 3, 3)$ in trinomial coefficients. (a) The candidate row subsets from trinomial coefficients. (b) The ternary covering array created.

3.3 Recursive methods

In this section we describe recursive constructions, recursive methods use small covering arrays as ingredients to construct larger instances.

3.3.1 Raise to a power the number of columns for any alphabet and any strength

Hartman (2005) presented a recursive construction which gives a method of squaring the number k of columns in a covering array of strength t while multiplying the rows N by a factor dependent only on t and v, but independent of k. This factor is related to the Turan numbers $T(t;v)$ that are defined to be the number of edges in the Turan graph. The Turan graph is the complete v-partite graph with t-vertices, having b parts of size $a+1$, and $v-b$ parts of size $a = \lfloor t/v \rfloor$ where $b = t - va$. Turan's theorem states that among all t-vertex graphs with no $v+1$ cliques, the Turan graph is the one with the most edges.

This method constructs a $CA(N_1(T(t,v)+1); t, k^2, v)$ Z from the below two ingredients:

1. A $CA(N_1; t, k, v)$ X
2. An $OA(N_; 2, T(t,v)+1, k)$ Y

Theorem 4 (Squaring covering arrays (Hartman, 2005)).

If $CAN(t,k,v) = N$ and there exist $T(t,v)-1$ mutually orthogonal Latin squares of side k (or equivalently $CAN(2,k,T(t,v)+1) = k^2$) then $CAN(t,k^2;v) \leq (T(t,v)+1)N$.

The procedure is as follows, see Theorem 4. Let X be a $CA(N;t,k,v)$ and let X^i be the i-*th* column of X. Let Y be an orthogonal array of strength 2 with $T(t,v)+1$ columns and entries from $\{1,2,\ldots,k\}$. We will construct a block array Z with k^2 columns and $T(t,v)+1$ rows. Each element in Z will be a column of X. Let X^Y be the block in the i-*th* row and j-*th* column of Z, see Figure 3.3.

If exists $X = CA(N;t,k,v)$, to raise to the power n the number of columns k, should be possible to construct $Y = CA(M;s,l,r)$ with the following properties.

1. r must have the cardinality k
2. $M = k^n$
3. $s = n$
4. $l = (n-1) \times T(v,t) + 1$

The procedure is similar to squaring. We will construct a block array Z with k^n columns and $N \times ((n-1) \times T(v,t)) + 1$ rows.

(a) $CA(N_1; t, k, v)$

$$\left(\begin{array}{c|c|c|c} X^1 & X^2 & \dots & X^k \end{array}\right)$$

(b) $OA(N_2; 2, T(t, v) + 1, k)$

$$\left(\begin{array}{c} y_{ij} \end{array}\right)$$

(c) $CA(N_1(T(t, v) + 1); t, k^2, v)$

$$\left(\begin{array}{c} z_{ij} = X^{y_{ji}} \end{array}\right)$$

Figure 3.3: The construction for Theorem 4.

(a) $CA(4; 2, 3, 2)$

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

(b) $OA(9; 2, 2, 3)^T$

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

(c) $CA(8; 2, 9, 2)$

$$\left(\begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{array}\right)$$

Figure 3.4: Example of the squaring covering arrays.

3.3.2 Products of covering arrays

Colbourn et al. (2006a) presented a product construction for $t = 2$. In general, the product of two covering arrays where $t = 2$ consists in obtaining a new covering array where the number of columns is equal to the product of the columns of the ingredients, and the number of rows is equal to the sum of the rows of each ingredient. The basic strategy of the product of covering arrays is described below.

When a covering array $CA(N_1; 2, k, v)$ and a covering array $CA(N_2; 2, \ell, v)$ both exist, it is an easy matter to construct a covering array $CA(N_1+N_2; 2, k\ell, v)$. To be specific, let $X = (x_{ij})$ be a $CA(N_1; 2, k, v)$ and let $Y = (y_{ij})$ be a $CA(N_2; 2, \ell, v)$. Form an $(N_1 + N_2) \times k\ell$ array $Z = (z_{i,j}) = X \otimes Y$ by setting $z_{i,(f-1)k+g} = x_{i,g}$ for $1 \leq i \leq N_1$, $1 \leq f \leq \ell$, and $1 \leq g \leq k$. Then set $z_{N+i,(f-1)k+g} = y_{i,f}$ for $1 \leq i \leq N_2$, $1 \leq f \leq \ell$, and $1 \leq g \leq k$. In essence, k copies of $Y = (y_{ij})$ are being appended to ℓ copies of $X = (x_{ij})$ as shown in Figure 3.5. Since two different columns of Z arise either from different columns of X or from two different columns of Y, the result is a covering array $CA(N_1 + N_2; 2, k\ell, v)$. Figure 3.6 shows the construction of the covering array $CA(9; 2, 12, 2)$ using as ingredients the covering arrays $CA(5; 2, 4, 2)$ and $CA(4; 2, 4, 2)$.

$$Z = \left(\begin{array}{cccc|cccc|c|cccc}
x_{11} & x_{12} & \cdots & x_{1k} & x_{11} & x_{12} & \cdots & x_{1k} & \cdots & x_{11} & x_{12} & \cdots & x_{1k} \\
x_{21} & x_{22} & \cdots & x_{2k} & x_{21} & x_{22} & \cdots & x_{2k} & \cdots & x_{21} & x_{22} & \cdots & x_{2k} \\
\vdots & & & & \vdots & & & & \cdots & \vdots & & & \\
x_{N_11} & x_{N_12} & \cdots & x_{N_1k} & x_{N_11} & x_{N_12} & \cdots & x_{N_1k} & \cdots & x_{N_11} & x_{N_12} & \cdots & x_{N_1k} \\
\hline
y_{11} & y_{11} & \cdots & y_{11} & y_{12} & y_{12} & \cdots & y_{12} & \cdots & y_{1\ell} & y_{1\ell} & \cdots & y_{1\ell} \\
y_{21} & y_{21} & \cdots & y_{21} & y_{22} & y_{22} & \cdots & y_{22} & \cdots & y_{2\ell} & y_{2\ell} & \cdots & y_{2\ell} \\
\vdots & & & & \vdots & & & & \cdots & \vdots & & & \\
y_{N_21} & y_{N_21} & \cdots & y_{N_21} & y_{N_22} & y_{N_22} & \cdots & y_{N_22} & \cdots & y_{N_2\ell} & y_{N_2\ell} & \cdots & y_{N_2\ell}
\end{array}\right)$$

Figure 3.5: Products of covering arrays, the structure of $X \otimes Y$.

(a) $CA(5; 2, 4, 2)$

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

(b) $CA(4; 2, 3, 2)$

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

(c) $CA(9; 2, 12, 2)$

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

Figure 3.6: Products of covering arrays of strength two.

3.3.3 Roux type constructions

Sloane (1993) published a method which improved some elements of the work reported in Roux (1987), see Theorem 5.

Theorem 5 (Roux (1987))**.**

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

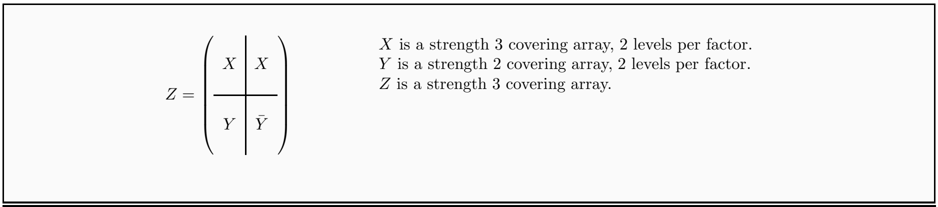

Figure 3.7: Original Roux construction.

This procedure constructs a $CAN(3, 2k, 2)$ by combining two covering arrays with the following characteristics: $CA(N_3; 3, k, 2)$ and $CA(N_2; 2, k, 2)$. It starts by appending $CA(N_2; 2, k, 2)$ to a $CA(N_3; 3, k, 2)$, which results into a $(N_3 + N_2) \times k$ array. Then this array is copied below itself, producing a $(N_3 + N_2) \times 2k$ array. Finally, the copied strength two array is replaced by its bit-complement array (i.e., switch 0 to 1 and 1 to 0). Figure 3.7 is an illustration of this construction. Figure 3.8 shows the construction of the covering $CA(13; 3, 8, 2)$ using as ingredients the covering arrays $CA(8; 3, 4, 2)$ and $CA(5; 2, 4, 2)$.

Theorem 6 proves a generalization of the Roux construction. It begins by placing two $CA(N_3; 3, k, v)$s side by side. We need a Y covering array $CA(N_2; 2, k, v)$ and a set π, where π is a cyclic permutation of the v symbols. Then for $1 \leq i \leq v - 1$, we append N_2 rows consisting of Y and $\pi^i(C)$ placed side by side. Figure 3.9 illustrates this construction. Figure 3.10 shows the construction of the covering array $CA(45; 3, 8, 3)$ using as ingredients the covering arrays $CA(27; 3, 4, 3)$ and $CA(9; 2, 4, 3)$.

Theorem 6 (Chateauneuf and Kreher (2002))**.**

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

(a) $CA(8;3,4,2)$

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

(b) $CA(5;2,4,2)$

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

(c) $CA(8+5;3,2\times 4,2)$

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

Figure 3.8: An example of the Roux construction. (a) shows a $CA(8;3,4,2)$. (b) shows a $CA(5;2,4,2)$. (c) shows the $CA(13;3,2,2)$ resulting from the Roux construction.

$$Z = \left(\begin{array}{c|c} X & X \\ \hline Y & Y^{\pi^1} \\ Y & Y^{\pi^2} \\ Y & Y^{\pi^{v-1}} \end{array}\right)$$

X is a $CA(N_3;3,k,v)$.
Y is a $CA(N_2;2,k,v)$ and $\pi = \{1,2,\ldots,v\}$ is a cyclic permutation of the v symbols.
Z is a $CA(N_3+(v-1)N_2;3,2k,v)$.

Figure 3.9: The construction for Theorem 6.

Cohen et al. (2008) generalized Theorem 6 to permit the number of factors to be multiplied by $l \leq 2$ rather than two, see Theorem 7; this is the *k-ary Roux construction*. To carry this out, it requires another combinatorial object. Let Γ be an Abelian group of order v, with $\odot$ as its binary operation.

Definition 13 (Difference Covering Array)**.**

A Difference Covering Array (DCA) $D = (d_{ij})$ over Γ, denoted by $DCA(N,\Gamma;2,k,v)$, is an $N \times k$ array with entries from Γ having the property that for any two distinct columns j and ℓ, $\{d_{ij} \odot d_{i\ell}^{-1} | 1 \leq i \leq N\}$ contains every non-identity element of Γ at least once. It denotes by $DCAN(2,k,v)$ the minimum N for which a $DCA(N,\Gamma;2,k,v)$ exists.

(a) $CA(27;3,4,3)$

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

(b) $CA(9;2,4,3)$

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

(c) $CA(27+2\times 9;3,2\times 4,3)$

$$\left(\begin{array}{cccc|cccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
2 & 2 & 2 & 0 & 2 & 2 & 2 & 0 \\
0 & 1 & 2 & 0 & 0 & 1 & 2 & 0 \\
1 & 2 & 0 & 0 & 1 & 2 & 0 & 0 \\
2 & 0 & 1 & 0 & 2 & 0 & 1 & 0 \\
0 & 2 & 1 & 0 & 0 & 2 & 1 & 0 \\
1 & 0 & 2 & 0 & 1 & 0 & 2 & 0 \\
2 & 1 & 0 & 0 & 2 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 2 & 2 & 1 & 1 & 2 & 2 & 1 \\
2 & 0 & 0 & 1 & 2 & 0 & 0 & 1 \\
0 & 2 & 0 & 1 & 0 & 2 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
2 & 1 & 2 & 1 & 2 & 1 & 2 & 1 \\
0 & 0 & 2 & 1 & 0 & 0 & 2 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
2 & 2 & 1 & 1 & 2 & 2 & 1 & 1 \\
0 & 2 & 2 & 2 & 0 & 2 & 2 & 2 \\
1 & 0 & 0 & 2 & 1 & 0 & 0 & 2 \\
2 & 1 & 1 & 2 & 2 & 1 & 1 & 2 \\
0 & 0 & 1 & 2 & 0 & 0 & 1 & 2 \\
1 & 1 & 2 & 2 & 1 & 1 & 2 & 2 \\
2 & 2 & 0 & 2 & 2 & 2 & 0 & 2 \\
0 & 1 & 0 & 2 & 0 & 1 & 0 & 2 \\
1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\
2 & 0 & 2 & 2 & 2 & 0 & 2 & 2 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 2 & 2 & 2 & 1 \\
2 & 2 & 2 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 2 & 1 & 1 & 2 & 0 & 2 \\
1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\
2 & 0 & 1 & 1 & 0 & 1 & 2 & 2 \\
0 & 2 & 1 & 2 & 1 & 0 & 2 & 0 \\
1 & 0 & 2 & 2 & 2 & 1 & 0 & 0 \\
2 & 1 & 0 & 2 & 0 & 2 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 2 \\
2 & 2 & 2 & 0 & 1 & 1 & 1 & 2 \\
0 & 1 & 2 & 1 & 2 & 0 & 1 & 0 \\
1 & 2 & 0 & 1 & 0 & 1 & 2 & 0 \\
2 & 0 & 1 & 1 & 1 & 2 & 0 & 0 \\
0 & 2 & 1 & 2 & 2 & 1 & 0 & 1 \\
1 & 0 & 2 & 2 & 0 & 2 & 1 & 1 \\
2 & 1 & 0 & 2 & 1 & 0 & 2 & 1
\end{array}\right)$$

Figure 3.10: An example of the Theorem 6. (a) shows a $CA(8;3,4,2)$. (b) shows a $CA(5;2,4,2)$. (c) shows the $CA(13;3,2,2)$ resulting from the Theorem 6 construction.

Definition 14.

A Covering Ordered Design (COD) denoted by $COD(N;t,k,v)$ is an $N\times k$ array such that every $N\times t$ subarray contains all non-constant t-tuples from v symbols at least once. We denote by $CODN(t,k,v)$ the minimum N for which a $COD(N;t,k,v)$ exists.

Definition 15.

A $QCA(N; k, \ell, v)$ is an $N \times k$ array with columns indexed by ordered pairs from $\{1, \dots, k\} \times \{1, \dots, \ell\}$, in which whenever $1 \leq i < j \leq k$ and $1 \leq a < b \leq \ell$, the $N \times 4$ subarray indexed by the four columns (i, a), (i, b), (j, b), (j, a) contains every 4-tuple (x, y, z, t) with $x - t \not\equiv y - z \pmod v$ at least once. $QCAN(k, \ell, v)$ denotes the minimum number of rows in such an array.

Theorem 7.

$CAN(3, k\ell, v) \leq CAN(3, k, v) + CAN(3, \ell, v) + CAN(2, \ell, v) \times DCAN(2, k, v)$.

To illustrate how this construction works, we suppose that all the following ingredients exist:

1. A covering array $CA(N_1; 3, \ell, v)$ W.
2. A covering array $CA(N_2; 3, k, v)$ X.
3. A covering array $CA(N_3; 2, \ell, v)$ Y.
4. A difference covering array $DCA(N_4; 2, k, v)$ U.

The result is a $CA(N_1 + N_2 + N_4N_3; 3, k\ell, v)$ Z, see Figure 3.11. Z is formed by vertical juxtaposing three arrays, Z_1 of size $N_1 \times k\ell$, Z_2 of size $N_2 \times k\ell$, and Z_3 of size $N_4N_3 \times k\ell$.

Z_1 is produced as follows. In row r and column (i, j) of Z_1 we place the entry in cell (r, j) of W. Thus Z_1 consists of k copies of W placed side by side.

Z_2 is produced as follows. In row r and column (i, j) of Z_2 we place the entry in cell (r, i) of X. Thus Z_2 consists of ℓ copies of the first column of X, then ℓ copies of the second column, and so on.

To construct Z_3, let $D = (d_{ij} | i = 1, \dots, N_4; j = 1 \dots, k)$ and $F = (f_{rs} | r = 1, \dots, N_3; s = 1, \dots, \ell)$. Choose a cyclic permutation π on the v symbols of the array. Then in row $(i-1)N_4 + r$ and column (j, s) of Z_3 place the entry $\pi^{d_{ij}}(f_{rs})$.

Martirosyan and Colbourn (2005) proposed recursive methods which generalize some Roux type constructions to produce a $CAN(t, 2k, v)$ for any $t \geq 4$ and $v \geq 2$. Some improvements to this procedure were presented later by Colbourn

$$
Z = \left(\begin{array}{c}
Z_1 = \left\{ \begin{array}{l} k \text{ copies of } W \\ N_1 \text{ rows} \end{array} \right. \\ \hline
Z_2 = \left\{ \begin{array}{l} \ell \text{ copies of } X \\ N_2 \text{ rows} \end{array} \right. \\ \hline
Z_3 = N_4 N_3 \text{ rows}
\end{array} \right)
$$

Figure 3.11: k-ary Roux construction.

et al. (2006b). The improved procedure permitted the authors to attain some of
628 the best-known bounds for binary covering arrays of strength four.

To describe how this construction works, we suppose that all the following ingre-
629 dients exist:

630 1. A covering array $CA(N_4; 4, k, v)$ C_4.
631 2. A covering array $CA(R_4; 3, \ell, v)$ B_4.
632 3. A covering array $DCA(S_1; 2, \ell, v)$ D_1.
633 4. A difference covering array $DCA(S_2; 2, k, v)$ D_2.
634 5. A covering ordered design $COD(N_3; 3, k, v)$ C_3.
635 6. A covering ordered design $COD(R_3; 3, \ell, v)$ B_3.
636 7. A difference covering array $QCA(M; k, \ell, v)$ G_5.

It produces a covering array $CA(N'; 4, k\ell, v)$ G where $N' = N_4 + R_4 + N_3S_1 +$
637 $R_3S_2 + M$. G is formed by vertically juxtaposing arrays G_1, G_2, G_3, G_4, and G_5.

- ▷ G_1 of size $N4 \times k\ell$. In row r and column (f, h) place the entry in cell (r, f)
638 of C_4. Thus G_1 consists of ℓ copies of C_4 placed side by side.
- ▷ G_2 of size $R_4 \times k\ell$. In row r and column (f, h) place the entry in cell (r, h)
of B_4. Thus G_2 consists of k copies of the first column of B_4, then k copies
639 of the second column, and so on.
- ▷ G_3 of size $N_3S_1 \times k\ell$. Index the N_3S_1 rows by ordered pairs from $\{1, \ldots, N_3\} \times$
$\{1, \ldots, S_1\}$. In row (r, s) and column (f, h) place $c_{r,f} + d_{s,h}$, where $c_{r,f}$ is
640 the entry in cell (r, f) of C_3 and $d_{s,h}$ is the entry in cell (s, h) of D_1.

- ▷ G_4 of size $R_3S_2 \times k\ell$. Index the S_2R_3 rows by ordered pairs from $\{1, \ldots, S_2\} \times \{1, \ldots, R_3\}$. In row (s, r) and column (f, h) place $b_{r,h} + d_{s,f}$, where $b_{r,h}$ is the entry in cell (r, h) of B_3 and $d_{s,f}$ is the entry in cell (s, f) of D_2.
- ▷ G_5 of size $M \times k\ell$.

3.4 Greedy methods

The majority of commercial and open source test data generating tools use greedy algorithms for covering arrays construction (AETG, TCG, ACTS and DDA), the greedy algorithms provide the fastest solving method.

3.4.1 Automatic Efficient Test Generator (AETG)

Cohen et al. (1996) presented a strategy called Automatic Efficient Test Generator (AETG). In AETG, covering arrays are constructed *one row at a time*. To generate a row, the first t-tuple is selected based on the one involved in most uncovered pairs. Remaining factors are assigned levels in a random order. Levels are selected based on the one that covers the most new t-tuples. For each row that is actually added to the covering array, there are a number, M, candidate rows that are generated and only a candidate that covers the most new t-tuples is added to the covering array. Once a covering is constructed, a number, R, of test suites are generated and the smallest test suite generated is reported. This process continues until all pairs are covered. Algorithm 1 shows the pseudocode of the AETG.

3.4.2 Test Case Generation (TCG)

Tung and Aldiwan (2000) proposed a tool called Test Case Generation (TCG). In TCG, one row is added at a time to a covering array until all pairs are covered. Before each row is added, a number of up to M candidate rows are generated and the best candidate (covering the most new pairs) is added. M is defined to be the maximum cardinality of factors (the maximum number of levels associated with any factor). To construct each row, factors are assigned levels in an order based on a non-ascending order of the cardinality of each factor. Each level for the factor is evaluated and a count of the number of pairs that are covered is used to determine whether or not to select a level for a factor. Algorithm 2 shows pseudocode for TCG.

Algorithm 1: AETG, Automatic Efficient Test Generator (Cohen et al., 1996).

```
begin
    set MinArray to ∞
    for i ← 1 to R do
        start with no tests in T
        N ← ∞
        while there are uncovered t-tuples in T do
            start with an empty test C and an empty test BestCandidate
            for j ← 1 to M do
                select the first pair that appears in the largest number of uncovered pairs
                while free factors remain do
                    randomly select a factor f
                    select a level v that is in the largest number of uncovered pairs with
                    uniform factors
                end while
                if C covers more t-tuples than BestCandidate then
                    BestCandidate ← C
                end if
            end for
            add test BestCandidate to T
            N ← N + 1
        end while
        if T has N < MinArray tests then
            MinArray ← N
            BestArray ← T
        end if
    end for
end
```

Algorithm 2: TCG, Test Case Generation (Tung and Aldiwan, 2000).

```
begin
    start with no tests in T
    sort factors in non-ascending order of cardinality
    while there are uncovered t-tuples in T do
        for i ← 1 to M do
            assign k0_{v_i} to k0
            for j ← 1 to k − 1 do
                select a level for k_i that covers the largest number of uncovered t-tuples in
                relation to uniform factors
                break ties by selecting the least recently used level
            end for
        end for
        add the candidate that covers the most uncovered t-tuples to T
    end while
end
```

3.4.3 Deterministic Density Algorithm (DDA)

Bryce and Colbourn (2007) presented an algorithm called Deterministic Density Algorithm (DDA). The DDA constructs one row of a covering array at a time using a steepest ascent approach. Factors are dynamically fixed one at a time in an order based on density. New rows are continually added until all interactions have been covered. The main advantage of DDA over other one-row-at-a-time methods is that it provides a worst-case logarithmic guarantee on the size N of the covering array. In order to make the previous discussion precise, consider the pseudocode in Algorithm 3. Four decisions must be made to instantiate this prototype:

1. *Factor density*, the manner in which densities are computed for factors
2. *Factor tie-breaking rule*, what tie-breaking is done when two or more maximum densities for factors are equal
3. *Level density*, the manner in which densities are calculated for levels
4. *Level tie-breaking rule*, what tie-breaking is done when two or more maximum densities for levels are equal

Algorithm 3: DDA, Deterministic Density Algorithm (Bryce and Colbourn, 2007).

```
begin
    start with empty test suite
    while uncovered pairs remain do
        compute factor density for each factor
        initialize new test with all factors not fixed
        while a factor remains whose level is not fixed do
            select such a factor f with largest density, using a factor tie-breaking rule
            compute level density for each level of factor f
            select a level ℓ for f with maximum density using a level tie-breaking rule
            fix factor f to level ℓ
            recompute densities for each factor
        end while
        add test to test suite
    end while
end
```

3.4.4 In-parameter-order (IPO)

Lei and Tai (1998) introduced a new algorithm called In-Parameter-Order (IPO), for pairwise testing. For a system with two or more input parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continues to do so for each additional parameter. Contrary to many other algorithms that build covering arrays *one row at a time*, the IPO strategy constructs them *one column at a time*. The extension of a test set for the addition of a new parameter includes the following two steps:

- ▷ *Horizontal growth*, which extends each existing test by adding one value of the new parameters
- ▷ *Vertical growth*, which adds new tests, if necessary, after the completion of horizontal growth

Assume that system $\mathcal{S}$ has parameters $f_1, f_2, \ldots, f_n$ with $n \leq 2$. Algorithm 4 shows IPO pseudocode for generating a pairwise test set T for $\mathcal{S}$.

Algorithm 4: IPO, In-Parameter-Order (Lei and Tai, 1998).

```
begin
    /* for the first two parameters f1 and f2                                  */
    T ← {(v1, v2) | v1 and v2 are values of f1 and f2 respectively}
    if n = 2 then
        stop
    end if
    /* for the remaining parameters                                           */
    for fi ← 3 to n do
        /* horizontal growth                                                  */
        foreach test(v1, v2, ..., vi−1) in T do
            replace it with v1, v2, ..., vi
            where vi is a value of fi
        end foreach
        /* vertical growth                                                    */
        while T does not cover all pairs between fi and each of f1, f2, ..., fi−1 do
            add a new test for f1, f2, ..., fi to T
        end while
    end for
end
```

Lei et al. (2008) introduced an algorithm for the efficient production of covering arrays, called In-Parameter-Order-General (IPOG), which generalizes the IPO strategy from pairwise testing to multi-way testing. The main idea is that covering arrays of $k-1$ columns can be used to efficiently build a covering array with degree k.

In order to construct a covering array, IPOG initializes a $v^t \times t$ matrix which contains each of the possible v^t distinct rows having entries from $\{0, 1, \ldots, v-1\}$. Then, for each additional column, the algorithm performs two steps, called *horizontal growth* and *vertical growth*. Horizontal growth adds an additional column to the matrix and fills in its values, then any remaining uncovered t-tuples are covered in the vertical growth stage. The choice of which rows will be extended with which values is made in a greedy manner: it picks an extension of the matrix that covers as many previously uncovered t-tuples as possible. Algorithm 5 shows the pseudocode of the IPOG algorithm. The algorithm takes two parameters:

1. An integer t specifying the strength of coverage

2. A parameter set ps containing the input parameters and their values

The output of this algorithm is a t-way test set for the parameters in set ps.

Algorithm 5: IPOG, In-Parameter-Order-General (Lei et al., 2008).

Input: Strenght t and set ps containing the input parameters and their values
Output: A t-way test set for the parameters in set ps
begin
  initialize test set ts to be an empty set
  sort the parameters in set ps in a non-increasing order of their domain sizes, and denote them as $P_1, P_2, \ldots, andP_k$
  add into test set ts a test for each combination of values of the first t parameters
  for $i = t + 1$ **to** k **do**
    let π be the set of all t-way combinations of values involving parameter P_i and any group of $(t-1)$ parameters among the first $i-1$ parameters
    `/* horizontal extension for parameter` P_i `*/`
    for $\tau = (v_1, v_2, \ldots, v_i - 1$ *in test set ts* **do**
      choose a value v_i of P_i and replace τ with τ' $= (v_1, v_2, \ldots, v_{i-1}, v_i)$ so that τ' covers the most number of combinations of values in π
      remove from ϕ the combinations of values covered by τ'
    end for
    `/* vertical extension for parameter` P_i `*/`
    for *each combinations* σ *in set* π **do**
      if *there exists a test* τ *in test set ts such that it can be changed to cover* σ **then**
        change test τ to cover σ
      else
        add a new test to cover σ
      end if
    end for
  end for
  return ts
end

IPOG is currently implemented in a software package called Advanced Combinatorial Testing System (ACTS), which was written in Java. Even if IPOG is a very fast algorithm for producing covering arrays it generally provides poorer quality results than other state-of-the-art algorithm like the algebraic procedures proposed by Chateauneuf and Kreher (2002).

3.4.5 Building-Block Algorithm (BBA)

Ronneseth and Colbourn (2009) introduced a new algorithm for constructing covering arrays, the Building-Block Algorithm (BBA). The BBA's fundamental idea is to combine smaller covering arrays by reordering the rows and then to append additional rows for the remaining uncovered pairs.

The BBA consists of four major steps:

1. Partition the k factors $\{f_1, f_2, \ldots, f_k\}$ into ϵ *factor groups* $\{G_1, G_2, \ldots, G_\epsilon\}$. Let $\phi(G_i)$ denote the collection of numbers of levels in the factors of G_i.
2. For each $1 \leq i \leq \epsilon$, construct M_i, an $MCA(n_i; t, \phi(G_i))$ called a *building block* for factor group G_i. All building blocks have the same strength as the original covering array. Let $\eta = max_{1 \leq i \leq \epsilon} n_i$.
3. Construct a *partial covering array*, $PMCA(\eta; t, k, (v_1 v_2 \ldots v_k))$, by combining the building blocks $M_1, M_2, \ldots, M_\epsilon$.
4. Complete the $PMCA(N; t, k, (v_1 v_2 \ldots v_k))$ by adding rows to cover the cross pairs left uncovered.

Several decisions must be made. An algorithm must choose ϵ and the assignment of the k factors to the ϵ factor groups. To construct the building blocks, an implementation of BBA may select any method (including applying itself recursively). The most important decision is how the rows are reordered and combined. Reordering can be done implicitly by selecting, for each factor group, an unused row from the corresponding building block. To do this, it can treat the factor groups in any order in order to select a row, and hence the algorithm must also determine in what order to consider the factor groups. The building blocks are rarely the same size, so the algorithm must decide, for building blocks M_i and M_j with $n_i > n_j$, how to combine rows in M_i with nonexistent rows in M_j after n_j rows have been fixed. If there are don not-care positions in the building blocks, the algorithm must also decide how and when to fix them.

Finally, additional rows are appended to cover as yet uncovered cross pairs. Algorithms that can complete a partial covering array are suitable, such as AETG, DDA, and TCG. Indeed, heuristic search approaches such as simulated annealing, tabu search, and hill climbing could complete the covering array. However, methods such as TConfig that generate an entire array, or IPO that adjoins factors, seem unsuited to this task. The entire algorithm is summarized in Algorithm 6.

3.4.6 Intersection Residual Pair Set Strategy (IRPS)

Younis et al. (2010) introduced a novel pairwise test data generation strategy called Intersection Residual Pair Set Strategy (IRPS). The IRPS for generating pairwise test data set takes the following steps:

1. Generates all pairs and stores them into compact linked list called Pi. For a test set with k parameters, the Pi list contains $(k-1)$ linked list. Each linked list contains nodes equal to the number of values defined by its parameter as well as an array of linked list that represents the pair of all other variables in the next linked lists.
2. Searches the Pi list and takes the desired weight of the candidate case as a test case then deletes it from the Pi list.

Algorithm 6: BBA, Building-Block Algorithm (Ronneseth and Colbourn, 2009).

```
begin
    Divide f_1, f_2, ..., f_k into ϵ factor groups G_1, G_2, ..., G_k
    for i ← 1 to ϵ do
        Compute M_i and mark all of its rows as unused
    end for
    while rows are unused in any building block do
        Mark all factor groups free
        for j ← 1 to ϵ do
            Select free factor group f to fix
            Select unused row from M_f to use
        end for
        Add newly created row to MCA
    end while
    Cover remaining uncovered tuples in MCA by adding additional test rows
end
```

3. Repeats step 2 until the *Pi* list is empty.

The generated pairs are stored in compact linked list called *Pi*, which is a linked list of linked lists. For a test set with k parameters, the *Pi* list contains $(k-1)$ linked list. Each linked list contains nodes equal to the number of values defined by its parameter as well as an array of linked list that represents the pair of all other variables in the next linked lists.

To understand how the *Pi* list works, consider a system with $t = 2$, $k = 4$, and $v = 3$, see Table 3.3. In this example, we have $\binom{4}{2}3^2 = 54$ possible pairs of combinations.

Table 3.3: Example for a system with $t = 2$, $k = 4$, and $v = 3$.

A	B	C	D
a_0	b_0	c_0	d_0
a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2

Table 3.4 shows the *Pi* linked list. Node $a0$ with the pairs linked list array contains the following pairs $(\{a0, b0\}, \{a0, b1\}, \{a0, b2\}, \ldots, \{a0, d2\})$. Here, this list contains only pairs that are based on a0. Similarly, the same observation can be seen with other nodes in the lists.

To describe the IRPS in detail, it is necessary to define a number of terminologies. The *weight* of the candidate test case is defined as the number of pairs that are covered by that candidate. For example, the test case combination of $a_0b_0c_0d_0$

Table 3.4: *Pi* linked list for storing combination pairs for a system with $t = 2$, $k = 4$, and $v = 3$.

index					
0		1		2	
a_0	b_0 b_l b_2 c_0 c_l c_2 d_0 d_l d_2	b_0	c_0 c_1 c_2 d_0 d_1 d_2	c_0	d_0 d_1 d_2
a_1	b_0 b_1 b_2 c_0 c_1 c_2 d_0 d_1 d_2	b_1	c_0 c_1 c_2 d_0 d_1 d_2	c_1	d_0 d_1 d_2
a_2	b_0 b_1 b_2 c_0 c_1 c_2 d_0 d_1 d_2	b_2	c_0 c_1 c_2 d_0 d_1 d_2	c_2	d_0 d_1 d_2

covers the pairs ($\{a_0, b_0\}$,$\{a_0, c_0\}$,$\{a_0, d_0\}$,$\{b_0, c_0\}$,$\{b_0, d_0\}$, and $\{c_0, d_0\}$) and the variables b_0, c_0, d_0 in node a_0, c_0, d_0 in node b_0, and finally d_0 in node c_0, so its $weight = 6$. The *maximum weight*, $wmax$, for k parameters can be calculated by (3.4):

$$wmax = \frac{k \times (k-1)}{2}. \tag{3.4}$$

Here, if $k = 4$, then $wmax = 4 \times 3/2 = 6$. The *miss* variable is defined as the difference between the *maximum weight* and the *weight* of the candidate test case. The intersection of node in the list i with the list $(i + 1)$ is defined as the intersection between the node and all nodes given by the first row. IRPS constructs a double linked list that stores the original i node and the intersection with the second node in $i + 1$ list, as well as the rest of the nodes. If the first row in the pairs array is empty, the intersection process will be performed with all values of the nodes in the next list and the miss variable is reduced by one (if $miss > 0$). Otherwise, the intersection process will be terminated and the iteration moves to the next node. The candidate test case is obtained by taking the node value in each node in the double linked list. For the last node, the candidate test case takes the current value and the first element in the pair array. The candidate test case is taken as a test case only if its weight satisfies the desired weight criteria. If not, the intersection process will continue with the other nodes in the list (by deleting the last node in the double linked list and replace it with the intersection with next node in the list, or when there is no next node in the list, the strategy will delete the last two nodes and continue with the iteration). In other words, the intersection process goes horizontally when the target weight is not found and grows vertically in recursive fashion. Finally, the *delete* operation operates by deleting each variable (if they exist) in each node. Table 3.5 shows the test set constructed using the IRPS construction and the values from Table 3.3 and the structure Table 3.4.

Table 3.5: The constructed test set.

No.	Test case				*miss*	*weight*
1	a_0	b_0	c_0	d_0	0	6
2	a_0	b_l	c_l	d_l	0	6
3	a_0	b_2	c_2	d_2	0	6
4	a_1	b_0	c_1	d_2	0	6
5	a_1	b_1	c_2	d_0	0	6
6	a_1	b_2	c_0	d_1	0	6
7	a_2	b_0	c_2	d_1	0	6
8	a_2	b_1	c_0	d_2	0	6
9	a_2	b_2	c_1	d_0	0	6

3.5 Metaheuristic methods

Some stochastic algorithms in artificial intelligence, such as *tabu search* (Nurmela, 2004; Gonzalez-Hernandez et al., 2010), *simulated annealing* (Cohen et al., 2003), *generic algorithms* and *ant colony optimization algorithm* (Shiba et al., 2004) provide an effective way to find approximate solutions. In these algorithms, the optimization focuses on one value of N at a time, attempting to find a covering array for that size.

3.5.1 Tabu search (TS)

Tabu Search (TS) metaheuristic is a local search optimization approach that copes with different problems of combinatorial optimization. The TS was proposed by Glover, 1986. The overall approach is to avoid entrainment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited (hence “tabu”). The TS method was partly motivated by the observation that human behavior appears to operate with a random element that leads to inconsistent behavior given similar circumstances. As Glover points out, the resulting tendency to deviate from a charted course, might be regretted as a source of error but can also prove to be source of gain. The TS method operates in this way with the exception that new courses are not chosen randomly. Instead the TS proceeds according to the supposition that there is no point in accepting a new poor solution unless it is to avoid a path already investigated. This insures new regions of a problems solution space will be investigated in with the goal of avoiding local minima and ultimately finding the desired solution.

The TS begins by marching to a local minima. To avoid retracing the steps used, the method records recent moves in one or more *tabu list*. The original intent of the list was not to prevent a previous move from being repeated, but rather to insure it was not reversed. The tabu lists are historical in nature and form the TS memory. The role of the memory can change as the algorithm proceeds. At initialization the goal is make a coarse examination of the solution space, known as "diversification", but as candidate locations are identified the search is more focused to produce local optimal solutions in a process of "intensification". In many cases the differences between the various implementations of the TS method have to do with the size, variability, and adaptability of the TS memory to a particular problem domain.

TS has been used successfully by Nurmela (2004) for finding covering arrays. This algorithm starts with an $N \times k$ randomly generated matrix that represents a potential covering array. The number of uncovered t-tuples is used to evaluate the cost of a candidate solution $(matrix)^2$. Next an uncovered t-tuple is selected at random and the rows of the matrix are searched to find those that require only the change of a single element in order to cover the selected t-tuple. These changes, called moves, correspond to the neighboring solutions of the current candidate solution. The variation of cost corresponding to each such move is calculated and the move having the smallest cost is selected, provided that the move is not tabu. If there are several equally good non-tabu moves, one of them is randomly chosen. Then another uncovered t-tuple is selected and the process is repeated until a matrix with zero cost (a covering array) is found or a predefined maximum number of moves is reached. The tabu condition prevents changing an element of the matrix, if it has been changed during the last T moves. This feature prevents looping and increases the exploration capacity of the algorithm.

The results have demonstrated that Nurmela's TS implementation is able to slightly improve some previous best-known solutions. However, an important drawback of this algorithm is that it consumes considerably much more computational time than any of the previously presented algorithms.

Walker II and Colbourn (2009) presented another TS implementation for covering arrays construction. It employs a compact representation of covering arrays based on permutation vectors and Covering Perfect Hash Families (CPHF) (Seroussi and Bshouty, 1988) in order to reduce the size of the search space. Using this algorithm, improved covering arrays of strengths three to five have been found, as well as the first arrays of strength six and seven found by computational search.

Gonzalez-Hernandez et al. (2010) proposed a TS approach to construct MCA. The key features of their TS implementation are: the use of mixture of three neighborhood functions to create neighbors, an efficient calculation of the objective function and a novel initialization function. Given that the performance of TS depends on the values of the probabilities assigned, they presented a fine tuning of the probabilities configurations based on a complete test set of discretized probabilities. The evaluation function $C(s)$ of a solution s is defined as the number of combination of symbols missing in the matrix M. Then, the expected solution will be zero missing. The pseudocode of their TS is shown in Algorithm 7. In this algorithm the function $F(s, \rho_1, \rho_2, \rho_3)$ makes a roulette-wheel selection with the values ρ_1, ρ_2, ρ_3; the result will indicate which neighborhood function will be used to create a neighbor. The function $NumEvalRequired(s, \rho_1, \rho_2, \rho_3)$ will determine the number of evaluations performed by the neighborhood function used by $F(s, \rho_1, \rho_2, \rho_3)$ to create a new neighbor.

Algorithm 7: MiTS, Tabu Searh for constructing mixed covering arrays (Gonzalez-Hernandez et al., 2010).

```
begin
    s ← s_0
    s_best ← s
    while C(s_best) > 0 and e < ℰ do
        s' ← F(s, ρ1, ρ2, ρ3)
        if C(s') ¡ C(s_best) then
            s_best ← s'
        end if
        if NotInTabuList(s') then
            s ← s'
            UpdateTebuList(s, s')
        end if
        e ← NumEvalRequired(s, ρ1, ρ2, ρ3)
    end while
end
```

Their TS approach was compared against IPOG using a benchmark taken from the literature. Their TS implementation improved the size of the matrices in comparison with the ones constructed by IPOG, finding the optimal solution in all the cases considered.

3.5.2 Ant colony optimization (ACO

Ant Colony Optimization (ACO) is a metaheuristic algorithm for the approximate solution of combinatorial optimization problems that has been inspired by the foraging behavior of real ant colonies proposed by Dorigo et al. (1996). The structured behavior of an ant colony is possible by a chemical substance called pheromone, which establish the best possible route from the colony to their food source. Real ants are capable of finding the shortest trajectory from a food source to their nest, without using visual cues by exploiting pheromone information. While walking, ants deposit pheromone on the ground, and follow, in probability, pheromone previously deposited by other ants.

The computational method follows the ant behavior by giving more pheromone to better solutions. Shiba et al. (2004) proposed the ACO to generate test cases using a one-test-at-a-time approach. In their algorithm a test case can be represented as a route from a starting point to the final objective. A given amount of ants start their travel to the final objective. Each time an ant reach to its final objective, it deposits a certain quantity of pheromone to each point visited. When a new ant starts, it will prefer those points where the scent of the pheromone is stronger.

3.5.3 Simulated annealing (SA)

Simulated Annealing (SA) is a general-purpose stochastic optimization method that has proven to be an effective tool for approximating globally optimal solutions to many types of NP-hard combinatorial optimization problems.

SA is a randomized local search method based on the simulation of annealing of metal. The acceptance probability of a trial solution is given by (3.5), where T is the *temperature* of the system, ΔE is the difference of the costs between the trial and the current solutions (the cost change due to the perturbation), (3.5) means that the trial solution is accepted by nonzero probability $e^{(-\Delta E/T)}$ even though the solution deteriorates (*uphill move*).

$$(P) = \begin{cases} 1 & if \Delta E < 0 \\ e^{(-\frac{\Delta E}{T})} & otherwise \end{cases} \tag{3.5}$$

Uphill moves enable the system to escape from the local minima; without them, the system would be trapped into a local minimum. Too high of a probability for the occurrence of uphill moves, however, prevents the system from converging. In SA, the probability is controlled by temperature in such a manner that at the beginning of the procedure the temperature is sufficiently high, in which a high probability is available, and as the calculation proceeds the temperature is gradually decreased, lowering the probability (Jun and Mizuta, 2005).

Stardom (2001) made a study of different metaheuristics including SA, TS and Genetic Algorithms (GA). He used a matrix of size $N \times k$ to represent the solution. His comparisons suggest that the SA algorithm was the best option to solve the CAC problem. Besides the quality of the obtained covering arrays are frequently optimal or near optimal, the output result from a SA algorithm depends directly on the selected number of rows. This is, the SA algorithm needs the parameter N for the required covering array to be searched. An extensive search must be performed for looking the best value for parameter N if that parameter is not set as an input parameter to the SA algorithm.

A SA metaheuristic has been applied by Cohen et al. (2003) for constructing covering arrays. Their SA implementation starts with a randomly generated initial solution M which cost $E(M)$ is measured as the number of uncovered t-tuples. A series of iterations is then carried out to visit the search space according to a neighborhood. At each iteration, a neighboring solution M' is generated by changing the value of the element $m_{i,j}$ by a different legal member of the alphabet in the current solution M. The cost of this iteration is evaluated as $\Delta E = E(M') - E(M)$. If ΔE is negative or equal to zero, then the neighboring solution M' is accepted. Otherwise, it is accepted with probability $P(\Delta E) = e^{-\Delta E/T_n}$, where T_n is determined by a cooling schedule. In their implementation, Cohen et al. use a simple linear function $T_n = 0.9998T_{n-1}$ with an initial temperature fixed at $T_i = 0.20$. At each temperature, 2000 neighboring solutions are generated. The algorithm stops either if a valid covering array is found, or if no change in the cost of the current solution is observed after 500 trials. The authors justify their choice of these parameter values based on some experimental tuning. They conclude that their SA implementation is able to produce smaller covering arrays than other computational methods, sometimes improving upon algebraic constructions. However, they also indicate that their SA algorithm fails to match the algebraic constructions for larger problems, especially when $t = 3$.

Cohen et al. (2008) presented a hybrid metaheuristic called Augmented Annealing. It employs recursive and direct combinatorial constructions to produce small building blocks which are then augmented with a simulated annealing algorithm to construct a covering array. This method has been successfully used to construct covering arrays that are smaller than those created by using their simple SA algorithm (Cohen et al., 2003).

Martinez-Pena et al. (2010) propose a SA algorithm for the construction of ternary covering arrays using a trinomial coefficient representation. This algorithm implements the following key features:

1. A novel representation of the search space using trinomial coefficients.
2. A mixture of neighborhood functions. A set of four neighborhood functions were implemented. They were able to form the ternary covering arrays by exploring and exploiting diverse zones of the search space.
3. An evaluation function that guides the search process. The evaluation function measures the number of missing combinations and the quality of the solution.

In order to provide a good global performance of the SA algorithm, they followed a fine tuning methodology for optimizing the assigned probabilities of execution for each of the four neighborhood functions using a linear Diophantine equation. The results obtained with this algorithm show that the best values of N were given by this SA implementation than the IPOG algorithm for all the instances $CA(t, t+1, 3)$ and $CA(t, t+2, 3)$. However, for a degree $k \geq t+3$ the size of the $CA(t, k, 3)$ instances obtained through IPOG were better.

Rodriguez-Tello and Torres-Jimenez (2009) present a new Memetic Algorithm (MA) designed to compute near-optimal solutions for the CAC. It incorporates several distinguished features including an efficient heuristic to generate a good quality initial population, and a local search operator based on a fine tuned simulated annealing algorithm employing a carefully designed compound neighborhood. From the data presented in this work the authors make the next observations: first, the solution quality attained by the proposed MA is very competitive with respect to that produced by the state-of-the-art techniques; second, in their experiment the IPOG procedure returns poorer quality solutions than their MA in 19 out 20 benchmark instances. Indeed, IPOG produces covering arrays which are in average 73.16% worst than those constructed with a MA.

3.6 Construction of orthogonal arrays of index unity using logarithm tables for Galois fields

A wide variety of problems found in computer science deals with combinatorial objects. Combinatorics is the branch of mathematics that deals with finite countable objects called combinatorial structures. These structures find many applications in different areas such as hardware and software testing, cryptography, pattern recognition, computer vision, among others.

Of particular interest in this section are the combinatorial objects called Orthogonal Arrays(OAs). These objects have been studied given of their wide range of applications in the industry, Gopalakrishnan and Stinson (2006) present their applications in computer science; among them are in the generation of error correcting codes presented by (Hedayat et al., 1999; Stinson, 2004), or in the design of experiments for software testing as shown by Taguchi (1994).

To motivate the study of the orthogonal arrays, it is pointed out their importance in the development of algorithms for the cryptography area. There, orthogonal arrays have been used for the generation of authentication codes, error correcting codes, and in the construction of universal hash functions (Gopalakrishnan and Stinson, 2006).

This section proposes an efficient implementation for the Bush's construction (Bush, 1952) of orthogonal arrays of index unity, based on the use of logarithm tables for Galois Fields. This is an application of the algorithm of Torres-Jimenez et al. (2011a). The motivation of this research work born from the applications of orthogonal arrays in cryptography as shown by Hedayat et al. (1999). Also, it is discussed an alternative use of the logarithm table algorithm for the construction of cyclotomic matrices to construct covering arrays (Colbourn, 2010).

3.6.1 The Bush's construction

The Bush's construction is used to construct $OA(v^t; t, v+1, v)$, where $v = p^n$ is a prime power. This construction considers all the elements of the Galois Field $GF(v)$, and all the polynomials $y_j(x) = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \ldots + a_1x + a_0$, where $a_i \in GF(v)$. The number of polynomials $y_j(x)$ are v^t, due to the fact that there are v different coefficients per each of the t terms.

Let's denote each element of $GF(v)$ as e_i, for $0 \leq i \leq v-1$. The construction of an orthogonal array following the Bush's construction is done as follow:

1. Generate a matrix $\mathcal{M}$ formed by v^t rows and $v+1$ columns;
2. Label the first v columns of $\mathcal{M}$ with an element $e_i \in GF(v)$;
3. Label each row of $\mathcal{M}$ with a polynomial $y_j(x)$;
4. For each cell $m_{j,i} \in \mathcal{M}$, $0 \leq j \leq v^t - 1, 0 \leq i \leq v-1$, assign the value u whenever $y_j(e_i) = e_u$ (i.e., evaluates the polynomial $y_j(x)$ with $x = e_i$ and determines the result in the domain of $GF(v)$); and
5. Assign value u in cell $m_{j,i}$, for $0 \leq j \leq v^t - 1, i = v$, if e_u is the leading coefficient of $y_j(x)$, i.e., $e_u = a_{t-1}$ in the term $a_{t-1}x^{t-1}$ of the polynomial $y_j(x)$.

The constructed matrix $\mathcal{M}$ following the previous steps is an orthogonal array. We point out in this moment that the construction requires the evaluation of the polynomials $y_j(x)$ to construct the orthogonal array. The following subsection describes the general idea of the algorithm that does this construction with an efficient evaluation of these polynomials.

This section presented a survey of some construction reported in the scientific literature that are used to generate orthogonal arrays. The following section will present an algorithm for the generation of logarithm tables of finite fields.

3.6.2 Algorithm for the construction of logarithm tables of Galois fields

In Barker (1986) a more efficient method to multiply two polynomials in $GF(p^n)$ is presented. The method is based on the definition of logarithms and antilogarithms in $GF(p^n)$. According with Niederreiter (1990), given a primitive element ρ of a finite field $GF(p^n)$, the discrete logarithm of a nonzero element $u \in GF(p^n)$ is that integer k, $1 \leq k \leq p^n - 1$, for which $u = \rho^k$. The antilogarithm for an integer k given a primitive element ρ in $GF(p^n)$ is the element $u \in GF(p^n)$ such that $u = \rho^k$. Table 3.6 shows the table of logarithms and antilogarithms for the elements $u \in GF(3^2)$ using the primitive element $x^2 = 2x+1$; column 1 shows the elements in $GF(3^2)$ (the antilogarithm) and column 2 the logarithm.

Table 3.6: Logarithm table of $GF(3^2)$ using the primitive element $2x + 1$.

Element $u \in GF(p^n)$	$\log_{2x+1}(u)$
1	0
x	1
$2x + 1$	2
$2x + 2$	3
2	4
$2x$	5
$x + 2$	6
$x + 1$	7

Using the definition of logarithms and antilogarithms in $GF(p^n)$, the multiplication between two polynomials $\mathcal{P}_1(x)\mathcal{P}_2(x) \in GF(p^n)$ can be done using their logarithms $l_1 = log(\mathcal{P}_1(x)), l_2 = log(\mathcal{P}_2(x))$. First, the addition of logarithms $l_1 + l_2$ is done and then the antilogarithm of the result is computed.

Torres-Jimenez et al. (2011a) proposed an algorithm for the construction of logarithm tables for Galois Fields $GF(p^n)$. The pseudocode is shown in Algorithm 8. The algorithm simultaneously finds a primitive element and constructs the logarithm table for a given $GF(p^n)$.

Algorithm 8: `BuildLogarithmTable(p,n)`, an algorithm for the construction of logarithm tables for Galois fields $GF(p^n)$ (Torres-Jimenez et al., 2011a).

Input: A prime number p and a power n.
Output: $\mathcal{L}$ logarithm table.

begin
  foreach $\rho \in GF(p^n) - 0$ **do**
    $\mathcal{L} \leftarrow \varnothing$
    $\mathcal{P}(x) \leftarrow 1$
    $k \leftarrow 0$
    while $(\mathcal{P}(x), k) \notin \mathcal{L}$ **and** $k < p^n - 1$ **do**
      $\mathcal{L} \leftarrow \mathcal{L} \bigcup (\mathcal{P}(x), k)$
      $k \leftarrow k + 1$
      $\mathcal{P}(x) \leftarrow p \cdot \mathcal{P}(x)$
    end while
    if $k = p^n - 1$ **then return** ρ
  end foreach
  return $\mathcal{L}$
end

Now, it follows the presentation of the core of this chapter, the efficient implementation of the Bush construction for orthogonal arrays, based on a modification of the algorithm presented in this section.

3.6.3 Efficient construction of orthogonal arrays

The idea that leads to an efficient construction of orthogonal arrays through the Bush's construction relies on the algorithm proposed in (Torres-Jimenez et al., 2011a). This algorithm computes the logarithm tables and the primitive element of a given Galois Field $GF(v)$. In this chapter, it is proposed an extension of this algorithm such that it can be used in combination with the Bush's construction to efficiently construct orthogonal arrays of index unity. The result is an algorithm that uses only additions and modulus operations to evaluate the polynomials $y_j(x)$.

Let's show an example of this contribution. Suppose that it is wanted to construct the $OA(4^3; 3, 5, 4)$. This array has an alphabet $v = p^n = 2^2 = 4$ and size 64×5. To construct it, it is required the polynomial $x + 1$ as the primitive element of $GF(2^2)$, and the logarithm table shown in Table 3.7(a) (both computed using the algorithm in (Torres-Jimenez et al., 2011a)). Table 3.7(b) is a modified version of the logarithm table that contains all the elements $e_i \in GF(2^2)$ (this includes e_0, the only one which can not be generated by powers of the primitive element).

Table 3.7: Logarithm table for $GF(2^2)$, with primitive element $x + 1$.

(a)

Power	**Polynomial in GF(2^2)**
0	1
1	x
2	$x+1$

(b)

Element $e_i \in$ GF(2^2)	**Polynomial in GF(2^2)**
e_0	0
e_1	1
e_2	x
e_3	$x+1$

The following step in the construction of the orthogonal array is the construction of the matrix $\mathcal{M}$. For this purpose, firstly it is labeled its first v columns with the elements $e_i \in GF(2^2)$; after that, the rows are labeled with all the polynomials of maximum degree 2 and coefficients $e_j \in GF(2^2)$. Next, it is defined the integer value u for each cell $m_{j,i} \in \mathcal{M}$, where $0 \leq j \leq v^t - 1$ and $0 \leq i \leq v - 1$, as the one satisfying $y_j(e_i) = e_u$. Finally, it is generated the values of cell $m_{j,i}$, where the column $i = v$, using the value of the leading coefficient of the polynomial $y_j(x)$, for each $0 \leq j \leq v^t - 1$. Table 3.8 shows part of the construction of the $OA(4^3; 3, 5, 4)$ through this method.

During the definition of values e_u, the polynomials $y_j(e_i)$ must be evaluated. For example, the evaluation of the polynomial $y_{14} = e_3 x + e_1$ at value $x = e_2$ yields $y_{14}(e_2) = e_3 x + e_1 = e_3 \cdot e_2 + e_1 = e_0$. To obtain the result e_0 it is necessary to

Table 3.8: Example of a partial construction of the $OA(4^3; 3, 4, 5)$, using the Bush's construction.

$\mathcal{M}$		**Elements of $GF(2^2)$**				
		e_0	e_1	e_2	e_3	
	$y_j(x)$ **Polynomial**	**0**	**1**	**x**	**x + 1**	
0	e_0	$\{u\|y_0(e_0) = e_u\}$	$\{u\|y_0(e_1) = e_u\}$	$\{u\|y_0(e_2) = e_u\}$	$\{u\|y_0(e_3) = e_u\}$	e_0
1	e_1	$\{u\|y_1(e_0) = e_u\}$	$\{u\|y_1(e_1) = e_u\}$	$\{u\|y_1(e_2) = e_u\}$	$\{u\|y_1(e_3) = e_u\}$	e_0
2	e_2	$\{u\|y_2(e_0) = e_u\}$	$\{u\|y_2(e_1) = e_u\}$	$\{u\|y_2(e_2) = e_u\}$	$\{u\|y_2(e_3) = e_u\}$	e_0
3	e_3	$\{u\|y_3(e_0) = e_u\}$	$\{u\|y_3(e_1) = e_u\}$	$\{u\|y_3(e_2) = e_u\}$	$\{u\|y_3(e_3) = e_u\}$	e_0
4	e_1x	$\{u\|y_4(e_0) = e_u\}$	$\{u\|y_4(e_1) = e_u\}$	$\{u\|y_4(e_2) = e_u\}$	$\{u\|y_4(e_3) = e_u\}$	e_0
5	$e_1x + e_1$	$\{u\|y_5(e_0) = e_u\}$	$\{u\|y_5(e_1) = e_u\}$	$\{u\|y_5(e_2) = e_u\}$	$\{u\|y_5(e_3) = e_u\}$	e_0
6	$e_1x + e_2$	$\{u\|y_6(e_0) = e_u\}$	$\{u\|y_6(e_1) = e_u\}$	$\{u\|y_6(e_2) = e_u\}$	$\{u\|y_6(e_3) = e_u\}$	e_0
7	$e_1x + e_3$	$\{u\|y_7(e_0) = e_u\}$	$\{u\|y_7(e_1) = e_u\}$	$\{u\|y_7(e_2) = e_u\}$	$\{u\|y_7(e_3) = e_u\}$	e_0
8	e_2x	$\{u\|y_8(e_0) = e_u\}$	$\{u\|y_8(e_1) = e_u\}$	$\{u\|y_8(e_2) = e_u\}$	$\{u\|y_8(e_3) = e_u\}$	e_0
9	$e_2x + e_1$	$\{u\|y_9(e_0) = e_u\}$	$\{u\|y_9(e_1) = e_u\}$	$\{u\|y_9(e_2) = e_u\}$	$\{u\|y_9(e_3) = e_u\}$	e_0
10	$e_2x + e_2$	$\{u\|y_{10}(e_0) = e_u\}$	$\{u\|y_{10}(e_1) = e_u\}$	$\{u\|y_{10}(e_2) = e_u\}$	$\{u\|y_{10}(e_3) = e_u\}$	e_0
11	$e_2x + e_3$	$\{u\|y_{11}(e_0) = e_u\}$	$\{u\|y_{11}(e_1) = e_u\}$	$\{u\|y_{11}(e_2) = e_u\}$	$\{u\|y_{11}(e_3) = e_u\}$	e_0
12	e_3x	$\{u\|y_{12}(e_0) = e_u\}$	$\{u\|y_{12}(e_1) = e_u\}$	$\{u\|y_{12}(e_2) = e_u\}$	$\{u\|y_{12}(e_3) = e_u\}$	e_0
13	$e_3x + e_1$	$\{u\|y_{13}(e_0) = e_u\}$	$\{u\|y_{13}(e_1) = e_u\}$	$\{u\|y_{13}(e_2) = e_u\}$	$\{u\|y_{13}(e_3) = e_u\}$	e_0
14	$e_3x + e_2$	$\{u\|y_{14}(e_0) = e_u\}$	$\{u\|y_{14}(e_1) = e_u\}$	$\{u\|y_{14}(e_2) = e_u\}$	$\{u\|y_{14}(e_3) = e_u\}$	e_0
15	$e_3x + e_3$	$\{u\|y_{15}(e_0) = e_u\}$	$\{u\|y_{15}(e_1) = e_u\}$	$\{u\|y_{15}(e_2) = e_u\}$	$\{u\|y_{15}(e_3) = e_u\}$	e_0
16	e_1x^2	$\{u\|y_{16}(e_0) = e_u\}$	$\{u\|y_{16}(e_1) = e_u\}$	$\{u\|y_{16}(e_2) = e_u\}$	$\{u\|y_{16}(e_3) = e_u\}$	e_1
17	$e_1x^2 + e_1$	$\{u\|y_{17}(e_0) = e_u\}$	$\{u\|y_{17}(e_1) = e_u\}$	$\{u\|y_{17}(e_2) = e_u\}$	$\{u\|y_{17}(e_3) = e_u\}$	e_1
18	$e_1x^2 + e_2$	$\{u\|y_{18}(e_0) = e_u\}$	$\{u\|y_{18}(e_1) = e_u\}$	$\{u\|y_{18}(e_2) = e_u\}$	$\{u\|y_{18}(e_3) = e_u\}$	e_1
19	$e_1x^2 + e_3$	$\{u\|y_{19}(e_0) = e_u\}$	$\{u\|y_{19}(e_1) = e_u\}$	$\{u\|y_{19}(e_2) = e_u\}$	$\{u\|y_{19}(e_3) = e_u\}$	e_1
20	$e_1x^2 + e_1x$	$\{u\|y_{20}(e_0) = e_u\}$	$\{u\|y_{20}(e_1) = e_u\}$	$\{u\|y_{20}(e_2) = e_u\}$	$\{u\|y_{20}(e_3) = e_u\}$	e_1
21	$e_1x^2 + e_1x + e_1$	$\{u\|y_{21}(e_0) = e_u\}$	$\{u\|y_{21}(e_1) = e_u\}$	$\{u\|y_{21}(e_2) = e_u\}$	$\{u\|y_{21}(e_3) = e_u\}$	e_1
22	$e_1x^2 + e_1x + e_2$	$\{u\|y_{22}(e_0) = e_u\}$	$\{u\|y_{22}(e_1) = e_u\}$	$\{u\|y_{22}(e_2) = e_u\}$	$\{u\|y_{22}(e_3) = e_u\}$	e_1
23	$e_1x^2 + e_1x + e_3$	$\{u\|y_{23}(e_0) = e_u\}$	$\{u\|y_{23}(e_1) = e_u\}$	$\{u\|y_{23}(e_2) = e_u\}$	$\{u\|y_{23}(e_3) = e_u\}$	e_1
24	$e_1x^2 + e_2x$	$\{u\|y_{24}(e_0) = e_u\}$	$\{u\|y_{24}(e_1) = e_u\}$	$\{u\|y_{24}(e_2) = e_u\}$	$\{u\|y_{24}(e_3) = e_u\}$	e_1
25	$e_1x^2 + e_2x + e_1$	$\{u\|y_{25}(e_0) = e_u\}$	$\{u\|y_{25}(e_1) = e_u\}$	$\{u\|y_{25}(e_2) = e_u\}$	$\{u\|y_{25}(e_3) = e_u\}$	e_1
26	$e_1x^2 + e_2x + e_2$	$\{u\|y_{26}(e_0) = e_u\}$	$\{u\|y_{26}(e_1) = e_u\}$	$\{u\|y_{26}(e_2) = e_u\}$	$\{u\|y_{26}(e_3) = e_u\}$	e_1
27	$e_1x^2 + e_2x + e_3$	$\{u\|y_{27}(e_0) = e_u\}$	$\{u\|y_{27}(e_1) = e_u\}$	$\{u\|y_{27}(e_2) = e_u\}$	$\{u\|y_{27}(e_3) = e_u\}$	e_1
28	$e_1x^2 + e_3x$	$\{u\|y_{28}(e_0) = e_u\}$	$\{u\|y_{28}(e_1) = e_u\}$	$\{u\|y_{28}(e_2) = e_u\}$	$\{u\|y_{28}(e_3) = e_u\}$	e_1
29	$e_1x^2 + e_3x + e_1$	$\{u\|y_{29}(e_0) = e_u\}$	$\{u\|y_{29}(e_1) = e_u\}$	$\{u\|y_{29}(e_2) = e_u\}$	$\{u\|y_{29}(e_3) = e_u\}$	e_1
30	$e_1x^2 + e_3x + e_2$	$\{u\|y_{30}(e_0) = e_u\}$	$\{u\|y_{30}(e_1) = e_u\}$	$\{u\|y_{30}(e_2) = e_u\}$	$\{u\|y_{30}(e_3) = e_u\}$	e_1
31	$e_1x^2 + e_3x + e_3$	$\{u\|y_{31}(e_0) = e_u\}$	$\{u\|y_{31}(e_1) = e_u\}$	$\{u\|y_{31}(e_2) = e_u\}$	$\{u\|y_{31}(e_3) = e_u\}$	e_1
	⋮	⋮	⋮	⋮	⋮	

multiply the polynomials e_3 and e_2, and to add the result to e_1. Here is where lies the main contribution shown in this chapter, it is proposed to use the primitive element and the logarithm table constructed by the algorithm in (Torres-Jimenez et al., 2011a) to do the multiplication through additions. To do that they are used equivalent powers of the primitive element of the elements $e_i \in GF(2^2)$ involved in the operation, e.g. instead of multiplying $(x+1)\cdot(x)$ we multiply $x^2 \cdot x^1$. Then, the sum of indices does the multiplication, and the antilogarithm obtains the correct result in $GF(2^2)$. For the case of $x^2 \cdot x^1$ the result is $x^3 = x^0 = e_1$. Finally, we add this result to e_1 to complete the operation (this yield the expected value e_0).

Note that whenever and operation yields a result outside of the field, a modulus operations is required.

The pseudocode for the construction of orthogonal arrays using the Bush's construction and the logarithm tables is shown in Algorithm 9. The logarithm and antilogarithm table $\mathcal{L}_{i,j}$ is obtained through the algorithm reported by Torres-Jimenez et al. (2011a). After that, each element e_i and each polynomial $y_j(x)$ in $GF(p^n)$ are considered as the columns and rows of $\mathcal{M}$, the orthogonal array that is being constructed. Given that the value of each cell $m_{i,j} \in \mathcal{M}$ is the index u of the element $e_u \in GF(p^n)$ such that $y_j(e_i) = e_u$, the following step in the pseudocode is the evaluation of the polynomial $y_j(x)$. This evaluation is done by determining the coefficient of each term $a_k \in y_j(x)$ and its index, i.e., the value of the element $e_l \in GF(p^n)$ that is the coefficient of a_k, and then adding it to $i \cdot d$ (the index of e_i raised to the degree of the term a_k). A modulus operation is applied to the result to obtained v, and then the antilogarithm is used over v such that the index it is able to get the value u of the element e_u. Remember that the algorithm `BuildLogarithmTable` simultaneously find the primitive element and computes the logarithm and antilogarithm tables.

Algorithm 9: `BuildOrthogonalArray(p,n)`, an algorithm for the construction of orthogonal arrays using the Bush's construction and the logarithm tables (Torres-Jimenez et al., 2012).

Input: A prime number p and a power n.
Output: An orthogonal array $\mathcal{M}$.

```
begin
    L ← BuildLogarithmTable(p, n)
    M ← ∅
    foreach element e_i ∈ GF(p^n) do
        c ← i
        foreach polynomial y_j(x) ∈ GF(p^n) do
            r ← j
            foreach term a_k ∈ y_j(x) do
                d ← GetDegree(a_k)
                l ← GetIndexCoefficient(a_k)
                v ← (i · d + l)mod(p^n − 1)
                s ← L_{v,1}
            end foreach
            m_{r,c} ← s
        end foreach
    end foreach
    return M
end
```

Note that in the pseudocode the more complex operation is the module between integers, which can be reduced to shifts when $GF(p^n)$ involves powers of two. This fact makes the algorithm easy and efficient for the construction of orthogonal arrays, requiring only additions to operate, and modulus operations when the field is over powers of primes different of two. After the construction of the orthogonal array, the number of operations required by the algorithm are bounded by $O(N \cdot t^2)$, due to it requires t operations for the construction of an orthogonal array matrix of size $N \times (t+1)$.

3.6.4 Efficient constructions of covering arrays

This section analyzes the case when Covering Arrays can be constructed from cyclotomy by rotating a vector created from an orthogonal array (Colbourn and Torres-Jimenez, 2010). It is another process that can be benefited from the previously constructed logarithm tables. The cyclotomy process requires the test of different cyclotomic vectors for the construction of covering arrays. This vectors can be constructed using the logarithm table. The rest of the section details a bit more about covering arrays and this process of construction.

The trivial mathematical *lower bound* for a covering array is $v^t \leq CAN(t,k,v)$, however, this number is rarely achieved. Therefore determining achievable lower bounds is one of the main research lines for covering arrays; this problem has been overcome with the reduction of the known upper bounds. The construction of cyclotomic matrices can help to accomplish this purpose.

The strategy to construct a cyclotomic matrix involves the identification of a good vector starter. This task can be facilitated using the logarithm table derived from a Galois field. The construction is simple. The first step is the generation of the logarithm table for a certain $GF(p^n)$. After that, the table is transposed in order to transform it into a vector starter v. Then, by using all the possible rotations of it, the cyclotomic matrix is constructed. Finally, the validation of the matrix is done such that a covering array can be identified.

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

Figure 3.12: Covering array where $N = 9$, $t = 2$, $k = 4$ and $v = 3$.

Figure 3.13 shows an example of a cyclotomic matrix.

(a) Vector Starter

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

(b) Cyclotomic matrix

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

Figure 3.13: Example of a cyclotomic vector V, or a vector starter, and the cyclotomic matrix formed with it. The matrix constitutes a $CA(13; 2, 13, 2)$.

The pseudocode to generate the cyclotomic vector and construct the covering array is presented in Algorithm 10. There, the algorithm `BuildLogarithmTable(p,n)` is used to construct the table of logarithm and antilogarithms $\mathcal{L}$, where the ith row indicate the element $e_i \in GF(p^n)$, and the column 0 its logarithm, and the column 1 its antilogarithm. The first step is the construction of the vector starter $\mathcal{V}$, which is done by transposing the logarithm table $\mathcal{L}_{*,0}$, i.e., the first column of $\mathcal{L}$. After that, the cyclotomic matrix $\mathcal{M}$ is constructed by rotating the vector starter p^n times, each time the vector rotated will constituted a row of $\mathcal{M}$. Finally, the cyclotomic matrix $\mathcal{M}$ must be validated as a covering array to finally return it; one strategy to do so is the parallel algorithm reported by Avila-George et al. (2010b).

For more details about this construction, the reader is referred to (Torres-Jimenez et al., 2012).

3.7 Verification of covering arrays

Some of the algorithms used to solve the CAC problem are approximated, meaning that rather than constructing optimal covering arrays, they construct matrices of size close to that value. Some of these approximated strategies must verify that the matrix they are building is a covering array. If the matrix is of size $N \times k$ and the interaction is t, there are $\binom{k}{t}$ different combinations which implies a cost of $O(N \times \binom{k}{t})$ for the verification. For small values of t and v the verification of covering arrays is overcame through the use of sequential approaches; however,

Algorithm 10: `BuildCoveringArray(p,n)`, an algorithm to generate a cyclotomic vector and then construct a covering array (Torres-Jimenez et al., 2012).

```
Input: A prime number p and a power n.
Output: A covering array M.
begin
    L ← BuildLogarithmTable(p, n)
    foreach e_i ∈ GF(p^n) do
        V_i ← L_{i,0}
    end foreach
    foreach e_i ∈ GF(p^n) do
        foreach e_j ∈ GF(p^n) do
            k ← (i + j)mod(p^n)
            m_{i,j} ← V_k
        end foreach
    end foreach
    if IsACoveringArray(M) then
        return M
    else
        return ∅
    end if
end
```

when we try to construct covering arrays of moderate values of t, v and k, the time spent by those approaches is impractical. Then, the necessity of parallel or Grid strategies to solve the verification of covering arrays appears, for more details please refer (Avila-George et al., 2010b; Avila-George et al., 2011; Avila-George et al., 2012d).

A matrix $\mathcal{M}$ of size $N \times k$ is a $CA(N; t, k, v)$ *if and only if* every t-tuple contains the set of combination of symbols described by $\{0, 1, \ldots, v-1\}^t$. Avila-George et al. (2010b) proposed a strategy that uses two data structures called P and J, and two injections between the sets of t-tuples and combinations of symbols, and the set of integer numbers, to verify that $\mathcal{M}$ is a covering array.

Let $\mathcal{C} = \{c_1, c_2, \ldots, c_{\binom{k}{t}}\}$ be the set of the different t-tuples. A t-tuple $c_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,t}\}$ is formed by t numbers, each number $c_{i,1}$ denotes a column of the matrix $\mathcal{M}$. The set $\mathcal{C}$ can be managed using an injective function $f(c_i) : \mathcal{C} \rightarrow \mathcal{I}$ between $\mathcal{C}$ and the integer numbers, this function is defined in (3.6):

$$f(c_i) = \sum_{j=1}^{t} \binom{c_{i,j} - 1}{i + 1}. \tag{3.6}$$

Now, let $\mathcal{W} = \{w_1, w_2, \ldots, w_{v^t}\}$ be the set of the different combination of symbols, where $w_i \in \{0, 1, \ldots, v-1\}^t$. The injective function $g(w_i) : \mathcal{W} \rightarrow \mathcal{I}$ is defined in

(3.7). The function $g(w_i)$ is equivalent to the transformation of a v-ary number to the decimal system.

$$g(w_i) = \sum_{j=1}^{t} w_{i,j} \cdot v^{t-i}. \tag{3.7}$$

The inverse $g^{-1}(w_i)$ is obtained using the same algorithm that maps a decimal number to a v-ary numeric system.

The use of the injections represents an efficient method to manipulate the information that will be stored in the data structures P and J used in the verification process of $\mathcal{M}$ as a covering array. The matrix P is of size $\binom{k}{t} \times v^t$ and it counts the number of times that each combination appears in $\mathcal{M}$ in the different t-tuples. Each row of P represents a different t-tuple, while each column contains a different combination of symbols. The management of the cells $p_{i,j} \in P$ is done through the functions $f(c_i)$ and $g(w_j)$; while $f(c_i)$ retrieves the row related with the t-tuple c_i, the function $g(w_i)$ returns the column that corresponds to the combination of symbol w_i.

Table 3.9: Mapping of the set $\mathcal{W}$ to the set of integers using the function $g(w_j)$ in $CA(9; 2, 4, 3)$ shown in Figure 2.6(b).

$\mathcal{W}$	$g(w_i)$	$\mathcal{I}$
w_1 ={0,0}	$0 \cdot 3^1 + 0 \cdot 3^0$	0
w_2 ={0,1}	$0 \cdot 3^1 + 1 \cdot 3^0$	1
w_3 ={0,2}	$0 \cdot 3^1 + 2 \cdot 3^0$	2
w_4 ={1,0}	$1 \cdot 3^1 + 0 \cdot 3^0$	3
w_5 ={1,1}	$1 \cdot 3^1 + 1 \cdot 3^0$	4
w_6 ={1,2}	$1 \cdot 3^1 + 2 \cdot 3^0$	5
w_7 ={2,0}	$2 \cdot 3^1 + 0 \cdot 3^0$	6
w_8 ={2,1}	$2 \cdot 3^1 + 1 \cdot 3^0$	7
w_9 ={2,2}	$2 \cdot 3^1 + 2 \cdot 3^0$	8

Table 3.9 shows an example of the use of the function $g(w_j)$ for the Covering Array $CA(9; 2, 4, 3)$ (shown in Figure 2.6(b)). Column 1 shows the different combination of symbols. Column 2 contains the operation from which the equivalence is derived. Column 3 presents the integer number associated with that combination.

The matrix P is initialized to zero. The construction of matrix P is direct from the definitions of $f(c_i)$ and $g(w_j)$; it counts the number of times that a combination of symbols $w_j \in \mathcal{W}$ appears in each subset of columns corresponding to a t-tuple c_i, and increases the value of the cell $p_{f(c_i),g(w_j)} \in P$ in that number.

Table 3.10(a) shows the use of injective function $f(c_i)$. Table 3.10(b) presents the matrix P of $CA(9; 2, 4, 3)$. The different combination of symbols $w_j \in \mathcal{W}$ are in the first rows. The number appearing in each cell referenced by a pair (c_i, w_j) is the number of times that combination w_j appears in the set of columns c_i of the matrix $CA(9;2,4,3)$.

Table 3.10: Example of the matrix P resulting from $CA(9;2,4,3)$ presented in Figure 2.6(b).

(a) Applying $f(c_i)$.

c_i		
index	t-tuple	$f(c_i)$
c_1	$\{1,2\}$	0
c_2	$\{1,3\}$	1
c_3	$\{1,4\}$	3
c_4	$\{2,3\}$	2
c_5	$\{2,4\}$	4
c_6	$\{3,4\}$	5

(b) Matrix P.

	$g(w_j)$								
$f(c_i)$	{0,0}	{0,1}	{0,2}	{1,0}	{1,1}	{1,2}	{2,0}	{2,1}	{2,2}
0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1

In summary, to determine if a matrix $\mathcal{M}$ is a covering array or not, the number of different combinations of symbols per t-tuple is counted using the matrix P. The matrix $\mathcal{M}$ will be a covering array *if and only if* the matrix P contains no zero in it. Several approaches can be followed to implement this strategy to verify a covering array. The traditional one is the sequential algorithm; one instruction at a time. The other two approaches are parallel computing and Grid computing. These strategies use the data structures described in this section and are discussed in the following subsections.

3.7.1 Sequential algorithm to verify covering arrays

Usually, the first programming model followed to solve a problem is the sequential one. This section presents the implementation details of the sequential approach to verify a covering array.

The Sequential Algorithm to Verify Covering Arrays (SAVCA) takes as input a matrix $\mathcal{M}$ and the parameters N, k, v, t that describe the covering array that $\mathcal{M}$ can be. Also, the algorithm requires the sets $\mathcal{C}$ and $\mathcal{W}$ and, without lost of generality, the values $\mathcal{K}_l$ and $\mathcal{K}_u$ that represent the first and last t-tuple to be verified. SAVCA outputs the total number of missing combinations in the matrix $\mathcal{M}$ to be a covering array. The algorithm first counts for each different t-tuple c_i the times that a combination $w_j \in \mathcal{W}$ is found in the columns of $\mathcal{M}$ corresponding to c_i. After that, SAVCA calculates the missing combinations $w_j \in \mathcal{W}$ in c_i. Finally, the algorithm transforms c_i into c_{i+1}, i.e., it determines the next t-tuple to be evaluated.

The pseudocode for SAVCA is presented in Algorithm 11. The matrix $\mathcal{M}$ can be stored column-wise to allow a more efficient management of the memory. Instead of completely allocating the matrix P in memory, SAVCA can manage it as a single vector P of size v^t and associate it with the current t-tuple analyzed, found

Algorithm 11: SAVCA, sequential algorithm to verify a covering array (Avila-George et al., 2010b).

1 $t\_wise(\mathcal{M}_{N,k}, N, k, v, t, \mathcal{W}, \mathcal{C}, \mathcal{K}_l, \mathcal{K}_u)$
  Output: Number of missing combination of symbols
2 $Miss \leftarrow 0$;
3 **foreach** $J \in \{c_i | c_i \in \mathcal{C}, \mathcal{K}_l \leq i \leq \mathcal{K}_u\}$ **do**
4     $Covered \leftarrow 0$;
5   **foreach** $w_j \in \mathcal{W}$ **do**
6         $P_{g(w_j)} \leftarrow$ Count(J, w_j);
7   **if** $P_{g(w_j)} > 0$ **then**
8             $Covered \leftarrow Covered + 1$;
9   **end if**
10   **end foreach**
11     $Miss \leftarrow Miss + v^t - Covered$;
12 **end foreach**
13 **return** $Miss$;

in vector J. Then, for each different t-tuple (lines 3 to 12) the algorithm performs the following actions: counts the expected number of times a combination w_j appears in the set of columns indicated by J (line 6); then, the counter *Covered* is increased in the number of different combinations with a number of repetitions greater than zero (line 8). After that, the algorithm calculates the number of missing combinations (line 11). The algorithm ends when all the t-tuples c_i, where $\mathcal{K}_l \leq i \leq \mathcal{K}_u$, have been analyzed. SAVCA sets the values of $\mathcal{K}_l$ and $\mathcal{K}_u$ to 0 and $\binom{k}{t} - 1$, respectively. The t-tuples are generated in lexicographical order.

3.7.2 Parallel approach to verify covering arrays

Parallel computing has been an area of active research interest and application for decades, mainly the focus of high performance computing, but it is now emerging as the prevalent computing paradigm due to the semiconductor industry shift to multi-core processors. As multi-core processors bring parallel computing to mainstream customers, the key challenge in computing today is to make the transition of sequential software to parallel software programming.

The main motivation is the idea that the verification problem is easily parallelizable. To show that, four different parallel implementations were proposed; each implementation was characterized by a distribution method of the workload among the different available cores. The challenge that all the distribution methods must confront was the designing of a workload distribution with a low communication cost. This task was accomplished through the design of a strategy that calculates the starting point for each core, given the set of t-tuples to be analyzed for the problem.

According with the definition presented in Section 3.7 for the problem of verifying a covering array, a matrix $\mathcal{M}$ is a covering array if and only if each t-tuple in $\mathcal{C}$ contains all the symbol combinations derived from $\{0, 1, \ldots, v-1\}^t$. Given that the symbol combination existing in a particular t-tuple does not affect other t-tuples, a workload distribution with low communication cost for a parallel approach can be achieved by uniformly distributing all the t-tuples in $\mathcal{C}$ among the available cores (denoted by $\mathcal{P}$). Following this way, the ith core must start the verification, of the matrix $\mathcal{M}$, at the t-tuple $\mathcal{K}_l = l \cdot \frac{|\mathcal{C}|}{\mathcal{P}}$. Some of the different ways in which the value of l can be defined resulted in the parallel implementations presented in this section.

To make the distribution of work, it is necessary to calculate the initial point $\mathcal{K}_l$ for each core. Therefore, a method to convert the scalar $\mathcal{K}_l$ to the equivalent t-tuple is necessary . Based on an index i, Algorithm 12 determines the tuple $c_i \in \mathcal{C}$ in lexicographical order of the t-tuples. This algorithm is used once in line 3 of Algorithm 11 to determine the first t-tuple and store it in the vector J. Once the first t-tuple is identified, the following tuples are taken following the lexicographical order. Algorithm 12 is of particular use when the initial t-tuple is not c_0.

Algorithm 12: Get initial t-tuple (Avila-George et al., 2011).

getInitialTuple$(k,\ t,\ c_i)$
 Output: Initial t-tuple each core
 $\Theta \leftarrow i$;
 $iK \leftarrow 1$;
 $iT \leftarrow 1$;
 $kint \leftarrow \binom{k-iK}{t-iT}$;
 foreach $i \leftarrow 0;\ i < t;\ i \leftarrow i + 1$ **do**
  while $\Theta \geq kint$ **do**
   $\Theta \leftarrow \Theta - kint$;
   $kint \leftarrow (kint \cdot ((k - iK) - (t - iT)))/(k - iK)$;
   $iK \leftarrow iK + 1$;
  end while
  $J_i \leftarrow iK - 1$;
  $kint \leftarrow (kint \cdot (t - iT))/(k - iK)$;
  $iK \leftarrow iK + 1$;
  $iT \leftarrow iT + 1$;
 end foreach
 return J

To explain the purpose of Algorithm 12, let's consider the $CA(9; 2, 4, 3)$ shown in Figure 2.6(b). This covering array has as set $\mathcal{C}$ the elements found in column 1 of Table 3.10(a). The algorithm `getInitialTuple` with input $k = 4,\ t = 2,\ \mathcal{K}_l = 3$ must return $J = \{1, 4\}$, i.e., the values of the t-tuple c_3. Algorithm 12 is optimized to find the vector $J = \{J_1, J_2, \ldots, J_t\}$ that corresponds to $\mathcal{K}_l$. The value J_i is calculated according to

$$J_i = \min_{j\geq 1}\left\{\Delta_i \leq \sum_{l=J_{i-1}+1}^{j}\binom{k-l}{t-i}\right\}$$

where

$$\Delta_i = \mathcal{K}_l - \sum_{m=1}^{i-1}\sum_{l=J_{m-1}+1}^{J_m-1}\binom{k-l}{t-m}$$

and

$$J_0 = 0.$$

The following paragraphs describe in detail the four proposed implementations.

Static Assignment of Tasks to Workers Approach (SATWA). This strategy is simple, it uses a master-worker scheme with a coarse-grain static distribution. Here, the set $\mathcal{C}$ is divided into $\mathcal{P}-1$ blocks, one block for each worker. The size of each block, i.e., the number of t-tuples to be analyzed by each worker, is defined according to (3.8).

$$\mathcal{B} = \left\lceil \frac{\mathcal{C}}{\mathcal{P}-1} \right\rceil. \tag{3.8}$$

Algorithm 13 shows the pseudocode corresponding to the strategy SATWA. One of the cores is the master, and it must distribute the workload among the rest of the cores, i.e., the workers. In line 3, it determines the size for each block. The distribution of the work is done from lines 4 to 7. Finally, the results from each worker are obtained from lines 8 to 11. Summarizing, in this parallel implementation, each worker counts the number of symbol combinations missing in each of its t-tuples, and reports it to the master, which finally sums up all of them and reports as the final result.

Dynamic Assignment of Tasks to Workers Approach (DATWA). This strategy uses a fine-grain dynamic distribution of the workload. It also follows a master-worker scheme, and defines the size of each block according to (3.9). DATWA uses blocks of t-tuples of smaller size than those used by SATWA, improving the balance in the workload among the workers but increasing the communication with the master:

$$\mathcal{B} = \left\lceil \frac{\mathcal{C}}{(\mathcal{P}-1)\times t\times v} \right\rceil. \tag{3.9}$$

Algorithm 14 shows the pseudocode corresponding to the DATWA strategy. Initially, in line 2, the size of each block $\mathcal{B}$ is compute, and every worker is assigned a

Algorithm 13: SATWA, static assignment of tasks to workers approach (Avila-George et al., 2012d).

Input: An array $\mathcal{M}$ of size $N \times k$ with alphabet v and strength t, the set of the different combination of symbols $\mathcal{W}$, the set of different t-tuples $\mathcal{C}$, the number of cores $\mathcal{P}$.
Output: $miss$, the number of missing combination of symbols

begin
    if $MASTER$ **then**
        $\mathcal{B} \leftarrow \lceil \frac{C}{\mathcal{P}-1} \rceil$, $miss \leftarrow 0$
        for $i \leftarrow 0$ **to** $i < \mathcal{P} - 1$ **do**
            $\mathcal{K}_l \leftarrow \mathcal{P}_i \times \mathcal{B}$, $\mathcal{K}_u \leftarrow \mathcal{K}_l + \mathcal{B}$
            send $task(\mathcal{K}_l,\mathcal{K}_u)$ to WORKER P_i
        end for
        for $i = 1$ **to** $i < \mathcal{P}$ **do**
            receive $partial\_miss$ from any WORKER
            $miss \leftarrow miss + partial\_miss$
        end for
    else
        receive $task(\mathcal{K}_l, \mathcal{K}_u)$ from MASTER
        $partial\_miss \leftarrow t\_wise(\mathcal{M}, N, k, v, t, \mathcal{W}, \mathcal{C}, \mathcal{K}_l, \mathcal{K}_u)$
        send $partial\_miss$ to MASTER
    end if
end

block of t-tuples according with their rank (see lines 15 to 21). Then, the master computes the first subset of t-tuples that is pending to be processed (line 4), and after that if waits for an available worker to assign it (lines 5 to 9). The algorithm iterates the assignment of pending blocks of t-tuples, until all of them have been processed. Finally, the master joins the results coming from each worker to count the total number of missing symbol combinations (lines 10 to 13).

Assigning Tasks by Blocks Approach (ATBBA). This strategy uses a coarse-grain static distribution as SATWA, but with the difference that all the cores have an assigned verification task. The set of t-tuples $\mathcal{C}$ is divided into $\mathcal{P}$ blocks, one for each core. The size of the block $\mathcal{B}$ is defined in (3.10). The block distribution model maintains the simplicity in the code.

$$\mathcal{B} = \left\lceil \frac{C}{P} \right\rceil \tag{3.10}$$

Algorithm 15 shows the pseudocode corresponding to the strategy ATBBA. Initially, the subset of t-tuples that will be verified by each core is computed in lines 2 and 3, according to their rank $\mathcal{P}_i$. After that, each core verifies its corresponding t-tuples, and reports the results to one of the processes, which accumulates the total number of missing symbol combinations (lines 4 to 13).

Algorithm 14: DATWA, dynamic assignment of tasks to workers approach (Avila-George et al., 2012d).

Input: An array $\mathcal{M}$ of size $N \times k$ with alphabet v and strength t, the set of the different combination of symbols $\mathcal{W}$, the set of different t-tuples $\mathcal{C}$, the number of cores $\mathcal{P}$
Output: $miss$, the number of missing combination of symbols

```
begin
    B ← ⌈ C / ((P−1)×t×v) ⌉
    if MASTER then
        K_l ← P × B, K_u ← K_l + B, miss ← 0
        repeat
            receive requests for task from any WORKER
            send task(K_l,K_u) to WORKER P_i
            K_l ← K_l + B, K_u ← K_l + B
        until K_l < C;
        for i = 1 to i < P do
            receive partial_miss from any WORKER
            miss ← miss + partial_miss
        end for
    else
        K_l ← P_i × B, K_u ← K_l + B, partial_miss ← 0
        repeat
            partial_miss ← partial_miss + t_wise(M, N, k, v, t, W, C, K_l, K_u)
            request task from MASTER
            receive task(K_l, K_u) from MASTER
        until K_l < C;
        send partial_miss to MASTER
    end if
end
```

Assigning Tasks by Cyclic Blocks Approach (ATBCBA). This strategy uses a cyclic fine-grain distribution for the workload. The granularity for each task in this scheme is defined according to (3.11):

$$\mathcal{B} = \left\lceil \frac{C}{\mathcal{P} \times t \times v} \right\rceil. \tag{3.11}$$

Algorithm 16 shows the pseudocode corresponding to the strategy ATBCBA. Line 2 shows the computation of the size of the block $\mathcal{B}$, i.e., the number of t-tuples from $\mathcal{C}$ to be analyzed. After that, the specific subsets of t-tuples to be verified for each core are determined in lines 3 to 6, based on the cyclic distribution. The computation of the initial t-tuple for each block analyzed by a core is done through (3.12), where n corresponds to the block number of that process:

$$\mathcal{K}_l = (n \times \mathcal{P} \times \mathcal{B}) + (\mathcal{P}_i \times \mathcal{B}). \tag{3.12}$$

Algorithm 15: ATBBA, Assigning tasks by blocks approach (Avila-George et al., 2012d).

Input: An array $\mathcal{M}$ of size $N \times k$ with alphabet v and strength t, the set of the different combination of symbols $\mathcal{W}$, the set of different t-tuples $\mathcal{C}$, the number of cores $\mathcal{P}$
Output: $miss$, the number of missing combination of symbols

begin
  $\mathcal{B} \leftarrow \lceil \frac{C}{\mathcal{P}} \rceil$
  $\mathcal{K}_l \leftarrow \mathcal{P}_i \times \mathcal{B}$, $\mathcal{K}_u \leftarrow \mathcal{K}_l + \mathcal{B}$
  $partial\_miss \leftarrow t\_wise(\mathcal{M}, N, k, v, t, \mathcal{W}, \mathcal{C}, \mathcal{K}_l, \mathcal{K}_u)$
  if $\mathcal{P}_i \neq \mathcal{P} - 1$ **then**
    send $partial\_miss$ to $\mathcal{P} - 1$
  else
    $miss \leftarrow partial\_miss$
    for $i = 1$ **to** $i < \mathcal{P}$ **do**
      receive $partial\_miss$ from $\mathcal{P}_i$
      $miss \leftarrow miss + partial\_miss$
    end for
  end if
end

Algorithm 16: ATBCBA, assigning tasks by cyclic blocks approach (Avila-George et al., 2012d).

Input: An array $\mathcal{M}$ of size $N \times k$ with alphabet v and strength t, the set of the different combination of symbols $\mathcal{W}$, the set of different t-tuples $\mathcal{C}$, the number of cores $\mathcal{P}$
Output: $miss$, the number of missing combination of symbols

begin
  $\mathcal{B} \leftarrow \lceil \frac{C}{\mathcal{P} * t * v} \rceil$, $partial\_miss \leftarrow 0$
  for $n \leftarrow 0$ **to** $n < t \times v$ **do**
    $\mathcal{K}_l \leftarrow (n \times \mathcal{P} \times \mathcal{B}) + (\mathcal{P}_i \times \mathcal{B})$, $\mathcal{K}_u \leftarrow \mathcal{K}_l + \mathcal{B}$
    $partial\_miss \leftarrow partial\_miss + t\_wise(\mathcal{M}, N, k, v, t, \mathcal{W}, \mathcal{C}, \mathcal{K}_l, \mathcal{K}_u)$
  end for
  if $\mathcal{P}_i \neq \mathcal{P} - 1$ **then**
    send $partial\_miss$ to MASTER
  else
    $miss \leftarrow 0$
    for $i = 1$ **to** $i < \mathcal{P}$ **do**
      receive $partial\_miss$ from any WORKER
      $miss \leftarrow miss + partial\_miss$
    end for
  end if
end

3.7.3 Grid approach to verify covering arrays

In order to fully understand the Grid implementation developed in this work, this subsection will introduce all the details regarding the Grid Computing Platform used.

The evolution of Grid Middlewares has enabled the deployment of Grid e-Science infrastructures delivering large computational and data storage capabilities. Current infrastructures, such as the one used in this work, European Grid Infrastructure (EGI), rely on gLite mainly as core middleware supporting several services in some cases. World-wide initiatives, such as EGI, aim at linking and sharing components and resources from several European National Grid Initiatives (NGI).

In the EGI infrastructure, jobs are specified through a job description language Pacini (2011) or JDL that defines the main components of a job: executable, input data, output data, arguments, and restrictions. The restrictions define the features a resource should provide, and could be used for meta-scheduling or for local scheduling (such as in the case of MPI jobs). Input data could be small or large and job-specific or common to all jobs, which affects the protocols and mechanisms needed. Executables are either compiled or multiplatform codes (scripts, Java, Perl), and output data suffer from similar considerations as input data.

The key resources in gLite middleware are extensively listed in the literature, and can be summarized as:

1. User Interface (UI): The access point to any gLite Grid, normally any machine where the user certificate is installed. It provides Command Line Interface Tools (CLI) to perform some basic Grid operations (submission, cancelation, monitoring, data management, retrieval of results).
2. Workload Management System / Resource Broker (WMS/RB): Meta-scheduler that coordinates the submission and monitoring of jobs.
3. Computing Elements (CE): The access point to a farm of identical computing nodes, which contains the Local Resource Management System (LRMS). The LRMS is responsible for scheduling the jobs submitted to the CE, allocating the execution of a job in one (sequential) or more (parallel) computing nodes. In the case that no free computing nodes are available, jobs are queued. Thus, the load of a CE must be considered when estimating the turnaround of a job.
4. Working Nodes (WN): Each one of the computing resources accessible through a CE. Due to the heterogeneous nature of Grid infrastructure, the response time of a job will depend on the characteristics of the WN hosting it.
5. Storage Element (SE): Storage resources in which a task can store long-living data to be used by the computers of the Grid. This practice is necessary due to the size limitation imposed by current Grid Middlewares in the job file attachment (10 MB in the gLite case). So, use cases which require the access to files which exceed that limitation are forced to use these Storage Elements. Nevertheless, the construction of covering arrays is not a data-intensive use case and thus the use of SEs can be avoided.

6. Logic File Catalog (LFC): A hierarchical directory of logical names referencing a set of physical files and replicas stored in the SEs.
7. Berkley Database Information System (BDII): Service point for the Information System which registers, through LDAP, the status of the Grid. Useful information relative to CEs, WNs and SEs can be obtained by querying this element.
8. Relational Grid Monitoring Architecture (R-GMA): Service for the registration and notification of information in most of the EGI services.
9. Virtual Organisation (VO): Subset of the computing and storage resources of the infrastructure dedicated to a certain scientific field (Life Sciences, Earth Sciences, Physics...).
10. Virtual Organisation Management System (VOMS): Authorization infrastructure to define the access rights to resources.

All these terms will be referenced along the text.

With respect to the job submission, there are different strategies in Grid environments that can be broken into two paradigms: asynchronous and synchronous ones. In this work, we use a synchronous mechanism known as *pilot jobs submission* that is based on a master-worker architecture and supported by the DIANE (DIANE, 2011) + Ganga (Moscicki et al., 2009) tools. In this schema, the processing begins with the creation of a master process (a server) in the UI, which will dispatch tasks to the worker agents until all the tasks have been completed, being then dismissed. The worker agents are jobs running on the WN (Working Nodes) of the Grid capable of communicating with the master. The mission of the master is to keep track of the tasks to ensure that all of them are successfully completed while workers provide the access to a CPU reached through scheduling. If for any reason a task fails or a worker losses contact with the master, the master will immediately reassign the task to another worker. The whole process is exposed in Figure 3.14.

Nevertheless, prior to beginning the execution of a experiment, it is mandatory to configure certain aspects. Firstly, the specification of a run must include the master and workers heartbeat timeout. It is also necessary to establish master scheduling policies such as the maximum number of times that a lost or failed task is assigned to a worker; the reaction when a task is lost or fails; and the number of resubmissions before a worker is decided to be removed. Finally, the master must know the arguments of the tasks (the covering array filename and the task id), the input files (the covering array source code, the execution script, and the covering array file), and the output file. The execution script is necessary for compiling on-the-fly the source code in every worker and then executing and then execute the covering array validation program with the arguments indicated by the master.

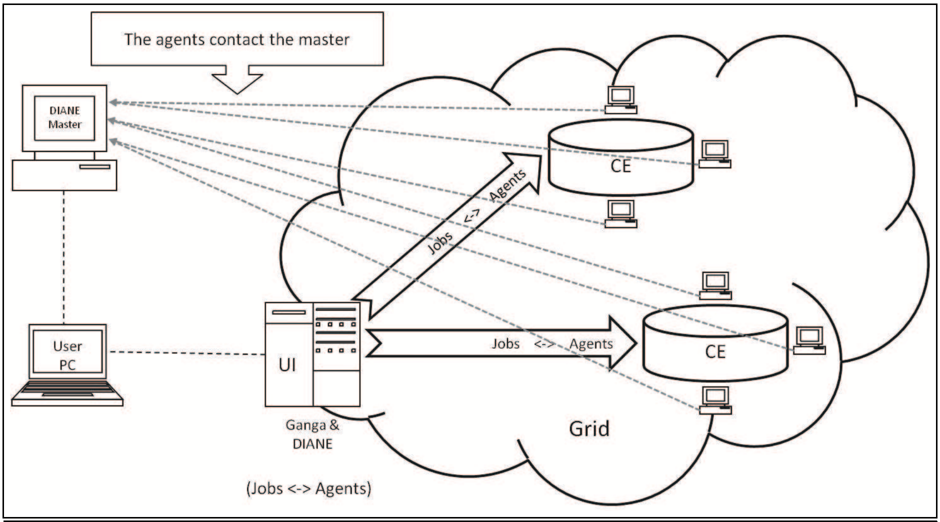

Figure 3.14: Pilot jobs schema offered by DIANE-Ganga.

At this point, the master can be started using the specification described above.
Upon checking that all is right, the master process will wait for incoming connec-
1209 tions from the workers.

Workers are generic jobs that can perform any operation requested by the master
which are submitted to the Grid. When a worker registers with the master, the
1210 master will automatically assign it a task.

This schema has several advantages derived from the fact that a worker can exe-
1211 cute more than one task. When a worker demands a new task it is not necessary
1212 to submit a new job. This way, the queuing time of the task is intensively re-
duced. Moreover, the dynamic behavior of this schema allows achieving better
1213 performance results, in comparison to the asynchronous schema.

However, there are also some disadvantages that must be mentioned. The first
1214 issue refers to the unidirectional connectivity between the master host and the
1215 worker hosts (Grid node). While the master host needs inbound connectivity,
1216 the worker node needs outbound connectivity. The connectivity problem in the
1217 master can be solved easily by opening a port in the local host; however, the
1218 connectivity in the worker will rely on the remote system configuration (the CE).
1219 So, in this case, this extra detail must be taken into account when selecting the
1220 computing resources. Another issue is defining an adequate timeout value. If, for
1221 some reason, a task working correctly suffers from temporary connection problems
1222 and exceeds the timeout threshold, it will cause the worker being removed by

the master. Finally, a key factor will be to identify the rightmost number of worker agents and tasks. In addition, if the number of workers is on the order of thousands, bottlenecks could be met, resulting in the master being overwhelmed by the excessive number of connections.

This Grid model of computation can also be applied to the process of verification of covering array. The Grid Algorithm to Verify Covering Arrays (GAVCA) uses a block partitioning scheme like the one described in the parallel ATBBA approach. The GAVCA model takes advantage of the huge number of cores that can be involved in the solution of the problem of verification of covering arrays. Each different core will output the number of missing combinations in a different file. At the end, these results are joined and the total number of missing combinations is counted and reported. Algorithm 17 shows the pseudocode of the GAVCA for the problem of verification of covering arrays; particularly, the algorithm shows the process performed by each core involved in the verification of covering arrays. The strategy followed by GAVCA is simple, each core determines the block of t-tuples to be analyzed by it (lines 2 and 4) and calls the SA for that specific block (line 5).

Algorithm 17: GAVCA, Grid approach to verify covering arrays. This algorithm assigns the set of t-tuples $\mathcal{C}$ to $\mathcal{P}$ different cores (Avila-George et al., 2012d).

Input: An array $\mathcal{M}$ of size $N \times k$ with alphabet v and strength t, the set of the different combination of symbols $\mathcal{W}$, the set of different t-tuples $\mathcal{C}$, the number of cores $\mathcal{P}$

Output: $miss$, the number of missing combination of symbols

1 **begin**

2 $\quad \mathcal{B} \leftarrow \lceil \frac{\mathcal{C}}{\mathcal{P}} \rceil$

3 $\quad \mathcal{K}_l \leftarrow \mathcal{P}_i \times \mathcal{B}$

4 $\quad \mathcal{K}_u \leftarrow \mathcal{K}_l + \mathcal{B}$

5 $\quad miss \leftarrow t\_wise(\mathcal{M}, N, k, v, t, \mathcal{W}, \mathcal{C}, \mathcal{K}_l, \mathcal{K}_u)$

6 **end**

For more details we refer the reader to Avila-George et al. (2010b); Avila-George et al. (2011); Avila-George et al. (2012d).

3.8 Summary

In this chapter, we have described in general terms the distinct types for constructing covering arrays: (1) Algebraic constructions, (2) Recursive constructions, (3) Greedy methods, and (4) Metaheuristic methods.

Algebraic constructions often provide a better bound in less computational time. Unfortunately, algebraic approaches often impose serious restrictions on the system configurations to which they can be applied. For example, many approaches for constructing orthogonal arrays require that the domain size be a prime number or a power of a prime number. This significantly limits the applicability of algebraic approaches for software testing.

There are sophisticated recursive constructions that combine small covering arrays. While the more sophisticated constructions yield substantially smaller covering arrays when they can be applied, these same constructions do not apply as generally as we require.

Greedy algorithms are more flexible than algebraic constructions and recursive constructions. These methods can generate any covering array using as input t, k, and v. The problem with these methods are the results, greedy methods rarely obtain optimal covering arrays.

Metaheuristic methods appear to produce smaller covering arrays compared to greedy algorithms but with more time to spend.

Finally, some of the algorithms used to solve the CAC problem are approximated, meaning that rather than constructing optimal covering arrays, they construct matrices of size close to that value. This chapter has presented a methodology to verify a given matrix as a covering array.

The remaining of this thesis focuses on building small and flexible interaction test suites using an improved simulated annealing algorithm.

Chapter 4

Methodology

In this chapter, we present the specific details that were involved in the development of the simulated annealing proposed to construct covering arrays. Section 4.1 presents an overview about the simulated annealing technique. Section 4.2 introduces an improved simulated annealing to construct covering arrays. Section 4.3 presents a Grid deployment of the parallel simulated annealing algorithm for constructing covering arrays, introduced in Section 4.2. In order to fully understand the Grid implementation developed in this work, this section will introduce all the details regarding the Grid Computing Platform used and then, the different execution strategies will be exposed. Finally, Section 4.4 introduces three parallel simulated annealing approaches to solve the CAC problem. The objective is to find the best bounds for covering arrays by using parallelism.

4.1 Simulated annealing overview

Often the solution space of an optimization problem has many local minima. A simple local search algorithm proceeds by choosing a random initial solution and generating a neighbor from that solution. The neighboring solution is accepted if it is a cost decreasing transition. Such a simple algorithm has the drawback of often converging to a local minimum. The simulated annealing algorithm (SA), though by itself it is a local search algorithm, avoids getting trapped in a local minimum by also accepting cost increasing neighbors with some probability. Simulated annealing is a general-purpose stochastic optimization method that has proven to be an effective tool for approximating globally optimal solutions to many types of NP-hard combinatorial optimization problems. In this section, we briefly review simulated annealing algorithm.

Simulated annealing is a randomized local search method based on the simulation of annealing of metal. A typical structure of simulated annealing consists of two nested loops, this method is represented in pseudocode format in Algorithm 18. It starts from an arbitrarily selected configuration so with an appropriate initial temperature (T_i) and works to minimize a given *cost function*.

At a fixed temperature, the inner loop (it represents a Markov chain) repeatedly executes the following three step operation, to be referred to as *iteration*, until an inner loop break condition is satisfied. It randomly perturbs the current solution (or configuration), evaluates the corresponding cost, and accepts the new solution with the probability given by (4.1), it means that the trial solution is accepted by nonzero probability $e^{(-\Delta E/T)}$ even though the solution deteriorates (*uphill move*), where ΔE is the difference of the costs between the trial and the current solutions (the cost change due to the perturbation), and T is the *temperature* of the system.

$$\mathbb{P} = \begin{cases} 1 & if \Delta E < 0 \\ e^{(-\frac{\Delta E}{T})} & otherwise \end{cases} \tag{4.1}$$

Uphill moves enable the system to escape from the local minima; without them, the system would be trapped into a local minimum. Too high of a probability for the occurrence of uphill moves, however, prevents the system from converging. In simulated annealing, the probability is controlled by temperature in such a manner that at the beginning of the procedure the temperature is sufficiently high, in which a high probability is available, and as the calculation proceeds the temperature is gradually decreased, lowering the probability (Jun and Mizuta, 2005).

The outer loop decreases temperature according to a *geometrical cooling scheme*, $T \leftarrow \alpha T$, where α, the *cooling coefficient*, satisfies $0 < \alpha < 1$. It can be said that SA consists of a sequential chain of consecutive perturbation, evaluation and decision steps.

Algorithm 18: Typical structure of simulated annealing.

begin
  choose the initial solution $s \leftarrow s_0$
  choose the initial temperature $T \leftarrow T_i$
  repeat
    repeat
      perturb the current solution s to s'
      evaluate the cost function $\Delta E \leftarrow E(s') - E(s)$
      accept the trial solution as a new solution by acceptance probability $min(1, e^{\frac{-\Delta E}{T}})$
    until termination condition is satisfied;
    temperature is lowered according to the cooling schedule $T \leftarrow \alpha T$
  until termination condition is satisfied;
end

The previously mentioned parameters, which control the execution of the nested loops are called *scheduling parameters*, i.e., *initial temperature* (T_i), *cooling coefficient* (α), and *equilibrium conditions* for the inner and outer loops. The execution time and solution quality are heavily dependent on the scheduling parameters. Next, we describe the developed simulated annealing algorithm to solve the CAC problem.

4.2 An improved simulated annealing to construct covering arrays

In this section we propose a simulated annealing to solve the CAC problem. Our approach constructs uniform and mixed covering arrays. Contrary to existing SA implementations for the CAC problem (Stardom, 2001; Cohen et al., 2003), the developed algorithm has the merit of improving two key features that have a great impact on its performance: an efficient method to generate initial solutions containing a balanced number of symbols in each column and a composed neighborhood function. Next all the implementation details of the proposed SA algorithm are presented.

4.2.1 Internal representation

The following paragraphs will describe each of the components of the Developed Sequential Simulated Annealing (DSSA). The description is done given the matrix representation of a covering array. A covering array can be represented as a matrix $\mathcal{M}$ of size $N \times k$, where the columns are the parameters and the rows are the cases of the test set that is constructed. Each cell $m_{i,j}$ in the array accepts values from the set $\{0, 1, \ldots, v_j - 1\}$ where v_j is the cardinality of the alphabet of j-th column.

In order to describe DSSA approach, we first introduce a list of sets ($\mathcal{C}, V, U, W$, and R) derived from an $MCA(N; t, k, v_1^1 v_2^2 \ldots v_g^w)$:

- ▷ Let $\mathcal{C} = \{c_1, c_2, \ldots, c_{\binom{k}{t}}\}$ be the set of the different t-tuples. A t-tuple $c_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,t}\}$ is formed by t numbers, each number $c_{i,j}$ denotes a column of matrix $\mathcal{M}$. The set $\mathcal{C}$ can be managed using an injective function $f(c_i) : \mathcal{C} \rightarrow \mathcal{I}$ between $\mathcal{C}$ and the integer numbers, this function is defined in (4.2).

$$f(c_i) = \sum_{j=1}^{j=t} \binom{c_{i,j} - 1}{i + 1} \tag{4.2}$$

- ▷ Let $V = \{v_1, v_2, \ldots, v_k\}$ be the vector that stores the cardinalities of the columns of $\mathcal{M}$.

- ▷ Let $U = \{u_1, u_2, \ldots, u_t\}$ be the vector that contains the t larger cardinalities, arranged in decreasing order, from the cardinalities of the columns of $\mathcal{M}$.

- ▷ Let $W = \{W_1, W_2, \ldots, W_{\binom{k}{t}}\}$ be the set in which each of its elements $W_i = \{w_1, w_2, \ldots, w_{\prod_{i=1}^{i=t} v_i}\}$ is a set containing the combinations of symbols that must be covered in the t-tuple $c_i \in C$, where v_i is the cardinality of the alphabet of column i in the mixed covering array that is constructed. Now, let $w_i = \{w_{i,1}, w_{i,2}, \ldots, w_{i,v_{max}}\}$ be the set of the different combinations of symbols, where $w_{i,j} \in \{0, 1, \ldots, v-1\}$, $v_{max} = \prod_{i=1}^{i=t} u_i$, and u_i is the i-th cardinality taken in decreasing order from the t larger cardinalities of the columns of $\mathcal{M}$. The injective function $g(w_i) : \mathcal{W} \to \mathcal{I}$ is defined in (4.3). The function $g(w_i)$ is equivalent to the transformation of a v-ary number to the decimal system:

$$g(w_i) = \sum_{j=2}^{t} J_j, \; s.t. \; J_j = w_{i,j-1} \times V_{c_{i,j}} + w_{i,j}. \tag{4.3}$$

- ▷ The use of the injections represents an efficient method to manipulate the information that will be stored in the data structure P used in the construction process of $\mathcal{M}$ as a covering array. The matrix P is of size $\binom{k}{t} \times v_{max}$. The matrix P counts the number of times that each combination appears in $\mathcal{M}$ in the different t-tuples. Each row of P represents a different t-tuple, while each column contains a different combination of symbols. The management of the cells $p_{i,j} \in P$ is done through the functions $f(c_i)$ and $g(w_j)$; while $f(c_i)$ retrieves the row related with the t-tuple c_i, the function $g(w_i)$ returns the column that corresponds to the combination of symbol w_i.

- ▷ The set $\mathcal{R} = \{r_1, r_2, \ldots, r_N\}$, where each element $r_i \in \mathcal{R}$ will be a test set of the covering array that will be constructed. The cardinality of the set $\mathcal{R}$ is N, the expected number of rows in the covering array.

4.2.2 Initial solution

The *initial solution* $\mathcal{M}$ is constructed by generating $\mathcal{M}$ as a matrix with maximum Hamming distance. The Hamming distance $d(x, y)$ between two rows $x, y \in \mathcal{M}$ is the number of elements in which they differ. Let r_i be a row of the matrix $\mathcal{M}$. To generate a random matrix $\mathcal{M}$ of maximum Hamming distance, follow these steps:

1. Generate the first row r_1 at random.
2. Generate two rows l_1, l_2 at random, which will be candidate rows.

3. Select the candidate row l_i that maximizes the Hamming distance according to (4.4) and added to the i-th row of the matrix $\mathcal{M}$.

$$g(r_i) = \sum_{s=1}^{i-1}\sum_{j=1}^{k} d(m_{s,j}, m_{i,j}), \textbf{where } d(m_{s,j}, m_{i,j}) = \begin{cases} 1 \textbf{ if } m_{s,j} \neq m_{i,j} \\ 0 \textbf{ Otherwise} \end{cases} \tag{4.4}$$

4. Repeat from step 2 until $\mathcal{M}$ is completed.

Figure 4.1 illustrates this method. Figure 4.1(a) shows an example of the Hamming distance between two rows r_1, r_2 that are already in the matrix $\mathcal{M}$ and two candidate rows l_1, l_2; the row l_1 is which maximizes the Hamming distance. Figure 4.1(b) shows the entire initial solution matrix $\mathcal{M}$ generated according with the method described for $MCA(16; 2, 6, 4^2 3^2 2^2)$, the last column shows the Hamming distance for each row of the matrix.

(a)

Rows

$$\begin{cases} r_1 = \{\ 1 \quad 1 \quad 1 \quad 2 \quad 0 \quad 0\ \} \\ r_2 = \{\ 3 \quad 2 \quad 0 \quad 1 \quad 1 \quad 1\ \} \\ l_1 = \{\ 0 \quad 3 \quad 2 \quad 0 \quad 0 \quad 1\ \} \\ l_2 = \{\ 0 \quad 3 \quad 2 \quad 2 \quad 1 \quad 0\ \} \end{cases}$$

Distances

$$\begin{cases} d(r_1, l_1) = 5 \\ d(r_2, l_1) = 5 \\ g(l_1) = d(r_1, l_1) + d(r_2, l_1) = 10 \\ \\ d(r_1, l_2) = 4 \\ d(r_2, l_2) = 5 \\ g(l_2) = d(r_1, l_2) + d(r_2, l_2) = 9 \end{cases}$$

(b)

1	1	1	2	0	0	
3	2	0	1	1	1	6
0	3	2	0	0	1	10
2	0	0	2	1	0	14
1	1	2	0	0	0	16
3	2	1	1	1	1	22
2	0	0	0	0	1	24
0	3	2	1	1	0	30
2	3	1	2	1	0	32
3	1	1	2	0	1	36
0	0	0	1	0	0	40
1	2	2	0	1	1	46
3	2	2	0	1	0	46
1	0	0	2	0	1	52
2	1	1	1	0	1	56
0	3	2	2	1	0	60

Figure 4.1: Example of how to construct a initial solution for $MCA(16; 2, 6, 4^2 3^2 2^2)$. (a) Example of the hamming distance between two rows r_1, r_2 that are already in the matrix $\mathcal{M}$ and two candidate rows l_1, l_2; (b) Initial solution $\mathcal{M}$ for $MCA(16; 2, 6, 4^2 3^2 2^2)$, the last column shows the Hamming distance for each row of the matrix.

Figure 4.2 contains the sets $\mathcal{C}, \mathcal{W}$, and $\mathcal{R}$ derived from the $MCA(16; 2, 6, 4^2 3^2 2^2)$ shown in Figure 4.1(b).

(a) $\mathcal{C}$

$$
\begin{aligned}
c_1 &= \{1,2\} \\
c_2 &= \{1,3\} \\
c_3 &= \{1,4\} \\
c_4 &= \{1,5\} \\
c_5 &= \{1,6\} \\
c_6 &= \{2,3\} \\
c_7 &= \{2,4\} \\
c_8 &= \{2,5\} \\
c_9 &= \{2,6\} \\
c_{10} &= \{3,4\} \\
c_{11} &= \{3,5\} \\
c_{12} &= \{3,6\} \\
c_{13} &= \{4,5\} \\
c_{14} &= \{4,6\} \\
c_{15} &= \{5,6\}
\end{aligned}
$$

(b) $\mathcal{W}$

$$
\begin{aligned}
w_1 &= \{(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),(2,0),(2,1),(2,2),(2,3),(3,0),(3,1),(3,2),(3,3)\} \\
w_2 &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0),(3,1),(3,2)\} \\
w_3 &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0),(3,1),(3,2)\} \\
w_4 &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1),(3,0),(3,1)\} \\
w_5 &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1),(3,0),(3,1)\} \\
w_6 &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0),(3,1),(3,2)\} \\
w_7 &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0),(3,1),(3,2)\} \\
w_8 &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1),(3,0),(3,1)\} \\
w_9 &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1),(3,0),(3,1)\} \\
w_{10} &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\} \\
w_{11} &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)\} \\
w_{12} &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)\} \\
w_{13} &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)\} \\
w_{14} &= \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)\} \\
w_{15} &= \{(0,0),(0,1),(1,0),(1,1)\}
\end{aligned}
$$

(c) $\mathcal{R}$

$$
\begin{pmatrix}
2 & 2 & 2 & 3 & 1 & 1 \\
4 & 3 & 1 & 2 & 2 & 2 \\
1 & 4 & 3 & 1 & 1 & 2 \\
3 & 1 & 1 & 3 & 2 & 1 \\
2 & 2 & 3 & 1 & 1 & 1 \\
4 & 3 & 2 & 2 & 2 & 2 \\
3 & 1 & 1 & 1 & 1 & 2 \\
1 & 4 & 3 & 2 & 2 & 1 \\
3 & 4 & 2 & 3 & 2 & 1 \\
4 & 2 & 2 & 3 & 1 & 2 \\
1 & 1 & 1 & 2 & 1 & 1 \\
2 & 3 & 3 & 1 & 2 & 2 \\
4 & 3 & 3 & 1 & 2 & 1 \\
2 & 1 & 1 & 3 & 1 & 2 \\
3 & 2 & 2 & 2 & 1 & 2 \\
1 & 4 & 3 & 3 & 2 & 1
\end{pmatrix}
$$

Figure 4.2: Example of the sets $\mathcal{C}, \mathcal{W}$, and $\mathcal{R}$. (a) shows an example of set $\mathcal{C}$; (b) shows an example of the set $\mathcal{W}$; (c) shows an example of the set $\mathcal{R}$.

4.2.3 Evaluation function

The *evaluation function* is used to estimate the goodness of a candidate solution. Previously reported metaheuristic algorithms for constructing covering arrays have commonly evaluated the quality of a potential solution (covering array) as the number of combination of symbols missing in the matrix $\mathcal{M}$ (Cohen et al., 2003; Nurmela, 2004; Shiba et al., 2004). Then, the expected solution will be zero missing. In the proposed simulated annealing implementation this evaluation function was also used.

For a particular matrix $\mathcal{M}$ that represents a mixed covering array, and sets $\mathcal{C}$ and $\mathcal{W}$ (previously described), a formal definition for this function is shown in (4.5):

$$
\begin{gathered}
E(\mathcal{M}, \mathcal{C}, \mathcal{W}) = \textstyle\sum_{\forall c \in \mathcal{C}} \sum_{\forall \mathcal{W}_i \in \mathcal{W}} \sum_{\forall w \in \mathcal{W}_i} g(\mathcal{M}, c, w), \\
\textbf{where } g(\mathcal{M}, c, w) = \begin{cases} 1 & \texttt{if } w \texttt{ in } c \texttt{ has not been covered yet in } \mathcal{M} \\ 0 & \textbf{otherwise} \end{cases}
\end{gathered}
\tag{4.5}
$$

The computational complexity of evaluating $E(\mathcal{M}, \mathcal{C}, \mathcal{W})$ is equivalent to (4.6), because the operation requires to examine the N rows of the matrix $\mathcal{M}$ and the $\binom{k}{t}$ different t-tuples.

$$
O\left(N\binom{k}{t}\right). \tag{4.6}
$$

With the aim of improving the time of this calculation, we implemented a matrix called P. Each element $p_{i,j} \in P$ contains the number of times that the i-th combination of symbols is found in the t-tuple $c_j \in \mathcal{C}$; the value of $p_{i,j}$ is not taken into account if the i-th combination of symbols must not be included in the t-tuple c_j.

An example of the use of the evaluation function $E(\mathcal{M}, \mathcal{C}, \mathcal{W})$ is shown in Table 4.1, where the number of missing symbol combinations in matrix $\mathcal{M}$ shown in Figure 4.2(c) is counted. Table 4.1(a) shows the use of injective function $f(c_i)$. Table 4.1(b) presents the matrix P. A symbol $*$ represents that a combination of symbols must not be satisfied in a certain combination c. Note that the matrix $\mathcal{M}$ still has 16 missing combinations making it a non mixed covering array.

Table 4.1: Matrix P of symbol combinations covered in $\mathcal{M}$ (see Figure 4.2(c)) and results from evaluating $\mathcal{M}$ with $E(\mathcal{M}, \mathcal{C}, \mathcal{W})$.

(a) Applying $f(c_i)$.

$\mathcal{C}$			
index	*t-tuple*		$f(c_i)$
c_1	1	2	0
c_2	1	3	1
c_3	1	4	3
c_4	1	5	6
c_5	1	6	10
c_6	2	3	2
c_7	2	4	4
c_8	2	5	7
c_9	2	6	11
$c_1 0$	3	4	5
$c_1 1$	3	5	8
$c_1 2$	3	6	12
$c_1 3$	4	5	9
$c_1 4$	4	6	13
$c_1 5$	5	6	14

(b) Matrix P.

	$g(w_{i,j})$															
$f(c_i)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	1	1	1	1	0	1	1	0	1	0	1	1	0
1	1	0	1	1	1	1	1	1	0	1	1	1	*	*	*	*
2	1	1	1	1	0	1	1	1	1	1	1	1	*	*	*	*
3	1	1	1	1	1	1	1	1	*	*	*	*	*	*	*	*
4	1	1	1	1	1	1	1	1	*	*	*	*	*	*	*	*
5	1	0	0	0	1	1	1	1	1	0	1	1	*	*	*	*
6	1	1	1	1	1	1	1	1	0	1	1	1	*	*	*	*
7	1	1	1	0	0	1	1	1	*	*	*	*	*	*	*	*
8	1	1	1	1	1	1	1	1	*	*	*	*	*	*	*	*
9	1	1	1	0	1	1	1	1	1	*	*	*	*	*	*	*
10	1	1	1	1	1	1	*	*	*	*	*	*	*	*	*	*
11	1	1	1	1	1	1	*	*	*	*	*	*	*	*	*	*
12	1	1	1	1	1	1	*	*	*	*	*	*	*	*	*	*
13	1	1	1	1	1	1	*	*	*	*	*	*	*	*	*	*
14	1	1	1	1	*	*	*	*	*	*	*	*	*	*	*	*

To avoid the expensive cost (see (4.6)) at every call of $E(\mathcal{M}, \mathcal{C}, \mathcal{W})$, the matrix P is used for a partial recalculation of the cost of $\mathcal{M}$, i.e., the cost of changing a symbol in a cell $m_{i,j} \in \mathcal{M}$ is determined and only the affected t-tuples in P are updated, modifying the results from $E(\mathcal{M}, \mathcal{C}, \mathcal{W})$ according to that changes. The cells in P that must be updated when changing a symbol from $m_{i,j} \in \mathcal{M}$ are the t-tuples that involve the column j of the matrix $\mathcal{M}$. On this way, the complexity taken for the update of $E(\mathcal{M}, \mathcal{C}, \mathcal{W})$ is reduced to (4.7):

$$O\left(2 \times \binom{k-1}{t-1}\right). \tag{4.7}$$

4.2.4 Neighborhood function

Given that the developed simulated annealing implementation is based on Local Search (LS) then a neighborhood function must be defined. The main objective of the neighborhood function is to identify the set of potential solutions which can be reached from the current solution in a LS algorithm. In case two or more neighborhoods present complementary characteristics, it is then possible and interesting to create more powerful compound neighborhoods. The advantage of such an approach is well documented in (Cavique et al., 1999). Following this idea, and based on the results of our preliminary experimentations, a neighborhood structure composed by two different functions is proposed for this simulated annealing algorithm implementation.

Two *neighborhood functions* were implemented to guide the local search of DSSA algorithm. The neighborhood function $\mathcal{N}_1(s)$ makes a random search of a missing t-tuple, then tries by setting the j-th combination of symbols in every row of $\mathcal{M}$. The neighborhood function $\mathcal{N}_2(s)$ randomly chooses a position (i, j) of the matrix $\mathcal{M}$ and makes all possible changes of symbol. During the search process a combination of both $\mathcal{N}_1(s)$ and $\mathcal{N}_2(s)$ neighborhood functions is employed by DSSA. The former is applied with probability $\mathbb{P}$, while the latter is employed at an $(1 - \mathbb{P})$ rate. This combined neighborhood function $\mathcal{N}_3(s, x)$ is defined in (4.8), where x is a random number in the interval [0, 1).

$$\mathcal{N}_3(s, x) = \begin{cases} \mathcal{N}_1(s) & \text{if } x \leq \mathbb{P} \\ \mathcal{N}_2(s) & \text{if } x > \mathbb{P} \end{cases} \tag{4.8}$$

4.2.5 Cooling schedule

The *cooling schedule* determines the degree of uphill movement permitted during the search and is thus critical to the simulated annealing algorithm's performance. The parameters that define a cooling schedule are: an initial temperature, a final temperature or a stopping criterion, the maximum number of neighboring solutions that can be generated at each temperature, and a rule for decrementing the temperature. The cooling schedule governs the convergence of the SA algorithm. At the beginning of the search, when the temperature is large, the probability of accepting solutions of worse quality than the current solution (uphill moves) is high. It allows the algorithm to escape from local minima. The probability of accepting such moves is gradually decreased as the temperature goes to zero.

Cooling schedules which rapidly decrement the temperature can lead the search process to get trapped in an early local minima. On the contrary, a very slow cooling of the temperature guides the algorithm towards non-promising searching regions, resulting often in a waste of computational time. A good selection of the cooling schedule is thus critical to the SA algorithm's performance.

The literature offers a number of different cooling schedules, see for instance (Aarts and Van Laarhoven, 1985; Atiqullah, 2004). They can be divided into two main categories: static and dynamic. In a static cooling schedule, the parameters are fixed and cannot be changed during the execution of the algorithm. With a dynamic cooling schedule the parameters are adaptively changed during the execution.

In DSSA we preferred a geometrical cooling scheme mainly for its simplicity. It starts at an initial temperature T_i which is decremented at each round by a factor α using the relation (4.9). For each temperature, the maximum number of visited neighboring solutions is L. It depends directly on the parameters (N, k, and $V4$) of the studied covering array. This is because more moves are required for covering arrays with alphabets of greater cardinality.

$$T_k = \alpha \, T_{k-1}. \tag{4.9}$$

4.2.6 Termination condition

The *stop criterion* for DSSA is either when the current temperature reaches T_f, when it ceases to make progress, or when a valid covering array is found. In the proposed implementation a lack of progress exists if after ϕ (frozen factor) consecutive temperature decrements the best-so-far solution is not improved.

4.2.7 Simulated annealing pseudocode

Algorithm 19 presents the simulated annealing heuristic as described above. The meaning of the four functions is obvious: INITIALIZE computes a start solution and initial values of the parameters T and L; GENERATE selects a solution from the neighborhood of the current solution, using the neighborhood function $\mathcal{N}_3(s, x)$; CALCULATE_CONTROL computes a new value for the parameter T (cooling schedule) and the number of consecutive temperature decrements with no improvement in the solution.

In the following sections we propose the use of Supercomputing and Grid Computing in order to accelerate the construction of covering arrays using the developed simulated annealing algorithm.

Algorithm 19: Sequential simulated annealing for the CAC problem

$INITIALIZE(M,T,L)$; `/* Create the initial solution. */`
$\mathcal{M}^{\star} \leftarrow \mathcal{M}$; `/* Memorize the best solution. */`
repeat
  for $i \leftarrow 1$ **to** L **do**
    $\mathcal{M}_i \leftarrow GENERATE(\mathcal{M})$; `/* Perturb current state. */`
    $\Delta E \leftarrow E(\mathcal{M}_i) - E(\mathcal{M})$; `/* Evaluate cost function. */`
    $x \leftarrow random$; `/* Range [0,1). */`
    if $\Delta E < 0$ **or** $e^{(-\frac{\Delta E}{T})} > x$ **then**
      $\mathcal{M} \leftarrow \mathcal{M}_i$; `/* Accept new state. */`
      if $E(\mathcal{M}) < E(\mathcal{M}^{\star})$ **then**
        $\mathcal{M}^{\star} \leftarrow \mathcal{M}$; `/* Memorize the best solution. */`
      end if
    end if
  end for
  $CALCULATE\_CONTROL(T, \phi)$
until *termination condition is satisfied*;

4.3 Grid approach

Simulated annealing is inherently sequential and hence very slow for problems with large search spaces. Several attempts have been made to speed up this process, such as development of special purpose computer architectures (Ram et al., 1996). As an alternative, we propose a Grid deployment of the parallel simulated annealing algorithm for constructing covering arrays, introduced in the previous section. In order to fully understand the Grid implementation developed in this work, this section will introduce all the details regarding the Grid Computing Platform used and then, the different execution strategies will be exposed.

4.3.1 Grid computing platform

The evolution of Grid Middlewares has enabled the deployment of Grid e-Science infrastructures delivering large computational and data storage capabilities. Current infrastructures, such as the one used in this work, EGI, rely on gLite mainly as core middleware supporting several services in some cases. World-wide initiatives, such as EGI, aim at linking and sharing components and resources from several European NGI.

In the EGI infrastructure, jobs are specified through a job description language (Pacini, 2011) or JDL that defines the main components of a job: executable, input data, output data, arguments, and restrictions. The restrictions define the features a resource should provide, and could be used for meta-scheduling or for local scheduling (such as in the case of MPI jobs). Input data could be small or large and job-specific or common to all jobs, which affects the protocols and mechanisms needed. Executables are either compiled or multiplatform codes (scripts, Java, Perl), and output data suffer from similar considerations as input data. In this section, it shows the details about Developed Grid Simulated Annealing (DGSA).

4.3.2 Preprocessing task: selecting the most appropriate compute elements

A production infrastructure such as EGI involves tens of thousands of resources from hundreds of sites, involving tens of countries and a large human team. Since it is a general-purpose platform, and although there is a common middleware and a recommended operating system, the heterogeneity in the configuration and operation of the resources is inevitable. This heterogeneity, along with other social and human factors such as the large geographical coverage and the different skills of operators introduces a significant degree of uncertainty in the infrastructure. Even considering that the service level required is around 95%, it is statistically likely to find in each large execution sites that are not working properly. Thus, prior to beginning the experiments, it is necessary to do empirical tests to define a group of valid computing resources (CEs) and this way facing resource setup problems. These tests can give some real information like computational speed, primary and secondary memory sizes and I/O transfer speed. These data, in case there are huge quantities of resources, will be helpful to establish quality criteria choosing resources.

4.3.3 Asynchronous schema

Once the computing elements, where the jobs will be submitted, have been selected, the next step involves correctly specifying the jobs. In that sense, it will be necessary to produce the specification using the job description language in gLite. An example of a JDL file can be seen in Figure 4.3.

```
Type = "Job";
VirtualOrganisation = "biomed";
Executable = "test.sh";
Arguments  = "16 21 3 2";
StdOutput  = "std.out";
StdError   = "std.err";
InputSandbox = {"/home/CA_experiment/DGSA.c",
                "/home/CA_experiment/N16k21v3t2.ca",
                "/home/CA_experiment/test.sh"};
OutputSandbox = {"std.out","std.err","N16k21v3t2.ca"};
```

Figure 4.3: JDL example for the case of $N = 16, k = 21, v = 3, t = 2$.

As it can be seen in Figure 4.3, the specification of the job includes: the virtual organisation where the job will be launched (VirtualOrganisation), the main file that will start the execution of the job (Executable), the arguments that will used for invoking the executable (Arguments), the files in which the standard outputs will be dumped (StdOutput y StdError), and finally the result files that will be returned to the user interface (OutputSandBox).

So, the most important part of the execution lies in the program (a shell-script) specified in the *Executable* field of the description file. The use of a shell-script instead of directly using the executable (DGSA) is mandatory due to the heterogeneous nature present in the Grid. Although the conditions vary between different resources, as it was said before, the administrators of the sites are recommended to install Unix-like operative systems. This measure makes sure that all the developed programs will be seamlessly executed in any machine of the Grid infrastructure. The source code must be dynamically compiled in each of the computing resources hosting the jobs. Thus, basically, the shell-script works like a wrapper that looks for a *gcc*-like compiler (the source code is written in the C language), compiles the source code and finally invokes the executable with the proper arguments (values of N, k, v and t respectively).

One of the most crucial parts of any Grid deployment is the development of an automatic system for controlling and monitoring the evolution of an experiment. Basically, the system will be in charge of submitting the different gLite jobs (the number of jobs is equal to the value of the parameter $S = number\ of\ workers$), monitoring the status of these jobs, resubmitting (in case a job has failed or it has been successfully completed but the simulated annealing algorithm has not already converged) and retrieving the results. This automatic system has been implemented as a master process which periodically (or asynchronously as the name of the schema suggests) oversees the status of the jobs.

This system must possess the following properties: completeness, correctness, quick performance and efficiency on the usage of the resources. Regarding the completeness, we have take into account that an experiment will involve a lot of jobs and it must be ensured that all jobs are successfully completed at the end. The correctness implies that there should be a guarantee that all jobs produce correct results which are comprehensive presented to the user and that the data used is properly updated and coherent during the whole experiment (the master must correctly update the file with the .ca extension showed in the JDL specification in order the Simulated Annealing algorithm to converge). The quick performance property implies that the experiment will finish as quickly as possible. In that sense, the key aspects are: a good selection of the resources that will host the jobs (according to the empirical tests performed in the preprocessing stage) and an adequate resubmission policy (sending new jobs to the resources that are being more productive during the execution of the experiment). Finally, if the on-the-fly tracking of the most productive computing resources is correctly done, the efficiency in the usage of the resources will be achieved.

Due to the asynchronous behavior of this schema, the number of slaves (jobs) that can be submitted (the maximum size of N) is only limited by the infrastructure. However, other schemas such as the one showed in the next point, could achieve a better performance in certain scenarios.

4.3.4 Synchronous schema

This schema a sophisticated mechanism known, in Grid terminology, as submission of *pilot jobs*. The submission of pilot jobs is based on the master-worker architecture and supported by the DIANE (DIANE, 2011) + Ganga (Moscicki et al., 2009) tools. When the processing begins a master process (a server) is started locally, which will provide tasks to the worker nodes until all the tasks have been completed, being then dismissed. On the other side, the worker agents are jobs running on the Working Nodes of the Grid which communicate with the master. The master must keep track of the tasks to assure that all of them are successfully completed while workers provide the access to a CPU previously reached through scheduling, which will process the tasks. If, for any reason a task fails or a worker losses contact with the master, the master will immediately reassign the task to another worker. The whole process is exposed in Figure 3.14. master is continuously in contact with the slaves.

However, before initiating the process or execution of the master/worker jobs, it is necessary to define their characteristics. Firstly, the specification of a run must include the master configuration (workers and heartbeat timeout). It is also necessary to establish master scheduling policies such as the maximum number of times that a lost or failed task is assigned to a worker; the reaction when a task is lost or fails; and the number of resubmissions before a worker is removed. Finally, the master must know the arguments of the tasks and the files shared by all tasks (executable and any auxiliary files).

At this point, the master can be started using the specification described above. Upon checking that all is right, the master will wait for incoming connections from the workers.

Workers are generic jobs that can perform any operation requested by the master which are submitted to the Grid. In addition, these workers must be submitted to the selected CEs in the pre-processing stage. When a worker registers to the master, the master will automatically assign it a task.

This schema has several advantages derived from the fact that a worker can execute more than one task. Only when a worker has successfully completed a task the master will reassign it a new one. In addition, when a worker demands a new task it is not necessary to submit a new job. This way, the queuing time of the task is intensively reduced. Moreover, the dynamic behavior of this schema allows achieving better performance results, in comparison to the asynchronous schema.

However, there are also some disadvantages that must be mentioned. The first issue refers to the unidirectional connectivity between the master host and the worker hosts (Grid node). While the master host needs inbound connectivity, the worker node needs outbound connectivity. The connectivity problem in the master can be solved easily by opening a port in the local host; however the connectivity in the worker will rely in the remote system configuration (the CE). So, in this case, this extra detail must be taken into account when selecting the computing resources. Another issue is defining an adequate timeout value. If, for some reason, a task working correctly suffers from temporary connection problems and exceeds the timeout threshold it will cause the worker being removed by the master. Finally, a key factor will be to identify the rightmost number of worker agents and tasks. In addition, if the number of workers is on the order of thousands (i.e. when N is about 1000) bottlenecks could be met, resulting on the master being overwhelmed by the excessive number of connections.

4.4 Parallel simulated annealing

Parallelization is recognized like a powerful strategy to increase algorithms efficiency; however, simulated annealing parallelization is a hard task because it is essentially a sequential process. The best parallel scheme is still the object of current research, since the "annealing community" has so far not achieved a common agreement with regards to a general approach for the serial simulated annealing.

In evaluating performance of a Parallel Simulated Annealing (PSA), it needs to consider solution quality as well as execution speed. The execution speed may be quantified in terms of *speed-up* ($\mathcal{S}$) and *efficiency* ($\mathcal{E}$). The $\mathcal{S}$ is defined as the ratio of the execution time (on one processor) by the sequential simulated annealing to that by the PSA (on $\mathcal{P}$ processors) for an equivalent solution quality. In the ideal case, $\mathcal{S}$ would be equal to $\mathcal{P}$. The $\mathcal{E}$ is defined as the ratio of the actual $\mathcal{S}$ to the ideal $\mathcal{S}(\mathcal{P})$.

Next, we propose three parallel implementations of the simulated annealing algorithm described in Section 4.2. For these cases, let $\mathcal{P}$ denote the number of processors and L the length of Markov chain.

4.4.1 Independent search approach

A common approach to parallelizing simulated annealing is the Independent Search Approach (ISA) (Aarts and Van Laarhoven, 1985; Lee and Lee, 1996; Czech, 2006). In this approach each processor independently perturbs the configuration, evaluates the cost, and decides on the perturbation. The processors $\mathcal{P}_i$, $i = 0, 1, \ldots, \mathcal{P}-1$, carry out the independent annealing searches using the same initial solution and cooling schedule as in the sequential algorithm. At each temperature $\mathcal{P}_i$ executes $N \times k \times v^2$ annealing steps. When each processor finishes, it sends its results to processor $\mathcal{P}_0$. Finally, processor $\mathcal{P}_0$ chooses the final solution among the local solutions.

We have implemented a simulated annealing algorithm using ISA approach for constructing covering arrays. In the developed implementation, the processors do not interact during individual annealing processes until all processors find their final solution. Then, the best of the solutions is saved and the others are discarded.

4.4.2 Semi-independent search approach

Aarts and Van Laarhoven (1985) introduced a new parallel simulated annealing algorithm named *division algorithm*. In the division algorithm, the number of iterations at each temperature is divided equally between the processors. After a change in temperature, each processor may simply start from the final solution obtained by that processor at the previous temperature. The best solution from all the processors is then taken to be the final solution. Another variant of this approach is to communicate the best solution from all the processors to each processor every time the temperature changes. Aarts and Van Laarhoven found no significant differences in the performance of these two variants.

We have developed an implementation of division algorithm; we named the implementation Semi-Independent Search Approach (SSA). In SSA, parallelism is obtained by dividing the effort of generation a Markov chain over the available processors. A Markov chain is divided into $\mathcal{P}$ sub-chains of the length $\lfloor L/\mathcal{P} \rfloor$. In this approach, the processors exchange local information including intermediate solutions and their costs. Then, each processor restarts from the best intermediate ones.

Compared to the ISA, communication overhead in this SSA approach would be increased. However, each processor can utilize the information from other processors such that the decrease in computations and idle times can be greater than the increase in communication overhead. For instance, a certain processor which is trapped in an inferior solution can recognize its state by comparing it with others and may accelerate the annealing procedure. That is, processors may collectively converge to a better solution.

4.4.3 Cooperative search approach

In order to improve the performance of the SSA approach, we propose the Cooperative Search Approach (CSA), it used asynchronous communication among processors accessing the global state to eliminate the idle times. Each processor follows a separate search path, accesses the global state which consists of the current best solution and its cost whenever it finished a Markov subchain and updates the state if necessary. Once a processor gets the global state, it proceeds to the next Markov subchain with any delay.

Unlike SSA, CSA having the following characteristics:

- ▷ Idle times can be reduced since asynchronous communications overlap a part of the computation.
- ▷ Less communication overhead, an isolated access to the global state is needed by each processor at the end of each Markov subchain.
- ▷ The probability of being trapped in a local optimum can be smaller. This is because not all the processors start from the same state in each Markov subchain.

For more details about this construction, the reader is referred to (Avila-George et al., 2012b). Additionally, the constructed covering arrays have been uploaded to the Covering Array Repository (CAR) described in Appendix A. This repository is available under request at http://www.tamps.cinvestav.mx/~jtj/CA.php.

4.5 Summary

In this chapter we introduced the simulated annealing technique and described its basic structure. We have presented an improved simulated annealing algorithm for constructing covering arrays. We proposed the use of Grid Computing in order to accelerate the DSSA. We ended with the presentation of three parallel simulated annealing approaches to construct covering arrays.

The next chapter presents the experimental results obtained from the implementation of simulated annealing algorithm, following the details described in the present chapter.

Chapter 5

Experimental results

This chapter presents the results obtained by the developed simulated annealing algorithm. Section 5.1 analyzes the global performance of the developed simulated annealing algorithm and the influences that some of its key features have on it. Section 5.2 presents a methodology for fine-tuning the developed simulated annealing approach. Section 5.3 presents the results obtained from the DSSA. The results are compared against the best algorithms obtained from the literature for constructing uniform and mixed covering arrays. Section 5.4 presents the results of comparing DGSA against two of the best algorithms from the literature; it created a new benchmark composed by 60 ternary covering arrays instances where $5 \leq k \leq 100$ and $2 \leq t \leq 4$. Section 5.5 presents the results of comparing Developed Parallel Simulated Annealing (DPSA) against the best bounds from the literature. Finally, Section 5.6 is designed to illustrate the development of test configurations for real software applications.

5.1 Analyzing the performance of simulated annealing

The purpose of this section is to experimentally analyze the global performance of the developed simulated annealing algorithm and the influences that some of its key features have on it. Next, we present the results of the experiments carried out for this purpose.

5.1.1 Influence of the initial solution

In this experiment we compare the performance of two different methods for constructing the initial solution of the developed simulated annealing. The first one is commonly used in the literature (Cohen et al., 2008), and creates the initial solution by assigning randomly a symbol in v_i at each element m_{ij} of the array. The second one is the procedure described in Section 4.2.2.

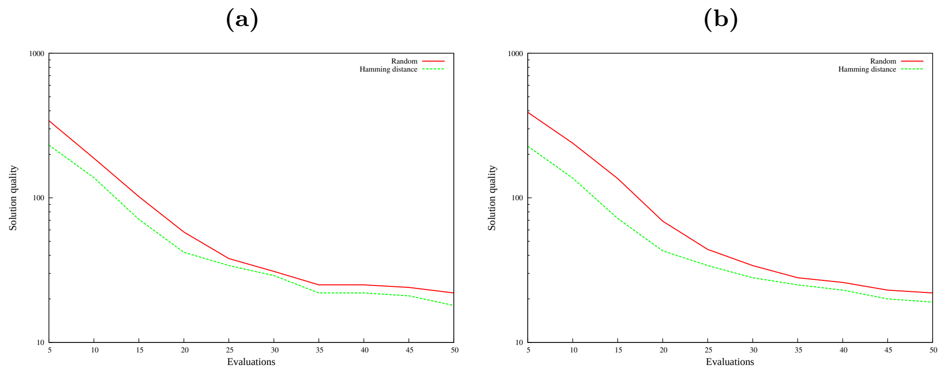

Figure 5.1: Performance comparison of two different initialization methods for the developed simulated annealing over the instances: (a) $MCA(29; 2, 22, 5^1 4^1 33^4 2^2)$ and (b) $MCA(137; 3, 9, 5^2 4^4 3^3)$.

Both initialization methods (called here *Maximum Hamming Distance* and *Random*, respectively) were integrated into the developed simulated annealing source code and executed 31 times over the next two mixed covering arrays: $MCA(137; 3, 9, 5^2 4^4 3^3)$ and $MCA(29; 2, 22, 5^1 4^1 33^4 2^2)$. The results achieved by the developed simulated annealing over these instances are illustrated in Figure 5.1. The plot represents the iterations of the developed simulated annealing against the average solution quality attained from the starting arrays generated with the compared initialization methods. Figure 5.1 discloses that the developed simulated annealing using *Maximum Hamming Distance* solutions performs much better than the simulated annealing algorithm that starts from a randomly generated solution.

5.1.2 Influence of the neighborhood functions

The neighborhood function is a critical element for the performance of any local search algorithm. In order to further examine the influence of this element on the global performance of the developed simulated annealing implementation we have performed some experimental comparisons using the following neighborhood functions (described in Section 4.2.4):

▷ $\mathcal{N}_1(M)$

▷ $\mathcal{N}_2(M, P)$

▷ $\mathcal{N}_3(M, P)$

For this experiment each one of the studied neighborhood functions was implemented within the developed simulated annealing algorithm, compiled and executed independently 31 times over the next two mixed covering arrays: $MCA(137; 3, 9, 5^2 4^4 3^3)$ and $MCA(29; 2, 22, 5^1 4^1 3 3^4 2^2)$. The results of this experiment are summarized in Figure 5.2. It shows the differences in terms of average solution quality attained by the developed simulated annealing, when each one of the studied neighborhood relations is used to solve the instances $MCA(137; 3, 9, 5^2 4^4 3^3)$ and $MCA(29; 2, 22, 5^1 4^1 3 3^4 2^2)$. From this graph it can be observed that the worst performance is attained by the developed simulated annealing approach when the neighborhood function called $\mathcal{N}_1$ is used. The functions $\mathcal{N}_2$ and $\mathcal{N}_3$ produce better results compared with $\mathcal{N}_1$ since they improve the solution quality faster. Finally, the best performance is attained by the developed simulated annealing algorithm when it is employed the neighborhood function $\mathcal{N}_3$, which is a compound neighborhood combining the complementary characteristics of both $\mathcal{N}_1$ and $\mathcal{N}_2$.

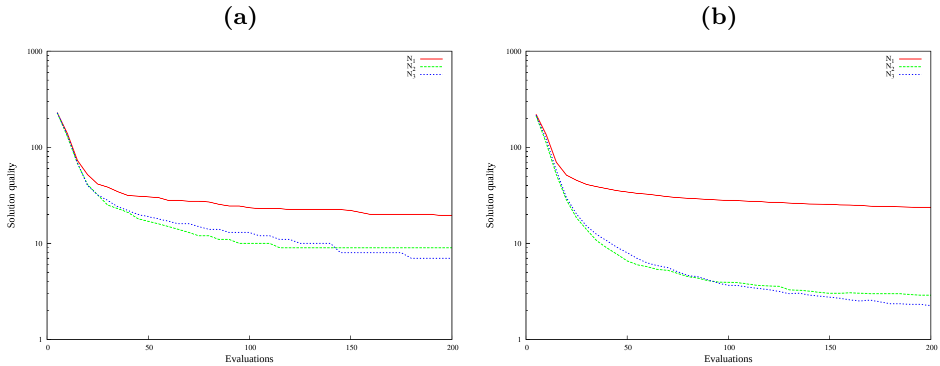

Figure 5.2: Performance comparison of four neighborhood functions using simulated annealing over the instances: (a) $MCA(29; 2, 22, 5^1 4^1 3 3^4 2^2)$ and (b) $MCA(137; 3, 9, 5^2 4^4 3^3)$.

5.2 Fine tuning of the neighborhood functions

It is well-known that the performance of a simulated annealing algorithm is sensitive to parameter tuning. In this sense, we follow a methodology for a fine tuning of the two neighborhood functions used in the developed simulated annealing algorithm. The fine tuning was based on the linear Diophantine equation (5.1), where x_i represents a neighborhood function and its value set to 1, $\mathbb{P}_i$ is a value in $\{0.0, 0.1, \ldots, 1.0\}$ that represents the probability of executing x_i, and q is set to 1.0 which is the maximum probability of executing any x_i.

$$\mathbb{P}_1 x_1 + \mathbb{P}_2 x_2 = q \tag{5.1}$$

A solution to the given linear Diophantine equation must satisfy (5.2). This equation has 11 solutions, each solution is an experiment that test the degree of participation of each neighborhood function in the developed simulated annealing implementation to accomplish the construction of an CA.

$$\sum_{i=1}^{2} \mathbb{P}_i x_i = 1.0 \tag{5.2}$$

It is well-known that the performance of a simulated annealing algorithm is sensitive to parameter tuning. In this sense, we follow a methodology for a fine tuning of the two neighborhood functions used in the developed simulated annealing algorithm. The fine tuning was based on the next linear Diophantine Equation, $\mathbb{P}_1 x_1 + \mathbb{P}_2 x_2 = q$. Where x_i represents a neighborhood function and its value set to 1, $\mathbb{P}_i$ is a value in $\{0.0, 0.1, .., 1.0\}$ that represents the probability of executing x_i, and q is set to 1.0 which is the maximum probability of executing any x_i. A solution to the given linear Diophantine Equation must satisfy $\sum_{i=1}^{2} \mathbb{P}_i x_i = 1.0$. This Equation has 11 solutions, each solution is an experiment that tests the grade of participation of each neighborhood function in the developed simulated annealing implementation to accomplish the construction of a mixed covering array.

Every combination of the probabilities was applied by the developed simulated annealing to construct the set of mixed covering arrays shown in Table 5.1(a) and each experiment was run 31 times, with the obtained data for each experiment we calculate the median. A summary of the performance of the developed simulated annealing with the probabilities that solved the 100% of the runs is shown in Table 5.1(b).

Finally, given the results shown in Figure 5.3, the best configuration of probabilities was $\mathbb{P}_1 = 0.3$ and $\mathbb{P}_2 = 0.7$ because it found the mixed covering arrays in smaller

Table 5.1: Fine tuning of the neighborhood functions. (a) A set of 7 mixed covering arrays configurations; (b) Performance of the developed simulated annealing with the 11 combinations of probabilities which solved the 100% of the runs to construct the mixed covering arrays listed in (a).

(a)

Id	**MCA description**
mca_1	$MCA(81; 2, 16, 9^2 8^2 7^2 6^2 5^2 4^2 3^2 2^2)$
mca_2	$MCA(42; 2, 19, 7^1 6^1 5^1 4^5 3^8 2^3)$
mca_3	$MCA(36; 2, 20, 6^2 4^9 2^9)$
mca_4	$MCA(30; 2, 19, 6^1 5^1 4^6 3^8 2^3)$
mca_5	$MCA(29; 2, 61, 4^{15} 3^{17} 2^{29})$
mca_6	$MCA(360; 3, 7, 10^1 6^2 4^3 3^1)$
mca_7	$MCA(49; 2, 10, 7^2 6^2 4^2 3^2 2^2)$

(b)

$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{mca_1}$	$\mathbf{mca_2}$	$\mathbf{mca_3}$	$\mathbf{mca_4}$	$\mathbf{mca_5}$	$\mathbf{mca_6}$	$\mathbf{mca_7}$
0	1	4789.763	3.072	46.989	12.544	3700.038	167.901	0.102
0.1	0.9	1024.635	0.098	0.299	0.236	344.341	3.583	0.008
0.2	0.8	182.479	0.254	0.184	0.241	173.752	1.904	0.016
0.3	0.7	224.786	0.137	0.119	0.222	42.950	1.713	0.020
0.4	0.6	563.857	0.177	0.123	0.186	92.616	3.351	0.020
0.5	0.5	378.399	0.115	0.233	0.260	40.443	1.258	0.035
0.6	0.4	272.056	0.153	0.136	0.178	69.311	2.524	0.033
0.7	0.3	651.585	0.124	0.188	0.238	94.553	2.127	0.033
0.8	0.2	103.399	0.156	0.267	0.314	81.611	5.469	0.042
0.9	0.1	131.483	0.274	0.353	0.549	76.379	4.967	0.110
1	0	7623.546	15.905	18.285	23.927	1507.369	289.104	2.297

time (median value). The values $\mathbb{P}_1 = 0.3$ and $\mathbb{P}_2 = 0.7$ were kept fixed in the following experiments.

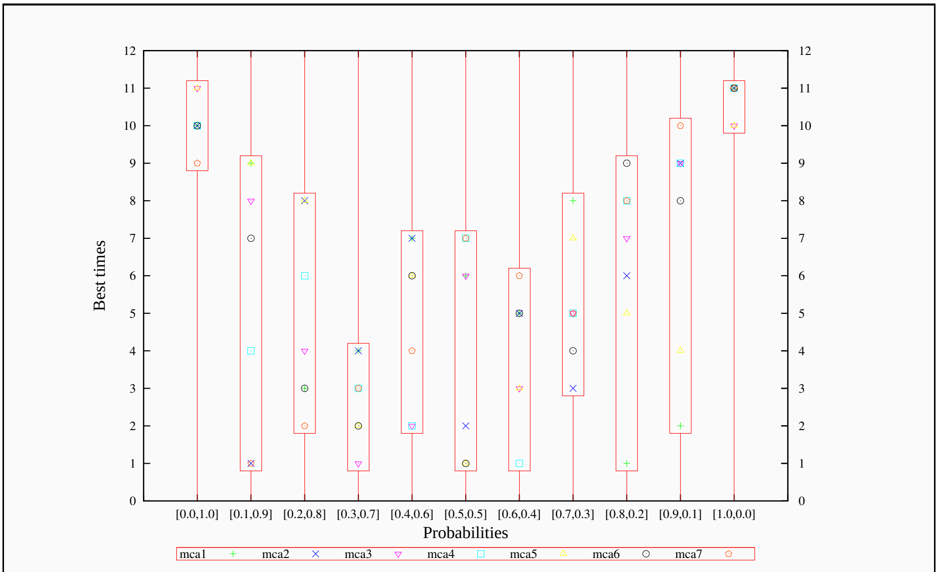

Figure 5.3: Performance of the developed simulated annealing algorithm. We used a Diophantine equation with 11 solutions, every combination of the probabilities was applied by the developed simulated annealing to construct the set of mixed covering arrays shown in Table 5.1(a). Each experiment was run 31 times and we used the median.

The following three sections presents the performance of the developed simulated annealing when solving benchmarks reported in the literature. Each section corresponds to a different implementation of the developed simulated annealing algorithm, i.e., Sequential, Grid, and Parallel. The results were compared against state-of-the-art algorithms for the construction of uniform and mixed covering arrays.

5.3 Sequential simulated annealing

The simulated annealing algorithm was coded in `C` and compiled with `gcc` using the optimization `flag -O3`. It was run sequentially into a CPU Intel(R) Xeon(TM) a 2.8 GHz, 2 GB of RAM with Linux operating system. In all the experiments the following parameters were used for DSSA:

- ▷ Initial temperature $T_i = 4.0$
- ▷ Final temperature $T_f = 1.0E - 10$
- ▷ Cooling factor $\alpha = 0.99$
- ▷ Maximum neighboring solutions per temperature $L = Nk\mathcal{V}^2$
- ▷ Frozen factor $\phi = 11$
- ▷ According to the results shown in Section 5.2, the neighborhood function $\mathcal{N}_1$ was applied using a probability $\mathbb{P} = 0.3$ and the neighborhood function $\mathcal{N}_2$ was applied using a probability $\mathbb{P} = 0.7$.

5.3.1 Uniform covering arrays

The results of DSSA are compared with those obtained by a tool called ACTS[1] (Automated Combinatorial Testing for Software) which was developed by the NIST (National Institute of Standards and Technology), an agency of the United States Government that works to develop tests, test methodologies, and assurance methods. The tool ACTS can compute tests for 2-*way* through 6-*way* interactions. The NIST reports that a comparison of ACTS with similar tools shows that ACTS produces smaller test suites; moreover it has over 800 users as of September 2011, in nearly all major industries. Due to these features, ACTS was selected as a point of comparison for our SA, the non deterministic algorithm IPOG-F was used in ACTS to solve all cases reported in this section.

The objective of this experiment is to make a fair comparison between IPOG-F and the developed sequential simulated annealing (DSSA) algorithm.

[1] http://csrc.nist.gov/groups/SNS/acts/index.html

The experimental comparison between SSA and IPOG-F was accomplished running once each compared method over 96 benchmark instances of strengths $3 \leq t \leq 6$, degrees $7 \leq k \leq 30$ and, $v = 2$. IPOG-F was executed with the parameter values suggested by its authors in (Forbes et al., 2008).

The results from this experiment are summarized in Table 5.2, which presents in the first three columns the strength t, the degree k, and the cardinality of the selected benchmark instances.

Table 5.2: Improved bounds on $CAN(t, k, 2)$ for strengths $3 \leq t \leq 6$ and degrees $7 \leq k \leq 30$ produced by DSSA. For each instance of strength t and degree k, the best solution, in terms of the size N, found by IPOG-F and DSSA are listed. Last column depicts the difference between the best result produced by DSSA and the best solution obtained by IPOG-F ($\Delta = DSSA - IPOG\text{-}F$).

(a)

t	k	*IPOG-F*	*DSSA*	Δ
3	7	16	10	-6
3	8	17	10	-7
3	9	17	10	-7
3	10	18	10	-8
3	11	18	12	-6
3	12	19	15	-4
3	13	20	15	-5
3	14	21	16	-5
3	15	21	16	-5
3	16	22	17	-5
3	17	24	17	-7
3	18	24	17	-7
3	19	24	17	-7
3	20	25	18	-7
3	21	25	18	-7
3	22	26	19	-7
3	23	26	20	-6
3	24	26	20	-6
3	25	27	21	-6
3	26	27	22	-5
3	27	28	22	-6
3	28	28	23	-5
3	29	28	23	-5
3	30	28	23	-5
Avg.		23.13	17.13	-6.00

(b)

t	k	*IPOG-F*	*DSSA*	Δ
4	7	32	21	-11
4	8	34	21	-13
4	9	37	21	-16
4	10	41	21	-20
4	11	43	21	-22
4	12	47	24	-23
4	13	49	32	-17
4	14	52	32	-20
4	15	53	32	-21
4	16	56	32	-24
4	17	57	35	-22
4	18	60	36	-24
4	19	62	36	-26
4	20	65	39	-26
4	21	68	42	-26
4	22	69	44	-25
4	23	70	44	-26
4	24	71	46	-25
4	25	74	50	-24
4	26	74	51	-23
4	27	76	51	-25
4	28	77	53	-24
4	29	78	53	-25
4	30	80	56	-24
Avg.		59.38	37.21	-22.17

(c)

t	k	*IPOG-F*	*DSSA*	Δ
5	7	57	42	-15
5	8	68	52	-16
5	9	77	54	-23
5	10	87	56	-31
5	11	95	56	-39
5	12	105	56	-49
5	13	111	56	-55
5	14	119	64	-55
5	15	127	79	-48
5	16	134	99	-35
5	17	140	104	-36
5	18	144	107	-37
5	19	148	116	-32
5	20	155	119	-36
5	21	160	122	-38
5	22	163	124	-39
5	23	168	132	-36
5	24	175	132	-43
5	25	181	132	-49
5	26	184	132	-52
5	27	188	132	-56
5	28	192	132	-60
5	29	196	132	-64
5	30	200	132	-68
Avg.		140.58	98.42	-42.17

(d)

t	k	*IPOG-F*	*DSSA*	Δ
6	7	79	64	-15
6	8	118	85	-33
6	9	142	108	-34
6	10	165	116	-49
6	11	192	118	-74
6	12	215	118	-97
6	13	237	118	-119
6	14	256	118	-138
6	15	276	128	-148
6	16	292	179	-113
6	17	309	235	-74
6	18	327	280	-47
6	19	343	299	-44
6	20	363	314	-49
6	21	375	330	-45
6	22	382	344	-38
6	23	397	357	-40
6	24	411	372	-39
6	25	426	385	-41
6	26	438	399	-39
6	27	449	410	-39
6	28	463	421	-42
6	29	474	435	-39
6	30	481	444	-37
Avg.		317.08	257.38	-59.71

From Table 5.2 we can clearly observe that in this experiment the IPOG-F algorithm consistently returns poorer quality solutions than DSSA.

5.3.2 Mixed covering arrays

The purpose of this experiment is to carry out a performance comparison of the best bounds achieved by DSSA with respect to those produced by the following state-of-the-art procedures: AETG (Cohen et al., 1996), TCG (Tung and Aldiwan, 2000), SA (Cohen et al., 2003), GA (Shiba et al., 2004), ACO (Shiba et al., 2004), DDA (Bryce and Colbourn, 2007), Tconfig (Williams, 2000), ACTS (Lei et al., 2007), AllPairs (McDowell, 2011), Jenny (Jenkins, 2011) and TS (Gonzalez-Hernandez et al., 2010). Table 5.3 displays the detailed computational results produced by this experiment. The benchmark is shown in the column two; from column 3 to 13 the results reported by some of the state-of-the-art approaches are presented. The previous best-known (β) solution is shown in column 14. The results of constructing the mixed covering arrays for the benchmark using DSSA are shown in column 15. The difference between the best result produced by DSSA and the previous best-known solution ($\Delta = \Theta - \beta$) is depicted in the last column. Next, Figure 5.4 compares the results shown in Table 5.3.

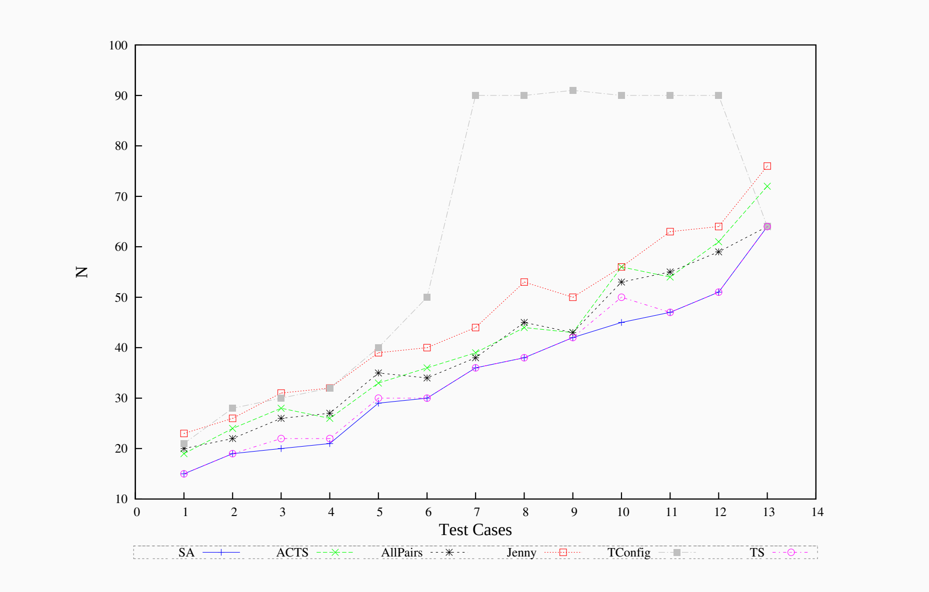

Figure 5.4: Graphical comparison of the best bounds achieved by DSSA with respect to those produced by the state-of-the-art procedures (TConfig (Williams, 2000), ACTS (IPOG) (Lei et al., 2007), AllPair (McDowell, 2011), Jenny (Jenkins, 2011) and TS (Gonzalez-Hernandez et al., 2010)), when the strength $t = 2$. Note that the performance of DSSA improves or equals the best-known solutions.

The empirical evidence presented in this section showed that DSSA improved the size of the mixed covering arrays in comparison with the tools that are among the best found in the state-of-the-art of the construction of mixed covering arrays. The performance of the proposed simulated annealing algorithm was assessed with a benchmark, composed by 19 mixed covering arrays of strengths two and three taken from the literature. The computational results are reported and compared with previously published ones, showing that our algorithm was able to find 4 new upper bounds and to equal 15 previous best-known solutions on the selected benchmark instances.

Table 5.3: For each instance shown in column 2, the best solution, in terms of the size N, found by AETG, TCG, SA, GA, ACO, DDA, Tconfig, ACTS, AllPairs, Jenny, TS and DSSA are listed. The * means that the solution is optimal. The difference between the best result produced by DSSA and the previous best-known solution ($\Delta = \Theta - \beta$) is depicted in the last column.

ID	MCA description	N													Improvements
		AETG[a]	TCG[b]	SA[c]	GA[d]	ACO[e]	DDA[f]	Tconfig[g]	ACTS[h]	AllPairs[i]	Jenny[j]	TS[k]	Best β	Our SA Θ	Δ
1	$t2k11v5^1 3^8 2^2$	20	20	15	15	16	21	21	19	20	23	15	15*	15	0
2	$t2k9v4^5 3^4$	-	-	-	-	-	25	28	24	22	26	19	19	19	0
3	$t2k75v4^1 3^{39} 2^{35}$	27	-	21	27	27	27	30	28	26	31	22	21	**20**	-1
4	$t2k21v5^1 4^4 3^{11} 2^5$	28	30	21	26	25	27	32	26	27	32	22	21	21	0
5	$t2k61v4^{15} 3^{17} 2^{29}$	37	33	30	37	37	35	40	33	35	39	30	30	**29**	-1
6	$t2k19v6^1 5^1 4^6 3^8 2^3$	35	-	30	33	32	34	50	36	34	40	30	30*	30	0
7	$t2k20v6^2 4^9 2^9$	-	-	-	-	-	-	90	39	38	44	36	36*	36	0
8	$t2k16v6^4 4^5 2^7$	-	-	-	-	-	-	90	44	45	53	38	38	38	0
9	$t2k19v7^1 6^1 5^1 4^5 3^8 2^3$	44	45	42	42	42	43	91	43	43	50	42	42*	42	0
10	$t2k14v6^5 5^5 3^4$	-	-	-	-	-	58	90	56	53	56	50	50	**45**	-5
11	$t2k18v6^7 4^8 2^3$	-	-	-	-	-	-	90	54	55	63	47	47	47	0
12	$t2k19v6^9 4^3 2^7$	-	-	-	-	-	-	90	61	59	64	51	51	51	0
13	$t2k8v8^2 7^2 6^2 5^2$	-	-	-	-	-	74	64	72	64	76	64	64*	64	0
14	$t3k9v4^5 3^4$	-	-	-	-	-	-	103	138	-	115	85	85	**80**	-5
15	$t3k6v5^2 4^2 3^2$	114	-	100	108	106	-	106	111	-	131	100	100*	100	0
16	$t3k7v10^1 6^2 4^3 3^1$	377	-	360	360	361	-	372	383	-	399	360	360*	360	0
17	$t3k12v10^2 4^1 3^2 2^7$	-	-	-	-	-	-	472	400	-	413	400	400*	400	0
18	$t3k14v6^5 5^5 3^4$	-	-	-	-	-	-	400	420	-	414	370	370	370	0
19	$t3k8v8^2 7^2 6^2 5^2$	-	-	-	-	-	-	594	614	-	645	540	540	535	-5

[a] Cohen et al., 1996.
[b] Tung and Aldiwan, 2000.
[c] Cohen et al., 2003.
[d] Shiba et al., 2004.
[e] Shiba et al., 2004.
[f] Bryce and Colbourn, 2007.
[g] Williams, 2000.
[h] Lei et al., 2007.
[i] McDowell, 2011.
[j] Jenkins, 2011.
[k] Gonzalez-Hernandez et al., 2010.

5.4 Grid simulated annealing

For this experiment we have obtained the ACTS and TConfig software. We create a new benchmark composed by 60 ternary covering arrays instances where $5 \leq k \leq 100$ and $2 \leq t \leq 4$.

The simulated annealing implementation reported by Cohen et al. (2003) for solving the CAC problem was intentionally omitted from this comparison because as their authors recognize this algorithm fails to produce competitive results when the strength of the arrays is $t \geq 3$.

The results from this experiment are summarized in Table 5.4, which presents in the first two columns the strength t and the degree k of the selected benchmark instances. The best size N found by the TConfig tool, IPOG-F algorithm and DGSA algorithm are listed in columns 3, 4, and 5 respectively. Next, Figure 5.4 compares the results shown in Table 5.4.

From Table 5.4, Figure 5.5, Figure 5.6, and Figure 5.7 we can observe that DGSA algorithm gets solutions of better quality than the other two tools. Finally, each of the 60 ternary covering arrays constructed by DGSA algorithm have been verified by the algorithm described in Section 3.7. In order to minimize the execution time required by DGSA algorithm, the following rule has been applied when choosing the rightmost Grid execution schema: experiments involving a value of the parameter N equal or less than 500 have been executed with the synchronous schema while the rest have been performed using the asynchronous schema.

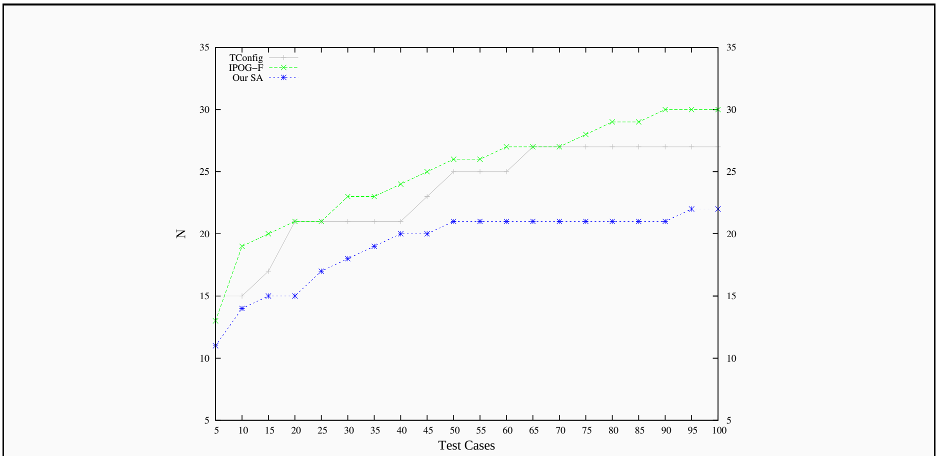

Figure 5.5: Graphical comparison of the performance among TConfig, IPOG-F and DGSA to construct ternary covering arrays when $5 \leq k \leq 100$ and $t = 2$.

Table 5.4: Comparison among TConfig, IPOG-F and DGSA to construct ternary covering arrays when $5 \leq k \leq 100$ and $2 \leq t \leq 4$.

(a) $CAN(2, k, 3)$

t	k	TConfig	IPOG-F	Our SA
	5	15	13	11
	10	15	19	14
	15	17	20	15
	20	21	21	15
	25	21	21	17
	30	21	23	18
	35	21	23	19
	40	21	24	20
	45	23	25	20
2	50	25	26	21
	55	25	26	21
	60	25	27	21
	65	27	27	21
	70	27	27	21
	75	27	28	21
	80	27	29	21
	85	27	29	21
	90	27	30	21
	95	27	30	22
	100	27	30	22

(b) $CAN(3, k, 3)$

t	k	TConfig	IPOG-F	Our SA
	5	40	42	33
	10	68	66	45
	15	83	80	57
	20	94	92	59
	25	102	98	72
	30	111	106	87
	35	117	111	89
	40	123	118	89
	45	130	121	90
3	50	134	126	101
	55	140	131	101
	60	144	134	104
	65	147	138	120
	70	150	141	120
	75	153	144	120
	80	155	147	129
	85	158	150	130
	90	161	154	130
	95	165	157	132
	100	167	159	133

(c) $CAN(4, k, 3)$

t	k	TConfig	IPOG-F	Our SA
	5	115	98	81
	10	241	228	165
	15	325	302	280
	20	383	358	330
	25	432	405	400
	30	466	446	424
	35	518	479	475
	40	540	513	510
	45	572	533	530
4	50	581	559	528
	55	606	581	545
	60	621	596	564
	65	639	617	581
	70	657	634	597
	75	671	648	610
	80	687	663	624
	85	699	678	635
	90	711	690	649
	95	723	701	660
	100	733	714	669

5.5 Parallel simulated annealing

5.5.1 Comparison of the ISA, SSA and CSA approaches

To test the performance of the ISA, SSA, and CSA approaches, we propose the construction of a covering array with $N = 80$, $t = 3$, $k = 22$ and $v = 3$. Each

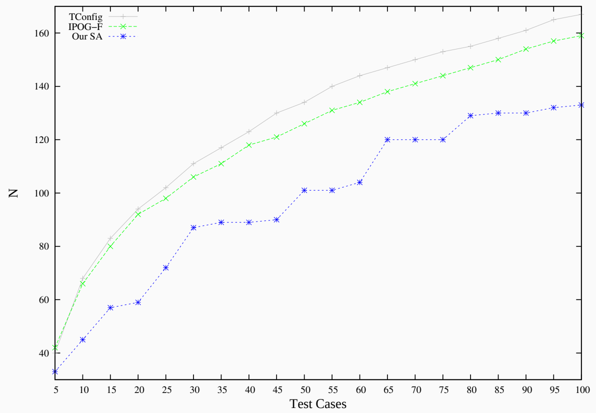

Figure 5.6: Graphical comparison of the performance among TConfig, IPOG-F and DGSA to construct ternary covering arrays when $5 \leq k \leq 100$ and $t = 3$.

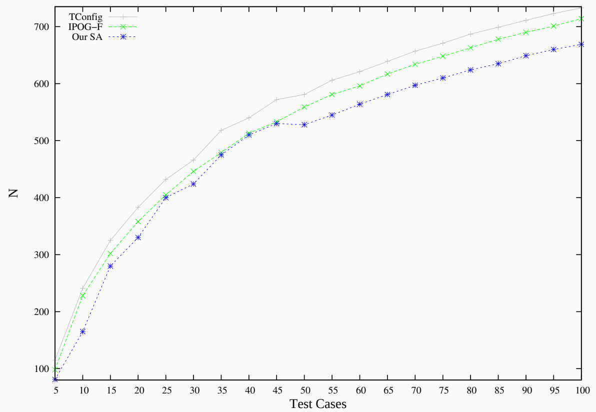

Figure 5.7: Graphical comparison of the performance among TConfig, IPOG-F and DGSA to construct ternary covering arrays when $5 \leq k \leq 100$ and $t = 4$.

approach was executed 31 times (for provide statistical validity to experiment) using $\mathcal{P} = \{4, 8, 16, 32\}$.

The performance of the algorithms has been compared based on median speed-up as a function of the number of processors, the results are shown in Figure 5.8.

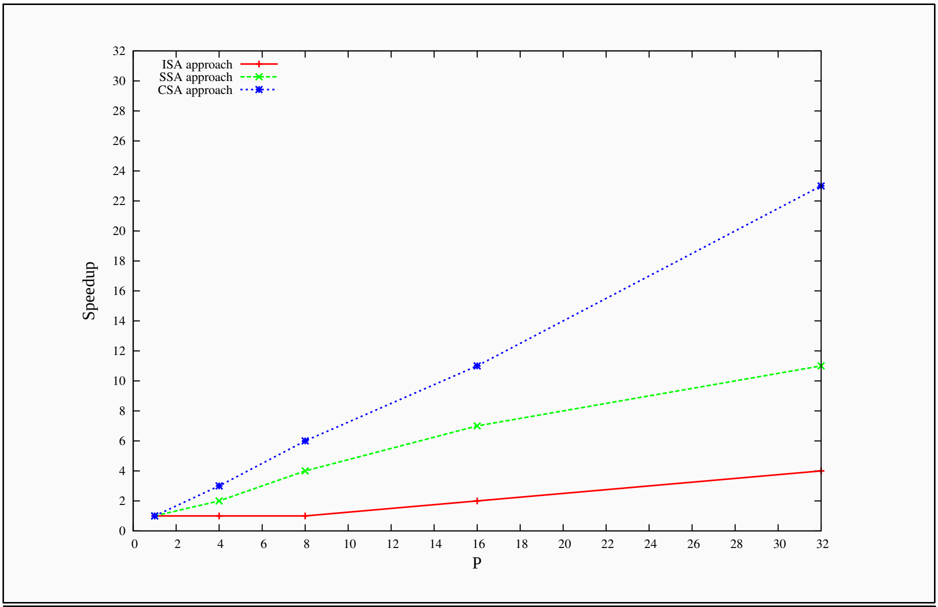

Figure 5.8: Median speed-up for the ISA, SSA and CSA approaches.

The ISA approach, had difficulty in handling the large problem instances, it does not scale. The SSA approach provides reasonable results, however, because it is a synchronous algorithm, the idle and communication times are inevitable. The CSA approach is who offers the best results, it reduces the execution time of the SSA approach by employing asynchronous information exchange.

In the next subsection, it is presented the third experiment of this work, the purpose is to measure the performance of the CSA algorithm against the best-known solutions reported in the literature.

5.5.2 Comparing the CSA approach against the state-of-the-art procedures

The purpose of this experiment is to carry out a performance comparison of the bounds achieved by the CSA approach with respect to the best-known solutions reported in the literature Colbourn, 2011., which were produced using the following state-of-the-art procedures: orthogonal array construction, Roux type constructions, doubling constructions, algebraic constructions, Deterministic Density Algorithm (DDA), Tabu Search and IPOG-F.

For this experiment we have fixed the maximum computational time expended by our PSA for constructing a CA to 72 hours and 50 processors. We create a new benchmark composed by 182 covering arrays distributed as follows:

- ▷ 47 covering arrays with strength $t = 3$, degree $4 \le k < 50$ and order $v = 3$
- ▷ 46 covering arrays with strength $t = 4$, degree $5 \le k < 50$ and order $v = 3$
- ▷ 45 covering arrays with strength $t = 5$, degree $6 \le k < 50$ and order $v = 3$
- ▷ 44 covering arrays with strength $t = 6$, degree $7 \le k < 50$ and order $v = 3$

The detailed results produced by this experiment are listed in Table 5.5. The first two columns in each subtable indicate the strength t and the degree k of the selected instances. Next two columns show, in terms of the size N of the covering arrays, the best-known solution reported in the literature and the improved bounds produced by the CSA approach. Last column depicts the difference between the best result produced by our CSA approach and the best-known solution ($\Delta = \beta - \vartheta$).

Table 5.5: It shows the improved bounds produced by our CSA approach. Column ϑ represents the best-known solution reported in the literature (Colbourn, 2011). Column β represents the best solution in terms of N produced by our CSA approach. Last column (Δ) depicts the difference between the best result produced by our CSA approach and the best-known solution ($\Delta = \beta - \vartheta$).

t	k	ϑ	β	Δ	t	k	ϑ	β	Δ
3	4	27	27	0	4	4			
3	5	33	33	0	4	5	81	81	0
3	6	33	33	0	4	6	111	111	0
3	7	40	39	-1	4	7	123	123	0
3	8	42	42	0	4	8	141	135	-6
3	9	45	45	0	4	9	159	135	-24
3	10	45	45	0	4	10	159	164	5
3	11	45	45	0	4	11	183	183	0
3	12	45	45	0	4	12	201	201	0
3	13	50	49	-1	4	13	219	219	0
3	14	51	50	-1	4	14	237	249	12
3	15	57	57	0	4	15	237	277	40
3	16	60	59	-1	4	16	237	277	40
3	17	60	59	-1	4	17	300	287	-13
3	18	60	59	-1	4	18	307	300	-7
3	19	60	59	-1	4	19	313	313	0
3	20	60	59	-1	4	20	315	321	6
3	21	66	67	1	4	21	315	338	23
3	22	66	71	5	4	22	315	347	32
3	23	69	71	2	4	23	315	359	44

continued on next page

continued from previous page

	t	k	ϑ	β	Δ	t	k	ϑ	β	Δ
	3	24	72	71	-1	4	24	377	375	-2
	3	25	75	72	-3	4	25	384	375	-9
	3	26	78	72	-6	4	26	393	387	-6
	3	27	81	79	-2	4	27	393	387	-6
	3	28	81	79	-2	4	28	393	392	-1
	3	29	87	84	-3	4	29	393	406	13
	3	30	87	84	-3	4	30	393	401	8
	3	31	90	88	-2	4	31	440	424	-16
	3	32	90	89	-1	4	32	445	431	-14
	3	33	90	89	-1	4	33	454	438	-16
	3	34	90	89	-1	4	34	462	447	-15
	3	35	90	89	-1	4	35	471	440	-31
	3	36	90	89	-1	4	36	471	456	-15
	3	37	90	89	-1	4	37	471	460	-11
	3	38	90	89	-1	4	38	471	465	-6
	3	39	90	89	-1	4	39	471	468	-3
	3	40	90	89	-1	4	40	499	472	-27
	3	41	98	94	-4	4	41	506	484	-22
	3	42	98	94	-4	4	42	509	488	-21
	3	43	100	99	-1	4	43	518	494	-24
	3	44	100	99	-1	4	44	522	497	-25
	3	45	103	99	-4	4	45	526	497	-29
	3	46	103	101	-2	4	46	530	506	-24
	3	47	106	101	-5	4	47	534	510	-24
	3	48	106	101	-5	4	48	542	516	-26
	3	49	108	101	-7	4	49	549	523	-26
	3	50	108	102	-6	4	50	549	525	-24
1759	5	6	243	243	0	6	6			
	5	7	351	351	0	6	7	729	729	0
	5	8	405	405	0	6	8	1152	1152	0
	5	9	483	405	-78	6	9	1431	1600	169
	5	10	483	405	-78	6	10	1449	1849	400
	5	11	705	550	-155	6	11	1449	2136	687
	5	12	723	600	-123	6	12	2181	2482	301
	5	13	723	828	105	6	13	2734	2744	10
	5	14	922	890	-32	6	14	2907	3220	313
	5	15	963	944	-19	6	15	3234	3338	104
	5	16	963	1025	62	6	16	3443	3672	229
	5	17	1117	1117	0	6	17	3658	3882	224
	5	18	1167	1165	-2	6	18	3846	4098	252
	5	19	1197	1190	-7	6	19	4054	4256	202
	5	20	1266	1257	-9	6	20	4486	4400	-86
	5	21	1317	1312	-5	6	21	4678	4600	-78
	5	22	1346	1319	-27	6	22	4853	4732	-121
	5	23	1405	1387	-18	6	23	4942	4941	-1
	5	24	1447	1420	-27	6	24	5193	5100	-93
	5	25	1486	1440	-46	6	25	5257	5238	-19
	5	26	1521	1493	-28	6	26	5709	5380	-329
	5	27	1538	1527	-11	6	27	5853	5810	-43
	5	28	1579	1555	-24	6	28	6003	5965	-38
	5	29	1615	1585	-30	6	29	6150	6110	-40
	5	30	1647	1616	-31	6	30	6281	6250	-31
	5	31	1681	1643	-38	6	31	6413	6393	-20
	5	32	1724	1671	-53	6	32	6535	6518	-17
	5	33	1783	1702	-81	6	33	6656	6642	-14
	5	34	1783	1724	-59	6	34	6772	6760	-12

continued on next page

continued from previous page

t	k	ϑ	β	Δ	t	k	ϑ	β	Δ
5	35	1851	1748	-103	6	35	6877	6871	-6
5	36	1882	1778	-104	6	36	6989	6978	-11
5	37	1909	1800	-109	6	37	7092	7086	-6
5	38	1937	1829	-108	6	38	7194	7187	-7
5	39	1960	1851	-109	6	39	7293	7284	-9
5	40	1986	1866	-120	6	40	7391	7385	-6
5	41	2023	1896	-127	6	41	7490	7478	-12
5	42	2046	1923	-123	6	42	7574	7569	-5
5	43	2069	1940	-129	6	43	7672	7661	-11
5	44	2091	2089	-2	6	44	7757	7748	-9
5	45	2112	2111	-1	6	45	7845	7836	-9
5	46	2130	2129	-1	6	46	7938	7928	-10
5	47	2150	2149	-1	6	47	8013	8005	-8
5	48	2174	2168	-6	6	48	8092	8089	-3
5	49	2191	2189	-2	6	49	8179	8176	-3
5	50	2213	2211	-2	6	50	8256	8253	-3

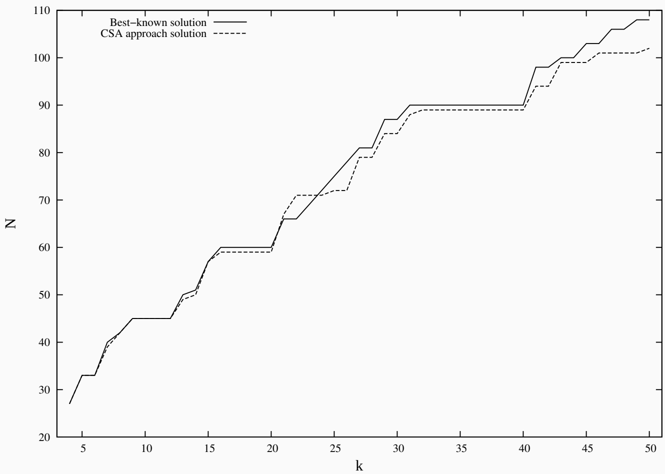

Figure 5.9: Graphical comparison of the quality solutions between CSA and the state-of-the-art (Colbourn, 2011) for strength $t = 3$, degree $4 \leq k < 50$ and order $v = 3$.

Figure 5.9, Figure 5.10, Figure 5.11, and Figure 5.12 compare the results shown in Table 5.5 involving the CSA algorithm and the best-known solutions. The analysis of the data presented let us to the following observation. The solutions quality attained by the CSA approach is very competitive with respect to that produced by the state-of-the-art procedures summarized in column 3 (ϑ). In fact, it is able to improve on 134 previous best-known solutions.

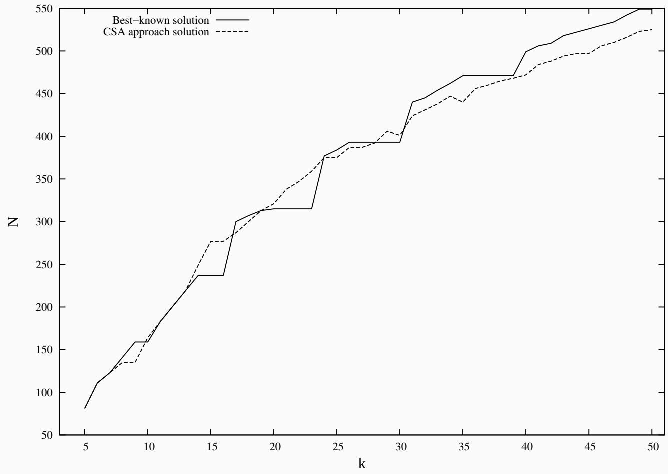

Figure 5.10: Graphical comparison of the quality solutions between CSA and the state-of-the-art (Colbourn, 2011) for strength $t = 4$, degree $5 \leq k < 50$ and order $v = 3$.

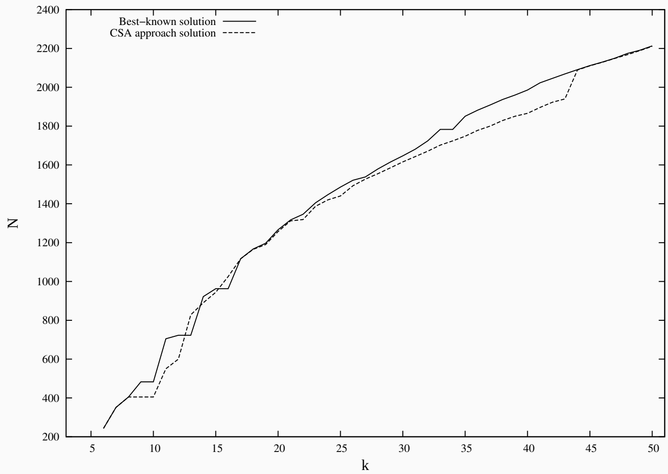

Figure 5.11: Graphical comparison of the quality solutions between CSA and the state-of-the-art (Colbourn, 2011) for strength $t = 5$, degree $6 \leq k < 50$ and order $v = 3$.

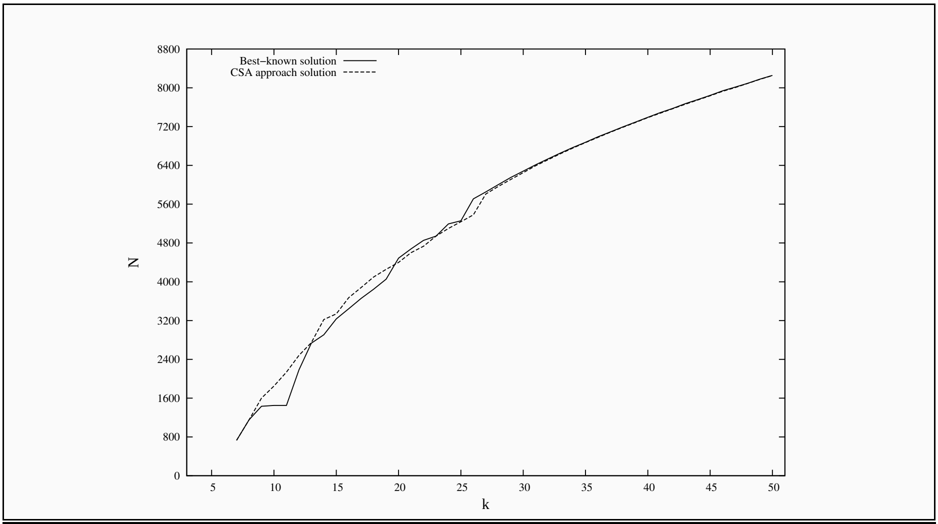

Figure 5.12: Graphical comparison of the quality solutions between CSA and the state-of-the-art (Colbourn, 2011) for strength $t = 6$, degree $7 \leq k < 50$ and order $v = 3$.

5.6 Constructing test-suites for different real-case software components

The purpose of this experiment is to evaluate the performance of the proposed simulated annealing through the construction of different test suites for two real-case softwares. The results of DSSA are compared with those obtained by a tool called ACTS[2] (Automated Combinatorial Testing for Software) which was developed by the NIST (National Institute of Standards and Technology), an agency of the United States Government that works to develop tests, test methodologies, and assurance methods. The tool ACTS can compute tests for 2-*way* through 6-*way* interactions. The NIST reports that a comparison of ACTS with similar tools shows that ACTS produces smaller test suites; moreover it has over 800 users as of September 2011, in nearly all major industries. Due to these features, ACTS was selected as a point of comparison for the developed simulated annealing, the non deterministic algorithm IPOG-F was used in ACTS to solve all cases reported in this section.

[2] http://csrc.nist.gov/groups/SNS/acts/index.html

5.6.1 Case 1: Test Suites for a Smartphone Application

The use of smartphones has increased in the last years, as can be seen in the first quarterly mobile study of 2012 provided by comScore, which notes that 104 million people in US were deemed smartphone owners, of which 50.1% have a mobile device with *Android*. Many resources are available online for this application, some of them define application permissions for system features. Table 5.6 shows the description of a file with 35 options from different parameters of Android. Table 5.7 indicates the name of each parameter and its possible configurations. For a more detailed parameters' explanation, see `http://developer.android.com/reference/android/package-summary.html`

Table 5.6: Android resource configuration file.

Constants					
int	HARDKEYBOARDHIDDEN_NO	int	NAVIGATIONHIDDEN_YES	int	SCREENLAYOUT_LONG_UNDEFINED
int	HARDKEYBOARDHIDDEN_UNDEFINED	int	NAVIGATION_DPAD	int	SCREENLAYOUT_LONG_YES
int	HARDKEYBOARDHIDDEN_YES	int	NAVIGATION_NONAV	int	SCREENLAYOUT_SIZE_LARGE
int	KEYBOARDHIDDEN_NO	int	NAVIGATION_TRACKBALL	int	SCREENLAYOUT_SIZE_MASK
int	KEYBOARDHIDDEN_UNDEFINED	int	NAVIGATION_UNDEFINED	int	SCREENLAYOUT_SIZE_NORMAL
int	KEYBOARDHIDDEN_YES	int	NAVIGATION_WHEEL	int	SCREENLAYOUT_SIZE_SMALL
int	KEYBOARD_12KEY	int	ORIENTATION_LANDSCAPE	int	SCREENLAYOUT_SIZE_UNDEFINED
int	KEYBOARD_NOKEYS	int	ORIENTATION_PORTRAIT	int	TOUCHSCREEN_FINGER
int	KEYBOARD_QWERTY	int	ORIENTATION_SQUARE	int	TOUCHSCREEN_NOTOUCH
int	KEYBOARD_UNDEFINED	int	ORIENTATION_UNDEFINED	int	TOUCHSCREEN_STYLUS
int	NAVIGATIONHIDDEN_NO	int	SCREENLAYOUT_LONG_MASK	int	TOUCHSCREEN_UNDEFINED
int	NAVIGATIONHIDDEN_UNDEFINED	int	SCREENLAYOUT_LONG_NO		

Table 5.7: Android configuration options.

	NAVIGATION	SCREENLAYOUT SIZE	KEYBOARD	ORIENTATION	SCREENLAYOUT LONG	TOUCHSCREEN	HARDKEYBOARDHIDDEN	KEYBOARDHIDDEN	NAVIGATIONHIDDEN
1	DPAD	LARGE	12KEY	LANDSCAPE	MASK	FINGER	NO	NO	NO
2	NONAV	MASK	NOKEYS	PORTRAIT	NO	NOTOUCH	UNDEFINED	UNDEFINED	UNDEFINED
3	TRACKBALL	NORMAL	QWERTY	SQUARE	UNDEFINED	STYLUS	YES	YES	YES
4	UNDEFINED	SMALL	UNDEFINED	UNDEFINED	YES	UNDEFINED			
5	WHEEL	UNDEFINED							

Derived of the information shown in Table 5.7, the total number of configurations is $3 \times 3 \times 4 \times 3 \times 5 \times 4 \times 4 \times 5 \times 4 = 172,800$. Taking into account that every configuration used in the testing process requires the verification of the output and the report of the failures (as the case), supposed that each takes at least 10 minutes, then it will take about 16 staff-years testing all cases; therefore to carry out the testing in this way is infeasible.

Instead of using the exhaustive approach to test this file for Android, the proposed simulated annealing and ACTS National Institute of Standards and Technology, 2011 were used to construct different test suites, which cover t-way combinations of values. Every test suite is represented by a $MCA(N; t, 9, 5^2 4^4 3^3)$, each instance was run 31 times (to provide statistical validity to the experiment). The minimum size achieved by each approach is shown in Figure 5.13.

(a)

t	TLB	IPOG-F	SA	Δ
2	25	29	25	-4
3	100	137	108	-29
4	400	625	540	-85
5	1600	2532	2189	-343
6	6400	8824	7880	-944

(b)

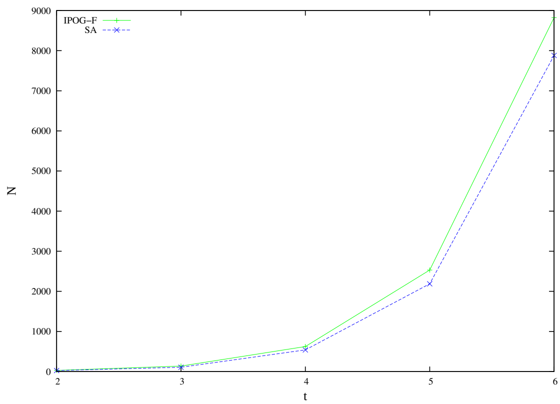

Figure 5.13: Size of each test suite $MCA(N; 2, 9, 5^2 4^4 3^3)$ for the resource configuration file for Android indicated in Table 5.6. Column one represents the strength t of each experiment, column two represents the *theoretical lower bound*, column three shows the results obtained by the IPOG-F algorithm, column four shows the results obtained by DSSA algorithm, finally, the last column depicts the difference between the best bound produced by DSSA and the best bound obtained by IPOG-F algorithm.

Table 5.8(a) shows the $MCA(25; 2, 9, 5^2 4^4 3^3)$ that covers 2-*way* interactions. Finally, to make the mapping between the mixed covering array and a test suite for Android applications every possible value of each parameter in Table 5.7 is labeled by the row number. Table 5.8(b) shows the corresponding pair-wise test suite; each of its twenty-five experiments is analogous to one row of the mixed covering array shown in Table 5.8(a).

Based on the results in Figure 5.13, it can be seen that DSSA was able to construct smaller test suites than those generated by ACTS. In comparison with the exhaustive approach, there is a decrease of 95.43% in the test suite with interaction of size 6; therefore in the example described above, instead of 16 staff-years of testing process, it would take less than 9 and a half months if it test suite is used; and it would take slightly more than 2 months for the case $t = 5$. These benefits are in terms of time; however the impact of the reduction of even a test case, can result in savings such that the salary and benefit costs for each tester, just to name a few.

Table 5.8: Test suite for Android applications (a) $MCA(25; 2, 9, 5^2 4^4 3^3)$; (b) Pairwise test suite for Android applications, each row corresponds to an experiment.

(a)

$$\begin{pmatrix}
2 & 0 & 1 & 0 & 3 & 1 & 2 & 0 & 2 \\
0 & 1 & 3 & 1 & 0 & 1 & 1 & 1 & 2 \\
3 & 1 & 2 & 0 & 3 & 3 & 1 & 0 & 0 \\
0 & 3 & 1 & 2 & 3 & 2 & 0 & 2 & 0 \\
0 & 4 & 0 & 0 & 1 & 0 & 1 & 0 & 2 \\
3 & 0 & 0 & 1 & 1 & 2 & 1 & 2 & 2 \\
1 & 1 & 1 & 2 & 1 & 2 & 2 & 1 & 1 \\
1 & 3 & 2 & 0 & 0 & 0 & 1 & 2 & 1 \\
3 & 3 & 3 & 3 & 2 & 0 & 2 & 0 & 1 \\
1 & 4 & 0 & 1 & 2 & 1 & 0 & 0 & 0 \\
0 & 0 & 2 & 1 & 2 & 3 & 2 & 1 & 1 \\
2 & 1 & 2 & 3 & 2 & 1 & 0 & 2 & 2 \\
2 & 2 & 3 & 0 & 1 & 0 & 0 & 1 & 0 \\
2 & 3 & 0 & 1 & 0 & 3 & 2 & 1 & 0 \\
2 & 4 & 3 & 2 & 3 & 2 & 1 & 2 & 1 \\
1 & 0 & 3 & 3 & 3 & 0 & 1 & 0 & 0 \\
4 & 1 & 0 & 2 & 3 & 0 & 2 & 1 & 1 \\
1 & 2 & 2 & 2 & 1 & 3 & 2 & 2 & 2 \\
0 & 2 & 0 & 3 & 0 & 2 & 0 & 0 & 0 \\
3 & 4 & 1 & 3 & 0 & 3 & 0 & 1 & 1 \\
4 & 0 & 3 & 2 & 0 & 3 & 0 & 0 & 2 \\
4 & 2 & 1 & 1 & 3 & 0 & 1 & 0 & 0 \\
3 & 2 & 1 & 2 & 2 & 1 & 1 & 1 & 1 \\
4 & 3 & 1 & 3 & 1 & 1 & 1 & 2 & 2 \\
4 & 4 & 2 & 0 & 2 & 2 & 2 & 0 & 0
\end{pmatrix}$$

(b)

NAVIGATION	SCREENLAYOUT SIZE	KEYBOARD	ORIENTATION	SCREENLAYOUT LONG	TOUCHSCREEN	HARDKEYBOARDHIDDEN	KEYBOARDHIDDEN	NAVIGATIONHIDDEN
TRACKBALL	LARGE	NOKEYS	LANDSCAPE	YES	NOTOUCH	YES	NO	YES
DPAD	MASK	UNDEFINED	PORTRAIT	MASK	NOTOUCH	UNDEFINED	UNDEFINED	YES
UNDEFINED	MASK	QWERTY	LANDSCAPE	YES	UNDEFINED	UNDEFINED	NO	NO
DPAD	SMALL	NOKEYS	SQUARE	YES	STYLUS	NO	YES	NO
DPAD	UNDEFINED	12KEY	LANDSCAPE	NO	FINGER	UNDEFINED	NO	YES
UNDEFINED	LARGE	12KEY	PORTRAIT	NO	STYLUS	UNDEFINED	YES	YES
NONAV	MASK	NOKEYS	SQUARE	NO	STYLUS	YES	UNDEFINED	UNDEFINED
NONAV	SMALL	QWERTY	LANDSCAPE	MASK	FINGER	UNDEFINED	YES	UNDEFINED
UNDEFINED	SMALL	UNDEFINED	UNDEFINED	UNDEFINED	FINGER	YES	NO	UNDEFINED
NONAV	UNDEFINED	12KEY	PORTRAIT	UNDEFINED	NOTOUCH	NO	NO	NO
DPAD	LARGE	QWERTY	PORTRAIT	UNDEFINED	UNDEFINED	YES	UNDEFINED	UNDEFINED
TRACKBALL	MASK	QWERTY	UNDEFINED	UNDEFINED	NOTOUCH	NO	YES	YES
TRACKBALL	NORMAL	UNDEFINED	LANDSCAPE	NO	FINGER	NO	UNDEFINED	NO
TRACKBALL	SMALL	12KEY	PORTRAIT	MASK	UNDEFINED	YES	UNDEFINED	NO
TRACKBALL	UNDEFINED	UNDEFINED	SQUARE	YES	STYLUS	UNDEFINED	YES	UNDEFINED
NONAV	LARGE	UNDEFINED	UNDEFINED	YES	FINGER	UNDEFINED	NO	NO
WHEEL	MASK	12KEY	SQUARE	YES	FINGER	YES	UNDEFINED	UNDEFINED
NONAV	NORMAL	QWERTY	SQUARE	NO	UNDEFINED	YES	YES	YES
DPAD	NORMAL	12KEY	UNDEFINED	MASK	STYLUS	NO	NO	NO
UNDEFINED	UNDEFINED	NOKEYS	UNDEFINED	MASK	UNDEFINED	NO	UNDEFINED	UNDEFINED
WHEEL	LARGE	UNDEFINED	SQUARE	MASK	UNDEFINED	NO	NO	YES
WHEEL	NORMAL	NOKEYS	PORTRAIT	YES	FINGER	UNDEFINED	NO	NO
UNDEFINED	NORMAL	NOKEYS	SQUARE	UNDEFINED	NOTOUCH	UNDEFINED	UNDEFINED	UNDEFINED
WHEEL	SMALL	NOKEYS	UNDEFINED	NO	NOTOUCH	UNDEFINED	YES	YES
WHEEL	UNDEFINED	QWERTY	LANDSCAPE	UNDEFINED	STYLUS	YES	NO	NO

5.6.2 Case 2: Test Suites for the module `Add park`

Currently there are several companies that provide custom application software development, maintenance, and support to add functionality and integrate disparate packaged applications that need to be enhanced to achieve business objectives. A real international company, which we named *xCompany* to avoid conflicts of interest, dedicated to custom software development is currently constructing an application to control the activities for a scientific park. This software is constituted by different modules, being one of them *Add park*. The parameters and their corresponding values are shown in Table 5.9.

Table 5.9: Parameters of the module: `Catalog of parks Add park`.

Parameter name	Values	#values
Name	alphanumeric, special, empty, length exceeds	4
Country	selected, unselected	2
State	selected, unselected	2
Population	selected, unselected	2
Category	aquatic, not defined, thematic	3
Address	alphanumeric, special, empty, length exceeds	4
Description	alphanumeric, special, empty, length exceeds	4
Services	alphanumeric, special, empty, length exceeds	4
Stock	checked, unchecked	2
Start of agreement	valid date, invalid date, empty	3
End of agreement	valid date, lower than start d., upper than start d., invalid date, empty	5
Business terms	alphanumeric, special, empty, length exceeds	4
Public terms	alphanumeric, special, empty, length exceeds	4
Description of the agreement	alphanumeric, special, empty, length exceeds	4
Contact Name	alphanumeric, special, empty, length exceeds	4
email contact	valid, invalid, empty	3
Description of the contact	alphanumeric, special, empty, length exceeds	4
Business name of contact	alphanumeric, special, empty, length exceeds	4
RFC	valid, invalid, empty	3
Bank	alphanumeric, special, empty, length exceeds	4
Account	alphanumeric, special, empty, length exceeds	4
CLABE	alphanumeric, special, empty, length exceeds	4

xCompany views Quality Assurance (QA) as an integrated system of management and testing that provides confidence that a software application will deliver its specified performance; so it has to carry out the testing process with this goal in mind. As mentioned in the Android case, test all configurations is infeasible, therefore our simulated annealing and ACTS were used to construct test suites for this module. Every test suite was represented by the $MCA(N; t, 22, 5^1 4^{13} 3^4 2^4)$ where $2 < t \leq 6$ and was solved 31 times. The minimum size obtained by the proposed simulated annealing and ACTS is shown in Figure 5.14. Table 5.10 shown the $MCA(N; 2, 22, 5^1 4^{13} 3^4 2^4)$ that covers 2-*way* interactions. The equivalent values for each test as been obtained as specified in Table 5.8(b).

Results in Figure 5.14 shows that the quality solution of our simulated annealing is better for all cases in comparison with those obtained by ACTS. The constructed test suites have the guarantee to cover all interactions of size t indicated in first column of Figure 5.14. It means that if exist a *functional failure* triggered by a particular configuration among t parameters, it will be evidenced by using the cre-

(a)

t	TLB	IPOG-F	SA	Δ
2	20	29	28	-1
3	80	206	169	-37
4	320	995	900	-95
5	1280	4912	4824	-88
6	5120	22252	21400	-852

(b)

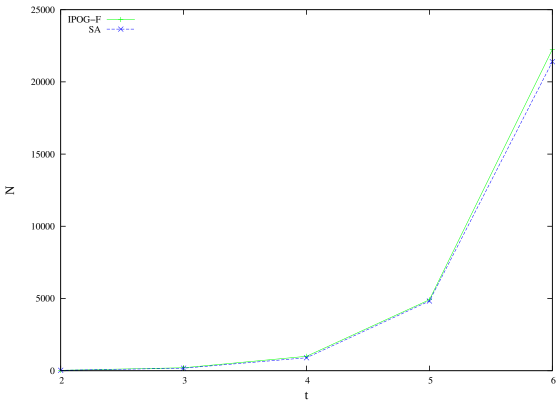

Figure 5.14: Minimum size for each test suite for the module described in Table 5.9. Column one represents the strength t of each experiment, column two represents the *theoretical lower bound*, column three shows the results obtained by the IPOG-F algorithm, column four shows the results obtained by DSSA algorithm, finally, the last column depicts the difference between the best bound produced by DSSA and the best bound obtained by IPOG-F algorithm.

ated test suites, thus the goal of *confidence that a software application will deliver its specified performance* is guaranteed to the extent of the degree of interaction.

5.7 Summary

This chapter presented the experiments carried out to assess the performance of the developed simulated annealing algorithm. The chapter was organized in six sections, the results are described in the following paragraphs.

The global performance of the developed simulated annealing algorithm and the influences that some of its key features have on it were presented in Section 5.1. The empirical experimentation disclosed that the developed simulated annealing using Maximum Hamming Distance solutions performs much better than the simulated annealing algorithm that starts from a randomly generated solution. The neighborhood function is a critical element for the performance of any local search algorithm. It was shown experimentally that our algorithm achieves its best performance using the neighborhood function $\mathcal{N}_3$, which is a compound neighborhood combining the complementary characteristics of both $\mathcal{N}_1$ and $\mathcal{N}_2$.

Table 5.10: $MCA(28; 2, 22, 5^1 4^{13} 3^4 2^4)$ that represents a test suite for the Module `Add park`

$$
\begin{pmatrix}
4 & 2 & 3 & 1 & 3 & 3 & 2 & 1 & 2 & 2 & 3 & 2 & 0 & 3 & 2 & 1 & 2 & 0 & 1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 & 3 & 2 & 2 & 3 & 1 & 2 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
3 & 2 & 1 & 0 & 2 & 3 & 0 & 2 & 3 & 0 & 1 & 1 & 2 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 2 & 3 & 3 & 0 & 2 & 0 & 3 & 2 & 1 & 0 & 3 & 3 & 3 & 0 & 2 & 0 & 2 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 1 & 2 & 0 & 3 & 1 & 1 & 2 & 2 & 2 & 3 & 2 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 3 & 2 & 0 & 2 & 2 & 0 & 3 & 0 & 3 & 2 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 2 & 3 & 1 & 3 & 0 & 3 & 0 & 0 & 3 & 1 & 1 & 2 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 3 & 0 & 3 & 3 & 2 & 2 & 1 & 1 & 3 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
3 & 3 & 2 & 2 & 3 & 1 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 3 & 1 & 3 & 3 & 2 & 0 & 0 & 3 & 3 & 2 & 2 & 0 & 2 & 2 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
2 & 3 & 3 & 3 & 1 & 1 & 1 & 2 & 1 & 2 & 2 & 3 & 2 & 2 & 0 & 1 & 2 & 0 & 1 & 0 & 0 & 1 \\
4 & 1 & 2 & 3 & 1 & 2 & 0 & 1 & 2 & 1 & 1 & 0 & 2 & 2 & 1 & 0 & 2 & 2 & 1 & 1 & 1 & 1 \\
2 & 0 & 2 & 0 & 1 & 0 & 2 & 2 & 3 & 1 & 0 & 2 & 0 & 0 & 0 & 2 & 1 & 2 & 1 & 1 & 0 & 1 \\
2 & 3 & 1 & 2 & 1 & 3 & 3 & 2 & 0 & 0 & 2 & 0 & 3 & 3 & 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 \\
2 & 2 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 3 & 1 & 2 & 2 & 1 & 2 & 0 & 2 & 1 & 0 & 0 & 0 & 1 \\
4 & 3 & 1 & 1 & 0 & 0 & 1 & 3 & 1 & 1 & 3 & 1 & 1 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 1 & 1 & 3 & 3 & 0 & 1 & 0 & 0 & 1 & 0 & 2 & 0 & 2 & 1 & 2 & 1 & 0 & 0 & 1 \\
1 & 3 & 0 & 0 & 3 & 0 & 1 & 1 & 3 & 0 & 1 & 2 & 3 & 3 & 1 & 2 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 2 & 1 & 2 & 1 & 3 & 3 & 1 & 3 & 3 & 0 & 2 & 3 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 1 \\
1 & 3 & 0 & 1 & 2 & 1 & 1 & 3 & 2 & 1 & 2 & 0 & 0 & 1 & 2 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
2 & 1 & 3 & 0 & 2 & 0 & 0 & 0 & 2 & 3 & 3 & 1 & 1 & 3 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
3 & 1 & 0 & 2 & 1 & 3 & 1 & 3 & 3 & 3 & 3 & 2 & 1 & 0 & 2 & 1 & 2 & 1 & 1 & 1 & 0 & 0 \\
4 & 0 & 2 & 2 & 2 & 2 & 1 & 0 & 0 & 3 & 3 & 3 & 3 & 1 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\
4 & 2 & 2 & 0 & 0 & 1 & 3 & 2 & 3 & 2 & 2 & 0 & 1 & 2 & 2 & 2 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 2 & 0 & 1 & 2 & 1 & 2 & 0 & 2 & 1 & 3 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 3 & 1 & 3 & 2 & 3 & 2 & 3 & 0 & 3 & 3 & 2 & 0 & 1 & 1 & 2 & 2 & 1 & 0 & 0 & 0 \\
1 & 2 & 3 & 2 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 2 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 2 & 3 & 0 & 1 & 0 & 1 & 1 & 0 & 2 & 0 & 3 & 1 & 1 & 2 & 0 & 2 & 2 & 1 & 1 & 1 & 1
\end{pmatrix}
$$

In order to provide a good global performance of the simulated annealing algo-
1837 rithm, Section 5.2 presented a fine tuning methodology for optimizing the assigned
probabilities of execution for each of the two neighborhood functions using a linear
1838 Diophantine equation.

Section 5.3 presented the results obtained from the sequential implementation of
1839 simulated annealing. The results were compared against the best algorithms ob-
1840 tained from the literature for constructing uniform and mixed covering arrays. The
computational results showed that our algorithm is able to improve the previous
1841 bounds or at least match them.

Section 5.4 presented the results of comparing our Grid implementation of simu-
1842 lated annealing algorithm against two of the best algorithms from the literature
1843 (IPOG-F and TConfig); we created a new benchmark composed by 60 ternary
1844 covering arrays instances where $5 \leq k \leq 100$ and $2 \leq t \leq 4$. The empirical evi-
1845 dence presented in this section showed that DGSA algorithm improved the size of
many covering arrays in comparison with the tools that are among the best found
1846 in the state-of-the-art of the construction of covering arrays.

Section 5.5 presented the results of comparing our parallel simulated annealing
1847 algorithm against the best bounds from the literature over a benchmark composed
1848 by 182 covering arrays. The quality solutions attained by our parallel approach is
1849 very competitive with respect to that produced by the state-of-the-art procedures,

in fact, it was able to improve on 134 previous best-known solutions and equaled
1850 the solutions for other 29 instances.

Finally, Section 5.6 showed two real examples of how to apply the combinatorial
1851 interaction testing.

The next chapter, and the last one, presents the main conclusions and contribu-
1852 tions derived from this research work.

Chapter 6

Conclusions

In this thesis we have examined the problem of constructing covering arrays for software interaction testing. We have proposed the development of an improved simulated annealing algorithm for constructing uniform and mixed covering arrays of strength $t >= 2$. In addition, we have proposed the use of Grid computing and Supercomputing to address the large amount of computing time necessary to obtain near-optimal covering arrays. Finally, the constructed covering arrays have been published in the repository described in Appendix A, in order that others can study the actual covering arrays, build new covering arrays from them, and also use these covering arrays without having to spend the computational resources.

6.1 Summary

Initially, we proposed an improved simulated annealing algorithm for constructing uniform and mixed covering arrays. The key features of the proposed simulated annealing are:

- ▷ An efficient method to generate initial solutions using maximum Hamming distance
- ▷ A carefully designed composed neighborhood function which allows the search to quickly reduce the total cost of candidate solutions, while avoiding to get stuck on some local minimal.
- ▷ An effective cooling schedule allowing our simulated annealing algorithm to converge faster, producing at the same time good quality solutions

In order to provide a good global performance of the proposed simulated annealing algorithm, we followed a fine tuning methodology for optimizing the assigned probabilities of execution for each of the three neighborhood functions using a linear Diophantine equation.

The empirical evidence presented in this work showed that our simulated annealing improved the size of the mixed covering arrays in comparison with the tools that are among the best found in the state-of-the-art of the construction of mixed covering arrays. The performance of the proposed simulated annealing algorithm was assessed with a benchmark, composed by 19 mixed covering arrays of strengths two and three taken from the literature. The computational results were reported and compared with previously published ones, showing that our algorithm was able to find 4 new upper bounds and to equal 15 previous best-known solutions on the selected benchmark instances.

Subsequently, we proposed the use of Grid computing and Supercomputing to address the large amount of computing time necessary to obtain near-optimal covering arrays, mainly for $t > 2$ and $v > 2$.

The main conclusion extracted from Grid implementation was the possibility of using two different schemas (asynchronous and synchronous) depending on the size of the experiment. On the one hand, the synchronous schema achieves better performance but is limited by the maximum number of slave connections that the master can keep track of. On the other hand, the asynchronous schema is slower but experiments with a huge value of slaves can be seamlessly performed. In order to show the performance of the Grid simulated annealing algorithm, we created a new benchmark composed by 60 ternary covering arrays instances where $5 \leq k \leq 100$ and $2 \leq t \leq 4$, and we have obtained the ACTS and TConfig softwares in order to compare Grid simulated annealing algorithm against them. The results showed that the Grid algorithm gets solutions of better quality than the other two tools.

Next, we proposed three approaches to parallelize the simulated annealing algorithm (independent search approach, semi-independent search approach, and cooperative search approach). From the experimental results, we found that the independent search approach is the worst performing offers, it does not scale. The semi-independent search approach offers reasonable execution times; compared to the independent search approach, communication overhead in the semi-independent search approach would be increased. However, each processor can utilize the information from other processors such that the decrease in computations and idle times can be greater than the increase in communication overhead. For instance, a certain processor which is trapped in an inferior solution can recognize its state by comparing it with others and may accelerate the annealing procedure. That is, processors may collectively converge to a better solution. The cooperative search approach is who offers the best results, it significantly reduces the execution time of the semi-independent search approach by employing asynchronous information exchange.

In order to show the performance of the cooperative search simulated annealing algorithm, we created a new benchmark composed by 182 covering arrays distributed as follows:

- ▷ 47 CAs with strength $t = 3$, degree $4 \leq k < 50$ and order $v = 3$
- ▷ 46 CAs with strength $t = 4$, degree $5 \leq k < 50$ and order $v = 3$
- ▷ 45 CAs with strength $t = 5$, degree $6 \leq k < 50$ and order $v = 3$
- ▷ 44 CAs with strength $t = 6$, degree $7 \leq k < 50$ and order $v = 3$

The analysis of results lead us to the following observation. The solutions quality attained by the cooperative search simulated annealing algorithm is very competitive with respect to that produced by the state-of-the-art procedures. In fact, it is able to improve on 134 previous best-known solutions. Even some of the best bounds were improved to reduce them hundreds of test cases.

These experimental results confirm the practical advantages of using our algorithm in the software testing area. It is a robust algorithm yielding smaller test suites than other representative state-of-the-art algorithms, which allows reducing software testing costs.

Finally, we presented two examples of generating test suits for real software components. The first one had been presented earlier by Kuhn et al. (2010), they constructed the test suite using the ACTS tool, setting IPOG-F as solution algorithm. Every test suite was solved 31 times by both approaches, the best solution for each case was registered, then the results of both approaches were compared, the results showed that our algorithm improves all previous bounds. The second example is a new benchmark, corresponding to a software system to control the activities for a scientific park, as the first case, the best solution achieved for each approach was registered, the results showed that the quality solution of our simulated annealing is better for all the cases in comparison with those obtained by ACTS.

6.2 Future work

The course of this research can take several ways, some of them are:

- ▷ Increase the experimentation for $v > 3$ and $t > 6$.
- ▷ Create a tool to merge the algebraic methods, recursive methods and metaheuristics methods, in order to create functional software tests.

Bibliography

Aarts, E. H. L. and P. J. M. Van Laarhoven (1985). "Statistical Cooling: A General Approach to Combinatorial Optimization Problems". In: *Philips Journal of Research* 40 (1985), pp. 193–226 (cit. on pp. 85, 91).

Atiqullah, Mir (2004). "An Efficient Simple Cooling Schedule for Simulated Annealing". In: *Proceedings of the International Conference on Computational Science and its Applications - ICCSA*. Vol. 3045. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 396–404. DOI: 10.1007/978-3-540-24767-8_41 (cit. on p. 85).

Avila-George, Himer et al. (2010a). *Verificación de Covering Arrays: Aplicando la Supercomputación y la Computación Grid*. LAP Lambert Academic Publishing, 2010. ISBN: 978-3-8433-5142-3.

Avila-George, Himer et al. (2010b). "Verification of General and Cyclic Covering Arrays Using Grid Computing". In: *Proceedings of the 3rd International Conference on Data Management in Grid and Peer-to-Peer Systems - GLOBE*. Vol. 6265. Lecture Notes in Computer Science. Bilbao, Spain, 30 August - 3 September: Springer-Verlag, 2010, pp. 112–123. ISBN: 978-3-642-15107-1. DOI: 10.1007/978-3-642-15108-8_10 (cit. on pp. XV, 61, 62, 65, 74).

Avila-George, Himer et al. (2011). "A parallel algorithm for the verification of Covering Arrays". In: *Proceedings of the 17th International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA*. Las Vegas, EEUU, July 18-21, 2011. ISBN: 1-60132-193-7. URL: http://world-comp.org/p2011/PDP8061.pdf (cit. on pp. XV, 62, 66, 74).

Avila-George, Himer et al. (2012a). "Grid Computing - Technology and Applications, Widespread Coverage and New Horizons". In: InTech, 2012. Chap. Using Grid Computing for the construction of ternary covering arrays. ISBN: 979-953-307-540-1 (cit. on pp. 139, 141).

Avila-George, Himer et al. (2012b). "Parallel Simulated Annealing for the Covering Arrays Construction Problem". In: *(to appear) Proceedings of the 18th International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA*. Las Vegas, EEUU, July 16-19, 2012 (cit. on pp. 92, 139, 141).

Avila-George, Himer et al. (2012c). "Simulated Annealing for Constructing Mixed Covering Arrays". In: *Proceedings of the 9th International Symposium on Distributed Computing and Artificial Intelligence - DCAI*. Vol. 151. Advances in Intelligent and Soft Computing. Salamanca, Spain, from 28th to 30th March: Springer Berlin / Heidelberg, 2012, pp. 657–664. ISBN: 978-3-642-28764-0. DOI: 10.1007/978-3-642-28765-7_79 (cit. on pp. 9, 139, 141, 144).

Avila-George, Himer et al. (2012d). "Supercomputing and Grid Computing on the verification of Covering Arrays". In: *The Journal of supercomputing* (2012). Published online: 18 April 2012, pp. 1–30. DOI: 10.1007/s11227-012-0763-0 (cit. on pp. XV, XVI, 62, 68–70, 74).

Avila-George, Himer et al. (April 2012). "A metaheuristic approach for constructing functional test-suites". In: *IET Software (submitted to a second round of revisions)* (April 2012).

Avila-George, Himer et al. (February 2012). "New bounds for ternary covering arrays using a parallel simulated annealing". In: *Submitted to: Mathematical Problems in Engineering* (February 2012).

Barker, H. A. (1986). "Sum and product tables for Galois fields". In: *International Journal of Mathematical Education in Science and Technology* 17 (1986), pp. 473 –485. DOI: 10.1080/0020739860170409 (cit. on p. 55).

Beizer, Boris (1990). *Software testing techniques*. New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN: 0-442-20672-0 (cit. on p. 3).

Bracho-Rios, Josue et al. (2009). "A New Backtracking Algorithm for Constructing Binary Covering Arrays of Variable Strength". In: *Proceedings of the 8th Mexican International Conference on Artificial Intelligence - MICAI*. Vol. 5845. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 397–407. DOI: 10.1007/978-3-642-05258-3_35 (cit. on pp. 139, 141).

Bryce, Renée C. and Charles J. Colbourn (2007). "The density algorithm for pairwise interaction testing". In: *Software Testing, Verification and Reliability*

17.3 (2007), pp. 159–182. ISSN: 0960-0833. DOI: 10.1002/stvr.365 (cit. on pp. XV, 9, 42, 43, 99, 101).

Bryce, Renée C. et al. (2010). "Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization". In: IGI Global, 2010. Chap. Combinatorial Testing, pp. 196–208 (cit. on pp. 6, 9).

Burr, Kevin and William Young (1998). "Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage". In: *Proceedings of the International Conference on Software Testing, Analysis, and Review - STAR*. West, 1998, pp. 503–513 (cit. on pp. 6, 18).

Bush, K. A. (1952). "Orthogonal Arrays of Index Unity". In: *Annals of Mathematical Statistics* 23.3 (1952), pp. 426–434. DOI: 10.1214/aoms/1177729387 (cit. on pp. 9, 26, 27, 54).

Cavique, L. et al. (1999). "Subgraph Ejection Chains and Tabu Search for the Crew Scheduling Problem". In: *The Journal of the Operational Research Society* 50.6 (1999), pp. 608–616. URL: http://www.ingentaconnect.com/content/pal/01605682/1999/00000050/00000006/2600728 (cit. on p. 84).

Cawse, James N. (2003). *Experimental Design for Combinatorial and High Throughput Materials Development*. John Wiley & Sons, Inc., 2003 (cit. on p. 18).

Chateauneuf, M. and D. L. Kreher (2002). "On the state of strength-three covering arrays". In: *Journal of Combinatorial Designs* 10.4 (2002), pp. 217–238. ISSN: 1520-6610. DOI: 10.1002/jcd.10002 (cit. on pp. 27, 36, 45).

Cheng, C. T. (2007). "The test suite generation problem: optimal instances and their implications". In: *Discrete Applied Mathematics* 155 (2007), pp. 1943–1957 (cit. on p. 26).

Cohen, David M. et al. (1996). "The Combinatorial Design Approach to Automatic Test Generation". In: *IEEE Software* 13.5 (1996), pp. 83–88. ISSN: 0740-7459. DOI: 10.1109/52.536462 (cit. on pp. XV, 9, 41, 42, 99, 101).

Cohen, Myra B. et al. (2003). "Augmenting simulated annealing to build interaction test suites". In: *Proceedings of the 14th International Symposium on Software Reliability Engineering - ISSRE*. IEEE Computer Society, 2003, pp. 394–405. DOI: 10.1109/ISSRE.2003.1251061 (cit. on pp. 8, 9, 22, 49, 53, 79, 82, 99, 101, 102).

Cohen, Myra B. et al. (2008). “Constructing strength three covering arrays with augmented annealing”. In: *Discrete Mathematics* 308.13 (2008), pp. 2709–2722. ISSN: 0012-365X. DOI: 10.1016/j.disc.2006.06.036 (cit. on pp. 37, 53, 94).

Colbourn, Charles J. (2004). “Combinatorial aspects of covering arrays”. In: *Le Matematiche* 59 (2004), pp. 121–167 (cit. on p. 26).

Colbourn, Charles J. (2010). “Covering arrays from cyclotomy”. In: *Designs, Codes and Cryptography* 55.2-3 (2010), pp. 201–219. ISSN: 0925-1022. DOI: 10.1007/s10623-009-9333-8 (cit. on p. 55).

Colbourn, Charles J. (2011). *Covering Array Tables for t=2,3,4,5,6.* Accessed on April 20. 2011. URL: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html (cit. on pp. 105, 106, 108–110, 138, 142).

Colbourn, Charles J. and Jose Torres-Jimenez (2010). “Error-Correcting Codes, Finite Geometries and Cryptography”. In: vol. 523. ISBN-10 0-8218-4956-5. Contemporary Mathematics, 2010. Chap. Heterogeneous Hash Families and Covering Arrays, pp. 3–15 (cit. on pp. 9, 21, 60).

Colbourn, Charles J. et al. (2005). “Progressive Ranking and Composition of Web Services Using Covering Arrays”. In: *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.* WORDS ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 179–185. ISBN: 0-7695-2347-1. DOI: 10.1109/WORDS.2005.47 (cit. on p. 6).

Colbourn, Charles J. et al. (2006a). “Products of mixed covering arrays of strength two”. In: *Journal of Combinatorial Designs* 12.2 (2006), pp. 124–138. DOI: 10.1002/jcd.20065 (cit. on pp. 34, 144).

Colbourn, Charles J. et al. (2006b). “Roux-type constructions for covering arrays of strengths three and four”. In: *Designs, Codes and Cryptography* 41 (1 2006), pp. 33–57. ISSN: 0925-1022. DOI: 10.1007/s10623-006-0020-8 (cit. on p. 39).

Czech, Zbigniew J. (2006). “Three Parallel Algorithms for Simulated Annealing”. In: *Parallel Processing and Applied Mathematics.* Vol. 2328. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 210–217. DOI: 10.1007/3-540-48086-2_23 (cit. on p. 91).

Dalal, S. R. et al. (1999). “Model-based testing in practice”. In: *Proceedings of the 21st international conference on Software engineering.* ICSE ’99. Los Angeles, California, United States: ACM, 1999, pp. 285–294. ISBN: 1-58113-074-0. DOI:

10.1145/302405.302640. URL: http://doi.acm.org/10.1145/302405.302640 (cit. on p. 6).

DIANE (2011). *Distributed Analysis Environment.* Accessed on June 6. 2011. URL: http://it-proj-diane.web.cern.ch/it-proj-diane/ (cit. on pp. 72, 89).

Dorigo, M. et al. (1996). “Ant system: optimization by a colony of cooperating agents”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 26.1 (1996), pp. 29–41. ISSN: 1083-4419. DOI: 10.1109/3477.484436 (cit. on p. 51).

Fisher, Ronald A. (1926). “The Arrangement of Field Experiments”. In: *Journal of the Ministry of Agriculture of Great Britain* 33 (1926), pp. 503–513. URL: http://hdl.handle.net/2440/15191 (cit. on p. 13).

Forbes, Michael et al. (2008). “Refining the In-Parameter-Order Strategy for Constructing Covering Arrays”. In: *Journal of Research of the National Institute of Standards and Technology* 113.5 (2008), pp. 287–297 (cit. on pp. 99, 138).

Frankl, P. G. and S. N. Weiss (1993). “An experimental comparison of the effectiveness of branch testing and data flow testing”. In: *IEEE Transactions on Software Engineering* 19.8 (1993), pp. 774 –787. ISSN: 0098-5589. DOI: 10.1109/32.238581 (cit. on p. 3).

Frankl, P. G. and E. J. Weyuker (1988). “An applicable family of data flow testing criteria”. In: *IEEE Transactions on Software Engineering* 14.10 (1988), pp. 1483 –1498. ISSN: 0098-5589. DOI: 10.1109/32.6194 (cit. on p. 2).

Glover, Fred (1986). “Future paths for integer programming and links to artificial intelligence”. In: *Computers & Operations Research* 13.5 (1986), pp. 533 –549. ISSN: 0305-0548. DOI: 10.1016/0305-0548(86)90048-1 (cit. on p. 49).

Gonzalez-Hernandez, Loreto et al. (2010). “Construction of mixed covering arrays of variable strength using a tabu search approach”. In: *Proceedings of the 4th international conference on Combinatorial optimization and applications - COCOA*. Vol. 6508. Lecture Notes in Computer Science. Kailua-Kona, HI, USA: Springer-Verlag, 2010, pp. 51–64. ISBN: 3-642-17457-4, 978-3-642-17457-5. DOI: 10.1007/978-3-642-17458-2_6 (cit. on pp. XV, 9, 49–51, 99–101, 141).

Gopalakrishnan, K. and D. R. Stinson (2006). “Applications of Orthogonal Arrays to Computer Science”. In: *International Conference on Discrete Mathematics - ICDM*. Lecture Notes Series in Mathematics. Ramanujan Mathematical

Society, 2006, pp. 149–164. URL: http://www.cs.ecu.edu/~gopal/icdm-pubver.pdf (cit. on p. 54).

Hartman, Alan (2005). “Software and Hardware Testing Using Combinatorial Covering Suites”. In: *Graph Theory, Combinatorics and Algorithms*. Vol. 34. Operations Research/Computer Science Interfaces Series. Springer US, 2005, pp. 237–266. ISBN: 978-0-387-25036-6. DOI: 10.1007/0-387-25036-0_10 (cit. on pp. 1, 33).

Hartman, Alan and Leonid Raskin (2004). “Problems and algorithms for covering arrays”. In: *Discrete Mathematics* 284.1-3 (2004), pp. 149 –156. DOI: 10.1016/j.disc.2003.11.029 (cit. on p. 144).

Hedayat, A. S. et al. (1999). *Orthogonal Arrays: Theory and Applications*. Springer-Verlag, 1999. ISBN: 978-0387987668 (cit. on pp. 14, 16, 18, 54).

Hnich, Brahim et al. (2006). “Constraint Models for the Covering Test Problem”. In: *Constraints* 11 (2 2006), pp. 199–219. ISSN: 1383-7133. DOI: 10.1007/s10601-006-7094-9 (cit. on p. 20).

Jenkins, Bob (2011). *Jenny: a pairwise testing tool*. Accessed on June 22. 2011. URL: http://burtleburtle.net/bob/math/jenny.html (cit. on pp. 99–101).

Jun, Yang and Satoshi Mizuta (2005). “Detailed Analysis of Uphill Moves in Temperature Parallel Simulated Annealing and Enhancement of Exchange Probabilities”. In: *Complex Systems* 15.4 (2005), pp. 349–358. URL: http://www.complex\-systems.com/abstracts/v15_i04_a04.html (cit. on pp. 52, 78).

Katona, G. O. H. (1973). “Two applications (for search theory and truth functions) of Sperner type theorems”. In: *Periodica Mathematica Hungarica* 3.1-2 (1973), pp. 19–26. DOI: 10.1007/BF02018457 (cit. on p. 27).

Kleitman, Daniel J. and Joel Spencer (1973). “Families of k-independent sets”. In: *Discrete Mathematics* 6.3 (1973), pp. 255–262. DOI: 10.1016/0012-365X(73)90098-8 (cit. on p. 27).

Kuhn, D. Richard et al. (2004). “Software Fault Interactions and Implications for Software Testing”. In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 418–421. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.24 (cit. on p. 6).

Kuhn, D. Richard et al. (2008). "Practical Combinatorial Testing: Beyond Pairwise". In: *IT Professional* 10.3 (2008), pp. 19–23. ISSN: 1520-9202. DOI: 10.1109/MITP.2008.54 (cit. on p. 7).

Kuhn, D. Richard et al. (2010). *Practical Combinatorial Testing*. Tech. rep. National Institute of Standards and Technology, 2010 (cit. on pp. 6, 121).

Lawrence, James et al. (2011). "A survey of binary covering arrays". In: *The Electronic Journal of Combinatorics* 18.1 (2011), p. 84 (cit. on p. 26).

Lee, Soo-Young and Kyung Geun Lee (1996). "Synchronous and Asynchronous Parallel Simulated Annealing with Multiple Markov Chains". In: *IEEE Transactions on Parallel and Distributed Systems* 7.10 (10 1996), pp. 993–1008. ISSN: 1045-9219. DOI: 10.1109/71.539732 (cit. on p. 91).

Lei, Yu and Kuo-Chung Tai (1998). "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing". In: *Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering - HASE*. IEEE Computer Society, 1998, pp. 254–261. ISBN: 0-8186-9221-9 (cit. on pp. XV, 26, 43, 44).

Lei, Yu et al. (2007). "IPOG: A General Strategy for T-Way Software Testing". In: *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems - ECBS*. Tucson, AZ, USA: IEEE Computer Society, 2007, pp. 549–556. DOI: 10.1109/ECBS.2007.47 (cit. on pp. 9, 99–101).

Lei, Yu et al. (2008). "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing". In: *Software Testing, Verification and Reliability* 18.3 (2008), pp. 125–148. ISSN: 1099-1689. DOI: 10.1002/stvr.381 (cit. on pp. XV, 44, 45).

Lobb, Jason R. et al. (2012). "Cover starters for covering arrays of strength two". In: *Discrete Mathematics* 312.5 (2012), pp. 943 –956. ISSN: 0012-365X. DOI: 10.1016/j.disc.2011.10.026 (cit. on p. 28).

Lopez-Escogido, D. et al. (2008). "Strength Two Covering Arrays Construction Using a SAT Representation". In: *roceedings of the 7th Mexican International Conference on Artificial Intelligence - MICAI*. Vol. 5317. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 44–53. DOI: 10.1007/978-3-540-88636-5_4 (cit. on p. 141).

Mala, D. Jeya et al. (2010). “Automated software test optimisation framework – an artificial bee colony optimisation-based approach”. In: *IET Software* 4.5 (2010), pp. 334–348. DOI: 10.1049/iet-sen.2009.0079 (cit. on p. 8).

Mandl, Robert (1985). “Orthogonal Latin Squares: an application of experiment design to compiler testing”. In: *Communications of the ACM* 28.10 (1985), pp. 1054–1058. DOI: 10.1145/4372.4375 (cit. on pp. 13, 16).

Martinez-Pena, Jorge and Jose Torres-Jimenez (2010). “A Branch and Bound Algorithm for Ternary Covering Arrays Construction Using Trinomial Coefficients”. In: *Research in Computing Science* 49 (2010), pp. 61–71. ISSN: 1870-4069 (cit. on pp. 31, 139, 141).

Martinez-Pena, Jorge et al. (2010). “A Heuristic Approach for Constructing Ternary Covering Arrays Using Trinomial Coefficients”. In: *Proceedings of the 12th Ibero-American conference on Advances in artificial intelligence - IBERAMIA*. Vol. 6433. Lecture Notes in Computer Science. Bahía Blanca, Argentina, November 1-5: Springer-Verlag, 2010, pp. 572–581. ISBN: 978-3-642-16951-9. DOI: 10.1007/978-3-642-16952-6_58 (cit. on pp. 53, 139, 141).

Martirosyan, Sosina S. and Charles J. Colbourn (2005). “Recursive constructions of covering arrays”. In: *Bayreuther Mathematische Schriften* 74 (2005), pp. 266–275 (cit. on p. 39).

McDowell, A. G. (2011). *All-Pairs Testing*. Accessed on June 21. 2011. URL: http://www.mcdowella.demon.co.uk/allPairs.html (cit. on pp. 9, 99–101).

McMinn, Phil (2004). “Search-based software test data generation: a survey”. In: *Software Testing, Verification and Reliability* 14.2 (2004), pp. 105–156. ISSN: 1099-1689. DOI: 10.1002/stvr.294 (cit. on p. 3).

Meagher, Karen and Brett Stevens (2005). “Group construction of covering arrays”. In: *Journal of Combinatorial Designs* 13.1 (2005), pp. 70–77. ISSN: 1520-6610. DOI: 10.1002/jcd.20035 (cit. on p. 27).

Mogyorodi, G. (2001). “Requirements-based testing: an overview”. In: *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*. Santa Barbara CA, USA, from 29 jul to 03 ago, 2001, pp. 286–295. DOI: 10.1109/TOOLS.2001.941681 (cit. on p. 4).

Moscicki, J. T. et al. (2009). “Ganga: A tool for computational-task management and easy access to Grid resources”. In: *Computer Physics Communications*

180.11 (2009), pp. 2303–2316. DOI: 10.1016/j.cpc.2009.06.016 (cit. on pp. 72, 89).

Moura, Lucia et al. (2003). “Covering arrays with mixed alphabet sizes”. In: *Journal of Combinatorial Designs* 11.6 (2003), pp. 413–432. ISSN: 1520-6610. DOI: 10.1002/jcd.10059 (cit. on p. 9).

National Institute of Standards and Technology (2011). *NIST Covering Array Tables*. Accessed on April 20. 2011. URL: http://math.nist.gov/coveringarrays/ (cit. on pp. 111, 138).

Niederreiter, H. (1990). “A short proof for explicit formulas for discrete logarithms in finite fields”. In: *Applicable Algebra in Engineering, Communication and Computing* 1.1 (1990), pp. 55–57. ISSN: 0938-1279. DOI: 10.1007/BF01810847 (cit. on p. 55).

Nurmela, Kari J. (2004). “Upper bounds for covering arrays by tabu search”. In: *Discrete Applied Mathematics* 138 (1-2 2004), pp. 143–152. ISSN: 0166-218X. DOI: 10.1016/S0166-218X(03)00291-9 (cit. on pp. 49, 50, 82).

Offutt, A. Jefferson et al. (1996). “An Experimental Evaluation of Data Flow and Mutation Testing”. In: *Software: Practice and Experience* 26.2 (1996), pp. 165–176. ISSN: 1097-024X. DOI: 10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K (cit. on p. 3).

Pacini, F. (2011). *Job Description Language HowTo*. Accessed on October 10. 2011. URL: http://server11.infn.it/workload-grid/docs/DataGrid-01-TEN-0102-0_2-Document.pdf (cit. on pp. 71, 86).

Phadke, Madhan Shridhar (1995). *Quality Engineering Using Robust Design*. Prentice Hall PTR, 1995. ISBN: 0137451679 (cit. on p. 18).

Quiz-Ramos, Pedro et al. (2009). “Constant Row Maximizing Problem for Covering Arrays”. In: *MICAI 2009: Proceedings of the Eighth Mexican International Conference on Artificial Intelligence*. IEEE Computer Society, 2009, pp. 159–164. ISBN: 978-0-7695-3933-1. DOI: 10.1109/MICAI.2009.28 (cit. on p. 141).

Ram, D. Janaki et al. (1996). “Parallel Simulated Annealing Algorithms.” In: *Journal of Parallel and Distributed Computing* 37.2 (1996), pp. 207–212. DOI: 10.1006/jpdc.1996.0121 (cit. on p. 86).

Rao, C. R. (1946). “Hypercube of strength ’d’ leading to confounded designs in factorial experiments”. In: *Bulletin of the Calcutta Mathematical Society* 38 (1946), pp. 67–78 (cit. on p. 16).

Reid, S. C. (1997). “An empirical analysis of equivalence partitioning, boundary value analysis and random testing”. In: *Proceedings of the Fourth International Software Metrics Symposium.* 1997, pp. 64–73. DOI: 10.1109/METRIC.1997.637166 (cit. on p. 4).

Rényi, A. (1971). *Foundations of Probability.* New York, USA: John Wiley & Sons, 1971 (cit. on p. 27).

Rodriguez-Tello, Eduardo and Jose Torres-Jimenez (2009). “Memetic Algorithms for Constructing Binary Covering Arrays of Strength Three”. In: *Proceedings of the 9th International Conference Evolution Artificielle - EA*. Vol. 5975. Springer-Verlag, 2009, pp. 86–97. DOI: 10.1007/978-3-642-14156-0_8 (cit. on pp. 54, 139, 141).

Ronneseth, Andreas H. and Charles J. Colbourn (2009). “Merging covering arrays and compressing multiple sequence alignments”. In: *Discrete Applied Mathematics* 157.9 (2009), pp. 2177 –2190. ISSN: 0166-218X. DOI: 10.1016/j.dam.2007.09.024 (cit. on pp. XV, 45, 47).

Roux, G (1987). “k-Propriétés dans les tableaux de n colonnes: cas particulier de la k-surjectivité et de la k-permutivité”. PhD thesis. Université de Paris, 1987 (cit. on p. 36).

Seroussi, G. and N. Bshouty (1988). “Vector sets for exhaustive testing of logic circuits”. In: *IEEE Transactions on Information Theory* 34 (1988), pp. 513–522 (cit. on pp. 26, 50).

Shasha, Dennis E. et al. (2001). “Using combinatorial design to study regulation by multiple input signals: A tool for parsimony in the post-genomics era”. In: *Plant Physiology* 127.4 (2001), pp. 1590–1594. DOI: 10.1104/pp.010683 (cit. on p. 18).

Sherwood, George (2011). *On the Construction of Orthogonal Arrays and Covering Arrays Using Permutation Groups.* Accessed April 20, 2011. 2011. URL: http://testcover.com/pub/background/cover.htm (cit. on p. 138).

Sherwood, George B. (2008). “Optimal and near-optimal mixed covering arrays by column expansion”. In: *Discrete Mathematics* 308.24 (2008), pp. 6022 –6035. ISSN: 0012-365X. DOI: 10.1016/j.disc.2007.11.021 (cit. on p. 9).

Shiba, T. et al. (2004). "Using artificial life techniques to generate test cases for combinatorial testing". In: *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01 - COMPSAC*. Hong Kong: IEEE Computer Society, 2004, pp. 72–77. DOI: 10.1109/CMPSAC.2004.1342808 (cit. on pp. 9, 49, 52, 82, 99, 101).

Sloane, N. J. A. (1993). "Covering arrays and intersecting codes". In: *Journal of Combinatorial Designs* 1.1 (1993), pp. 51–63. ISSN: 1520-6610. DOI: 10.1002/jcd.3180010106 (cit. on pp. 20, 36).

Stardom, John (2001). "Metaheuristics and the Search for Covering and Packing Arrays". MA thesis. Simon Fraser University, 2001 (cit. on pp. 52, 79).

Stinson, D. R. (2004). "Orthogonal Arrays and Codes". In: *Combinatorial Designs*. Springer-Verlag, New York, 2004. Chap. 10, pp. 225–255. ISBN: 978-0-387-21737-6 (cit. on p. 54).

Taguchi, G. (1994). *Taguchi Methods: Design of Experiments*. American Supplier Institute, 1994 (cit. on p. 54).

Tang, D. T. and L. S. Woo (1983). "Exhaustive Test Pattern Generation with Constant Weight Vectors". In: *IEEE Transactions on Computers* 32.12 (1983), pp. 1145–1150. ISSN: 0018-9340. DOI: 10.1109/TC.1983.1676175 (cit. on pp. 29, 31).

Tatsumi, K. (1987). "Test case design support system". In: *Proceedings of the International Conference on Quality Control - ICQC*. Tokyo, 1987, pp. 615–620. URL: http://www.pairwise.org/docs/icqc87.pdf (cit. on p. 16).

Torres-Jimenez, Jose et al. (2004). "Computation of Ternary Covering Arrays Using a Grid". In: *Proceedings of the Second Asian Applied Computing Conference - AACC*. Vol. 3285. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 240–246. DOI: 10.1007/978-3-540-30176-9_31 (cit. on p. 141).

Torres-Jimenez, Jose et al. (2010). "Optimization of investment options using SQL". In: *Proceedings of the 12th Ibero-American conference on Advances in artificial intelligence - IBERAMIA*. Vol. 6433. Lecture Notes in Computer Science. Bahía Blanca, Argentina, November 1-5: Springer-Verlag, 2010, pp. 30–39. ISBN: 978-3-642-16951-9. DOI: 10.1007/978-3-642-16952-6_4.

Torres-Jimenez, Jose et al. (2011a). "Construction of logarithm tables for Galois Fields". In: *International Journal of Mathematical Education in Science and*

Technology 42.1 (2011), pp. 91–102. DOI: 10.1080/0020739X.2010.510215 (cit. on pp. XV, 54, 56–59).

Torres-Jimenez, Jose et al. (2011b). "MAXCLIQUE Problem Solved Using SQL". In: *Proceedings of the third International Conference on Advances in Databases, Knowledge, and Data Applications - DBKDA*. St. Maarten, The Netherlands Antilles, January 23-28: IARIA, 2011, pp. 83–88. ISBN: 978-1-61208-115-1. URL: http://www.thinkmind.org/download.php?articleid=dbkda_2011_4_40_30097.

Torres-Jimenez, Jose et al. (2012). "Cryptography and Security in Computing". In: InTech, 2012. Chap. Construction of Orthogonal Arrays of Index Unity using Logarithm Tables for Galois Fields, pp. 71–90. ISBN: 978-953-51-0179-6. URL: http://www.intechopen.com/download/pdf/29702 (cit. on pp. XV, 59, 61, 62).

Torres-Jimenez, Jose et al. (September 2011). "CINVESTAV Covering Arrays Repository". In: *submitted to: IET Software* (September 2011).

Tung, Yu-Wen and W. S. Aldiwan (2000). "Automating test case generation for the new generation mission software system". In: *Proceedings of the IEEE Aerospace Conference*. Vol. 1. IEEE Press, 2000, pp. 431–437. DOI: 10.1109/AERO.2000.879426 (cit. on pp. XV, 9, 41, 42, 99, 101).

Vadde, K. K. and V. R. Syrotiuk (2004). "Factor interaction on service delivery in mobile ad hoc networks". In: *IEEE Journal on Selected Areas in Communications* 22.7 (2004), pp. 1335 –1346. ISSN: 0733-8716. DOI: 10.1109/JSAC.2004.829351 (cit. on p. 18).

Walker II, Robert A. and Charles J. Colbourn (2009). "Tabu search for covering arrays using permutation vectors". In: *Journal of Statistical Planning and Inference* 139.1 (2009), pp. 69 –80. ISSN: 0378-3758. DOI: 10.1016/j.jspi.2008.05.020 (cit. on p. 50).

Weyuker, E. J. (1998). "Testing component-based software: a cautionary tale". In: *IEEE Software* 15.5 (1998), pp. 54 –59. ISSN: 0740-7459. DOI: 10.1109/52.714817 (cit. on p. 1).

Williams, Alan W. (2000). "Determination of Test Configurations for Pair-Wise Interaction Coverage". In: *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques - TestCom*. Ottawa, Canada: Kluwer, 2000, pp. 57–72. ISBN: 0-7923-7921-7.

URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.9168 (cit. on pp. 9, 99–101).

Yan, Jun and Jian Zhang (2008). "An efficient method to generate feasible paths for basis path testing". In: *Information Processing Letters* 107.3–4 (2008), pp. 87 –92. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2008.01.007 (cit. on p. 3).

Yilmaz, Cemal et al. (2006). "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces". In: *IEEE Transactions on Software Engineering* 32.1 (2006), pp. 20–34. DOI: 10.1109/TSE.2006.8 (cit. on pp. 6, 18).

Younis, Mohammed I. et al. (2010). "Assessing IRPS as an efficient pairwise test data generation strategy". In: *International Journal of Advanced Intelligence Paradigms* 2.1 (2010), pp. 90–104. ISSN: 1755-0386. DOI: 10.1504/IJAIP.2010.029443 (cit. on p. 46).

Appendix A

Covering arrays repository

This appendix shows in detail a new *Covering Arrays Repository* (CAR) which is a very large database that contains a wide variety of covering arrays. Also, it discusses the importance of CARs in the construction of better covering arrays. The covering arrays that can be found in the repository have alphabet value of $v = \{2, 3, \ldots, 27\}$, strength of $t = \{2, 3, \ldots, 17\}$ and some of them involve up to 20000 parameters, or columns. The size N of some of the covering arrays included in the repository are the best upper bounds known in the literature. Moreover, the files containing the matrices of those covering arrays are available to be downloaded in the repository. In general, the appendix presents a complete description of the new repository that includes: the strategies that have been used to construct the covering arrays; a set of tables showing some of the upper bounds in the covering arrays that can be found there; a brief description of the web-based interface of it and a graphical summary of the covering arrays that it contains.

The remaining of this appendix is structured as follows: Section A.1 describes the relevant work related with the existence of repositories of covering arrays. Section A.2 shows the theoretical basis and structure of the repository and presents a summary of the new upper bounds that can be found in the repository. Section A.3 presents two cases of study for the construction of covering arrays, reported in the literature, that were benefited from covering arrays matrices to improve upper bounds of other covering arrays. Finally, conclusions and further work are presented in Section A.4.

A.1 Repositories for covering arrays

As it has been pointed out in the last section, there exist a lot of approaches devoted to the construction of covering arrays. While some approaches proposes new benchmarks for the problem of constructing covering arrays, some others improves the size of the existing covering arrays. However, a closer looks in the results reported by those works will show us that the research on the construction of covering arrays lacks from a place where all those covering arrays can be located, by other researches, or at least the best known covering arrays.

In general, given that the covering arrays have been studied from several viewpoints and constructed from a wide variety of approaches, to get a covering array table from determined values of (t, k, v) becomes difficult and sometimes impossible. The repositories of covering arrays that can be found on the Web partially overcome this situation. Some of these repositories contain information of the optimal size of the covering arrays and how can they be constructed, others have available the covering arrays tables. To the best of our knowledge, the repositories found in the literature are: the one maintained by Colbourn (2011), the covering arrays tables listed in the National Institute of Standards and Technology (2011) and the covering arrays described by Sherwood (2011).

The repository maintained by Charlie Colbourn publishes the current best known upper bounds for covering arrays. The repository does not contain the covering array tables, instead it records the information of the smallest covering arrays that have been constructed, and reported in the literature, for a wide variety of values of (t, k, v). For each value (t, k, v), reported in the repository, a reference to the technique used to construct the covering array is presented.

The NIST repository is maintained through the Automated Combinatorial Testing for Software (ACTS) project at NIST. This repository stores covering arrays tables that have been constructed by the IPOG-F algorithm (Forbes et al., 2008). The IPOG-F algorithm is fast and can construct small covering arrays however, the difference between the size of the covering arrays constructed by IPOG-F and the minimum size possible of the covering arrays tends to grow rapidly with increasing value of (t, k, v). Then, the NIST repository has the advantage of proportioning the covering array tables but with the disadvantage that they commonly are not the best possible ones.

The Sherwood repository has the peculiarity that it presents covering array tables and describes the method used to construct them in the same website. The method of construction is based on orthogonal arrays and permutation vectors and strategies that combines them to produce new covering arrays. The website does not provide the covering arrays tables, instead it gives the methods that must be used to construct them. The covering arrays described in this repository have the minimum number of rows possible, for the values of (t, k, v) considered.

Table A.1 present a brief comparison of the repositories already described. The comparison is made in terms of the covering arrays tables that can be found in each of them. The column 1 shows the repository; the columns 2, 3 and 4 contains the strengths t, alphabets v and columns k values, respectively, for which a covering array table is reported in the repository. Finally, the column 5 shows a reference of a constructed covering array table is available in the repository or not.

Table A.1: Description of the repositories for covering arrays available in the Web.

Repository	t	k	v	**CA tables**
Colbourn tables	$\{2,\ldots,6\}$	up to 20000	$\{2,\ldots,24\}$	no
NIST	$\{2,\ldots,6\}$	up to 74	$\{2,\ldots,6\}$	yes
Sherwood tables	$\{2,\ldots,4\}$	up to 273	$\{2,\ldots,13\}$	no

Summarizing, the NIST repository is the only one from which we can download explicit covering array tables however, these tables are not the best ones that can be found in the literature (Bracho-Rios et al., 2009; Rodriguez-Tello and Torres-Jimenez, 2009; Avila-George et al., 2012c; Avila-George et al., 2012a; Avila-George et al., 2012b; Martinez-Pena et al., 2010; Martinez-Pena and Torres-Jimenez, 2010). The Sherwood repository reports some of the best covering array tables, in terms of their size, but these tables are not constructed (instead, a method to construct them is provided) and only covers a reduce number of covering array tables (in comparison with the other two repositories). Finally, the Charlie Colbourn repository contains the best found upper bounds for a range of values of (t, k, v) wider than those reported in the NIST and Sherwood repositories. Also the Charlie Colbourn repository includes a reference to the approach followed to achieve those upper bounds. However, the repository does not includes the tables for the values of (t, k, v) that it reports.

In conclusion, the scientific community lack from a repository that offers explicit covering arrays tables for a wide range of (t, k, v) and guarantees that the sizes of the table are competitive among the best known values reported in the literature. For this purpose, in this appendix is proposed a new repository that includes constructed covering array tables for values of (t, k, v). The next section presents the new repository and compares it with the existing ones. The section begins with the presentation of the techniques used for the construction of covering arrays, next, it describes the web tools developed for its management and access. Finally, it shows covering array tables with new upper bounds when $t \leq 6$.

A.2 CINVESTAV covering arrays repository

This section presents the new covering array repository (CAR), which is available under request at http://www.tamps.cinvestav.mx/~jtj/CA.php. The covering array repository is maintained by the Group of Optimal Experimental Design (GOED)[1].

The remaining of the section describes in dept details about the repository concerning: the approaches used to construct the covering arrays, the web interface to access the repository, a graphical presentation of some upper bounds for the size of the covering arrays that can be found and a comparison against the state-of-the-art repositories.

A.2.1 Algorithms

The algorithms that have been used to construct the covering arrays located in the repository are based in a wide variety of approaches. A wide variety of approaches form been follow in order to construct the covering arrays stored in the repository. Table A.2 summarizes some of the most recent approaches that have been followed to support the construction of the covering arrays of the repository.

Basically, the algorithms for the construction of covering arrays varies from exact to approximated algorithms. The exact approaches construct optimal solution for small covering arrays. The approximated algorithms allows the construction of larger covering arrays than those produced by the exact algorithm but with the disadvantage that they are not of the optimal size. Among the approaches referred for the construction of covering arrays are also those ones that contributes in the generation of new covering array by identifying special structures of them that could help in the reduction of rows (as the case of the covering arrays with a large number of constant rows) and to verify rapidly that a matrix is a covering array (like the verification approaches).

The following subsection describes the graphical web interface of the repository.

A.2.2 Repository description

The repository has a multi-parametric interface to find a specific covering array (see Figure A.1), the queries can be done by v, t, k or any combination of them.

grows logarithmically as the number of columns grows linearly.

[1]Found at the *Centro de Investigación y de Estudios Avanzados del IPN* (CINVESTAV)

Table A.2: Algorithms used to construct the covering arrays of the repository reported in this appendix, they has been constructed by GOED.

Algorithm	**Description**
CRMP (Quiz-Ramos et al., 2009)	Exact approach for the maximization of the number of constant rows in a CA.
B&B (Bracho-Rios et al., 2009; Martinez-Pena and Torres-Jimenez, 2010)	Exact approach for the construction of covering arrays.
SA (Avila-George et al., 2012c; Avila-George et al., 2012b)	Simulated annealing algorithm to construct covering arrays.
SA & SAT models (Lopez-Escogido et al., 2008)	An approach to construct covering arrays using the propositional satisfiability problem (SAT).
SA & Trinomial Coefficients (Martinez-Pena et al., 2010)	A Heuristic approach for constructing covering arrays using trinomial coefficients.
MA (Rodriguez-Tello and Torres-Jimenez, 2009)	Memetic algorithm to construct covering arrays.
MiTS (Gonzalez-Hernandez et al., 2010)	Tabu search algorithm to construct mixed covering arrays.
Grid (Torres-Jimenez et al., 2004; Avila-George et al., 2012a)	Grid approaches for covering arrays.

A.2.3 Scope and upper bounds of the repository

In this section, we show the kind of covering arrays that can be found in this repository and the new upper bounds. The kind of covering arrays that can be found in this repository are briefly described in Table A.3.

Table A.3: Description of the covering array repository.

Level	**Columns**	**Cardinality**	**Strength**
Fixed	$3 \leq k \leq 20.000$	$2 \leq v \leq 27$	$2 \leq t \leq 17$
Mixed	$4 \leq k \leq 75$	$2 \leq max\ v \leq 11$	$2 \leq t \leq 6$

Table A.4 shows covering array tables which contains new upper bounds and they can be downloaded from our repository.

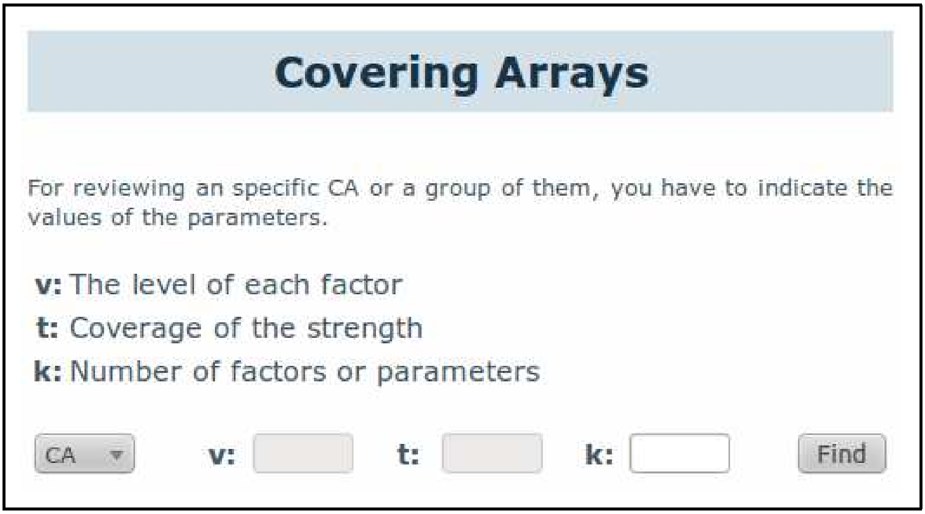

Figure A.1: Multi-parametric repository interface.

Table A.4: New upper bounds achieved with the algorithms described in Table A.2.

$CAN(t, k, v)$	Upper bounds *
CAN(3,k,2)	*26*
CAN(4,k,2)	*28*
CAN(5,k,2)	*21*
CAN(6,k,2)	*64*
CAN(2,k,3)	*15*
CAN(2,k,4)	*19*
CAN(2,k,5)	*20*
CAN(2,k,6)	*40*
CAN(2,k,7)	*10*
CAN(2,k,8)	*2*
CAN(2,k,9)	*3*
CAN(2,k,10)	*14*
CAN(2,k,11)	*6*
CAN(2,k,12)	*2*
CAN(2,k,14)	*13*
CAN(2,k,15)	*4*

* See Colbourn tables (Colbourn, 2011).

A.3 Cases of study

In this section, we present two approaches for the construction of covering arrays, reported in the literature, that were benefited from covering arrays matrices to

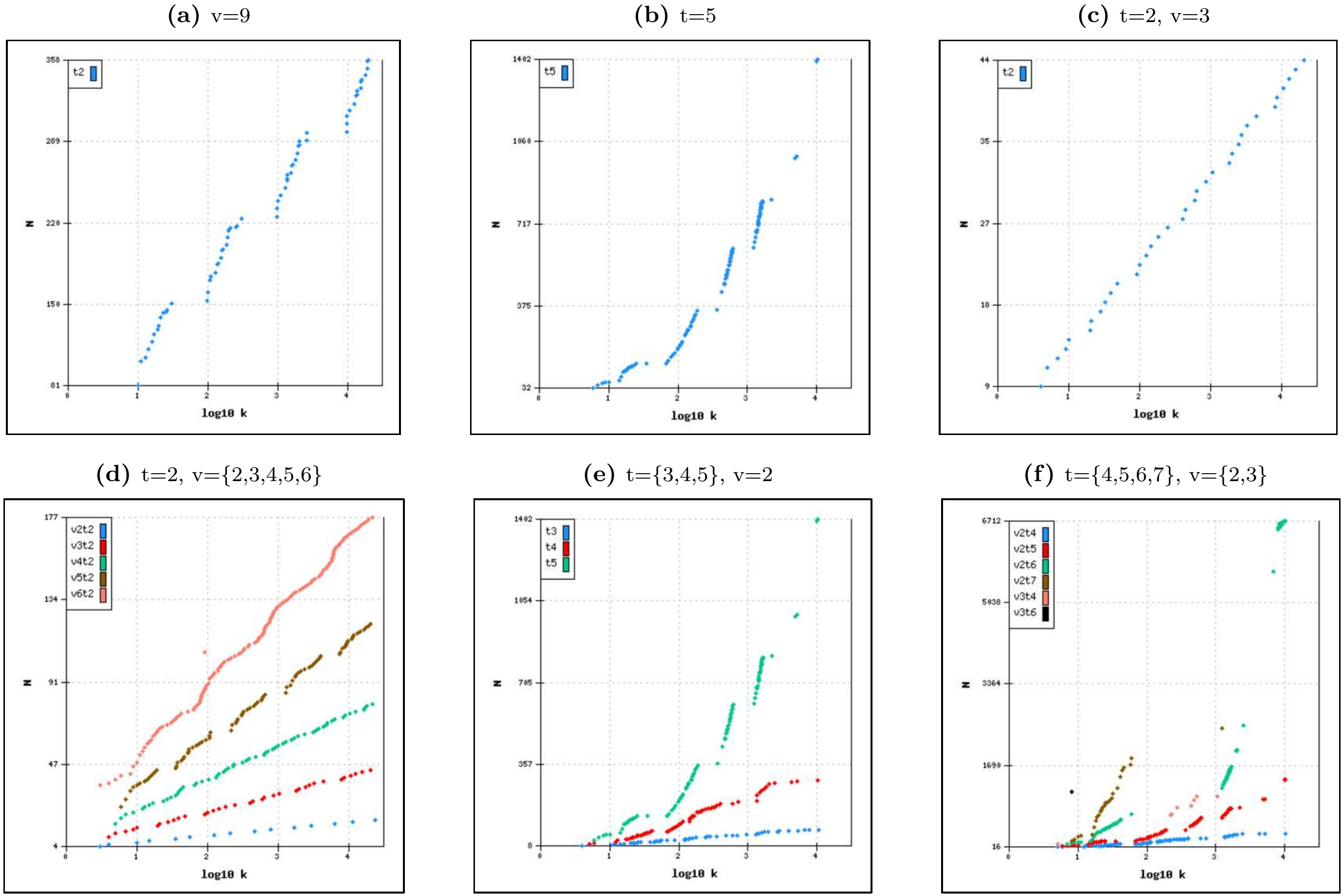

Figure A.2: A.1a Example using single v value. A.1b Example using single t value. A.1c Example using single t value and single v value. A.1d Example using single t value and multiple v values. A.1e Example using multiple t values and single v value. A.1f Example using multiple t values and multiple v values.

improve upper bounds of other covering arrays. For each approach, we present a summary of the evidence about the best upper bounds attained by it.

A.3.1 Algebraic constructions

The use of algebraic constructions to improve the upper bounds of covering arrays. The algebraic constructions are deterministic approaches that uses covering arrays to construct larger ones. Examples of such methods are the Hartman Style Raising Procedures (Hartman and Raskin, 2004) and the product of covering arrays (Colbourn et al., 2006a). This subsections discusses how such methods, combined with matrices of covering arrays as the ones found at our repository, can be used to improve existing upper bounds for covering arrays.

Our first case of study involves the product of covering arrays. Colbourn et al. (2006a) describes several methods that constructs large covering arrays from two relatively small covering arrays. One of them is the *Direct Product*, this method constructs a $CA(N+M;2,kl,v)$ from $CA_1(N;t,k,v)$, $CA_2(M;t,l,v)$. The other method, called $PCA \times PCA$, involves the product of special structures called Partitioned Covering Arrays (or PCAs), which can yield better results than the ones obtained with the direct product (i.e. the covering arrays produced would have less rows).

Table A.5 presents the number of covering arrays whose best upper bounds so far have been obtained using the algebraic methods of PCA $\times$ PCA, and the Direct product generalized. The column 1 shows the different covering arrays analyzed; column 2 presents the number of upper bound due to PCA $\times$ PCA; and column 3 shows the upper bounds derived from Direct product generalized.

Note that in the description presented in Table A.5, for some alphabets the number of upper bounds due to PCA $\times$ PCA and Direct product generalized are large. If we can have access to the small matrices that have produced those bounds, then the large matrices could also be constructed. This fact reflects the importance of a repository that keeps available matrices of covering arrays of size competitive with the best upper bounds known so far, as the one presented in this appendix.

A.3.2 Metaheuristics

However, the algebraic constructions are not the only ones that can be benefited from the repositories. Also, the construction of covering arrays by metaheuristics can be enhanced by the use of existing covering arrays when the latter ones are taken as initial solutions by the algorithms.

The method presented by Avila-George et al. (2012c) is an example of the use of covering arrays as initial solutions. This method has improved upper bounds for

Table A.5: Summary of the number of best upper bounds obtained using the methods *PCA* × *PCA* and *Direct product generalized*, in the construction of covering arrays. The information is presented for covering arrays of strength $t = 2$ and different alphabets $v = \{3, 4, \ldots, 24\}$.

	New Upper Bounds	
$CAN(t, k, v)$	$PCA \times PCA$	*Direct product generalized*
$CAN(2, k, 3)$	7	0
$CAN(2, k, 4)$	30	0
$CAN(2, k, 5)$	52	0
$CAN(2, k, 6)$	13	57
$CAN(2, k, 7)$	34	44
$CAN(2, k, 8)$	30	64
$CAN(2, k, 9)$	41	7
$CAN(2, k, 10)$	4	72
$CAN(2, k, 11)$	12	65
$CAN(2, k, 12)$	1	78
$CAN(2, k, 13)$	2	53
$CAN(2, k, 14)$	0	85
$CAN(2, k, 15)$	0	76
$CAN(2, k, 16)$	14	51
$CAN(2, k, 17)$	14	66
$CAN(2, k, 18)$	1	133
$CAN(2, k, 19)$	14	106
$CAN(2, k, 20)$	0	158
$CAN(2, k, 21)$	0	127
$CAN(2, k, 22)$	0	132
$CAN(2, k, 23)$	9	85
$CAN(2, k, 24)$	1	100

existing covering arrays. These kind of strategies have shown a good performance in the construction of covering arrays; it is so because a great variety of upper bounds have been obtained with them.

A.4 Conclusions

The main contributions presented in this appendix are listed in the following paragraphs.

This appendix presents a brief summary of the different strategies used for the construction of covering arrays. These strategies are grouped in exact, deterministic and non-deterministic approaches. It also presents a review of the available repositories of covering arrays and compares them in terms of strength t, number of columns k, alphabet v and in the availability of their matrices.

Through this appendix we describe a new repository that provides to the research community a great list of covering arrays, with a wide variety of strengths t and alphabets v. With this repository, others can use these arrays without having to spend the computational resources for constructing them and use them to construct new covering arrays. The main characteristic of this new repository is that it has a web interface for the management of the covering array matrices; also, these matrices are available under request at http://www.tamps.cinvestav.mx/~jtj/CA.php. Besides the easy to use interface, and for given values of t, k, v, the repository presents multidimensional graphs that describes the upper bounds in the size of covering arrays.

An important contribution of the repository described in this appendix is that it also contains matrices for Mixed Covering Arrays (MCAs), i.e., matrices for covering arrays with different alphabets in the columns. In the repository we can find matrices of covering arrays of strengths t, columns k and alphabets v of 17, 20000 and 27, respectively (i.e. it covers a wider ranges of values for t, k, v than the other repositories). For the case of mixed covering arrays, the values for t, k, v are $6, 75$ and 11, respectively.

We point out in a case of study, the benefits that can be achieved from the existence of the repository. Particularly, the repositories found applications in the construction of covering arrays as ingredients of algebraic methods, which are methods for the construction of covering arrays that uses smaller covering arrays as inputs. Another application of the repositories is in metaheuristics, where their matrices can be used as inputs of such methods to improve existing upper bounds.

Index