DESIGNING TEST SUITES FOR SOFTWARE INTERACTION TESTING

MYRA B. COHEN

A thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in Computer Science,
The University of Auckland, 2004

Abstract

Testing is an expensive but essential part of any software project. Having the right methods to detect faults is a primary factor for success in the software industry. Component based systems are problematic because they are prone to unexpected *interaction faults*, yet these may be left undetected by traditional testing techniques. In all but the smallest of systems, it is not possible to test every component interaction. One can use a reduced test suite that *guarantees* to include a defined subset of interactions instead.

A well studied combinatorial object, the *covering array*, can be used to achieve this goal. Constructing covering arrays for a specific software system is not always simple and the resulting object may not closely mirror the real test environment. Not only are new methods for building covering arrays needed, but new tools to support these are required as well. Our aim is to develop methods for building smaller test suites that provide stronger *interaction coverage*, while retaining the flexibility required in a practical test suite. We combine ideas from combinatorial design theory, computational search, statistical design of experiments and software engineering.

We begin with a description of a framework for greedy algorithms that has formed the basis for several published methods and a widely used commercial tool. We compare this with a meta-heuristic search algorithm, *simulated annealing*. The results suggest that simulated annealing is more effective at finding smaller test suites, and in some cases improves on combinatorial methods as well.

We then develop a mathematical model for *variable strength* interaction testing.

This allows us to balance the cost and the time of testing by targeting individual subsets of components. We present construction techniques using meta-heuristic search and provide the first known bounds for objects of this type.

We end by presenting some new cut-and-paste techniques that merge recursive combinatorial constructions with computational search via a process we term *augmented annealing*. This method leverages the computational efficiency and optimality of size obtained through combinatorial constructions while benefiting from the generality of a meta-heuristic search. We present examples of specific constructions and provide new bounds for strength three covering arrays. The results presented provide the foundations for an interaction testing toolkit.

Acknowledgements

This thesis, although attributed to one person, is the end result of collaboration with many people. I would like to extend a thank you to all of them.

I am indebted to my two supervisors Peter Gibbons and Rick Mugridge without whom this research and thesis would not exist. They have been a continual source of support and encouragement and have provided me with insights from differing points of view. Their teamwork has been very important to me. I particularly appreciate the prompt feedback when deadlines were quickly approaching. I would like to thank Peter Gibbons for his meticulous mathematical approach, for sharing his knowledge about computational search, for his careful reading and detailed comments on all of my writing, and for introducing me to the Ironman. I would like to thank Rick Mugridge for his continual re-factoring, for making me think from different angles, for his high standards and for his ability to provide a software engineering perspective to our work.

I am very grateful to Charlie Colbourn who has been integral in guiding my research and academic career. He has contributed in many ways to this thesis. He has taught me to appreciate quality research, has believed in my abilities, has offered me many exciting career opportunities, has pushed me to attain high standards and has acted like a third supervisor. He has taught me the meaning of the word mentor.

I thank the many co-authors whose discussions and contributions have enriched the content of this thesis. They are Peter Gibbons, Rick Mugridge, Charlie Colbourn, Alan Ling, James Collofello, Renee Turban, Adam Porter and Cemal Yilmaz. I would also like to thank the anonymous referees who have given us feedback on our research along the way and to thank my external examiners who provided thoughtful and detailed comments resulting in many improvements to the final version.

I would like to thank the Computer Science Department at the University of Auckland for supporting me during this process, especially John Hosking who has been HOD since my arrival.

Thanks to G. Dobbie for being a role model and a good friend. I thank her for all of her encouragement and support, careful proof-reading and comments and for the many miles run together. I am grateful to Violet Syrotiuk for her hospitality, her advice and for telling me to "just do it".

Others who have provided encouragement, offered feedback and have supported me along the way include Sanjoy Baruah, Miriam Walker, Jennifer Lennon, Ann Cameron, Jing Sun, Ute Lörch, Robert Berks, Peter Dance, Tim and Elizabeth Budd, Susie, Sarah, the lady with the umbrella and the guys from St. Johns. I would also like to thank my family for providing me with the necessary skills to achieve this goal.

Lastly I would like to thank Mark Wirzman who has been a constant source of support throughout this process. This thesis would not have been possible without him.

The research in this thesis was supported in part by the Consortium for Embedded and Internetworking Technologies and by ARO grant DAAD 19-1-01-0406.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Testing can account for 30 to 50 percent of software development costs [5, 13]. But the cost of failures is even higher. In 2002, the National Institute for Standards in the United States, explored the financial impact of an inadequate software testing infrastructure. Failures in mission critical software were not considered because it was impossible to assign dollar figures to lost lives. Even with the absence of this factor it was determined that the *annual* cost of insufficient software testing methods and tools in the United States is between 22.2 to 59.5 billion US dollars [13, 14]. This study highlights the need for more efficient and more effective methods of software testing. Given the magnitude of the problem, even small advances in testing can equate to large cost savings.

In "Testing: A Roadmap" [50] M.J. Harrold points out that software quality will become the main criteria for success in the software industry. She refers to software testing as the critical element in software quality. She also highlights an important issue, the gap in *technology transfer.* This is the long lag time that exists for software research techniques to move into the industrial sector. This gap is an important issue that must be addressed. Development of tools to support current research enables the faster transfer of software testing advances into practice.

These issues provide strong motivation for new innovations in software testing, and for the development of tools that support them. In essence they provide an excellent guideline for effective research in this field. This thesis embraces these ideas as it explores one method of software testing, *interaction testing*. It examines current methods of building interaction test suites and presents new techniques to generate stronger test suites that are more cost-effective to run. It then lays the groundwork for extending these ideas to a practical test environment and for the development of new tools that will exploit the results. The ultimate aim of the research ideas presented here is the development of a comprehensive toolkit for software interaction testing.

1.1 Motivation

A major initiative in software engineering is component based development. Component based development allows one to build systems incrementally using previously developed modules or parts. It supports and encourages re-use. As pre-built and pre-tested modules are integrated to create a new system, new code is added and the software is customized for the situation at hand.

Components that are re-used include individual software modules, entire software libraries or hardware parts such as memory and controllers. In fact, many systems today combine both hardware and software elements causing the term "component" to have a liberal and varied interpretation.

1.1.1 An Example Component Based System

In this section we present an example component based system that will be used throughout the rest of this thesis to illustrate ideas and concepts. This example is based on an advertisement for a real product, although we have simplified it for our purposes.

Company X builds integrated RAID controllers for the PC market. The controllers are made from pre-built, pre-tested hardware components. The company develops software to run the controller hardware interfaces and to configure the

	COMPONENT			
	RAID Level	Operating System	Memory Config	Disk Interface
VALUE	RAID 0	Windows XP	64 MB	Ultra 320-SCSI
	RAID 1	Linux	128 MB	Ultra 160-SCSI
	RAID 5	Novell Netware 6.x	256 MB	Ultra 160-SATA

Table 1.1: RAID integrated controller system: 4 components, each with 3 values

RAID system. This is provided when the controller is sold to the customer. The software is written to work on a variety of commercial operating systems and to support several different hard disk interfaces. To make the product scale, the company allows the system to be sold with differing amounts of memory. When the company sells this controller, the intention is to support many different system configurations so that the user can customize the system to fit their environment. For instance the user may be running Windows XP and want to use RAID Level 5 with Ultra 320 SCSI hard disks. Another user may be running Linux and want to use RAID Level 1, but may use the same type of hard disk.

Table 1.1 shows a simple system of this type[1]. This system consists of four components (RAID level, operating system, memory configuration and hard disk interface). Each of these has three supported values. The system user may select RAID Level 0,1 or 5. They can run this controller software on Windows XP, Linux or Novell Netware 6.x. In addition, they can purchase the system with one of three levels of memory and can use one of three different hard disk interfaces. Company X most likely builds many types of RAID controllers and supplies accompanying software to support a variety of large and small systems. If they use component based development then they will re-use the components in each of the controllers. And many of the components will be used for other types of controllers as well.

In the example given, there are $3^4 = 81$ possible system configurations. Of

[1]The real system supports at least five levels of RAID and at least five or six operating systems as well as a variety of other components not mentioned here.

course, most real systems will be more complex than this. Suppose there are 10 components and each of these has five different supported values. The number of system configurations has grown to $5^{10} = 9765620$.

1.1.2 Development Using Components

One advantage of component based software development is that components are built and tested once, after which they can be used repeatedly as building blocks. Given the high cost of initial design, development and testing, this model has the potential to save on overall system development costs. Consequently, many organizations are choosing a component based approach for both development and acquisition. The United States military, for instance, has recently moved toward using Commercial-Off-the-Shelf components (COTS), for both its software and hardware purchases [42]. However component based development brings new challenges that must be addressed, especially in the area of software testing [9, 100].

Re-use is where the savings occur, but when one re-uses components there is still a considerable amount of testing to be done. In the component example given, each hardware component has been individually tested. The software module controlling RAID level has been tested for each level of RAID. The operating systems are not built by this company, but will be hosting the software. They have been fully tested elsewhere, but it is unlikely that they were built with the knowledge that this particular RAID controller software would eventually run on their system. This system, as we have seen, has 81 possible combinations of settings. Before company X releases the controller it needs to test the software running it, in all of these possible states. Otherwise unexpected results may occur. What happens when we have 10 components with 5 values each? Testing all states is no longer possible, therefore decisions must be made about what *can* be tested given the time and money allocated. This is a central theme for the research that follows.

1.2 Software Testing: An Overview

Testing is pervasive throughout the software development life cycle. In all but the smallest systems, however, it is impossible to test all possible inputs. Therefore testing must be done efficiently and systematically to optimize its effectiveness within the given time and budget constraints [5]. There are many ways to look at the testing process. Essentially however, all testing returns to the system functionality or *system requirements*. When software is built, it is built to perform to a set of specifications. This is the minimum standard that must be satisfied for system testing. Testing to ensure that all system requirements are met is fundamental to software development.

In the broadest sense, testing may be viewed as structural (or *white box*) or it may be viewed as functional (or *black box*). White box tests are designed using knowledge of the data structures and algorithms within the program, while black box testing uses only system specifications. The testing process proceeds from testing individual modules, through to integration and system testing. As individual modules are built *unit testing* occurs on these. Unit testing is done primarily by developers. At this point a combination of white and black box testing can occur. As the system grows, tests are conducted as the modules are combined together. This type of testing may be done by testing teams, and may no longer involve the original developers. This is called *integration testing*. Testing of the entire integrated system is called *system testing*. Integration and system testing are commonly restricted to black box testing. They rely heavily on documentation of system specifications. The challenge in component based testing is the *integration* and *system* test [100], because individual components have already been tested.

Testing at the integration and system level is dependent on the quality of the unit tests and on the complexity of interactions that occur between components. Usually the source code for software components is unavailable. Therefore testing must be functional. One implication is that it is harder to use standard metrics to determine how complex a system is, and to test the thoroughness of individual test suites.

One issue that arises in development of components at the unit level is that they are tested without any *a priori* knowledge of their final *operational uses* [100]. Since exhaustive testing is impossible, one common method that focuses testing where it is most useful is to base it on the operational profile. This examines the module at hand and classifies sections of the code based on the percentage of time it is used under expected system use. The idea is to concentrate testing to uncover faults in areas of the code base that are used most often. The result of this is that less testing may occur in areas that are used rarely in initial development. This may not be a problem when the final operational uses are known ahead of time, but in component based development this is not possible, and therefore constitutes a risk.

When components are moved to a new operational environment, the component interactions change, i.e. different areas of code are likely to be exercised. Previously untested code may suddenly be heavily used and undetected faults in this area are likely to be exposed [100], leading to unexpected outcomes in component behavior. System testing should prevent this from happening, but as the number of components in a system grows, so does the number of possible *interactions* between them. And as the software complexity increases these interactions and uses may become more complex. This leaves component based systems prone to unexpected interaction faults.

1.3 Software Testing: Code Coverage

To quantify the quality of testing, *code coverage* metrics have been defined. Code coverage quantifies the amount of underlying program code that is executed by a set of tests. It is usually presented as a percentage where 100% indicates complete coverage. Higher code coverage is often used to indicate more thorough testing [5]. We examine a few specific metrics here; ones that have been used to equate the effectiveness of the interaction testing methods examined in the rest of this thesis.

For a more complete discussion of code coverage metrics see [5].

Some code coverage metrics are based on a program's control flow. A *process block* is defined as a sequence of program statements that contain no decisions or junctions [5]. The simplest or weakest metric of code coverage is *statement* or *block* coverage. This metric guarantees that each statement or process block has been tested at least once. This is usually the minimum acceptable level of code coverage [5]. *Decision coverage* indicates that all branches in the program's control flow have been tested at least once [5]. It is a stronger measure than block coverage.

Other types of code coverage are based on the data flow of a program. The data flow of a program graphs the paths between objects or variables from their definitions to their uses. This is often called a *du* or *definition-use*. Two basic types of uses can be defined. A *c-use* is a computational use of an object. This occurs when an object is used on the right hand side of an assignment statement. A *p-use* is a predicate use of that object. This occurs when the object is used in a boolean evaluation to determine the output of a branch. An assignment statement is a re-definition. The phrases "*all c-uses*" and "*all p-uses*" mean that all paths from definitions to c-uses or definitions to p-uses, respectively are tested [5, 55]. A stronger metric *all-uses* includes all c-uses and all p-uses.

1.4 Interaction Testing

In an ideal test environment, we need to do more than just test that individual system requirements are met. We must also ensure that our products do not fall prey to interaction faults. These are unexpected interactions between components. Examples are improper data validation, incorrect assumptions about values that will be returned and failure to pass values needed by secondary structures. If these assumptions are not properly documented then they may expose errors due to faulty initializations or data overflow.

Interaction testing is a form of functional testing. Although studies have used metrics to examine how well underlying code is covered, this type of testing is done independently from the control flow and data flow graphs of the program. It does not replace other methods, rather it complements them. Our focus of software testing is this type of test, the *interaction test.*

Returning to the RAID example, there are four components, each with three different values, resulting in 81 possible system configurations. Each of the system tests must be run in each of these 81 configurations in order to detect any unexpected interaction faults that will occur between components. But testing of all system configurations is usually impossible. For instance when we extended our system to have 10 components with 5 values it gave us 9765620 configurations which is clearly infeasible to test.

When this is the case, the goal is to generate test suites that test (*cover*) as many interactions as possible given time and cost constraints. This can be done randomly or one can select a particular set of rules that helps to choose the best possible *interaction coverage.*

Interaction testing has been presented as a problem of component based development. In the example component based system *interactions* are viewed as interactions among *values* of components in a system, but the ideas are analogous to those of a discrete set of functional inputs to a system.

There are many places where interaction testing occurs [8, 10, 19, 23, 24, 37, 40, 68, 99, 101, 108]. It can be used for example to examine all of the inputs to a system, or at a white box level as variable inputs. It can also be used to test GUI event interactions. In interaction testing one examines the *interaction coverage*, i.e. the number of interactions in the system that have been tested.

1.5 Statistical Design of Experiments

When testing all combinations is not possible, a method for selecting a subset of these is needed. One method of selecting a systematic and repeatable set of test cases for interaction testing is derived from statistical methods. The area of Design of Experiments (*DOE*) was introduced by Sir R. A. Fisher in the 1920's [85]. His goal was to determine optimum water, sunshine, fertilizer and soil conditions for producing the best crops.

Fisher called each of these four elements a *factor* in the system. Each value of water, sunshine, etc. was considered a *treatment*. By testing all combinations of these factors, he showed that one could determine the effect on the *response variable* (in this case, the crop output). Fisher developed methods, called *fractional factorial designs*, to reduce the size of the experiments when all combinations of tests was prohibitive in size [85, 97]. This is analogous to our software testing system when the number of components and values makes testing all combinations impossible. We can use the same concepts to create a reduced set of system configurations for testing our software systems; one that guarantees a set of rules are upheld [85, 97]. For instance one can decide to test all *pairs* of interactions between factors. Using these methods we can quantify and repeat which interactions in our system are tested.

Traditionally this approach uses continuous values for the response variable, such as those in Fisher's original crop experiments. In software testing we use discrete binary valued outputs (i.e. pass or fail) for response variables since one is only looking to find test failures. The DOE method has been adapted slightly in this fashion for use in software testing.

Returning to the example RAID system, suppose it is not possible to test all 81 interactions. One can instead decide to test all *pairs* or *triples* of interactions. Although this does not completely test all interactions, empirical evidence shows that this may still provide good error detection and code coverage [10, 23, 39, 44]. The advantage of this method is that these are repeatable and systematic tests. We can quantify what has and has not been tested and we can repeat our tests as modules are changed. All possible two way interactions for this system can be covered with only nine system configurations and all three way interactions with 27

System Configuration	RAID Level	Operating System	Memory Config	Disk Interface
1	RAID 5	Novell	128 MB	Ultra 160-SATA
2	RAID 5	Novell	64 MB	Ultra 320
3	RAID 1	Novell	256 MB	Ultra 320
4	RAID 1	Windows XP	128 MB	Ultra 320
5	RAID 5	Linux	256 MB	Ultra 160-SATA
6	RAID 1	Novell	128 MB	Ultra 160-SCSI
7	RAID 0	Linux	64 MB	Ultra 160-SATA
8	RAID 0	Windows XP	128 MB	Ultra 160-SATA
9	RAID 1	Linux	128 MB	Ultra 160-SATA
10	RAID 0	Novell	128 MB	Ultra 320
11	RAID 5	Linux	64 MB	Ultra 160-SCSI
12	RAID 5	Linux	128 MB	Ultra 320
13	RAID 0	Novell	64 MB	Ultra 160-SCSI
14	RAID 1	Windows XP	256 MB	Ultra 160-SATA
15	RAID 1	Linux	64 MB	Ultra 320
16	RAID 5	Novell	256 MB	Ultra 160-SCSI
17	RAID 1	Linux	256 MB	Ultra 160-SCSI
18	RAID 5	Windows XP	256 MB	Ultra 320
19	RAID 5	Windows XP	64 MB	Ultra 160-SATA
20	RAID 0	Novell	256 MB	Ultra 160-SATA
21	RAID 0	Windows XP	256 MB	Ultra 160-SCSI
22	RAID 0	Linux	128 MB	Ultra 160-SCSI
23	RAID 1	Windows XP	64 MB	Ultra 160-SCSI
24	RAID 5	Windows XP	128 MB	Ultra 160-SCSI
25	RAID 0	Windows XP	64 MB	Ultra 320
26	RAID 1	Novell	64 MB	Ultra 160-SATA
27	RAID 0	Linux	256 MB	Ultra 320

Table 1.2: Test suite covering all 3-way interactions for Table 1.1

system configurations. Table 1.2 is an example of a reduced test suite for Table 1.1. Each of the 27 test configurations in this table has 4 components with one value from each component selected. The first test configuration, (RAID 5, Novell, 128 MB, Ultra 160-SATA), covers six *pairs* of interactions (RAID 5 with Novell, RAID 5 with 128 MB of memory, RAID 5 with an Ultra 160-SATA disk interface, Novell with 128 MB of memory, Novell with an Ultra 160-SATA interface, and 128 MB of memory with an Ultra 160-SATA interface) or four *triples* of interactions (RAID 5 and Novell with 128 MB, RAID 5 and Novell with Ultra 160-SATA, RAID 5 and 128 MB with Ultra 160-SATA, and Novell and 128 MB with Ultra 160-SATA).

DOE methods were first used by Mandl [68] in software systems to test combinations of values in a compiler program. Taguchi popularized this method in industrial testing and called it *Robust Design* [80, 97]. Brownlie *et al.* adapted this method to testing input interactions for testing an internal email system at AT&T. D. Cohen *et al.* [22, 23, 24, 25, 40] and Dalal *et al.* [39] used these concepts to build a commercial software test generator that is available today from Telcordia, Inc. Moreover, DOE methods have recently been proposed as part of the standard toolkit for software testers [58] and are included in the Six Sigma methodology for testing quality [7]. This is strong evidence that this technique for software interaction testing is moving into the broader software testing community [7, 58].

In the course of this work we have tried to quantify sizes of “real” problems from industry but this is often domain specific or proprietary in nature. One such example [33] indicates that the company has approximately 50 binary valued components that can be configured in one of their software products. Running a full set of test suites for just one configuration involves approximately 4000 man-hours of time. In this environment only specific configurations are currently tested and certified. In Section 2.2.5 we will examine some more examples of real systems.

Design of experiments benefits from a wide body of literature describing several mathematical objects, or *combinatorial designs*, that have the properties needed for these experiments. For instance if one wants to test all two-way or n-way interactions in a system, then it may be possible to map the problem to a known mathematical object, called an *orthogonal array.* This has been exploited by the software testing community. Unfortunately, many of the results on combinatorial designs have been developed from a mathematical point of view. They may be limited to a subset of situations that can arise in testing or may not be flexible enough for use in practical software testing environments where additional constraints occur. Additionally, the ability to build these objects is not always straight forward thereby creating a longer technology transfer gap.

1.6 Research Aims

This brings us to the focus of this thesis. At the current time there are two distinct areas of active research on combinatorial designs for software testing. The mathematics community is focusing on building smaller designs of higher interaction strength [17, 18, 89, 95, 96]. Interaction strength determines how many system interactions are tested. The goal of much of this research is producing mathematically optimal objects without a concern for accessibility or generality. The software testing community is focusing on greedy search algorithms to build these in a more flexible environment, one that more closely matches real testing needs [23, 25, 39, 44, 98, 102, 109, 110]. In addition, more powerful search techniques such as simulated annealing have been employed recently [91, 92], but quite often these results are mathematically sub-optimal.

Ideally we would like to combine these ideas and build higher strength interaction tests that are minimal and efficient and easy to generate. This thesis aims to move the research in the two disciplines closer together. In doing so, it attempts to bridge the gap between software engineering research and combinatorial design research. The mathematical models explored move closer to that of a real software testing problem and the solutions provided are closer to mathematically optimal ones.

There are many areas that deserve further research in examining software interaction testing, including empirical studies, the development of testing toolkits and determining the best way to model and map the software test as an interaction problem. Our focus lies in supplying the foundations for the second problem, that of creating a useful toolkit to build efficient interaction test suites. The methods of building interaction test suites are varied. Some are mathematically optimal, but require in-depth mathematical knowledge and are inflexible when real system constraints are added. Some are computationally very efficient, but may produce larger test suites, or may not adapt well to a real software test environment. Some provide flexibility and robustness for testing, but produce overly large test suites. A trade-off must occur between the computational power, the ability to model a real test environment and the cost of running the final test suites.

1.7 Overview of Thesis

The main contributions of this thesis lie in three primary areas; meta-heuristic search, software engineering and combinatorial mathematics. In meta-heuristic search we have developed a guided search technique, *augmented annealing.* This combines computational search with mathematical constructions. In software engineering we have defined a model for software interaction testing that allows us to weigh the cost and level of testing required to produce the best coverage where it can be most beneficial. We call this the *variable strength covering array.* We have made an initial investigation into its use for modeling testing problems. In combinatorial mathematics the main contribution is the development of new constructions and bounds for covering arrays of strength three. Beyond that this thesis attempts to integrate several disciplines to enhance our techniques of software integration testing. In the course of this work we have discovered new upper bounds for 21 covering arrays, provided the first known bounds for 16 variable strength arrays and have presented over 80 bounds for two other objects used in constructions of covering arrays, two-one covering arrays and difference covering arrays without zero differences. Although we focus this thesis on its applicability to software testing, the techniques and methods developed may be useful in a broader context [15, 19, 61, 86, 108].

The rest of this thesis is structured as follows. Chapter 2 begins with a description of the mathematical objects underlying our research. It presents some general results for constructing these mathematically and explores the related work in applying these to software testing. Chapter 3 examines computational methods for building interaction test suites. We explore known algorithms and search techniques as well as describe some useful data structures. Chapter 4 compares implementations of several computational methods. Chapter 5 examines a new model for software interaction testing. It presents both computational methods and results for variable strength covering arrays. Chapter 6 describes some new mathematical constructions that leverage computational search and develops the idea of augmented annealing. Chapter 7 presents our conclusions and future work.

Chapter 2

Background

Mathematical models have been used for statistical design of experiments in many disciplines [15, 54]. Some of these include manufacturing, testing of chemical interactions and testing of pharmaceuticals. Recent work extends these ideas to testing inputs into a biological system [86].

The literature in this area is broad and varied. We use this chapter to bring together mathematical results that can be used for software interaction testing, and use this to lay the mathematical foundation for the rest of the thesis. We begin with some known combinatorial objects and show how these can be mapped to represent interaction test suites. We present a sample of known results and conclude with a discussion of how some of these have been applied to real software testing environments.

2.1 Definitions and Examples

Returning to the example RAID integrated controller system, one can define a *test suite* as follows. Each test suite is an $N \times k$ array. It has N *test configurations.* A test configuration is one combination of the k component values, e.g. RAID 0, Windows XP, 128 MB memory and Ultra 320-SCSI. In Table 1.2 there are 27 test configurations. At this stage we shall assume that each component can take on v

values. The components in this system are *RAID level*, *operating system*, *memory configuration* and *disk interface*, with each component having three possible values. For example the component *operating system* can take on the values XP, Linux and Novell. The rest of this thesis will use the terms *component* and *value* for consistency although the terms *factor* and *level* are often used in the literature for *component* and *value* respectively.

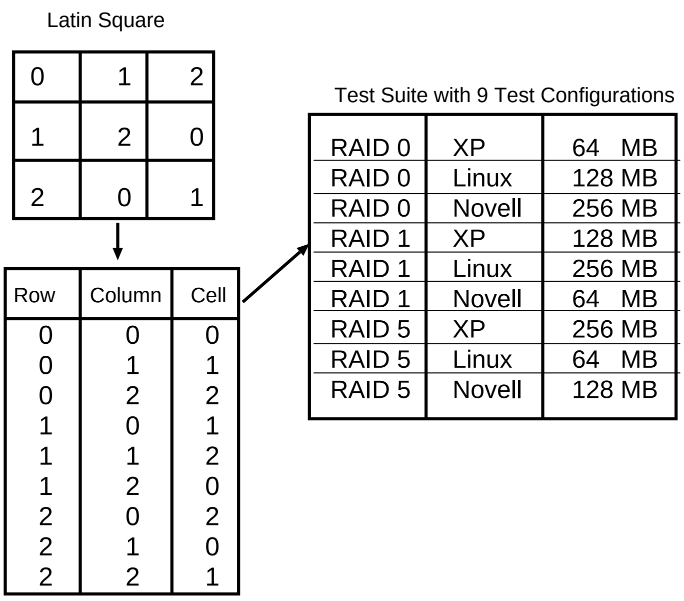
This test suite provides us with a set of independent system configurations. We can add constraints to this test suite. In our system we will require that all t-way combinations of component values occur (i.e. for each set of t components every t-tuple of component values is represented). We call t the *strength* of the test suite. In this example, each test configuration represents a single setting of the system. Therefore multiple tests cases will be run for each of the designated test configurations. It is also possible to use this model without loss of generality to represent individual *inputs* to a system. In this instance each test configuration represents an individual test case and is run as-is (i.e. we have a one-to-one relationship between test case and tests run). We will use this model of a system test throughout the rest of this thesis.

2.1.1 Orthogonal Latin Squares

Definition 2.1.1 *A latin square of order s is an $s \times s$ array with entries from a set S of cardinality s with the condition that for all i in S, i appears exactly once in each row and each column of the array. Two latin squares are orthogonal if, when superimposed on each other, the ordered pairs created in each cell cover all s^2 combinations of symbols[1, 54].*

Definition 2.1.2 *A set of* mutually orthogonal latin squares *or* MOLS *has the property that the squares in the set are pairwise orthogonal [54]. A MOLS(s, w)* [1] *is a set of w latin squares of order s in which any pair are orthogonal.*

[1] *This is often written as w MOLS(s).*



0	1	2
1	2	0
2	0	1

Row	Column	Cell
0	0	0
0	1	1
0	2	2
1	0	1
1	1	2
1	2	0
2	0	2
2	1	0
2	2	1

RAID 0	XP	64 MB
RAID 0	Linux	128 MB
RAID 0	Novell	256 MB
RAID 1	XP	128 MB
RAID 1	Linux	256 MB
RAID 1	Novell	64 MB
RAID 5	XP	256 MB
RAID 5	Linux	64 MB
RAID 5	Novell	128 MB

Figure 2.1: A latin square of order 3 used to define a test suite

Table 2.1 is a MOLS$(3, 2)$. Mandl [68] first proposed using latin squares and mutually orthogonal latin squares for testing compiler software. A latin square of order s can test all pairs of interactions in a system with three components, each with s values. To do this, map the values of two components to the symbols $(0, 1, ..., s-1)$. Map the values of the third component to the s symbols found in the cells of the latin square. Using these mappings, enumerate all of s^2 column and row locations using the first two components. The third component takes the value of the corresponding cell entry.

Figure 2.1 shows how one can use a latin square to set up a test suite for the first three components of the RAID example. We have chosen *RAID Level* as the row index. RAID 0 maps to 0, RAID 1 maps to 1 and RAID 5 maps to 2. *Operating system* is the column index. Windows XP maps to 0, Linux maps to 1 and Novell maps to 2. *Memory configuration* is the cell entry. 64 MB maps to 0, 128 MB maps to 1 and 256 MB maps to 2. A set of test configurations for pairwise coverage is shown.

In order to use MOLS for software testing of pairs of interactions, k orthogonal latin squares of size s are needed to test $k + 2$ components each with s values.

0	1	2
1	2	0
2	0	1

0	1	2
2	0	1
1	2	0

Table 2.1: Two mutually orthogonal latin squares of order 3

The cell entries represent k components in the system, and the column and row indices represent the remaining two components. As was the case with the previous example, two components are mapped to the column and row indices. The rest of the components are mapped to the symbols found in the cells of the latin squares. Table 2.1 can be translated into a test suite for the entire RAID system example in Table 1.1. Each test configuration is a 4-tuple of type (row,column,cell entry of 1st latin square, cell entry of 2nd latin square).

First designate each of the levels of each component as an integer between 0 and 2. We have chosen the following mapping: RAID 0 = 0, RAID 1 = 1,RAID 5 = 2, Windows XP=0, Linux=1, Novell=2, 64 MB=0, 128 MB =1, 256 MB =2, Ultra 320=0, Ultra 160-SCSI=1 and Ultra 160-SATA=2. Next enumerate all of the column and row locations for the latin squares. Use *RAID level* as the row and *operating system* as the column index. Map the other components, *memory configuration* and *disk interface* to the cell entries obtained when the latin squares are superimposed (see Table 2.2). This gives us 9 test configurations which cover all pairs of interactions in the RAID system (see Table 2.3). The first test configuration is row 0, column 0 (RAID 0, Windows XP). The cell entry for the array is (0,0) which is 64 MB and Ultra 320.

For any set of MOLS(s, w), $w \leq s-1$ [54]. A MOLS$(s, s-1)$ exists when s is a prime power [54]. No mutually orthogonal latin squares exist when $s = 2, 6$. For all other values of s there is *at least* one set of pairwise orthogonal latin squares [54]. For all $s > 2$ except $3, 6, 10$ a MOLS$(s, 3)$ exists, but there is limited knowledge about the largest w for which MOLS(s, w) exists when s is not a prime power [54]. The limited existence of mutually orthogonal latin squares constrains our ability to

(0,0)	(1,1)	(2,2)
(1,2)	(2,0)	(0,1)
(2,1)	(0,2)	(1,0)

Table 2.2: Superimposed MOLS from Table 2.1

RAID Level (row)	Operating System (col)	Memory Config (latin Sq 1)	Disk Interface (latin Sq 2)
RAID 0	XP	64 MB	Ultra 320
RAID 0	Linux	128 MB	Ultra 160-SCSI
RAID 0	Novell	256 MB	Ultra 160-SATA
RAID 1	XP	128 MB	Ultra 160-SATA
RAID 1	Linux	256 MB	Ultra 320
RAID 1	Novell	64 MB	Ultra 160-SCSI
RAID 5	XP	256 MB	Ultra 160-SCSI
RAID 5	Linux	64 MB	Ultra 160-SATA
RAID 5	Novell	128 MB	Ultra 320

Mappings
RAID 0 = 0
RAID 1 = 1
RAID 5 = 2
Windows XP =0
Linux =1
Novell =2
Ultra 320 = 0
Ultra 160-SCSI =1
Ultra 160 SATA =2

Table 2.3: Pairwise test suite derived from Table 2.2

use these for the general software testing problem. Therefore we need to find other objects that have the desired property.

2.1.2 Orthogonal Arrays

Definition 2.1.3 *An orthogonal array $OA_\lambda(N; t, k, v)$ is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all ordered subsets of size t from v symbols* exactly λ *times [54].*

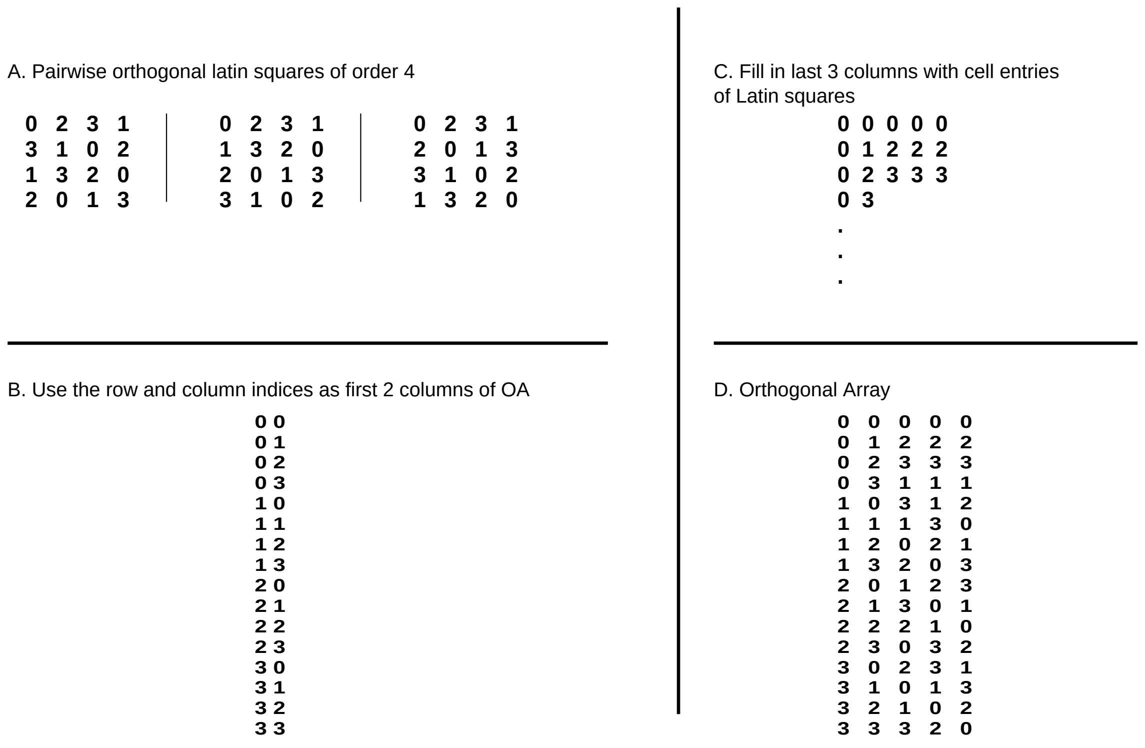
As $\lambda = \frac{N}{v^t}$ in an orthogonal array we often leave out the parameter N, denoting the array by $OA_\lambda(t, k, v)$. λ is often called the *index* of the array. When we drop λ from our notation, it is assumed that is has the value one. Table 2.4 is an example of an $OA(2, 4, 3)$.

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

Table 2.4: Orthogonal array $OA_1(2,4,3)$

An $OA(2, k+2, s)$ is equivalent to k orthogonal latin squares of order s. When a $MOLS(k, s)$ exists, we can transform k mutually orthogonal latin squares of order s into an orthogonal array $OA(s^2; k+2, s, 2)$ [1]. Figure 2.2 shows this translation. We use the column and row indices of the latin square as the first 2 columns. Then we add a column to the orthogonal array for each of the k latin squares. This column will contain the value found in the cell of the corresponding latin square indexed by the first two columns. To use this for software testing one defines a mapping from the components to the symbols.

There are various other mathematical constructions for orthogonal arrays. Therefore we are not limited by the existence of specific $MOLS$. However, they still do not exist for all combinations of the parameters t, k and v. The use of orthogonal arrays for testing has been discussed in the literature [8, 102], but these may be of less interest in interaction testing than some other objects because they can lead to overly large test suites with $\lambda > 1$. If there is no orthogonal array of index 1 then one approach used is to find the smallest value for λ where one exists. For software testing we are primarily concerned with the situation when $\lambda = 1$, i.e. everything is tested *once*. In situations where an orthogonal array with $\lambda = 1$ does exist, clearly this is the optimal test suite. However, there are many values of t, k and v for which an orthogonal array with $\lambda = 1$ does not exist so we must resort to a less restrictive structure; one that requires that subsets are instead covered *at least* once. We present this object next.



A. Pairwise orthogonal latin squares of order 4

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

C. Fill in last 3 columns with cell entries of Latin squares

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

B. Use the row and column indices as first 2 columns of OA

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

D. Orthogonal Array

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

Figure 2.2: Translating 3 orthogonal latin squares of order 4 into an $OA(2, 5, 4)$

2.1.3 Covering Arrays

Definition 2.1.4 *A* covering array*, $CA_\lambda(N; t, k, v)$, is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all ordered subsets from v symbols of size t* at least *λ times.*

When $\lambda = 1$ we use the notation $CA(N; t, k, v)$. In such an array, t is called the *strength*, k the *degree* and v the *order*.

Figure 2.3 is an example of a $CA(5; 2, 4, 2)$. Since we are interested in examining interactions between components, the symbol mappings for covering arrays are always arbitrary. As long as we consistently use the same symbol set for each component, we maintain the desired properties.

There are often a number of different ways to represent the same combinatorial object. Several other combinatorial objects have been defined with the same effective properties as a covering array. A *strength t transversal cover*, a *qualitatively*

Covering Array

1	2	5	6
1	3	4	7
0	2	5	7
0	3	5	6
0	2	4	6

Component 1: {0,1}
Component 2: {2,3}
Component 3 :{4,5}
Component 4: {6,7}

Figure 2.3: $CA(5; 2, 4, 2)$

independent system and a *t-surjective array* [25, 89, 96] are three such objects. In the rest of this document we will use the covering array definition for consistency.

D. Cohen *et al.* [23, 24] have developed the commercial product *Automatic Efficient Test Generator* (AETG) to construct covering arrays for software interaction testing. Williams *et al.* [102, 104] use orthogonal arrays as well as covering arrays to design tests for the interactions of nodes in a network. Stevens *et al.* [94] suggest creating a knowledge system for the tester that contains the best known covering arrays applicable to testing.

Covering arrays only suit the needs of software testers when all components have the same number of values. However, this is often not the case. The following scenarios point to a more realistic test environment:

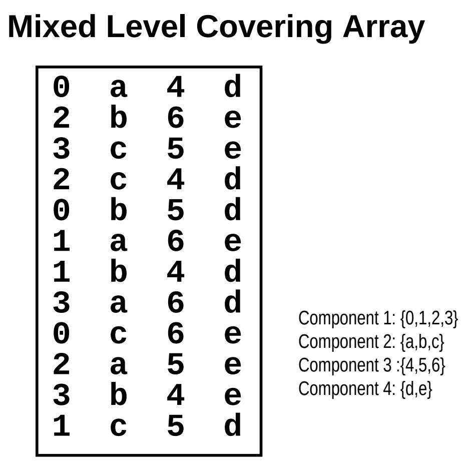
1. Components do not all have an equal number of values. For instance, we may have 6 levels of RAID, 4 operating systems, 3 memory configurations and only 2 hard disk interfaces.

2. Certain value combinations can never occur. These are called *avoids.*

3. There are aggregate conditions among several components.

4. A set of fixed test configurations are added to each test suite, regardless of the interaction strength required. These are called *seeded* test configurations.

These issues cause real testing environments to deviate from the covering array



Mixed Level Covering Array

0	a	4	d
2	b	6	e
3	c	5	e
2	c	4	d
0	b	5	d
1	a	6	e
1	b	4	d
3	a	6	d
0	c	6	e
2	a	5	e
3	b	4	e
1	c	5	d

Component 1: {0,1,2,3}
Component 2: {a,b,c}
Component 3 :{4,5,6}
Component 4: {d,e}

Figure 2.4: $MCA(12; 2, 4^1 3^2 2^1)$

definition. Here we restrict our subsequent discussion to the first scenario, which we feel is the most important deviation from a *fixed level* covering array; however we address some of these other issues later on.

2.1.4 Mixed Level Covering Arrays

When the the number of component values varies this can be handled with the *mixed level covering array.* Several authors have suggested its use for software testing (see [17, 91, 105]), but few results are known about upper bounds and how to construct these.

Definition 2.1.5 *A* mixed level *covering array,*

$$MCA_\lambda(N; t, k, (v_1, v_2, ..., v_k)),$$

is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

1. *Each column $i \quad (1 \leq i \leq k)$ contains only elements from a set S_i of size v_i.*

2. *The rows of each $N \times t$ sub-array cover all t-tuples of values from the t columns at least λ times.*

When $\lambda = 1$ we can omit the subscript. The array can now be represented as $MCA(N; t, k, (v_1, v_2, ..., v_k))$. We use a shorthand notation to describe mixed covering arrays by combining equal consecutive entries in $(v_i : i \leq 1 \leq k)$. For example three consecutive entries each equal to 2 can be written as 2^3. Consider an $MCA(N; t, (w_1^{k_1}, w_2^{k_2}, ..., w_s^{k_s}))$. This can be written as an $MCA(N; t, k, (v_1, v_2, ..., v_k))$ (see Figure 2.4).

In this array we have:

1. $k = \sum_{i=1}^{s} k_i$ *and* $v = \sum_{i=1}^{s} k_i w_i = \sum_{i=1}^{k} v_i$.
2. *Each column* i $(1 \leq i \leq k)$ *contains only elements from a set* S_i *where* $| \cup_{i=1}^{k} S_i| = v$.
3. *The columns are partitioned into* s *groups* $g_1, g_2, ...g_s$ *where group* g_i *contains* k_i *columns. The first* k_1 *columns belong to the group* g_1*, the next* k_2 *columns belong to group* g_2*, and so on.*
4. *If column* $r \in g_i$*, then* $|S_r| = v_i$.

In this thesis we often represent the vector $(w_1^{k_1}, w_2^{k_2}, ..., w_s^{k_s})$ as the string $w_1^{k_1} w_2^{k_2} ... w_s^{k_s}$. Although we use the order of the components in our notation, there is nothing structural that requires the underlying objects to use the same order. When components of a covering array are permuted their properties hold. This notation can be used for a fixed-level covering array as well. $CA(N; t, v^k)$ indicates that there are k parameters each containing a set of v symbols. This makes it easier to see that the values from different components can come from different sets.

2.2 Covering Array Bounds

The problem of finding the correct covering array for a particular test suite is not always easy. The trivial mathematical lower bound for a fixed level covering array is v^t. For a mixed level covering array we have the bound $\prod_{i=1}^{t} v_i$, where $v_1 \geq v_2 \geq \cdots \geq v_k$. This number, however, is often not achievable. Therefore a primary avenue of research for covering arrays is to determine the achievable lower bounds. When we have a provably optimal lower bound for a particular covering array this is known as the *covering array number*, denoted $CAN(t, k, v)$. For example, $CAN(2, 5, 3) = 11$ [17, 89].

Most of the time we either do not know the covering array number, or cannot construct a covering array with the known minimum number of rows. Instead researchers present the best known upper bound for covering arrays. The best upper bound is the number of rows in the smallest constructible array. This is the number we will focus on since the purpose is to build and use these for software testing. When we refer to the *best bound* in the rest of this thesis we will refer to the best upper bound for a covering array.

Because N is often unknown we can shorten our notation for both a covering array and mixed level array by leaving out the N and denoting these objects as $CA(t, k, v)$ and $MCA(t, k, (v_1, v_2, ..., v_k))$ or $MCA(t, (w_1^{k_1}, w_2^{k_2}, ..., w_s^{k_s}))$.

There are several types of results known for covering arrays. These include probabilistic bounds that provide us with the minimum value of N, but do not give us any method for construction of an optimal array. There are constructive results which provide us with a direct way to create such an object using mathematical principles (we call these *algebraic constructions*), and finally, there are computational results that are produced as the end product of a search. Of these, the last two are probably the most useful for us, although knowing the probabilistic bounds helps to guide us in a search for new constructions.

Not only are the techniques to build covering arrays varied, but there is no single location for the best known results. N. Sloane [89] has an excellent summary of results for binary covering arrays of strength 2 and 3. B. Stevens [92] summarizes known results for $t = 2$ in his Ph.D. thesis and M. Chateauneuf [16] presents a summary for $t = 2$ and $t = 3$ in his Ph.D. thesis. G. Sherwood [88] maintains a web site of orthogonal arrays and covering arrays built from permutation groups. A. Hartman [51] presents a survey of results for both mixed and fixed level arrays, which he terms “covering suites”.

In this section we briefly describe some of the known bounds for covering arrays, most of which have been obtained through algebraic constructions. In Chapter 3 we examine computational methods.

2.2.1 Known Results for Strength Two Arrays

As reported by Sloane [89], the first known results on covering array numbers are due to Rényi [83] who solved the case for $t = v = 2$ when N is even. Kleitman and Spencer [60] and Katona [59] simultaneously solved the case for all N. They showed that the maximum value of k for a particular value of N is:

$$k = \binom{N-1}{\lceil \frac{N}{2} \rceil}$$

For a large k, N grows logarithmically. The minimal N [59, 60, 89] satisfies:

$$N = \log_2 k + \frac{1}{2} \log_2 \log_2 k$$

Sloane [89] also mentions more results in the case of $v > 2$. In 1990 Gargano, Körner and Vaccaro [45] gave a probabilistic bound for $t = 2$, $v > 2$.

$$N = \frac{v}{2} \log k(1 + o(1))$$

However, this result was not constructive, i.e. they did not provide any method to produce an array with such a bound [89]. They did provide a construction that achieved $N = 2.07 \log k(1 + o(1))$ [89]. Other constructions are given for individual values of k by Poljak, Pultr, Rödl and Tuza [81, 82]. Östergård [79] showed that $N \leq 11$ for $k = 5$ and $v = 3$, while Sloane [89] reports that Applegate showed

$N = 11$ for the same parameters. Sloane [89] shows that when $t = 2, v = 3$ and $k = 3(N-1)$, where N is the size of an optimal $CA(2, k, 2)$, three copies of the $CA(2, k, 2)$ can be combined. One row of the $CA(2, k, 2)$ can be relabeled to form a row of zeros without changing the properties of the array. The zero rows are removed. One copy of the remaining $CA(2, k, 2)$ is written using symbols 0 and 1, one is written using symbols 1 and 2 and one using symbols 0 and 2. This gives us the following bound:

$$k = \binom{N-1}{\lceil \frac{N}{2} \rceil}$$

D. Cohen *et al.* [25, 16] present a construction to show that:

$$CAN(2, k_1k_2 + 1, v) \leq CAN(2, k_1, v) + CAN(2, k_2, v) - v$$

In [93] Stevens *et al.* use group divisible designs to obtain the following bound:

$$CAN(2, q + 2, q - 1) \leq q^2 - 1$$

Stevens *et al.* extend the constructions beyond $v = 3$ using recursive constructions that utilize block structures with disjoint blocks [92, 95].

In addition to the work on finding upper bounds when $t = 2$ the problem of lower bounds has also been tackled. Stevens, Moura and Mendelsohn [92, 96] examined lower bounds for covering arrays. They proved:

$$CAN(2, k, v) \geq v^2 + 3 \quad \text{for} \quad k \geq v + 2 \geq 5$$

and for a large $k \geq 2^{2v-2-\frac{2}{v}}$

$$CAN(2, k, v) \geq \lceil \frac{v}{2} \log k \rceil + v + 1$$

Williams [51, 102] presents a recursive construction giving us the following bound when v is a prime power:

$$CA(2, vk+1, v) \leq CA(2, k, v) + v^2 - v$$

2.2.2 Known Results for Strength Three Arrays

More recently, authors have moved to the case of $t = 3$. We mention a few of these bounds here. G. Roux [84, 89] presents an upper bound of $7.56444 \log k$ for the case when $t = 3$ and $v = 2$. Roux also provided the following using a constructive method which will be revisited again in Chapter 6 [84, 89].

$$CA(3, 2k, 2) \leq CA(3, k, 2) + CA(2, k, 2)$$

In [18] Chateauneuf, Colbourn and Kreher present several constructions for strength 3 arrays using transitive groups. They provide the following constructive bounds

$$CAN(3, k, v) \leq \begin{cases} \frac{(2v-1)(q^3-q)+v}{(\log(2v-1))^2}(\log k)^2 & \text{if} \quad v \equiv 0, 1 \mod 3 \\ \frac{(2v-1)(q^3-q)+v}{(\log(2v-3))^2}(\log k)^2 & \text{if} \quad v \equiv 2 \mod 3 \end{cases}$$

where $q \geq v-1$ is a prime power. Chateauneuf and Kreher [17] provide a summary of the known results for $t = 3$. They present some new constructions based on known methods such as symbol collapsing and row collapsing as well as results obtained from perfect hash families. In addition they generalize the construction from Roux for $v = 2$ to that of any $v > 2$ to give the result [17]:

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

and they extend the group construction from [18]. In this paper they provide a list of the best known upper and lower bounds for covering arrays of strength three that were known in 2002 [17]. In [69], Martirosyan and van Trung extend the Roux construction to present bounds for $t > 3$. Hartman and Raskin [53] also present a construction based on Roux for $t = 4$.

What is important about these results is the variety of methods used to build them. There are recursive constructions that rely on other mathematical objects, simple constructions, and ones that employ complex mathematical concepts. To date there has been no detailed analysis of these arrays using other methods such as computational search. Therefore many constructions that do exist for strength three covering arrays may not be optimal. What is known about strength three arrays is still in its early stages of research. We see many opportunities here for improving and simplifying these techniques.

2.2.3 Mixed Level Covering Arrays

Less is known about bounds for mixed level covering arrays and orthogonal arrays. In [54] Hedayat and Sloane discuss mixed level orthogonal arrays. Sloane *et al.* expand this work and present bounds for mixed level orthogonal arrays using linear programming [90]. In [107], H. Xu presents an algorithm for mixed level orthogonal arrays of small sizes. Stardom [91] and Chateauneuf [16] suggest the need for extending their work on fixed level covering arrays to mixed level, but most of the methods known for constructing these apply only when $t = 2$ and the results are limited. Recently, Moura *et al.* [73] presented some algebraic constructions for mixed level covering arrays. These are limited to strength 2, but they have solved the problem for constructing minimal covering arrays when $k \leq 4$ and have solutions for many cases when $k = 5$. To date, most of the methods and results for mixed level arrays rely on algorithms built for software interaction testing. These will be explored in Chapter 3.

2.2.4 Covering Arrays for Software Testing

The ideas of experimental design, many of which were popularized and brought into the field of manufacturing by Taguchi [97] as *Robust Design*, have been used to test various aspects of computer software. We review the main areas where these have been used and highlight some empirical studies in this section.

In 1985, Mandl [68] applied the principles of experimental design to test compiler software. He proposed using orthogonal latin squares as a basis for designing test suites for compiler software. He used this to test correctness of the operator ordering of enumerated values in Ada. Tatsumi *et al.* [99] proposed using orthogonal arrays to test inputs to software as an additional black box testing method. Brownlie *et al.* [8] proposed the term *Robust Testing* which is modified from that of Taguchi [97]. They extended the work of Mandl and Tatsumi by applying orthogonal array testing (OATS) to an internal AT&T electronic mail system. The components in this system were both hardware components such as the CPU type and software components such as those developed to perform system functions. They compared the use of orthogonal arrays with a traditional test plan constrained by time and resources. They determined that the OATS method found 22% more faults in one half of the testing time than did the traditional approach [8].

D. Cohen and Dalal *et al.*[22, 23, 24, 40] at Bellcore (now Telcordia) developed the Automatic Efficient Test Case Generator (AETG). This is a patented commercial test case generator that uses covering arrays to design test suites. The patent also describes the use of some simple constructions and test case merging [22]. AETG allows *seeded test cases* (fixed test configurations that are included regardless of the covering array properties), and *aggregate conditions* (multiple components are joined together and act as a single component), as well as *constraints* (avoids). The AETG tool moves the use of covering arrays for software testing a step beyond the mathematical model. In [23] D. Cohen *et al.* suggest a hierarchical test system to allow two levels of interaction testing. This idea will be pursued further in Chapter 5. They also prove that it is possible to build test suites to test all pairwise combinations of parameters within a logarithmic bound on the size of k [23].

Experiments that determine the effectiveness of the AETG tool in testing user interface modules and several Unix system commands have been reported [23, 24]. Previously undetected faults were found when they applied interaction test suites to already tested software programs. In addition code coverage metrics were gathered. The pairwise method showed 92% block coverage, 85% decision coverage, 49% p-uses and 72% c-uses. This coverage was comparable with experiments which tested all combinations of interactions, and better than random test suites of comparable sizes [23, 24].

Burroughs *et al.*[11] describe how one can use covering arrays for protocol conformance testing in telecommunication systems. They show that covering arrays can be used to reduce the number of tests, while increasing the quality of testing by including all pairwise combinations. Traditional protocol conformance testing does not typically use this type of a heuristic to select tests. Instead it might test all pairwise combinations from the two largest components, or it might use an even weaker condition; including all values of each component at least once [11].

In [44] Dunietz *et al.* conducted empirical studies to examine the degree of code coverage for various strength covering arrays. They show that when $t = 2$ good block coverage is obtained, but higher strengths ($t > 3$) are required to obtain good path coverage [44]. Dalal *et al.* [39] used AETG to generate test cases for a telephone software system. In this study previously tested software was used. Approximately 15% of test cases revealed new system faults. In addition, several failures were found to occur only under certain combinations of values [39].

Burr *et al.* [10] examined the effectiveness of pairwise testing and measured the obtainable code coverage. In their experiments they tested a Nortel email system. They determined that pairwise coverage provided higher block and decision coverage than traditional methods of testing. In addition they found previously undetected faults in a mature software system.

More recently others have taken these ideas and extended them to other types of software testing. Williams *et al.* [104] use orthogonal arrays to test interchangeable network components. Kobayashi *et al.* [62] examine the use of covering arrays for logic testing of software. They found that combinatorial testing was often comparable to specification-based testing, and was always better than random testing. White [101] uses MOLS to test GUI event interactions and Chays *et al.* [19, 20] propose the use of orthogonal arrays to test databases. Daich [35, 36] has developed a spreadsheet macro that uses orthogonal arrays to build test suites.

Kuhn *et al.* [66] characterized faults in a large open source software system. In this study they examined bug reports for several systems. They determined that pairwise testing of combinations would have found 76% of the faults. Higher levels of interaction coverage found more faults. For instance when $t = 3$, 95% of the faults would have been found and when $t = 6$, 100% would have been found. This study highlights the need for more than pairwise interaction testing, a point that will be revisited in later chapters.

Yilmaz *et al.* [108] present a method to help with fault characterization in a distributed testing environment. Covering arrays of varying strengths were used to build reduced test configuration schedules. Classification trees were used to characterize (or localize) the test configurations causing faults. Initial results show that this method has promise to locate faults caused by interactions in large configuration spaces.

In *Lessons Learned in Software Testing* [58], a book for software practitioners, five pages are devoted to the pairwise testing method, indicating that the use of covering arrays for interaction testing is finding a niche in traditional software testing. However, the manner for constructing these test suites is ad hoc. This points to a need for reliable and flexible methods for building covering arrays.

2.2.5 Software Testing Parameters

A survey of software systems over a variety of domains begins to show how applicable the methods that follow are to real software testing environments. Often the number of components in specific systems and the testing time is proprietary in nature or a company cannot easily quantify where this type of testing can be used successfully. An important factor is the method used to model the system, since often a system can be modeled in more than one way. This can lead to different parameters for the same system. We highlight two published examples to give a flavor for relative cost reduction in testing time that can be obtained if we use covering arrays to design test suites. We then quantify the size of the parameters for some common software applications.

In [56], Huller examines a ground system for satellites developed at Raytheon Company. He quantifies the cost savings that could be obtained through a reduction in testing time if a pairwise testing strategy is used. The Raytheon system has five components, three with three values each, and two with two values each. Although there are only 144 combinations of these parameters, this is an expensive system to test, therefore all 144 combinations would not normally be tested in practice. A covering array for this system of size 12 is constructed as a comparison for cost savings against the normal test strategy adopted by this company. The standard testing strategy requires 370.5 hours at a cost of $\$36,100$ US dollars. The same system, using only 12 test configurations would have required 118 hours and $\$11,900$ US dollars to test. This is a savings of almost 70%. Methods presented in later chapters of this thesis can construct a covering array for the same set of parameters with only 9 test configurations, providing an even larger reduction in time and money.

In [70], Memon *et al.* describe the *software configuration space* as a challenge to modern software development; i.e. software that must run on many hardware platforms and work on many operating systems. These systems often have highly configurable options that can be selected either at compile time or at run time. The authors suggest that web servers such as Apache, object request brokers such as TAO and databases such as Oracle, have dozens or even hundreds of options. In their study they examine ACE + TAO which are middleware projects aimed at distributed software applications. The ACE+TAO system has over one million lines of code, and is supported on multiple operating systems by multiple compilers. There are several hundred components each of which can take on different values independently.

Compilation of the system for their study requires 4 hours. The static configuration space has 17 "options" with two values each, and 35 inter-option constraints resulting in over $82,000$ valid configurations. A simplified model is used that only examines 10 of the compile time options with additional constraints added between certain options. This reduces the model to less than 100 configurations [70]. Once the system is compiled in a particular configuration, a set of 96 tests has to be compiled and run. There are 6 run time options in this system, with a range of two to four values each. This creates 648 combinations of CORBA run time policies. Each of these has to be tested with each of the valid compilation configurations (29 configurations compiled successfully). The compilation of the test cases in each configuration for this system requires an additional 3.5 hours and running the tests requires 30 minutes. In total 8 hours is needed to compile and run the tests for each configuration of the system [70]. The total configuration space for this study includes $18,792$ configurations which requires $9,400$ hours of computer time to compile and test[108].

The same system is examined in [108] by Yilmaz *et al.* to determine if test schedules defined by various strength covering arrays provide a useful method to find and locate option related faults using only a subset of the configuration space. In this study it is determined that covering arrays in conjunction with classification trees are useful for finding and locating potential option related failures in the system. This study models the system as an $MCA(N; t, 29^1 4^1 3^4 2^1)$. For this system, covering arrays of various strengths are constructed using methods described in later chapters of this thesis. Strength two arrays perform as well as higher strength arrays in finding the failures, but higher strength arrays provide better fault localization.

The covering array for $t = 2$ reduces the configuration space to 116 configurations which is a 99% reduction in the number of configurations. This has potential to reduce the testing cost by almost one year of machine time. Additionally, methods to build variable strength arrays, described in Chapter 5 of this thesis, may provide better localization if higher strength covering arrays can be included for the same cost. This is yet to be explored.

There are many systems that have configurable options resulting in large configuration spaces. The following examples have been obtained from online documentation for some common applications:

- SQL Server 7.0 [72] has 47 configurable options. Of these 10 are binary, while the rest have a range of values. The number of equivalence classes per option would depend on how each one is modeled.
- Oracle 9 [78] has 211 initialization parameters.
- The Apache HTTP Sever Version 1.3 [2] has 85 core configuration options, 15 of which are binary.
- The gcc-3.3.1 compiler [48] has over 1000 command line flags that control 14 options. More than 50 of these flags are used to control the optimization option alone.

The examples in this section highlight the size of problems and potential savings if efficient methods for building covering arrays are developed.

2.2.6 Limitations of Covering Arrays for Testing

Some common themes evolve from these studies. The first is that the use of covering arrays for testing software systems is a complimentary testing method to the existing ones. In some of these studies it has been used after other test techniques have been applied, and has often found a different subset of faults. In [108] Yilmaz *et al.* show how covering arrays can be used for fault characterization. However, they also show that for certain faults, this is not possible. With only five pages in *Lessons*

Learned in Software Testing [58] from the book's 280+ pages dedicated to this topic, it is clearly viewed as only one of a multitude of testing techniques. Therefore we put the use of covering arrays for software testing into perspective. While this is a complimentary tool, and one that can be beneficial under severe time constraints, their use is not meant to replace other standard software testing techniques.

A second theme emerges. In all of these studies the modeling of the test problem to form a covering array or similar object is a separate and difficult issue. In fact, Dalal *et al.* point this out in [39]. Without correct modeling none of these methods can be effective. The inputs or components that should be designated as *factors* and the determination of their unique values is not always an easy decision. Sometimes this may be obvious, but different models for the same problem may reveal different test suites and different results. In this thesis we do not attempt to discuss or address the modeling question. This is an interesting area of research and one that enhances our work.

2.3 Summary

In this chapter we have presented the mathematical background and given a short historical perspective to our research. The primary mathematical object that we will examine from now on is the covering array. The orthogonal array is really a special case of the covering array and is subsumed by this. We will discuss both fixed and mixed level arrays since our aim is to move toward a real testing environment. The rest of this thesis focuses on *building* small, flexible interaction test suites in an efficient manner.

Chapter 3

Computational Techniques

In the previous chapter we defined mathematical models for combinatorial test suites and presented a sample of known results. This gives a flavor for the scope and depth of algebraic constructions. One striking feature is the variety of mathematical methods and extensive background knowledge that is required to master these. A construction that produces optimal results for one set of parameters, t, k and v, may not work well for a different set. Just this problem alone, that of knowing what method to use for a particular test suite, is enormous. This is magnified for mixed level covering arrays, where very little has been published until recently. An additional difficulty is that the results for known methods and bounds of covering arrays are scattered throughout the literature. If a software tester wants to build a covering array for a specific problem, without a tool, then they have a daunting task in front of them.

This leads us to examine more general approaches for building interaction test suites, ones that work well across fixed and mixed level arrays of all combinations of parameters. The logical direction for this type of solution is to use a computational search method. There are a variety of search techniques that can be used to build covering arrays. These may not always find the smallest size array, and they may require longer computational times than constructions, but they work across a broad set of problems and may not fall prey to the complexities of the mathematical approach.

3.1 Overview

When we refer to "computational methods" in this thesis we mean *computational search*. Computational methods are directed by specific properties of the output object as it is being constructed, such as the number of t-sets still left uncovered. Algebraic methods, on the other hand, are driven entirely by the input parameters and are based on mathematical reasoning. Given a set of parameters the outcome is known ahead of time. In this sense they are direct construction techniques. Algebraic methods can be incorporated into computational tools as is described in Section 3.4.

Computational methods are more flexible than algebraic methods. They can work across a broad range of values for t, k and v and they are suitable for both mixed and fixed level arrays. In addition it is easier to add constraints to computational methods such as including a set of fixed or *seeded* test configurations. Many algebraic constructions would be forced to append seeded test configurations to an already completed test suite, whereas computational methods can incorporate these as the suite is being built.

Exhaustive search will always produce the smallest possible test suite. Unfortunately this is not feasible except for trivially small arrays. Given a set of uncovered pairs, and a specific integer n, answering the question of whether or not a test configuration exists that covers at least n pairs is NP-complete [31, 87, 109]. Furthermore, Seroussi and Bshouty [87] suggest that the problem of finding the minimum size of a covering array for an arbitrary set of pairs is at least as hard as this problem. This implies that an efficient method for determining the smallest set of test configurations for a particular set of parameters t, k and v is unlikely to exist [31]. Therefore computational methods use *heuristics* (rules) to intelligently narrow a large solution space. If good heuristics are used, then relatively quick convergence for a close-to-optimal solution occurs.

There are two constraints imposed when writing an algorithm to find covering arrays. The first and primary goal is to cover all t-sets of interactions. A solution is not considered valid unless this is satisfied. The second goal is to find the smallest N. The approach used to determine if all t-sets are covered is a generic one that applies for all algorithms. Some methods for doing this are discussed at the end of this chapter. The approach used to figure out the best N will vary depending on the type of algorithm selected. We will see two main approaches for this. One is to incrementally add test configurations until we have satisfied the first constraint. The other is to select N first and then see if the values of the components can be manipulated to cover all t-sets.

There will always be a trade-off in building covering arrays using computational methods. The time taken to build a covering array must be weighed against the need for small test suites. If the tests to be run require manual intervention, in component configurations, or in test outcome assessment, then finding minimal test suites may be of utmost importance. In the situation, however, where the test suites are inexpensive to run, the computational time of finding a test suite may be more important. We focus on situations where the testing is expensive.

Existing methods to build covering arrays include greedy algorithms [23, 31, 109, 110] and meta-heuristic search [27, 76, 91, 92] as well as integer programming [106]. In addition several tools that implement algebraic constructions have been developed [53, 102]. In this chapter, we begin by examining several *greedy algorithms* that have been used successfully to find covering arrays. We also briefly discuss two tools that implement algebraic methods. We then examine a search based approach. We begin with a *heuristic* method and follow this with several *meta-heuristic* search algorithms. We end the chapter by examining some data structures and algorithms that reduce the technical difficulties in implementing many of the computational methods.

Integer programming has only been shown to work on very small problems so far [106]. Williams and Probert experimented with integer programming in [106].

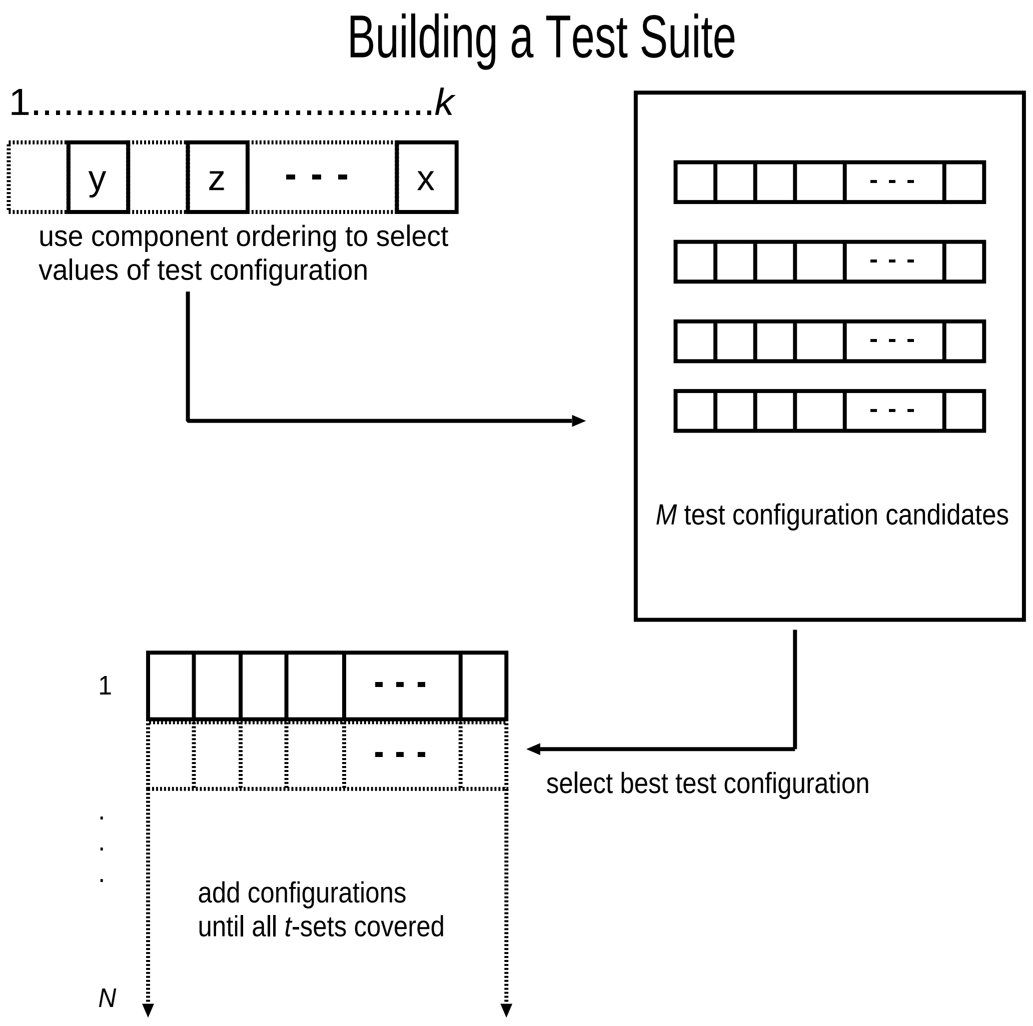

Building a Test Suite
1..................................k
y
z
- - -
x
use component ordering to select values of test configuration
- - -
M test configuration candidates
1
- - -
select best test configuration
add configurations until all t-sets covered
N

Figure 3.1: General framework for greedy algorithms

In their representation of the problem, the number of variables grows exponentially with the number of components and values, making run times infeasible except for very small problems. For instance, the representation of a $CA(2,5,3)$ required 243 variables. The solver ran for more than 6.5 hours at which point it was terminated. The solution found at that time was of size 13 which is greater than best reported bound of 11 [17]. It is possible that an alternative representation of this problem for integer programming may produce different results, but we do not explore that here. Readers are referred to [106] for more information.

3.2 A Framework for Greedy Algorithms

Greedy algorithms are often used when no known polynomial time algorithm exists for a specific problem. A generic greedy algorithm goes through a series of steps making the locally optimal decision at each point in time [34]. Because these do not explore the entire search space, but instead select the next "best" solution, they usually run in polynomial time, but do not guarantee a global optimum. Instead the hope is that the use of good heuristics will give a relatively close to optimal solution. Greedy algorithms have been used successfully to find covering arrays. They often require less computational time than other computational techniques, but do not always produce the smallest possible covering arrays [27].

We begin by defining a framework for one set of greedy algorithms. It is our observation that three of the greedy algorithms presented in the literature, the *Automatic Efficient Test Generator* (AETG) [23], the *Test Case Generator* (TCG) [110] and the *Deterministic Density Algorithm* (DDA) [31] are essentially the same algorithm with variations at certain key decision points. We have abstracted the common elements and present them as a generic framework. The AETG algorithm was first described by D. Cohen *et al.* in [22, 23, 24]. The others were presented as variations on AETG [31, 110]. One additional greedy method has been described in the literature (In Parameter Order [109]), but as it uses a different approach it will be described separately.

Figure 3.1 illustrates this framework. It uses an incremental approach to find the best N. We begin with an empty test suite. As each test configuration is added, N is incremented. We stop when all t-sets are covered. The test suite being built is shown in the lower left corner of Figure 3.1. At this point one test configuration has been selected and the algorithm is currently selecting the second. The top left corner of this figure shows a single test configuration. We begin with an *ordering* of the components and use this order to select a value for each component by choosing the value that will give us the "best" solution. In this framework, a temporary *pool*, or collection of M test configuration *candidates* is created before adding the "best" one to the final test suite, i.e. we build M test candidates in parallel, but use only one. This step is shown in the right portion of Figure 3.1.

Algorithm 1 Greedy framework

```
$N = 0$
[SET $M$]
while ( uncovered $t$-sets ) {
  for $i = 1$ to $M$ {
      for $j = 1$ to $k$ {
          [SELECT NEXT COMPONENT TO FIX]
          select [BEST] value
          use [VALUE TIE BREAKING RULES]
      }
      add test configuration to pool of candidates
  }
  select test configuration that covers most new $t$-sets
  use [TEST CONFIGURATION TIE BREAKING RULES]
  fix $N$th test configuration
  update counts
  $N++$
}
```

We have defined 5 *decision points* in this framework. These account for the majority of the differences between the three algorithms. Pseudo-code is given in Algorithm 1. The decision points are contained in square brackets and are in bold font. They are [**SET** M], [**SELECT *NEXT* COMPONENT TO FIX**], [**BEST**],[***VALUE* TIE BREAKING RULES**] and [***TEST CONFIGURATION* TIE BREAKING RULES**].

The algorithm begins with an empty test suite, i.e. $N = 0$. Next the algorithm determines the size of M [**SET** M] and the main loop begins. For each iteration of the loop one test configuration candidate is added to the test suite. Inside this loop is the logic for creating each individual test configuration. For each of the M candidates, a component ordering, [**SELECT *NEXT* COMPONENT TO FIX**] is chosen. This may be the same for all M candidates or may be dynamically determined. One value for the next component is selected. The value chosen is based on the [**BEST**] rule for the specific algorithm. When more than one value is best, [***VALUE*** **TIE BREAKING RULES**] are employed. Once M candidates are built, the "best" test configuration is chosen. The [***TEST CONFIGURATION*** **TIE BREAKING RULES**] are used when more than one satisfies this condition. The chosen test configuration is added to the test suite, all counts and data structures are updated, and N is incremented.

The main differences between these three greedy algorithms are found in the five decision points. Each of these is discussed in more detail next. This is followed by a discussion of each individual algorithm. We restrict our discussion of the framework to already implemented algorithms, but there are many more variations and combinations of decision points for this framework. The effect that each one has on computational time and the size of N is an interesting and open question.

3.2.1 Selecting the Number of Candidates (M)

The greedy framework creates one test candidate at a time. However, a *pool* of candidate test configurations may be built in parallel to increase the chances of finding the best next test candidate. If M is set to be too large, then this will have a negative impact on the algorithm's run time. If M is set to be too small, then one may not get the best result for N. These factors must be balanced when we determine how to use this decision point. Selecting a good value for M seems to be the key to AETG's and TCG's effectiveness, although they use different methods [23, 110]. DDA on the other hand creates only one test candidate each time ($M = 1$).

3.2.2 Selecting a Component Ordering

One of the most important steps in these greedy algorithms is selecting the next value for a particular component. Since the components are selected one at a time and a value is chosen, based on the other components that have previously been selected, the component ordering plays a major role in how effective the algorithm is. Component orderings that provide one with the best chance of making good decisions early in the algorithm are preferred. A good choice is to select early on components that are *free*, i.e. they have combinations with other components left to cover. If, instead, a component is selected first that has had all of its potential combinations with other components covered, then this is a bad choice.

Some of the possible methods for ordering are deterministic, based on a heuristic, based on a program parameter or random. A combination of these methods can also be used. When the order is deterministic we mean that the ordering is based on the position of the component in the array, such as first, last or lexicographical. In this situation, since the ordering is not based on any parameter of the covering array and does not change during run time, the exact order selected is arbitrary. By changing the underlying representation of the components, one can change this ordering. Although slight differences in performance may occur for different fixed orderings, they are not algorithmic in nature, but due to the data representation used. When a deterministic ordering is used we will just state it as lexicographical, but mean this to be any fixed ordering.

AETG uses a combined heuristic-random method for this decision point. TCG uses a static component ordering for all test configurations, but the ordering is based on the parameters k and v. Therefore we say that it uses a heuristic to set this order [23, 110]. DDA, like AETG uses a dynamic approach (i.e. each test configuration has its own ordering), but uses a different heuristic to select the component ordering [31].

3.2.3 Selecting the *Best* Value

Once a component is selected for the current test configuration, the *value* for that component must be chosen. In this framework the algorithm systematically tries each value for the component. The value that is *best* is chosen. AETG and TCG define best to be the value that creates the most new t-sets with the already "fixed" components [23, 110]. They do not consider any information about components yet to be selected. DDA on the other hand, uses a *density* calculation that acts like a "lookahead" to the other "non-fixed" components.

3.2.4 Value Tie Breaking

When there is more than one value that satisfies *best*, a decision about how to break it must be made. The simplest and fastest method is to use some deterministic ordering, such as lexicographical ordering. This makes the algorithm deterministic and repeatable. Random tie breaking can also be chosen, or as is done in TCG a heuristic can be used [110]. The method chosen for this decision point is a trade-off between the size of the final covering array, the run time and the repeatability of the algorithm. If randomness is used, then the final test suite may be smaller, but the same array will not be produced by each run of the algorithm. If lexicographic tie breaking is used, then this is fast and repeatable, but may not provide the tightest bound on N. The last option, using a heuristic, is an alternate approach that might improve the goodness of the solution and maintain repeatability.

3.2.5 Test Configuration Tie Breaking

The last decision point shown occurs when there is more than one test configuration in the pool of candidates that covers the most new t-sets. Once again it can be random, heuristic or lexicographical. All of the algorithms we describe use either random or lexicographical, but using a heuristic may also work. This is open for exploration.

3.2.6 Algorithm Repetitions

In several of the decision points listed, one option is randomness. For instance, tie breaking is often done randomly. In AETG the ordering of components has a random element to it. When *any* randomness is involved the final size of N can vary between different runs of the same algorithm. Therefore one last element to be considered is the number of times that the algorithm itself is run. This is not shown as a decision point in the framework, since it is not an actual variation, but rather an experimental aspect. It will, however, play an important role in performance. The number of algorithm repetitions will affect both the run time and the final size of N.

3.2.7 The Framework in Practice

In this section we present details of the three algorithms described in the literature. We refer back to the framework to highlight specific decisions that are made and to show differences. Although we describe the known algorithms here, there are many more variations to this framework. A systematic exploration of this framework is needed to determine which combinations of parameters provides the best performance with respect to algorithmic efficiency and the size of the final covering array. We do not attempt to explore this here, but see it as future work.

Automatic Efficient Test Generator(AETG)

The decision points for AETG are shown in Algorithm 2. Since AETG is a commercial patented product some variations that are not detailed in the literature are most likely to occur in practice [22, 23]. When we refer to the details of this algorithm, we are referring to the one described in [23].

The algorithm is based on a proof that the greedy approach can provide a logarithmic bound of N given k [23]. However, as is pointed out in [31], the algorithm as is implemented does not guarantee this. It seems to achieve the bound in practice due to decisions made in this framework. One place that has an impact on the performance of AETG is in the selection of the size of M for the pool of test candidates. The tuning of the parameter M is important in the AETG approach. In [23] the value of 50 has been suggested for M as an experimentally derived best value. Although we do not know what value is used in practice, we adopt this. There may be other values of M which work better on certain parameter sizes, but that is open

Algorithm 2 AETG algorithm details

```
SET $M$ {
  $M = 50$
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
SELECT NEXT COMPONENT TO FIX {
  if ( $j < t$ ) {
      create set of (component,value) pairs with the greatest
         number of new $t$-sets to cover
      select a (component,value) pair randomly from this set
  }
  else
      randomly select next component
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
BEST {
  value that creates most new $t$-sets with $j - 1$ fixed components
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
VALUE TIE BREAKING RULES {
  use random tie breaking
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
TEST CONFIGURATION TIE BREAKING RULES {
  use random tie breaking
}
```

for exploration.

AETG uses a mixed heuristic and random ordering of components. It selects a different ordering of components for each of the M test candidates, in each iteration of the algorithm. For instance if M is 50, then for each of the 50 candidates, a new ordering is created. The ordering of components is important and therefore intelligent heuristics should be used. The AETG algorithm maintains a set of uncovered t-sets, $\mathcal{U}$. The first (component,value) pair is selected so that it is a value which appears the greatest number of times in this set. This is an attempt to maximize the potential for covering the most new t-sets later on. The rest of the components are ordered randomly. The algorithm description in the literature does not say specifically what happens for the case of $t > 2$, although in [22] it is suggested that the

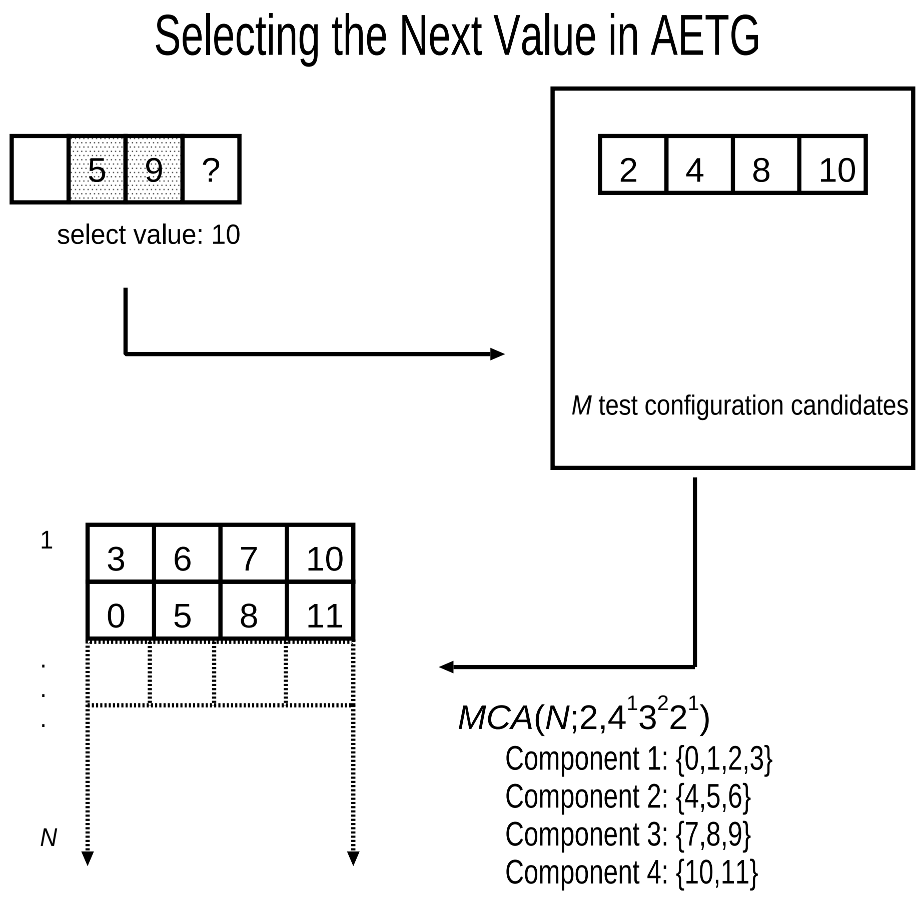


Figure 3.2: AETG algorithm illustrating one step in creating a test configuration candidate

first t-1 components are selected from $\mathcal{U}$. Therefore we can extend the heuristic so that each of the first t-1 (component,value) pairs is selected to maximize the number of times the value occurs in $\mathcal{U}$.

AETG decides on the *best* value by selecting the one that covers the most new t-sets with the components that are already fixed in that test configuration. It randomly breaks ties for values and test configurations.

A single step in the algorithm for pairwise coverage is shown in Figure 3.2. In this example, we are trying to build an $MCA(N; 2, 4^1 3^2 2^1)$. The first two test configurations have already been committed. We have created one of the M test candidates and are in the process of creating a second one. First we must create a component ordering. The first component in this order is 2, since the value "9" is among the values that occurs most often in $\mathcal{U}$, the set of uncovered t-sets. The next component to be fixed (based on the random permutation of the remaining components) is component 1. Any of the values for this component will create a new pair with the value 9 so one is randomly selected. In this case 5 was chosen. The next component to be fixed is 3 (once again random). This component has two possible values. If 10 is selected, then 2 new pairs will result, but if 11 is selected then only one will be covered since the pair (5,11) has already been used in the second test configuration. Therefore 10 is selected.

The commercial AETG program is general in t. This is the only one of these three greedy algorithms that has a working implementation beyond $t = 2$. It extends the base algorithm further by providing support for seeded test configurations, aggregate test cases and *avoid* conditions. These are combinations that cannot occur and therefore should not be tested. In addition, if the parameters, t, k and v match certain constraints then algebraic constructions using projective planes may be employed. D. Cohen *et al.* suggest in [22, 25] that there may also be post processing, test configuration merging steps.

Test Case Generator (TCG)

Tung *et al.* [110] suggest a deterministic algorithm that they claim creates smaller test suites than AETG and uses less computational power. It is not apparently clear however that the first claim is true, due to the limited set of experimental results presented. This is explored further in Chapter 4. TCG was used to test software for the Mission Data System in the Jet Propulsion Laboratory [110]. The decision point details are given in Algorithm 3. The authors use a deterministic fixed ordering of the components. They sort the components in non-increasing order by the v_i's, i.e. the number of values for each component. Therefore components with more values will always be used first.

Unlike AETG this ordering does not change dynamically during run time. Each of the M test candidates uses the same ordering of components. In TCG the first component i is fixed as the one with the greatest number v_i of values. M is set to v_i. Unlike AETG, M is dependent on the parameters of the test suite to be built. Although the algorithm does not explicitly state this, we interpret the fixing of M to mean that we rotate through each value of the first component for the M test candidates.

Since this algorithm optimizes its component ordering based on the differences in numbers of values for components, the question remains as to whether it works as well as some of the other algorithms for fixed level covering arrays, i.e. when all components have an equal number of values.

Algorithm 3 TCG algorithm details

```
SET $M$ {
  sort components in descending order
    by number of values in component
  $M$ = number of values in first component
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
SELECT NEXT COMPONENT TO FIX {
  use the next component in sorted order
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
BEST {
  value that creates most new $t$-sets with $j - 1$ fixed components
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
VALUE TIE BREAKING RULES {
  select value that is least used
  if ( still tied )
     select either random or lexicographical order
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
TEST CONFIGURATION TIE BREAKING RULES {
  select either random or lexicographical order
}
```

A value is determined as *best* when it covers the most new t-sets with the already fixed components. This is the same as in AETG. A heuristic is used in *value tie breaking*, however. The number of times individual values are used is stored and maintained. When there is more than one value that is *best*, the value that has been *least-used* is chosen. This count is slightly different from the heuristic chosen in AETG for component ordering (the value that occurs most often in uncovered t-sets), because not all values will be involved in the same number of t-sets in a mixed level covering array. This is illustrated further in our data structures presented at the end of this chapter. Although not directly stated in the algorithm as written [110], we interpret the number of times a symbol is used, to be the *number of times it contributes to a new t-set.* Therefore we count its usage based on how many t-sets it covers. For instance if a symbol is used once, and there are 3 other components, then it will be used three times for pairwise coverage, i.e. one for each new pair. The next time it is used, it is possible that one of the pairs is already covered. If this is the case, then we only count the "new" pairs.

When a second tie occurs, the TCG algorithm uses either a deterministic (i.e. lexicographical) ordering or a random selection. The algorithm as described in [110] used a deterministic ordering, but the authors suggest using a random ordering to improve results. Therefore a choice is shown in our framework description.

Figure 3.3 shows one step in the TCG algorithm. This example builds a test suite for an $MCA(N; 2, 4^1 3^2 2^1)$. Five test configurations have already been committed in this example. The component ordering is based on the number of values per component. In this example the first component selected is number 1, followed by the two components with 3 values, and finally the component with 2 values. These are shown in sorted order in the diagram. M is set to equal the largest number of values (in this case 4). The first of the M test candidates has been added to the pool.

We have added the next value from the first component, 1, to this test candidate. The next component to be fixed is the second one. In this example either 4 or 5 can be selected. They both will create a new pair with the only fixed component so far. Since we have more than one value that satisfies the criteria, TCG breaks the tie by selecting the one that has been least used. In this example, 5 has been used

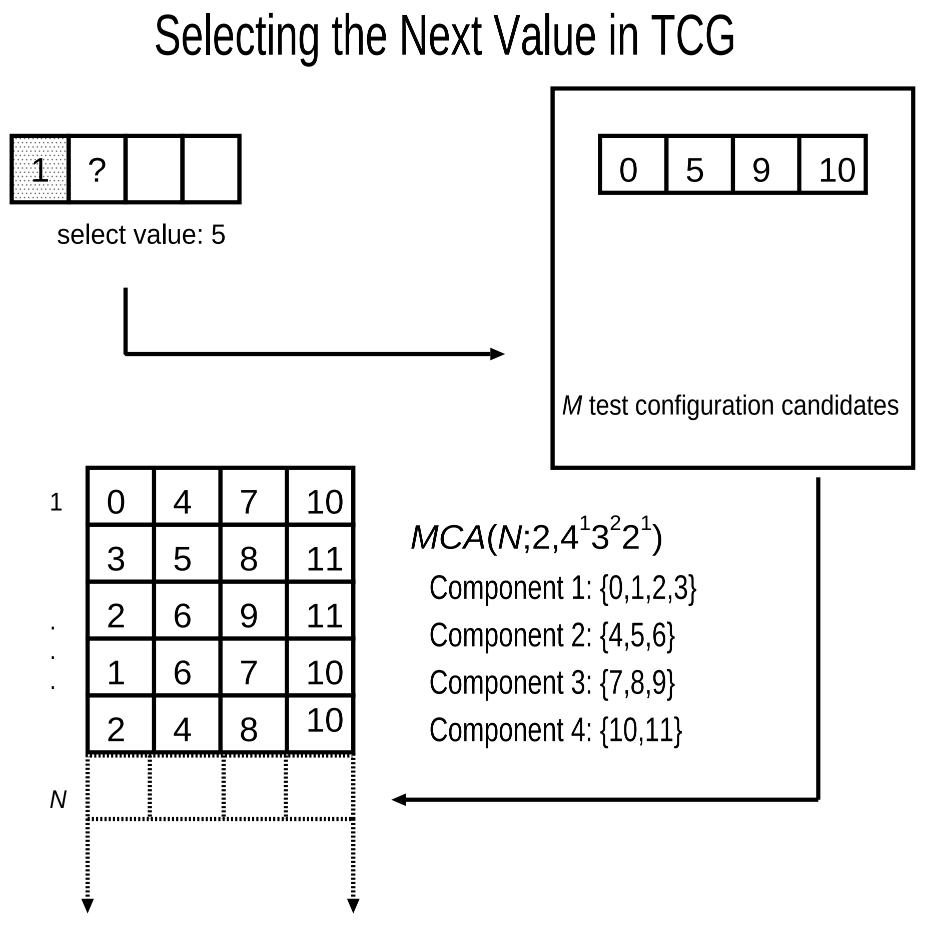


Figure 3.3: TCG algorithm. Illustrating one step in creating a test configuration candidate

in three pairs, while 4 has been used in 5, so we select the value 5.

The Deterministic Density Algorithm

The AETG authors, D.Cohen *et al.* [23], prove a worst case logarithmic bound on the number of test configurations required to build a test suite as a function of k. To do this they describe a method for constructing covering arrays. The following is a summary of their proof paraphrased from [31]. Let $\mathcal{P}$ be the collection of pairs still uncovered after a certain number of test configurations are selected. Let $\mathcal{L}$ be the product of the number of values found in the two components containing the greatest number of values. There are a total of $\prod_{i=1}^{k} v_i$ possible configurations that can be chosen. A test configuration that selects the largest number of pairs found in $\mathcal{P}$ is chosen. This is repeated until the collection of uncovered pairs $\mathcal{P}$ is empty. This does not guarantee a minimum size test suite, but it does guarantee that at each stage *at least* $\frac{|\mathcal{P}|}{\mathcal{L}}$ pairs are covered [23, 31]. It provides us with a worst case logarithmic function of k (the number of components).

As pointed out by Colbourn, Cohen and Turban in [31], however, the AETG algorithm does not actually provide this guarantee since it does not "look ahead". In practice the algorithm may adhere to the bound, but it is not guaranteed.

They have developed the deterministic density algorithm (DDA) to overcome this limitation. It is a greedy algorithm based on AETG that produces mixed and fixed level arrays of strength 2. They define *densities* which are meant to reflect the potential contribution a component and value pair will make to the number of t-sets that are left to be covered. In this manner, they can provide a logarithmic worst case bound on N in relation to k. The algorithm is completely deterministic. In all decision points in the algorithm where randomness can be used, DDA uses a fixed deterministic rule to make the selection. We represent this as lexicographical order. In DDA M is set to one (i.e. only one test configuration candidate is created and used at each step).

In order to understand DDA we begin with some definitions:

Definition 3.2.1 *The* local density *is* $\delta_{i,j} = \frac{r_{i,j}}{v_i v_j}$*, where* $r_{i,j}$ *is the number of uncovered pairs involving a value for component* i *and a value for component* j*.*

Definition 3.2.2 *The* global density *is* $\delta = \sum_{1 \leq i < j \leq k} \delta_{i,j}$*.*

The densities represent the fraction of pairs of components that are yet to be covered. This gives us a method to look ahead beyond the currently fixed components. At each stage of the algorithm a test configuration that covers *at least* δ pairs is chosen. The framework details are shown in Algorithm 4.

The actual algorithm implementation modifies these computations slightly to optimize the densities, while still providing the logarithmic guarantee [31]. In the

Algorithm 4 DDA algorithm

```
SET $M$ {
  $M = 1$
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
SELECT NEXT COMPONENT TO FIX {
  select component with greatest component density
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
BEST {
  value that creates greatest local density
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
VALUE TIE BREAKING RULES {
  lexicographical order
}
– – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
TEST CONFIGURATION TIE BREAKING RULES {
  lexicographical order
}
```

proof of the algorithm there is no need to order the components, as is done in the framework, but this might improve the performance by producing a smaller N. Therefore two additional densities are defined:

Definition 3.2.3 *The* component density *is* $\delta_i = \sum_{1 \leq j \leq k, j \neq i} \delta_{i,j}$.

Definition 3.2.4 *The* value density *is* $\rho_{i,s,\sigma}$ = *the number of uncovered pairs involving some level of component* i *and the* σ *value of component* s.

The component densities are calculated and the component with the greatest density is *fixed* next. For each component a value is selected by summing all of the value densities for each value. Whenever a component is already fixed, only the value densities between it and other components are calculated. This is because only those pairs associated with the fixed value can change. The component value that produces the greatest sum is selected. Lexicographical tie breaking is used for both value and test configuration tie breaking.

When calculating all of the densities, further "practical" considerations are made. In a mixed level array the local density scales by the total number of uncovered pairs for each component combination. When two components that have fewer values are combined, the denominator is smaller. For instance in an $MCA(N; 2, 4^1 2^1 3^1)$, the first two components have 8 possible pairs, while the first and last components have 12 possible pairs between them. This means that pairs in the first two components contribute $\frac{1}{8}$ while pairs in the first and last components contribute $\frac{1}{12}$. In practice, however it will probably be *harder* to cover all the pairs between the first and last components. Therefore the same denominators are used across all of the densities to fix this problem. We use three variables to represent these denominators. Let v_max = the maximum number of values for any component. In the $MCA(N; 2, 4^1 2^1 3^1)$ $v\_max = 4$. We also use the square of this value, v_max^2. For example, in the $MCA(N; 2, 4^1 2^1 3^1)$ $v\_max^2 = 16$. Lastly we maintain a constant variable v_fixed and set this equal to 1.

The following rules apply when each of the specific denominators are used:

1. When calculating the component densities, if neither of the components are fixed, then v_max^2 is used. If one is already fixed, then v_max is used.

2. When calculating the local densities to select a value, v_max is used if neither of the components is fixed yet. When one is already fixed, v_fixed is used.

Figure 3.4 shows a step in the DDA algorithm. Tables 3.5 - 3.7 give the various densities for this step. We are creating an $MCA(N; 2, 4^1 2^1 3^1)$. The first test configuration has already been chosen. Therefore the pairs $\{0, 4\}, \{0, 6\}$ and $\{4, 6\}$ have been covered. Table 3.5 shows the component densities after this test configuration has been added. At this point no components are fixed so that all of the denominators use $v\_max^2 = 16$. The first component has the greatest density, so this one is fixed first. Once a value has been chosen (value 1 has been selected), the next component to be fixed is chosen. This brings us to the state of Figure 3.4. Table 3.6 shows the density calculations for the two remaining components. Since

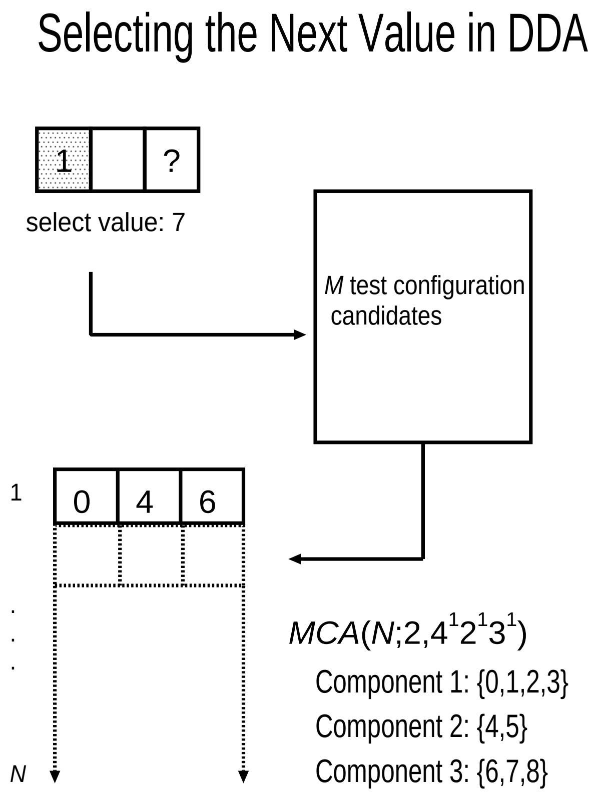


Figure 3.4: DDA algorithm. Illustrating one step in creating a test configuration

the first component is fixed, the denominator for densities using this component is $v\_max = 4$. In this case, the density of the last component (1.0625) is greater so we fix that one next. Next we examine the value densities for each of the three values of this component with the other two components. These are shown in Table 3.6. In this example, since the first component is fixed, the denominator is $v\_fixed = 1$. The denominator for the density with the second component remains $v\_max = 4$. There is a tie between the second and third value of this component. They both have a density equal to 1.5. Since our tie breaking is lexicographic we select the first one, which gives us the symbol 7.

Preliminary results using DDA to generate tests compared to AETG and TCG suggest it is competitive [108].

Densities	Calculation
$\delta_{1,2}$	$\frac{7}{16}$
$\delta_{1,3}$	$\frac{11}{16}$
$\delta_{2,3}$	$\frac{5}{16}$
Totals	
δ_1	$\frac{7}{16} + \frac{11}{16} = 1.125$
δ_2	$\frac{7}{16} + \frac{5}{16} = .75$
δ_3	$\frac{11}{16} + \frac{5}{16} = 1$

Figure 3.5: Selecting first component of second test configuration in Figure 3.4

Densities	Calculation
$\delta_{1,2}$	$\frac{2}{4}$
$\delta_{1,3}$	$\frac{3}{4}$
$\delta_{2,3}$	$\frac{5}{16}$
Totals	
δ_2	$\frac{2}{4} + \frac{5}{16} = .8125$
δ_3	$\frac{3}{4} + \frac{5}{16} = 1.0625$

Figure 3.6: After first component selected in Figure 3.4

Densities	Calculation
$\rho_{2,0,0}$	$\frac{1}{1}$
$\rho_{2,0,1}$	$\frac{1}{1}$
$\rho_{2,0,2}$	$\frac{1}{1}$
$\rho_{2,1,0}$	$\frac{1}{4}$
$\rho_{2,1,1}$	$\frac{2}{4}$
$\rho_{2,1,2}$	$\frac{2}{4}$
Totals	
$\sum_{i=0}^{1} \rho_{2,i,0}$	$\frac{1}{1} + \frac{1}{4} = 1.25$
$\sum_{i=0}^{1} \rho_{2,i,1}$	$\frac{1}{1} + \frac{2}{4} = 1.5$
$\sum_{i=0}^{1} \rho_{2,i,2}$	$\frac{1}{1} + \frac{2}{4} = 1.5$

Figure 3.7: Selecting the value for component 2 in Figure 3.4

3.3 In Parameter Order (IPO)

We present a different greedy algorithm next for finding covering arrays. The In Parameter Order (IPO) *pairwise* test generation strategy developed by Tai and Yu uses a different greedy approach [98, 109]. The authors have built a tool (PairTest) that implements the IPO framework [98, 109]. IPO works for mixed and fixed level arrays of strength two. Assume that there are more than 2 components. The framework begins by creating a test suite for the smallest number of components (i.e. 2). This is an enumeration of all pairs that must be covered. The framework then expands the test suite horizontally (i.e. by component) and then vertically (i.e. by test configuration) until all pairs are covered. We have included the framework from [109] as Algorithm 5.

Figure 3.8 illustrates this framework for an $MCA(N; 2, 2^2, 3^1)$ as described in [109]. The first step of this algorithm is to enumerate the 4 pairs $(0,2), (0,3), (1,2), (1,3)$

Algorithm 5 In parameter order greedy algorithm [109]

```
begin
  { for the first two components $p_1$ and $p_2$}
  $\mathcal{T} := \{(v_1, v_2)\} | v_1$ and $v_2$ are values of $p_1$ and $p_2$
                    respectively }
   if $n = 2$ then stop;
   for component $p_i, i = 3, 4, ...n$ do
   begin
       horizontal growth
       for each test configuration $(v_1, v_2, ..., v_{i-1})$ in $\mathcal{T}$ do
           replace it with $(v_1, v_2, ..., v_{i-1}, v_i)$;
           where $v_i$ is a value of $p_i$ ;
      vertical growth
      while $\mathcal{T}$ does not cover all pairs between $p_i$ and each of $p_1, p_2, ..., p_{i-1}$ do
          add a new test configuration for $p_1, p_2, ...p_i$ to $\mathcal{T}$;
    end
end
```

from the first two components. This is shown in part **A**. The next step is to expand horizontally, by adding the third component. Suppose the first test configurations are now $(0, 2, 4), (0, 3, 4), (1, 2, 5)$ and $(1, 3, 5)$ as shown in part **B** of this figure. We are still missing the following six pairs : $(0, 5), (0, 6), (1, 4), (1, 6), (2, 6), (3, 6)$. These six pairs can be covered with the four test configurations shown in part **C**. This completes the covering array.

The authors present an optimal algorithm for vertical expansion and one for horizontal growth. The horizontal growth algorithm however has an exponential running time. Therefore they also provide an algorithm that is polynomial, but that may not produce an optimal result. This is the algorithm that is used in practice [98, 109].

One advantage of IPO is that it is deterministic. This translates to faster run times than the algorithms that employ randomness. The other advantage is that when one adds additional components to an already tested system, the old test configurations can be reused, since the algorithm simply expands the test suite horizontally and then vertically if needed.

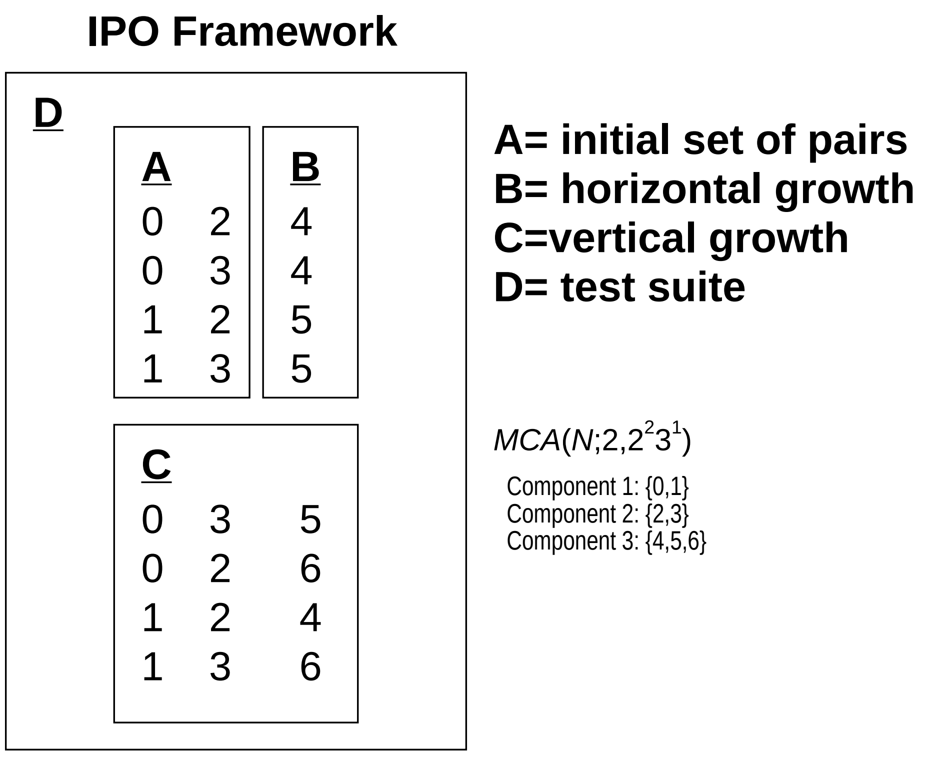


Figure 3.8: IPO framework illustrating horizontal and vertical expansion

3.4 Algebraic Tools

We briefly describe two computational tools that use algebraic and combinatorial recursive constructions. These are deterministic algorithms that use mathematical methods to build covering arrays. The first, TConfig, works only for fixed strength arrays of strength two. The second, CTS, is general in t and builds both mixed and fixed level covering arrays.

3.4.1 TConfig

TConfig was developed by Williams *et al.* [102]. It implements an algebraic construction to provide pairwise coverage. Several smaller building blocks are required, a *reduced array* and a *basic array* to implement this construction. We do not define these here, but comment that they are orthogonal arrays with certain rows removed and columns duplicated. This allows one to build a covering array as follows. Multiple copies of an orthogonal array are concatenated horizontally together. Then the reduced arrays and basic arrays are used to fill in the missing pairs. As in IPO, there is an idea of expanding horizontally and vertically, but with a slightly different result [102]. For a full description of this algorithm see ([102]). Results of TConfig compared with In Parameter Order showed that in many cases it improves upon those bounds [102]. We will include some results from TConfig in Chapter 4. However there is no extension for mixed level arrays for this algorithm.

3.4.2 Combinatorial Test Services

Another computational approach is to implement many algebraic methods and combinatorial recursive constructions as a knowledge base and select the best one for a particular set of parameters, t, k and v. This is the approach used by Hartman and Raskin in [53]. They have developed a C++ library called Combinatorial Test Services, *CTS*. The tool begins by analyzing the parameters and decides which is the "best" construction method. It includes methods to build covering arrays for any strength t as well as for both mixed and fixed level arrays. In addition, test case seeding is handled. This tool also provides a method to "avoid" specific configurations. All of the constructions are deterministic thereby making the time to build covering arrays minimal. However, only a subset of construction methods are included in this tool. This means that the results will sometimes be sub-optimal [53]. In addition, there are times when constructions do not produce the smallest arrays. This brings us to our next type of computational method, heuristic search.

3.5 Heuristic Search

Although greedy methods have been used successfully to find covering arrays, there are standard combinatorial optimization techniques that cannot be ignored. These are heuristic search techniques in which a "neighborhood" of solutions is explored to find the best fit. Once again these are approximation algorithms, but they have been used successfully for other similar types of problems [77, 79]. There is an emerging field of software engineering called *search based software engineering* that applies specialized search algorithms to a variety of software engineering problems [49]. Successful applications of these search techniques include evolutionary testing, program slicing, and structural testing.

Heuristic search algorithms are used commonly for problems that are NP-complete. They use heuristics and exploit randomness to search and narrow a large solution space. At each stage of the algorithm a move is made to the new local optimum. We describe one heuristic search technique next.

3.5.1 Hill Climbing

Hill climbing is a variant of the state space search technique for solving combinatorial optimization problems. With a general optimization problem the hope is that the found solution is close to an optimal one. With many design problems we *know* (from the cost) when we have reached an optimal solution. On the other hand, approximations in these cases are of little value.

An optimization problem can be specified as a set Σ of feasible solutions (or states) together with a cost $c(S)$ associated with each $S \in \Sigma$. An optimal solution corresponds to a feasible solution with overall (i.e. global) minimum cost. Sometimes hill climbing uses the maximum cost for its optimal solution (hence "hill climbing"), but here we stick to minimizing the cost. We define, for each $S \in \Sigma$, a set T_S of transformations (or transitions), each of which can be used to change S into another feasible solution S'. The set of solutions that can be reached from S by applying a transformation from T_S is called the neighborhood $N(S)$ of S.

We start by randomly choosing an initial feasible solution and then generate a randomly chosen transformation of the current feasible solution S. If the transformation results in a feasible solution S' of equal or lower cost, then S' is accepted as the new current feasible solution. If S' is of higher cost, then we reject this solution and check another randomly chosen neighbor of the current feasible solution.

This allows us to randomly *walk* around Σ, without reducing the goodness of our current solution. Hill climbing has the potential to get stuck in a local minimum (or to *freeze*), so stopping heuristics are required. To increase the chance of forming a good solution we repeat the random walk (or *trial*) a number of times, each time beginning with a random initial feasible solution.

In the hill climbing algorithm for constructing covering arrays the current feasible solution is an approximation S to a covering array in which certain t-subsets are not covered. The cost function is based on the number of t-subsets that are not covered, so that a covering array itself will have a cost of zero. A potential transformation is made by selecting one of the k-sets belonging to S and then replacing a random component value in this k-set by a random component value not in the k-set (i.e. we select a column and row from our $N \times k$ array and change the value of the component). The number of blocks (N) remains constant throughout the hill climbing trial.

Since N is fixed, but is unknown, we have used the method described by Stardom and Stevens [91, 92] to determine our array size. We set loose upper and lower bounds on the size of an optimal array and then use a binary search process to find the smallest covering array in this interval. Our lower bound is set initially to the mathematical lower bound for the parameters t and v. The upper bound is set to be “sufficiently large”. We then run many iterations of our hill climb. For each iteration we set N to be halfway between the current upper and lower bounds. The hill climb either finds a valid covering array or freezes. If it finds a valid array, then we want to search for a smaller one, so we set the upper bound equal to N. If the hill climb freezes then we change the lower bound to N. The process ends when the upper and lower bounds cross. An alternative method is to start with the size of a known test suite and search for a solution. This of course uses less computational resources, but the required test suite size must be known ahead of time. Ideally in a real system this is the method which we would use.

3.6 Meta-heuristic Search

One class of heuristic search algorithms employ higher level strategies to allow the algorithm to escape local optima [46]. These algorithms are collectively known as *meta-heuristic search* algorithms. The term *meta-heuristic* was introduced by Glover [47] in 1986 when describing *tabu search* (see Section 3.6.3). A meta-heuristic describes a strategy to guide and modify other heuristics [47]. These go beyond the normal heuristics that often get stuck in local optima. We explore a few of these algorithms next and discuss how they have been used to find covering arrays. One meta-heuristic search algorithm, simulated annealing, has been used by Nurmela and Östergård [77] to construct *covering designs* which have a structure very similar to covering arrays. We do not report their results because the objects they build are sufficiently different from covering arrays. We examine this algorithm next.

3.6.1 Simulated Annealing

Simulated annealing uses the same approach as hill climbing but allows the algorithm, with a controlled probability, to make choices that reduce the quality of the current solution. The idea is to avoid getting stuck in a bad configuration while continuing to make progress. If the transformation results in a feasible solution S' of higher cost, then S' is accepted with probability $e^{-(c(S')-c(S))/K_BT}$, where T is the controlling temperature of the simulation and K_B is a constant (the Boltzmann constant). The temperature is lowered in small steps with the system being allowed to approach "equilibrium" at each temperature through a sequence of transitions (or Markov chain) at this temperature. Usually this is done by setting $T := \alpha T$, where α (the *control decrement*) is a real number slightly less than 1. After an appropriate stopping condition is met, the current feasible solution is taken as an approximation to the solution of the problem at hand. Again, we improve our chances of obtaining a good solution by running a number of trials.

Stardom [91] has recently compared simulated annealing with other types of local search, such as tabu search and genetic algorithms, for finding covering arrays of strength 2. Stardom has reported several new upper bounds (including $CAN(2, 16, 8) \leq 113$ and $CAN(2, 18, 11) \leq 225$) using a simulated annealing algorithm. We compare some of our annealing results with those of Stardom in Chapter 4.

3.6.2 Great Deluge Algorithm

One further variant of hill climbing and simulated annealing is the *great flood* or *great deluge* algorithm first introduced by G. Dueck [43]. It is also termed *threshold accepting.* This follows a strategy similar to simulated annealing but often displays more rapid convergence. Instead of using probability to decide on a move when the cost is higher, a worse feasible solution is chosen if the cost is less than the current threshold. This threshold value is sometimes referred to as the *water level* which, in a profit maximizing problem, would be rising rather than falling (as is happening in this case). As the algorithm progresses, the threshold is reduced, moving it closer to the optimal cost. This algorithm has not been used for finding covering arrays and so it may provide some interesting new results.

3.6.3 Tabu Search

Tabu search has been used successfully by K. Nurmela [76] for finding strength 2 covering arrays. This has produced some new bounds such as $CAN(2, 20, 3) \leq 15$ and $CAN(2, 10, 7) \leq 63$. Stardom [91] also explored tabu search and determined that it works well when the neighborhood is small.

In tabu search, N is a set containing all neighbors of S that can be reached with one transition from the current solution S. C is the set of solutions that satisfy a particular selection criteria. For instance, C might be the set of solutions that contain a specific interaction t-set. The union of these sets is the neighborhood of interest. It is explored completely for the best solution. A *tabu* list, L, is also maintained. This is a list of all solutions that have been chosen within a given time T (i.e. a portion of the algorithm's history). The tabu list prevents the algorithm from getting stuck in an infinite loop.

At each step of the algorithm the intersection of sets N and C are explored and the best solution that is not found in L is chosen. In Nurmela's implementation S is an $N \times k$ array. The cost is the number of uncovered t-sets. A covering array has a cost of zero. N contains all arrays that can be found by changing one entry in S. At each step of the algorithm a random uncovered t-set is selected. C contains all arrays that cover the randomly selected t set. $N \cap C \setminus L$ contains the solutions to be explored. These are the arrays within one move of the current solution, that contain this t-set and are not "tabu". The move that will provide the largest decrease in the cost is chosen.

Most of the results presented by Nurmela are for small fixed level covering arrays of strength 2, but he does include two examples for mixed level arrays. We include these in our comparisons in Chapter 4.

3.6.4 Genetic Algorithms

Genetic algorithms model the biological evolutionary process. A population is composed of many individuals. In this case the population is a set of feasible solutions. Pairs of solutions (*parents*) are selected. A crossover and recombination stage take place and a set of children are formed. The fittest children from this population (i.e the ones that cover the most t-sets) are chosen and the process is repeated. Genetic algorithms have been very successful in search based software engineering [49]. Stardom [91] examined the effectiveness of simulated annealing, tabu search and genetic algorithms to find strength 2 covering arrays. In this work the genetic algorithms performed poorly against the other two. Given that this was a preliminary study, there is still room for exploration of this method for finding covering arrays. The use of genetic algorithms for finding covering arrays is open for future exploration.

3.7 Data Structures for Computational Methods

The algorithms presented in this chapter may contain some complex data structures. They all require efficiency in order to run in acceptable time frames. The heuristic and meta-heuristic search algorithms must make thousands or millions of transitions before they find a solution or are frozen. At the same time one would like to be able to encode these algorithms so that they are general in t. AETG already has this ability and as new methods for designing interaction test suites are developed it would be desirable to retain this ability. Furthermore, Chapter 5 presents a model for software testing where the need to vary t is imperative. The desire for this generality sometimes costs us efficiency.

We can use existing methods to handle these problems since these are common in combinatorial mathematics. We present the main ones we have used to increase our generality while maintaining efficiency here; ones that are relevant to the algorithms used in the following chapters. In addition, we present a data structure that helps the AETG component ordering heuristic work efficiently.

3.7.1 Ranking

If the goal when implementing one of these algorithms is to create a general algorithm that will work with any size t, then one can employ standard combinatorial techniques to store t-sets as integer values. This will avoid the problem of forcing structural looping constraints any time one wants to compute t-combinations. For instance when a new component is fixed in one of the greedy methods, the new t-sets it creates must be compared with all of the previously fixed components. This is a dynamic situation. Each time, the combinations to be tested must be computed, and all of them must be checked. Of course one can use fixed loops if we know we have only pairwise or only three way coverage, but this restricts the ability of the program to generalize. We also want to store our t-sets in some data structure that allows efficient access. This is especially important in a meta-heuristic search where this structure is accessed many times for each potential transition.

The simplest data structure to use is a t-dimensional array, with each dimension of size $k \times v$. The indices represent values of each component. We can store our component values as unique integers from 0 to $V-1$ where $V = \sum_{i=1}^{k} v_i$. This means, the task of determining if a specific t-set is covered can be done in constant time. However, a t-way array requires certain structural constraints. This is harder to implement in a dynamic environment unless limits are placed on the size of t. Instead we can use a technique called *ranking* from combinatorial mathematics. Ranks are integer values that correspond to the positions of t-sets in the lexicographic ordering of all t-sets [64]. Although we do not actually use *all possible* t-sets in this ordering for covering arrays, we can still use the ranks of the t-sets that are needed. We will return to this idea.

Suppose we have a set $V = \{0, 1..., |V| - 1\}$, where $|V| = \sum_{i=1}^{k} v_i$. There are $\binom{|V|}{t}$ possible t-subsets of this group of symbols. Name this set W. We can represent the members of W as a list. Represent each $S = s_1, ..., s_t \in W$ as an ordered list $[s_1, s_2, ..., s_t]$ where $s_1 < s_2 < ... < s_t$. The elements of W can now be written as a list ordered in increasing lexicographical order.

Table 3.9 is an example of a set of symbols $V = \{0, 1, 2, 3, 4, 5\}$ with t-sets of size 3. The table contains all possible t-subsets of these symbols, in lexicographical order.

Using this method we can represent each member of W as a unique integer value. If we try to map this table to the covering array, $CA(3, 3, 2)$, we can see that only a portion of the $\binom{|V|}{t}$ subsets are needed, since the covering array only examines interactions *between* components. Any t-sets that consist of more than one value from the same component will not be used. In this example there are $\binom{|V|}{t}$ or 20 ranks, but only 8 of these are used in the covering array. These t-sets are starred in Table 3.9.

Ranking, un-ranking (translation from an integer rank to a t-set) and successor (the next t-set in lexicographical order) algorithms can be found in [64]. By using

$V = \{0,1,2,3,4,5\}$,$t = 3$

S	Rank(S)
$\{0,1,2\}$	0
$\{0,1,3\}$	1
$\{0,1,4\}$	2
$\{0,1,5\}$	3
$\{0,2,3\}$	4
$\{0,2,4\}*$	5
$\{0,2,5\}*$	6
$\{0,3,4\}*$	7
$\{0,3,5\}*$	8
$\{0,4,5\}$	9
$\{1,2,3\}$	10
$\{1,2,4\}*$	11
$\{1,2,5\}*$	12
$\{1,3,4\}*$	13
$\{1,3,5\}*$	14
$\{1,4,5\}$	15
$\{2,3,4\}$	16
$\{2,3,5\}$	17
$\{2,4,5\}$	18
$\{3,4,5\}$	19

$CA(N;3,3,2)$
Component 1: $\{0,1\}$
Component 2: $\{2,3\}$
Component 3: $\{4,5\}$

Figure 3.9: Lexicographic ordering of t-sets

these we can generalize our search algorithms to work for any t. To store our ranks we use a one dimensional integer array that is of size $\binom{|V|}{t}$. The array index is the rank. In this array we store the number of times a t-set has been used. For instance, if we use the example from Table 3.9 and if the t-set $\{0,2,4\}$ occurs in exactly one test configuration, then the array value for index 5 of our array would contain a “1”. As this array is sparsely populated (see Figure 3.9), a more efficient storage structure can be developed to improve the usage of space. This inefficiency of space was chosen in our initial prototype implementations to favor the generality of the solution. The ranking algorithm which must invoked each time we determine if a t-set is covered requires $O(t \times v)$ time assuming that we can calculate $\binom{v}{t}$ in constant time (see Section 3.7.2 for details of this method). Since t is usually very small (e.g. 2-4) and this is a worst case bound, the increase in computational complexity does not seem to degrade the performance of our algorithms. This is a situation where we have chosen to accept some loss of efficiency for the benefit of generality.

The use of ranking is not limited to storing our t-sets. We can use this anytime we need to perform a function that would normally require different looping structures depending on the size of t. For instance if we want to calculate all of the t-sets covered in a single test configuration (an array of size k), we have $\binom{k}{t}$ combinations of locations (indices) in this array. If one is using a fixed t then this can be done using nested loops. However, we want to keep our code general. The first t locations of this array (i.e. $\{0,1\}$ when $t = 2$, or $\{0,1,2\}$ when $t = 3$) are equivalent to a rank of 0. We can loop through all $\binom{k}{t}$ combinations by repeatedly applying the successor algorithm. The successor algorithm returns a t-set representing the next subset in our lexicographical order. The time complexity of the successor algorithm is $O(t)$. Since t is generally very small, we have chosen to use the successor method for generality. We use the elements of this subset as indices of our test configuration. All ranks between 0 and $\binom{k}{t}-1$ are used (unlike the ranks for interactions).

When we use the ranking algorithms to determine the change of cost (delta), as happens whenever a new value for a component is added (in the greedy algorithms), or when a component value is changed (in hill climbing or simulated annealing), we can make use of the general nature of these algorithms to avoid unecessary computations. To begin with, we always restrict our computation to that of delta (instead of recalculating the entire cost each time). Since we are only changing an individual component we do not need to calculate the delta in this test configuration by computing *all* t-sets containing this element. This would require us to compute the rank and perform other computations $\binom{(k-1)}{t}$ times. Instead we can compute all of the t-1 subsets that *do not* contain this component and then combine the new component value with each of these to create the changed t-sets. This requires only $\binom{k-1}{(t-1)}$ computations. When k gets large this has potential to save a lot of computational time.

3.7.2 Other Utility Structures

Further combinatorial methods can be used in order to increase efficiency. Nurmela *et al.* [77] provides an algorithmic method to build a binomial lookup table which can then be used any time we need to determine the value of $\binom{n}{r}$. We have used this method to pre-compute the binomial values for $\binom{t}{v}$ ahead of time. This is done once each time the algorithm starts. It uses the input parameters t and v to determine the size of the table. It gives us constant time access during the rest of the algorithm for any $\binom{n}{r}$. In the simulated annealing implementation, one can also employ the method used by Nurmela *et al.* [77] for storing an approximation of exponential values as a table. We pre-compute the values of e^x for a large range of integer values $(0, 1, ..., r-1)$ at the start of our program. We store these in a $1 \times r$ array. When we need to find e^x we cast x to an integer value and access the xth element of the array. This operation takes constant time. Rounding to the nearest integer slightly alters the probability of making an uphill move, but given the heuristic nature of this algorithm, and the large effect that varying parameters of the cooling schedule has, this change has an insignificant effect on performance.

3.7.3 Structures for Least Used t-sets

As part of the greedy framework described, AETG uses a heuristic to select a component ordering. The first component and its value is selected as a component-value combination that occurs in the most uncovered t-sets. It does this for every one of the M test candidates for each new test configuration. Since this is done many times in a run of the AETG algorithm, especially if we use multiple repetitions of the entire algorithm, then a method for efficiently handling this is needed. Although this structure is specific to the AETG algorithm at present, an expansion and exploration of the greedy framework might use this component ordering heuristic in combination with other decision points, so it is worth exploring.

In AETG, the set of *least used* requires an efficient structure so that it is easy to

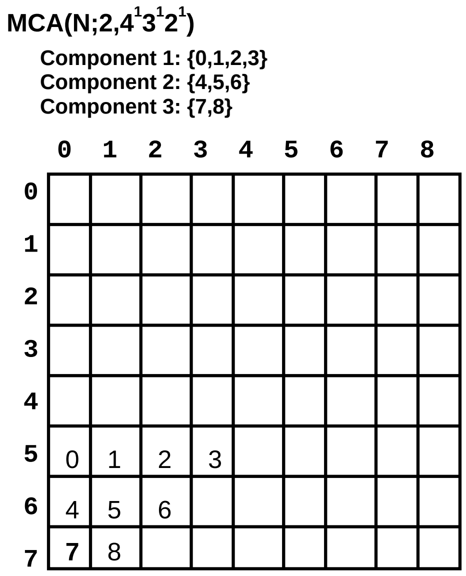


Figure 3.10: Initial state of *Used*

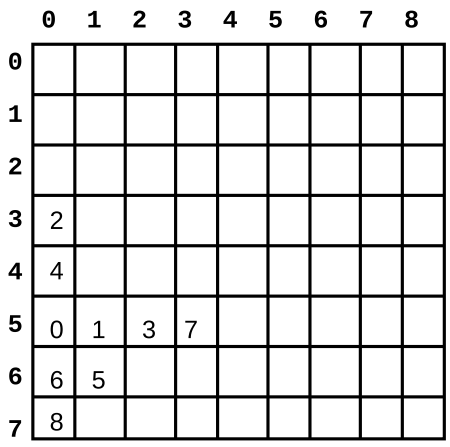


Figure 3.11: State after test configuration (2 4 7) committed

keep track of the t-sets and to randomly select one from among this set each time. Of course updating this structure must also be fast and efficient.

At the start of the algorithm we compute how many times a symbol will occur in a t-set. In a mixed level array this is not the same for each value. For instance if we have an $MCA(2, 3^2 2^2)$, the values in components with 2 values will occur in 8 pairs, while the ones in components with 3 values will occur in 7. We calculate the maximum times each value will occur, max, and then create a $(max + 1) \times V$ array called *Used* where $V = \sum_{i=1}^{k} v_i$. We fill in each row i of *Used* with the symbols that have i t-sets left to cover. In the previous example row 8 would contain all the values from the 2 components with 3 values and row 7 would contain the others. We keep track of max, which changes as the algorithm progresses. At the end of the algorithm $max = 0$. To make things easier, we also maintain information for each value of a component so that we know its location in the array.

At each point when a new symbol is needed from *least*, we randomly select one of the symbols from the max row. As long as we know how many values are in

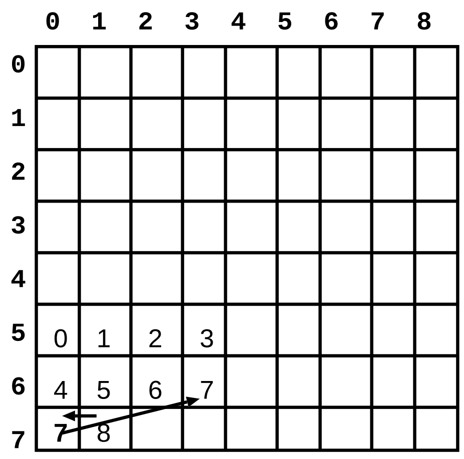


Figure 3.12: First step in update for (2,4,7)

the *max* row, we can do this in constant time. In order to update a value (this is done each time it is used in a previously uncovered t-set) we percolate the values up to the correct rows. At the end of the algorithm, all of the symbols are in row 0. Figures 3.10 and 3.11 shows this structure at two points in the algorithm. In Figure 3.10, representing the first stage, the algorithm is just starting and the values are all unused. After the first test configuration is committed (Figure 3.11) the symbols move to their new location. At the next point in this algorithm the component value 8 is chosen since this is currently the only value in the *max* row (row 7).

The *percolate* step can be completed in constant time for each update. To percolate we move the symbol to the end of the previous row and update its new position. This is shown in Figure 3.12. Assume that the symbol 7 has been used in a new t-set and that we need to update it. It moves to the end of the previous row. Next we replace the symbol that was moved with the *last* symbol in its original row. In this case the 8 will move to the cell that contained 7. If 7 had been the last symbol in this row, then they value for *max* would have been decremented and this row would no longer be used. In order to implement this in constant time we use data structures to keep track of each symbol's column and row location in this array as well as the length of each row and the current maximum row. We have chosen to use more space as a trade-off for time efficiencies.

In this implementation the components that have the least number of values (in a mixed level array) are actually fixed first at the start of the algorithm, since they are the ones that will be involved in the most t-sets. In TCG this would never occur since the fixed component ordering always selects the component with highest cardinality first. Further exploration of the framework may lead to more interesting combinations of these two heuristics.

3.8 Summary

In this section we have presented computational techniques to find covering arrays. We presented a general framework for several greedy algorithms. We presented one other greedy algorithm from the literature and mention some tools that use algebraic constructions. We then explored some heuristic and meta-heuristic search algorithms. We ended with a discussion of some data structures and algorithms that are useful when implementing these. The next chapter compares implementations of some of these algorithms.

Chapter 4

Comparison of algorithms

In the previous chapter we explored a variety of computational methods for building covering arrays. It is natural to ask how the various methods compare with each other and how they compare with algebraic constructions. As our ultimate aim is a toolkit for testers then other important questions arise as well. Which methods are best suited for particular types of arrays? Do certain methods work better for fixed versus mixed level arrays or for large versus small numbers of components and values? Do these answers change when we move to a higher strength array? What are the computational costs for the various algorithms?

As a first step in exploring these questions we have implemented our own versions of two of the greedy algorithms from the framework presented in Chapter 3, and have implemented a hill climbing and simulated annealing algorithm. We compare results from these with some of the bounds published in the literature across a range of fixed and mixed level arrays of strength two and three. Results from this chapter are published in [27].

4.1 Introduction

A variety of computational techniques exist to produce covering arrays. Often, the results that are presented provide new bounds, but are restricted to experiments using only a single technique. There are very few studies that compare results for both mixed and fixed level arrays. Furthermore most of the results using computational methods are limited to the case where $t = 2$. This chapter attempts to address these issues by examining a variety of algorithms across a broad range of covering arrays.

We have selected one heuristic search algorithm, *hill climbing*, and one meta-heuristic search algorithm, *simulated annealing*. We compare these with our own implementations of two greedy algorithms, AETG and TCG. We call these implementations *mAETG* and *mTCG* in order to distinguish them as our own implementations. We chose simulated annealing because it has been shown to work well for fixed level strength two arrays [91, 92]. Hill climbing was chosen for its simplicity and similarities to simulated annealing. We selected AETG and TCG for comparisons since they can be used with both mixed and fixed level arrays and are variations on the same framework. We compare these with the real implementations of AETG, TCG and IPO as reported in the literature and with results of algebraic constructions. We have also included some results reported by Hartman *et al.* in [53] for CTS and TConfig, for tabu search by Nurmela[76] and from the tool created by James Bach called ALLPAIRS [3]. We examine both fixed and mixed level arrays of strength two and three.

We begin with a description of each of our algorithms and then present the experimental results.

4.2 Hill Climbing

We have implemented hill climbing for t-way coverage. We use methods described in Chapter 3 to store t-sets as integer ranks. Our initial array consists of a random set of values for each component. We use a binary search technique to find the best N, but do not re-randomize our array between iterations of each hill climb. For each trial of the hill climb we randomly select a column and a row in the covering array. A random value is chosen for the component that corresponds to this column of the array. If changing this value will give us a cost that is equal or less than the current cost we move to the new solution. In our initial version of a hill climb we set our "frozen" counter quite high (100,000). This was done to prevent the program from terminating before it found a good solution.

4.3 Simulated Annealing

The simulated annealing algorithm uses the same data structures as the hill climbing algorithm. However, we set frozen to be much lower (500). In simulated annealing we have found that using a starting temperature of 0.20 with a cooling factor of 0.9998 works very well. For harder problems a slower cooling (0.99999) with a starting temperature of 0.030 is sometimes employed. We cool every two to three thousand moves. There is a balance needed in the slowness of cooling and the starting temperature. If we start at a temperature that is too high, then we move to bad solutions much more often and take longer to find an equilibrium. If we cool too quickly, we do not allow the algorithm to find its way out of local optimums. The idea is to keep a small probability of making a bad move for as long as possible.

4.4 mTCG

mTCG only supports pairwise coverage, but works for both fixed and mixed level covering arrays. We store the counts for our t-sets in this implementation as a $v \times v$ array. This provides fast access, but cannot easily be generalized.

For our version of the mTCG algorithm, a few minor refinements were applied. The algorithm in [110] does not fully describe how ties are handled. To handle this problem there are two places in the framework description for TCG (see Algorithm 3) where it states to use *either* a random selection or lexicographical order. For our implementation we have selected *random* for both cases. When choosing a value to add to an individual test configuration, TCG uses a heuristic. It selects the value that has been used the least number of times so far. However, it is possible to have a tie for values that satisfy this heuristic, i.e. a second level tie. Therefore we needed to implement a tie breaking strategy for this scenario. We chose to do this randomly in our version of the algorithm. The other place that a tie can occur is when more than one of the M candidate test configurations produces the most new pairs. We again chose a random tie-breaking scheme. Since we have added randomness to our algorithm, we used a series of repeated runs, only keeping the smallest covering array produced. We have also tightened the definition of *least used* to count a symbol as being used *only* in the case when it contributes to a new uncovered interaction pair.

4.5 mAETG

The mAETG algorithm uses standard combinatorial techniques to store a t-set as a rank. This means that the algorithm can handle arbitrary t-way coverage. Although the implementation of the mAETG algorithm is as described in the literature [23], we acknowledge that the actual commercial product is patented and may include some simple construction techniques as well as post-processing stages. These are not included in our implementation.

The real AETG always returns the same size covering array for a given set of parameters. This might be due to post processing or it might be because a library of known good random seeds is used. The exact method is not published. In our implementation we use multiple repeats of the algorithm and select the smallest one. One additional heuristic has been added in our version of the AETG algorithm, for the case of $t > 2$. In the algorithm described in [23], it is unclear what happens when choosing the ordering of components between 2 and $t-1$. The first component-value pair is always chosen as one of the values found in the most uncovered t-sets. We maintain this first step, then continue to choose a component-value pair from the same set until we have the first $t-1$ symbols fixed. We then continue to follow the algorithm as stated and use a random order for the rest of the components.

Our results for $t = 3$ seem to improve on those reported in the literature for the commercial AETG program (see Table 4.4). This may be attributed partially to this enhancement.

4.6 Experiments

We have implemented the mTCG algorithm for pairwise coverage and the mAETG algorithm for t-way coverage. In addition we have implemented a hill climbing and simulated annealing program for building t-way covering arrays. All of the programs are written in C++ and run on Linux using an INTEL Pentium IV 1.8 GHZ processor with 512 MB of memory. We are interested in the smallest value of N obtained by each algorithm although other metrics could be explored.

All of our algorithms build both fixed and mixed level arrays, but we have emphasized the fixed level cases in our reporting in order to make comparisons with results in the literature. We have selected the subset of covering arrays to use in our experiments to match those that have been used in other experimental papers.

The mTCG algorithm uses 5,000 repeats keeping only the best array at the end. All 5,000 iterations were included in the total time reported in Table 4.5. The results for mAETG use 300 repeats. Again we report only the smallest array found at the end. All of these iterations were counted as part of the total time reported in Table 4.5. We selected the number of repeats for each of these two algorithms so that they consistently gave us the closest results to those reported in the literature for the real algorithms.

4.7 Results

For all of the algorithms we ran a series of trials, but report only the best test suite obtained (the one with the smallest number of rows). We use the abbreviations *HC* for hill climbing, *SA* for simulated annealing, *TS* for tabu search and *AP* for

	Minimum Number of Test Configurations in Test Suite						
	TCG[1]	AETG[1]	AP	mTCG	mAETG	HC	SA
$MCA(2, 5^1 3^8 2^2)$	20	19	21	18	20	16	**15**
$MCA(2, 7^1 6^1 5^1 4^5 3^8 2^3)$	45	45	53	**42**	44	**42**	**42**
$MCA(2, 5^1 4^4 3^{11} 2^5)$	30	30	33	25	28	23	**21**
$MCA(2, 6^1 5^1 4^6 3^8 2^3)$	33	34	39	32	35	**30**	**30**

Table 4.1: Comparisons for 2-way coverage
1. Source = Yu-Wen et al.[110]

	Minimum Number of Test Configurations in Test Suite								
IPO[1]	AETG[1]	TConfig[2]	CTS[2]	AP	TS[3]	mTCG	mAETG	HC	SA
$CA(2, 4, 3)$									
9	9	9	9	10	NA	9	9	9	9
$CA(2, 3^{13})$									
17	**15**	**15**	**15**	22	NA	17	17	16	16
$MCA(2, 4^{15} 3^{17} 2^{29})$									
34	41	40	39	41	**29**	34	37	30	30
$MCA(2, 4^1 3^{39} 2^{35})$									
26	28	30	29	30	**21**	26	27	**21**	**21**
$CA(2, 100, 4)$									
53	NA	**43**	**43**	52	NA	56	56	47	45
$CA(2, 20, 10)$									
212	**180**	231	210	230	NA	213	198	189	183

Table 4.2: Comparisons for 2-way coverage
1. Source = Tai *et al.*[98]
2. Source = Hartman *et al.* [53]
3. Source = Nurmela [76]

ALLPAIRS [3] in Tables 4.1-4.4. The best known bounds are shown in bold in these tables and subsequent tables.

Table 4.1 compares our results with those reported by Yu *et al.* in [110]. mTCG and mAETG produce similar results to those reported. In all of the four test suites, however both hill climbing and simulated annealing algorithms improve on the bounds given by these other algorithms. The hill climbing and simulated annealing algorithms both produced similar lower bounds, but in our experimentation we found that quite often the simulated annealing produced these in fewer trials. More experimentation is needed here.

Table 4.2 compares our results with those reported by Tai *et al.* for IPO in [98] and for CTS and TConfig reported by Hartman *et al.* [53] as well as those obtained from ALLPAIRS [3]. We also include a few results found by tabu search (labeled TS) as reported by Nurmela [76]. It should be noted that the results for IPO vary slightly between the two papers [98, 109]. We have chosen to use the results from the journal version of this paper for consistency [98]. For these examples our algorithms produce arrays at least as small as those produced by the IPO algorithm. In two cases the reported commercial AETG results are smaller than ours. For the first of these arrays, $CA(2, 3^{13})$, there is clearly a good algebraic construction since both CTS and TConfig have the same bound as AETG of 15. It is possible that the real AETG tool uses this construction also. We have results for only two of these covering arrays using tabu search, but for one array, $MCA(2, 4^{15}3^{17}2^{29})$, tabu search provides the smallest bound of any method and for the other it matches the smallest bound found by both simulated annealing and hill climbing.

Simulated annealing consistently does as well or better than hill climbing, so we report only those results for the next two tables. Table 4.3 compares results for some fixed level arrays reported by Stardom in [91]. We have also included some results from CTS for a comparisons with algebraic methods. The arrays we have included in this table are ones for which Stardom reported a new bound. These are arrays where there may not be an optimal algebraic construction. Therefore it is not surprising that CTS does not perform as well for these arrays. In each case our annealing program has improved upon his results. The results in [76] for tabu search do not report arrays of this size. Since the results in [91] were obtained from a similar algorithm, we attribute this to the need for better tuning of the annealing program parameters. We have not yet fine tuned the cooling schedule which plays a role in the quality of the final results. This is open for future work.

Table 4.4 compares our results against some known strength-three algebraic constructions reported by Chateauneuf *et al.* in [17]. For these arrays the expectation was that it would be difficult to match the known results. One surprise was that the mAETG algorithm found consistently smaller arrays than those reported using

	Minimum Number of Test Configurations in Test Suite				
	Stardom[1]	mAETG	SA	CTS[2]	AP
$CA(2,16,6)$	65	70	**62**	88	80
$CA(2,16,7)$	88	94	**87**	91	110
$CA(2,16,8)$	113	120	**112**	120	128
$CA(2,17,8)$	116	123	**114**	120	128

Table 4.3: Comparisons for 2-way coverage

1. Source = Stardom[91]
2. Source = Hartman *et al.* [53]

the commercial AETG product. This may be due to the additional heuristic added when choosing the first $t-1$ symbols for each test suite.

As expected, our simulated annealing algorithm did not perform as well as most of the algebraic constructions. In the case of a $CA(N;3,6,6)$, however, we have found a smaller array using simulated annealing. Further experimentation is needed with a more refined algorithm. Of course, in many cases constructions are not known (or may not exist) which is true in the last two entries of this table. For these arrays, simulated annealing finds an *optimal* solution. There are very few known constructions for mixed-level covering arrays. Therefore a fixed level array of a larger size would need to be constructed and used in its place. This may require more test configurations than a mixed level array found by computational search. Additionally, a real test suite may include special seeded test configurations that are required regardless of the interaction coverage. Once again current constructions do not handle this issue. We address this in Chapter 6.

Sample performance results for the mTCG, mAETG and simulated annealing algorithms are included to give a flavor for the time required to run each of these. Table 4.5 presents run times of our three algorithms for a selection of arrays above. Variation in the run times of these algorithms depends somewhat on program parameter settings such as cooling temperature and the value for *frozen*, as well as the number of iterations performed. The performance results presented for simulated annealing reflect the total time taken to find all arrays through a binary search pro-

	Minimum Number of Test Configurations in Test Suite				
	Construction[1]	AETG[1]	CTS[2]	mAETG	SA
$CA(N; 3, 6, 3)$	**33**	47	45	38	**33**
$CA(N; 3, 6, 4)$	**64**	105	**64**	77	**64**
$CA(N; 3, 6, 5)$	**125**	NA	**125**	194	152
$CA(N; 3, 6, 6)$	305	343	342	330	**300**
$CA(N; 3, 6, 10)$	1331	1508	**1330**	1473	1426
$CA(N; 3, 7, 5)$	**185**	229	225	218	201
$MCA(N; 3, 3^2 4^2 5^2)$	NA	NA	NA	114	**100**
$MCA(N; 3, 10^1 6^2 4^3 3^1)$	NA	NA	NA	377	**360**

Table 4.4: Comparisons for 3-way coverage

1. Source = Chateauneuf *et al.*[17]
2. Source = Hartman *et al.* [53]

	CPU User Time in Seconds		
	mTCG	mAETG	SA
$MCA(2, 5^1 3^8 2^2)$	6	58	214
$MCA(2, 7^1 6^1 5^1 4^5 3^8 2^3)$	57	489	874
$MCA(2, 5^1 4^4 3^{11} 2^5)$	33	368	379
$MCA(2, 6^1 5^1 4^6 3^8 2^3)$	42	376	579
$CA(2, 20, 10)$	1,333	6,001	10,833
$CA(3, 6, 6)$	NA	359	13,495

Table 4.5: Comparisons of run times

cess. Therefore the numbers reported in Table 4.5 may be reduced if tighter bounds are used as a starting point.

Clearly mTCG uses the least CPU time, while simulated annealing consumes the most. The hardest problem presented ($CA(3, 6, 6)$) took simulated annealing a little under 4 hours, but it produced a previously unknown bound. It highlights again the need for a balance between the cost of building the test suite and the cost of running the tests.

4.8 Summary

We have presented a comparison of greedy algorithms, heuristic and meta-heuristic search and algebraic constructions for covering arrays of strength two and three. Preliminary results on hill climbing and simulated annealing for mixed level covering arrays are presented which suggest they provide tighter bounds than their greedy counterparts. Their run times are however longer. In most cases algebraic methods were best, when they existed, but we did find one new bound using simulated annealing. More experimentation and tuning of these algorithms is required. In addition, performance measures need to be applied to determine the overall usefulness of each approach.

Chapter 5

Variable Strength Arrays

In many situations pairwise coverage is effective for testing [23]. However, we must balance the need for stronger interaction testing with the cost of running tests. For instance a $CA(N; 2, 5, 4)$ can be achieved with as little as 16 test configurations, while a $CA(N; 3, 5, 4)$ requires at least 64 test configurations. In order to appropriately use our resources we want to focus our testing where it has the most potential value. This chapter presents variable strength interaction testing. We develop the concept, describe a model to formalize our discussion, present some successful construction methods and provide initial upper bounds for these objects. The results from this chapter are published in [26, 27].

5.1 Introduction

The recognition that all software does not need to be tested equally is captured by James Bach in the concept of risk-based testing [4]. Risk-based testing prioritizes testing based on the probability of a failure occurring and the consequences should the failure occur. High risk areas of the software are identified and targeted for more comprehensive testing.

The following scenarios point to the need for a more flexible way of examining interaction coverage.

- We completely test a system, and find a number of components with pairwise interaction faults. We believe this may be caused by a bad interaction at a higher strength, i.e. some triples or quadruples of a group of components. We may want to revise our testing to handle the “observed bad components” at a higher strength.

- We thoroughly test another system but have now revised some parts of it. We want to test the whole system with a focus on the components involved in the changes. We use higher strength testing on certain components without ignoring the rest.

- We have computed software complexity metrics on some code, and find that certain components are more complex. These warrant more comprehensive testing.

- We have certain components that come from automatic code generators and have been more/less thoroughly tested than the human generated code.

- One part of a project has been outsourced and needs more complete testing.

- Some of our components are more expensive to test or to change between configurations. We still want to test for interactions, but cannot afford to test more than pairwise interactions for this group of components.

While the goal of testing is to cover as many component interactions as possible, trade-offs must occur. In his chapter we present one method for handling *variable interaction strengths* while still providing a base level of coverage for the entire system. We define the variable strength covering array, provide some initial bounds for these objects and outline a computational method for creating them.

5.2 Example Variable Strength System

In a real test environment, one would like the flexibility to offer a guarantee that *subsets* of components have higher interaction strengths than the whole system. As defined, a covering array only guarantees a coverage of t-subsets. Instead we may also want coverage of some subsets of size t' for values of $t' > t$.

Restricting testing to pairwise coverage does not guarantee that faults due to three or four-way interactions will be found. A trade off has to occur between the time and cost of testing and the required strength of guaranteed coverage. Williams *et al.* [105] describe a method to quantify the coverage for a particular interaction level. One can determine how many pairs, or n-way interactions are covered at each stage when building a test suite. For instance if there are four components, any new test configuration can contribute at most $\binom{4}{2}$, or 6 new covered pairs. Further, if each component has three values, there are a total of $\binom{4}{2}\ 3^2 = 54$ possible pairs that must be covered. Therefore any one new test configuration increases coverage by at most 11.1% [105]. A similar method is described by Dunietz *et al.* [44] who point out that although we guarantee 100% two-way coverage by using a covering array we are getting partial higher strength coverage as well.

As it is usually too expensive to test all components using three or four-way coverage there is a benefit from doing this for part of the system. For instance, a particular subset of components may have a higher interaction dependency or a certain combination of components may have more serious effects in the event of a failure. Consider a subset of components that control a safety-critical hardware interface. We want to use stronger coverage in that area. However, the rest of our components may be sufficiently tested using pairwise interaction. We can assign a coverage *strength* requirement to each subset of components as well as to the whole system.

For the safety critical system, we might require that the whole system has 100% coverage for two-way interactions, while the safety-critical subset has 100% coverage for four-way interactions. The final test suite may, however, have 60% coverage for four-way interactions over all components.

Component			
RAID Level	**Operating System**	**Memory Config.**	**Disk Interface**
RAID 0	Linux	128 MB	Ultra 160-SCSI
RAID 0	Novell	128 MB	Ultra 160-SCSI
RAID 5	Linux	64 MB	Ultra 160-SCSI
RAID 1	XP	128 MB	Ultra 160-SCSI
RAID 0	Novell	256 MB	Ultra 320
RAID 0	XP	128 MB	Ultra 160-SCSI
RAID 1	Novell	256 MB	Ultra 160-SCSI
RAID 0	Linux	64 MB	Ultra 320
RAID 5	XP	256 MB	Ultra 320
RAID 0	XP	64 MB	Ultra 160-SATA
RAID 5	Novell	128 MB	Ultra 320
RAID 5	XP	128 MB	Ultra 160-SATA
RAID 1	Linux	256 MB	Ultra 160-SCSI
RAID 5	XP	64 MB	Ultra 160-SATA
RAID 0	XP	256 MB	Ultra 160-SATA
RAID 1	Linux	128 MB	Ultra 320
RAID 0	Novell	64 MB	Ultra 320
RAID 1	Novell	64 MB	Ultra 160-SATA
RAID 1	Linux	64 MB	Ultra 160-SCSI
RAID 5	Novell	64 MB	Ultra 160-SCSI
RAID 1	XP	256 MB	Ultra 160-SATA
RAID 1	Novell	128 MB	Ultra 160-SCSI
RAID 1	XP	64 MB	Ultra 160-SATA
RAID 5	Linux	256 MB	Ultra 320
RAID 5	Linux	128 MB	Ultra 320
RAID 5	Novell	256 MB	Ultra 160-SCSI
RAID 0	Linux	256 MB	Ultra 160-SATA

Table 5.1: Variable strength array for Table 1.1

We return to the integrated RAID controller software described in Chapter 1. The requirement is to test all pairwise interactions. Perhaps, however, it is known that there are more likely to be interaction problems between the three components, *RAID level*, *operating system* and *memory configuration*. One may want to test these interactions more thoroughly than the rest. It may be too expensive to run tests involving all three-way interactions among all components. In this instance we can

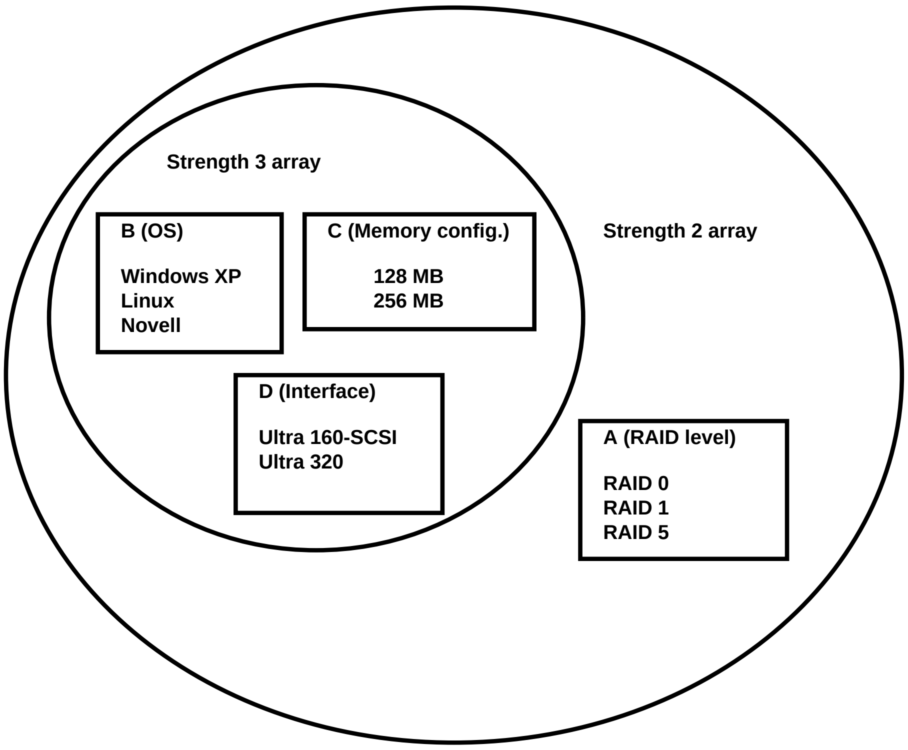


Figure 5.1: Model of variable strength covering array

obtain three-way interaction testing of the first three components while maintaining two-way coverage for the rest. We still have a minimal three-way coverage guarantee across all the components but do not need to use 81 test configurations required for testing *all* three-way interactions. The test suite shown in Table 5.1 shows a test suite with this specification using only 27 test configurations. The first three components (Raid level, OS and memory configuration) contain all three-way interactions, while the entire system has three-way coverage.

5.2.1 A Theoretical Framework

In order to address this type of problem we present a new model for interaction testing, illustrated in Figure 5.1. In this example (a modified version of the RAID example) we have four components (A,B,C,D). Component A (RAID level) and B (operating system) have three levels each while component C (memory configuration) and D (disk interface) have two. The system has a total of 37 interaction pairs,

Component			
A	**B**	**C**	**D**
RAID 0	Linux	256 MB	Ultra 160-SCSI
RAID 1	XP	256 MB	Ultra 320
RAID 2	XP	128 MB	Ultra 160-SCSI
RAID 0	Novell	128 MB	Ultra 320
RAID 1	Linux	128 MB	Ultra 160-SCSI
RAID 2	Linux	256 MB	Ultra 320
RAID 2	Novell	256 MB	Ultra 320
RAID 1	Novell	256 MB	Ultra 160-SCSI
RAID 0	XP	128 MB	Ultra 320
RAID 0	XP	256 MB	Ultra 160-SCSI
RAID 2	Linux	128 MB	Ultra 320
RAID 1	Novell	128 MB	Ultra 160-SCSI

Table 5.2: Variable strength array for Figure 5.1

and 60 interaction triples. We require that the subset $\{B, C, D\}$ has three-way coverage, while the entire system $\{A, B, C, D\}$ is has two-way coverage. We require 100% two-way coverage and a minimal three-way coverage of $\frac{12}{60} = 20\%$. As we shall see, the actual three-way coverage of a test suite may be much higher. In this example we show just one subsystem of higher coverage, but in general the model may have multiple subsystems.

A mixed level covering array to handle 100% of two-way interactions in the example from Figure 5.1 can be created using only nine tests. This is shown as the first nine rows of Table 5.2. To handle all three-way interactions, as many as 18 tests are needed. To increase the strength of the strength two array to cover all three-way interactions of $\{B, C, D\}$, we need to add only three more tests for a total of 12. These are the last three rows shown in Table 5.2. This actually covers $\frac{42}{60} = 70\%$ of the three-way interactions for our system while covering all of the three-way interactions for our subsystem.

Gargano, *et al.* [45] were the first to describe a variable strength array. They limit their discussion to the special case in which the whole system has strength zero and the smaller subsets have strength two. In [23] D. Cohen *et al.* present the idea of a hierarchical system for t-set coverage in AETG, but do not formalize a model for this. They also do not elaborate on construction methods other than to suggest an approach using the stronger subset as a seed. A software tool for incorporating variable strength coverage was developed at IBM by Ram Biyani [52]. The tool is called *Test Optimizer for Functional Usage* or *TOFU.* It allows one to select the strength of coverage for different subsets of components. TOFU is currently only available for internal use at IBM [52]. We devote the rest of this chapter to developing these ideas further.

5.2.2 An Ad Hoc Method

Most software interaction test generators provide only a fixed level of interaction strength. We might use this to build two separate test suites and run each independently, but this is a more expensive operation and does not really satisfy the desired criteria. We could instead just default to the higher strength coverage with more tests which in some cases may be the best option, but this leads to larger test suites. The ability to tune a test suite for specific levels of coverage is highly desirable, especially as the number of components and levels increases. Therefore it is useful to define and create test suites with flexible strengths of interaction coverage and to examine some methods for building these.

The variable strength covering array model in Figure 5.1 can be viewed as a collection of covering arrays contained inside a larger covering array. We can begin by building each individual array separately and then combine these in order to gain the additional coverage needed for the whole system. Or we can begin with a covering array for the whole system and alter it to obtain the higher strength coverage required for the designated component subsets.

We outline an ad hoc construction here for the example given in Table 5.2. In order to construct the variable strength array we use the mAETG algorithm to find a strength two covering array. Any of the computational methods described in

Chapter 3 can be used for this step. We then try to remove duplicate pairs in a way that leads to new, uncovered triples. For instance, in Table 5.2 we can see that had the last row of the two-way coverage been (RAID 0, XP, 128 MB, Ultra 160-SCSI) instead of (RAID 0, XP, 128 MB, Ultra 320), it could have been changed to (RAID 0, XP,128 MB, Ultra 320) with the resulting system still providing pairs (RAID 0, Ultra 160-SCSI), (XP, Ultra 160-SCSI), (128 MB, Ultra 160-SCSI) (found in rows $0, 2$ respectively). Thus we could replace the Ultra 160-SCSI with Ultra 320 without reducing our pairwise coverage, while at the same time covering a new triple. This step could be performed using a simulated annealing algorithm. Since all of the interactions between component A and all other components have been covered in the first part of the array, we can choose any valid values for that component in the last three test configurations. In this example we used a random selection.

An alternate way to approach this is to begin by enumerating all of the interactions for the higher strength subsets. For instance, we could build a mixed level covering array of strength three for components B, C and D and then fill horizontally by adding a column and the symbols needed so that all missing pairs are covered.

One further option is to simply extend the simulated annealing algorithm to compute the cost as a function of both levels of interaction and build the suite directly in that manner. Since annealing has worked well for fixed strength arrays this is our choice of construction from now on.

Table 5.3 provides some preliminary test suite sizes for variable strength arrays with two different strengths using simulated annealing. In this table we have restricted ourselves to a single subset of higher interaction strength, but we require a more flexible model. In the next section we address this. We present a notation for variable strength arrays. It defines interaction test suites that have multiple disjoint subsets of higher interaction strength. The best method for building a variable strength test suite is still open and the discovery of good algorithms and constructions for these is an interesting problem. We discuss this problem further in the following sections.

Variable Strength Arrays		
Base Array	Subset of Higher Coverage	Test Suite Size
$CA(N;2,11,6)$	$CA(N;3,6,6)$	302
$CA(N;2,20,4)$	$CA(N;3,9,4)$	125
$CA(N;2,10,3)$	$CA(N;3,6,3)$	33
$MCA(N;2,5^1 4^4 3^{11} 2^5)$	$MCA(N;3,5^1 4^2 3^5)$	80
$MCA(N,2,3^5 4^{10} 5^2)$	$MCA(N;3,3^2 4^2 5^2)$	100

Table 5.3: Test suite sizes for variable strength arrays using simulated annealing

5.3 A Model for Variable Strength Arrays

Definition 5.3.1 *A variable strength covering array, denoted as a $VCA(N;t,(v_1,v_2,..,v_K),C)$, is an $N \times K$ mixed level covering array, of strength t containing C, a vector of covering arrays each of strength $> t$ and defined on a subset of the K columns.*

We add some restrictions to this model as an initial investigation step. We will require that our subsets of columns are disjoint (see Figure 5.1). The more general model (see Figure 5.2 on page 98) allows for overlapping subsets of columns, but we leave the construction of examples of this type as future work. In order to simplify the discussion for the specific covering arrays presented, we reorder the components of the array so that the disjoint subsets are consecutive from left to right. This ordering is restricted to the representation of the array. The actual column ordering of the array is arbitrary.

An example of a $VCA(27;2,3^9 2^2,(CA(3,3^3),CA(3,3^3),CA(3,3^3)))$ can be seen in Table 5.4. The overall array is a mixed level array of strength two with nine components containing three symbols and two components containing two. There are three sub-arrays each with strength three. All three-way interactions among components $0-2, 3-5, 6-8$ are included. All two-way interactions for the whole system are also covered. This has been achieved with 27 rows which is the optimal size for a $CA(3,3^3)$. A covering array that would cover all three-way interactions for all 11 components, on the other hand, might need as many as 52 rows.

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

Table 5.4: $VCA(27; 2, 3^9 2^2, (CA(3, 3^3), CA(3, 3^3), CA(3, 3^3)))$

5.4 Construction Methods

We chose to extend our simulated annealing algorithm to build variable strength covering arrays. Good data structures are required to enable the relative cost of the new feasible solution to be calculated efficiently, and the transition (if accepted) to be made quickly. We build an exponentiation table prior to the start of the program. This allows us to approximate the transition probability value using a table lookup. We use ranking algorithms from [64] to hold the values of our t-sets. This makes it possible to generalize our code easily for different strengths without changing the base data structures. When working with variable strength covering arrays it is an important feature. To calculate the change in cost for each transition, we do not need to recalculate all of the t-sets in a test configuration, but instead only calculate the t-sets that occur with the changed value. To accomplish this, we gather all t-1 subsets that do not include the "changed" component and combine each of these with our changed component. This method is described more thoroughly in Chapter 3. The delta in our cost function is counted as the change in t-sets from our current solution. Since at any point we can make changes to both the base array and one of the higher strength arrays, these changes are added together. In our model the higher strength arrays are disjoint. Therefore, to create a variable strength array with multiple subsets of higher strength, we do not need to do any more work than for a single subset of higher strength. This is because only one subset of higher strength can be affected by any single change of a symbol during annealing. The cost of calculating a change always involves at most two cost calculations; one for the base array and one for the higher strength array.

A constant is set to determine when our program is frozen. This is the number of consecutive trials allowed where no change in the cost of the current solution has occurred. For most of our trials this constant has been set to 1,000. The cooling schedule is very important in simulated annealing. If we cool too quickly, then we freeze too early because the probability of allowing a worse solution drops too quickly. If we cool too slowly or start at too high a temperature, then we allow too many poor moves and fail to make progress. If we start at a low temperature and cool slowly, then we can maintain a small probability of a bad move for a long time, thereby allowing us to avoid a frozen state, and at the same time continuing to make progress. We have experimented using fixed strength arrays and compared our results with known algebraic constructions (see [27]). We have found that a starting temperature of approximately 0.20 and a slow cooling factor, α, of between 0.9998 and 0.99999, which applied every 2,500 iterations, works well. Using these parameters, the annealing algorithm completes in a "reasonable" computational time on a PIII 1.3GHz processor running Linux. For instance, the first few VCA's in Table 5.5, complete in seconds, while the larger problems, such as the last VCA

in Table 5.5, complete within a few hours.

As there is randomness inherent in this algorithm, we run the algorithm multiple times for any given problem.

5.5 Results

Table 5.5 gives the minimum, maximum and average sizes obtained after 10 runs of the simulated annealing algorithm for each of the associated VCA's. Each of the 10 runs uses a different random seed. A starting temperature of 0.20 and a decrement parameter of 0.9998 is used in all cases. In two cases a smaller sized array was found during the course of our overall investigation, but was not found during one of these runs. These numbers are included in the table as well and labeled with an asterisk, since these provide a previously unknown bound for their particular arrays. In each case we show the number of tests required for the base array of the single lower strength. We then provide some examples with variations on the contents of C. Finally we show the arrays with all of the components involved in a higher strength coverage. We have only shown examples using strength two and three, but our methods should generalize for any strength t. A sample of these arrays are included in the Appendix.

What is interesting in Table 5.5 is that the higher strength covering arrays often drive the size of the final test suite. Such is the case in the first and second VCA groups in this table. We can use this information to make decisions about how many components can be tested at higher strengths. Since we must balance the strength of testing with the final size of the test suite we can use this information in the design process.

Of course there are cases where the higher strength covering arrays do not determine the final test suite size since the number of test cases required is a combination of the number of levels *and* the strength. In the last VCA group in Table 5.5 the two components each with 10 levels require a minimum of 100 test cases to cover

VCA	**C**	**Min Size**	**Max Size**	**Avg Size**
$VCA(2, 3^{15}, C)$	( )	16	17	16.1
	$(CA(3, 3^3))$	27	27	27
	$(CA(3, 3^3), CA(3, 3^3))$	27	27	27
	$(CA(3, 3^3), CA(3, 3^3), CA(3, 3^3))$	27	27	27
	$(CA(3, 3^4))$	27	27	27
	$(CA(3, 3^5))$	33	33	33
	$(CA(3, 3^4), CA(3, 3^5), CA(3, 3^6))$	33* 34	35	34.8
	$(CA(3, 3^6))$	33* 34	35	34.9
	$(CA(3, 3^7))$	41	42	41.4
	$(CA(3, 3^9))$	50	51	50.8
	$(CA(3, 3^{15}))$	67	69	67.6
$VCA(2, 4^3 5^3 6^2, C)$	( )	36	36	36
	$(CA(3, 4^3))$	64	64	64
	$(MCA(3, 4^3 5^2))$	100	104	101
	$(CA(3, 5^3))$	125	125	125
	$(CA(3, 4^3), CA(3, 5^3))$	125	125	125
	$(MCA(3, 4^3 5^3 6^1))$	171	173	172.5
	$(MCA(3, 5^1 6^2))$	180	180	180
	$(MCA(3, 4^3 5^3 6^2))$	214	216	215
$VCA(2, 3^{20} 10^2, C)$	( )	100	100	100
	$(CA(3, 3^{20}))$	100	100	100
	$(MCA(3, 3^{20} 10^2))$	304	318	308.5

Table 5.5: Table of sizes for variable strength arrays after 10 runs

* The minimum values for these VCA's were found during a separate set of experiments

all pairs. In this case we can cover all of the triples from the 20 preceding components with the same number of tests. In such cases, the quality of the tests can be improved without increasing the number of test cases. We can set the strength of the 20 preceding components to the highest level that is possible without increasing the test count.

Both situations are similar in the fact that they allow us to predict a minimum size test suite based on one of the covering arrays in the system. Since there are

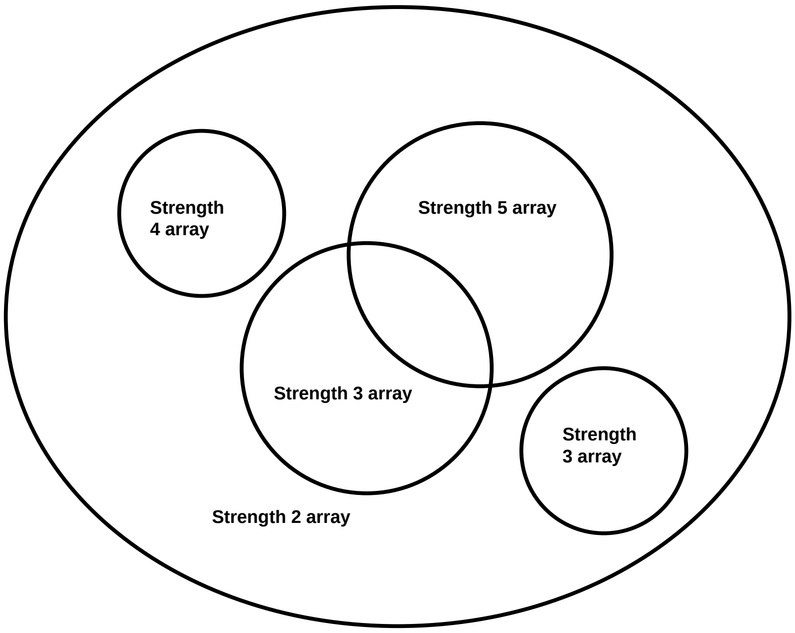


Figure 5.2: Alternative model for a variable strength covering array

better known bounds for fixed strength arrays we can use this information to drive our decision making processes in creating test suites that are both manageable in size while providing the highest possible interaction strengths.

5.6 Summary

We have presented a combinatorial object, the variable strength covering array, which can be used to define software component interaction tests, and have discussed one computational method to produce them. We have presented some initial results with sizes for a group of these objects. These arrays allow one to guarantee a minimum strength of overall coverage while varying the strength among disjoint subsets of components. Although we present these objects for their usefulness in testing component based software systems they may be of use in other disciplines that currently employ fixed strength covering arrays.

The constraining factor in the final size of the test suite may be one of the higher strength covering arrays. We can often get a second level of coverage for almost no extra cost. We see the potential to use these when there is a need for higher strength, but we cannot afford to create an entire array of higher strength due to cost limitations.

Where the constraining factor is the large number of values in a set of components at lower strength, it may be possible to increase the strength of some subset of the components without additional cost, improving the overall quality of the tests.

One method of constructing fixed strength covering arrays described by Chateauneuf *et al.* [17] is to combine smaller arrays or related objects and to fill the uncovered t-sets to complete the desired array. Since the size of a VCA may be dependent on the higher strength arrays, we believe that building these in isolation followed by annealing or other processes to fill in the missing lower strength t-sets will provide fast and efficient methods to create optimal variable strength arrays. We explore some of these methods in Chapter 6. “Seeding” our test suite with the most constrained object first often creates smaller test suite sizes. Further experimentation is needed here.

We end this chapter by remarking that our model for variable strength covering arrays is a starting point. The best model in a practical software test environment is still unknown. One alternate way to view this model is shown in Figure 5.2. In this case we have overlapping areas of coverage. The entire system has strength two coverage. There are four sub-systems that have higher strength. Two of these have overlapping instead of disjoint components. Examining which model is the best match for real software testing problems is an open and interesting topic. We leave this to future work.

Chapter 6

Cut-and-Paste Techniques for Strength Three Covering Arrays

This chapter moves away from the traditional algebraic and computational methods presented so far. It brings us a step closer to a toolkit for software testers. It merges the ideas and concepts from previous chapters to develop new methods for constructions of covering arrays. It uses algebraic methods to decompose the problems thereby requiring only small building blocks. It does not enforce the mathematical rigidity these often require. Instead it extends the role of search algorithms to general tools for building components. When these do not exist mathematically, the "closest" object can be built instead. It explores the use of meta-heuristic search to build *partial covering arrays*, *difference covering arrays*, to include *seeded test configurations*, and to find complete test suites with added constraints such as disjoint rows. The results in this chapter have been published in [28, 29]. Examples of the objects presented in this chapter are given in the Appendix.

6.1 Introduction

In Chapter 2 we cite empirical evidence that suggests arrays of higher strength are desirable in many contexts. In Chapter 4 we showed that simulated annealing performs well when the search space is relatively small and there are abundant solutions. In this case annealing often produces smaller test suites than other computational methods and sometimes improves upon known combinatorial constructions. But as the search space increases and the density of potential solutions becomes sparser the algorithm may fail to find a good solution or may require extremely long run times. It often cannot do as well as combinatorial constructions, especially when t=3. Careful tuning of the parameters of temperature and cooling improves upon the results, but at a potentially high computational cost. Recursive and direct combinatorial constructions often provide a better bound in less computational time than meta-heuristic search [17, 53, 95]. However, they are not as general and must be tailored to the problem at hand. An in-depth knowledge is often needed to decide which construction best suits a particular problem.

In this chapter we develop a new strategy, *augmented annealing*, which takes advantage of the strengths of both algebraic constructions and computational search. The idea of using small building blocks to construct a larger array is used often in combinatorial constructions. We refer to these techniques in general as *cut-and-paste* methods. But techniques to obtain a general solution often result in objects that are larger than need be. As our aim is to construct an individual object (and not prove the general existence of a class of objects), we can relax the construction and build an object that fits our criteria.

In the rest of this chapter we use combinatorial constructions and augment them with meta-heuristic search to construct strength three arrays. We have used this method successfully to construct objects with bounds lower than those of simulated annealing alone and in many cases have improved upon results for known combinatorial constructions [29]. Note that our method does not hinge on the use of simulated annealing as the computational search technique. Any search technique may be substituted without changing the basic paradigm.

6.2 Augmented Annealing

We outline the primary ideas in augmented annealing next. Details of this process are given in Section 6.4. Consider a typical recursive construction. The problem is decomposed by using a “master” structure that is used to determine the placement of certain “ingredients”. In this prototype scheme, a number of fatal problems can arise. A decomposition imposed by the master may not cleanly separate the ingredients, so that ingredients overlap or interact. The character and extent of the interaction results in either a specialized definition of allowed ingredients, or (as in our covering problem) additional coverage not required in the problem statement. Combinatorial constructions focus on proving general results, and hence often permit an overlap that is asymptotically small. However, for instances that are themselves small, the overlap can mean the difference between a good solution and a poor one.

Even more severe problems arise. It may happen that in a combinatorial construction, we have no general technique for producing the needed ingredients. When this occurs, the construction simply fails, despite its “success” at constructing a large portion of the object sought. Current techniques often abandon the combinatorial construction at this point and employ computational search.

Augmented annealing suggests a middle road. We use a combinatorial construction to decompose the problem, but then use simulated annealing (or any other search technique) to:

1. produce ingredients for which no combinatorial construction is known;
2. minimize overlap between and among ingredients; and
3. complete partial structures (*seeded tests*) when no combinatorial technique for completion is available.

This enables us to use combinatorial decompositions to reduce a problem to a number of smaller subproblems, on which simulated annealing can be expected to be both faster and more accurate than on the problem as a whole. By having simulated annealing use knowledge about which t-tuples really need to be covered, we avoid much duplicate coverage in general constructions.

Kuhfeld [65] describes a use of simulated annealing to select parameters for general constructions, and to iteratively improve upon solutions found. This is incorporated into a macro written for the SAS statistical software application called "%MktEx". This macro builds a variety of experimental designs for market research problems. It iterates through a series of steps, starting with algebraic constructions. It continues until the best design is found for a set of parameters and restrictions. Some of the iterations may include simulated annealing [65].

In the remainder of this chapter we illustrate this idea using three combinatorial constructions; the first class of constructions is discussed in [29], while the second and third are discussed in [28].

6.3 Constructions

We present several combinatorial constructions in this section that involve decomposing the covering array into smaller objects. We extend the "traditional" approach used in recursive constructions by allowing small pieces to be built using computational search.

6.3.1 Ordered Design Construction

Definition 6.3.1 *An* ordered design $OD_\lambda(t,k,v)$ *is a* $\lambda \cdot \binom{v}{t} \cdot t! \times k$ *array with* v *entries such that*

1. *each column has* v *distinct entries, and*
2. *every* t *columns contain each row tuple of* t *distinct entries precisely* λ *times.*

When $\lambda = 1$ *we write* $OD(t,k,v)$.

An $OD(3,q+1,q+1)$ exists when q is a prime power [32]. We use an ordered design as an ingredient for building a $CA(3,q+1,q+1)$ since it already covers all triples with distinct entries, having the minimal number of blocks. This handles many but not all of the triples required. The covering array is completed by covering the remaining triples. We describe a general construction next.

Construction 1 $CAN(3, q+1, q+1) \leq q^3 - q + \binom{q+1}{2} \times CAN(3, q+1, 2)$ *when q is a prime power.*

To create a $CA(3, q+1, q+1)$, begin with a $OD(3, q+1, q+1)$ of size $N_3 = (q+1) \times q \times (q-1) = q^3 - q$. This covers all triples of the form (a, b, c) where $a \neq b \neq c \neq a$. To complete the covering array we need to cover all of the triples of the form (a, a, b), (a, b, b), (a, b, a) and (a, a, a). These are exactly the triples covered by a $CA(N_2; 3, q+1, 2)$ on symbol set $\{a, b\}$. Since a and b can be any of $\binom{q+1}{2}$ combinations we append $\binom{q+1}{2}$ $CA(N_2; 3, q+1, 2)$s to the N_3 rows of the ordered design. This gives us a $CA(3, q+1, q+1)$.

Unnecessary coverage of triples occurs. In fact, any triple of the form (a, a, a) is covered at least q times rather than once. We therefore relabel entries in the $CA(N_2; 3, q+1, 2)$s to form one constant row, i.e. $(a, a, .., a)$ in each. We can do this so that each of these constant rows will contain triples already covered in at least one other of the $CA(N_2; 3, q+1, 2)$. Deleting them reduces the number of rows required by $\binom{q+1}{2}$. We can save even more:

Construction 2 $CAN(3, q+1, q+1) \leq q^3 - q + \binom{q+1}{2} \times CAN(3, q+1, 2) - (q^2 - 1)$ *when q is a prime power and there are two disjoint rows in the $CA(3, q+1, 2)$.*

In Construction 1 we exploit overlap in coverage of triples that occurs if each of the $CA(N_2; 3, q+1, 2)$s has two disjoint rows. In this case we re-map the two disjoint rows, without loss of generality, to the form $(a, a, ..., a)$ and $(b, b, ..., b)$. We remove the $2 \times \binom{q+1}{2} = q^2 + q$ rows and add back in $q+1$ rows of the form $(a, a, ..., a)$.

We give an example using $CA(3, 6, 6)$. The ordered design has 120 rows. There are 15 combinations of two symbols. In Construction 1, we create a $CA(3, 6, 2)$ with 12 rows. We therefore add back in 180 rows. This gives us a $CA(3, 6, 6)$ of size 300.

This is smaller than the bound reported by a construction in [17], and matches that found by annealing in [27]. Removing 15 constant rows lowers this bound to 285. For Construction 2, we find a $CA(12; 3, 6, 2)$ having two disjoint rows (see Table 6.4). Therefore we remove 30 rows of the type $(a, a, ..., a)$ for a total of 270 rows. We add back in six rows, one for each symbol, to achieve a covering array of size 276. This improves on both reported bounds above.

We generalize further.

Definition 6.3.2 *A (2,1)-*covering array*, denoted by $TOCA(N; 3, k, v; \sigma)$ is an $N \times k$ array containing σ or more disjoint constant rows, in which every $N \times 3$ sub-array contains every 3-tuple of the form (a, a, b), (a, b, a), and (b, a, a) with $a \neq b$, and every 3-tuple of the form (a, a, a).*

Definition 6.3.3 *$TOCAN(3, k, v; \sigma)$ denotes the minimum number N of rows in such an array.*

Definition 6.3.4 *A set $\mathcal{B}$ of subsets of $\{1, \ldots, k\}$ is a* linear space *of order k if every 2-subset $\{i, j\} \subseteq \{1, \ldots, k\}$ appears in exactly one $B \in \mathcal{B}$.*

Construction 3 *Let q be a prime power. Let $\mathcal{B} = \{B_1, \ldots, B_b\}$ be a linear space on $K = \{1, \ldots, k\}$. Let $L \subseteq K$, where L can possibly be the empty set, $\emptyset$. Suppose that for each $B_i \in \mathcal{B}$ there exists a $TOCA(N_i; 3, q + 1, |B_i|; |B_i \cap L|)$. Then there exists a $CA(q^3 - q + |L| + \sum_{i=1}^{b}(N_i - |B_i \cap L|); 3, q + 1, q + 1)$.*

We start with an $OD(q^3-q; 3, q+1, q+1)$ and for each $B_i \in B$, we construct the TOCA on the symbols of B_i with the constant rows (to be removed) on the symbols of $B_i \cap L$. Then $|L|$ constant rows complete the covering array.

6.3.2 Constructing an OD

There are two problems with the previous construction. The first is that we must construct the ordered design (i.e. the existence is not enough) if we want to use this

+	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	2	0	4	5	3	7	8	6
2	2	0	1	5	3	4	8	6	7
3	3	4	5	6	7	8	0	1	2
4	4	5	3	7	8	6	1	2	0
5	5	3	4	8	6	7	2	0	1
6	6	7	8	0	1	2	3	4	5
7	7	8	6	1	2	0	4	5	3
8	8	6	7	2	0	1	5	3	4

Table 6.1: GF(9) addition table

in a practical environment. Here we outline a method which has been adapted from [12]. Let q be a prime or power of a prime, and set $Q = \{0, 1, \ldots, q-1\}$. We add a $(q+1)$st symbol ∞ to this set. Now we generate certain permutations defined on $Q \cup \{\infty\}$ as follows. Choose all possible 4-tuples of values (a, b, c, d) where $b, c, d \in Q$ and $a \in \{0, 1\}$ subject to the conditions:

1. when $a = 1$, include all permutations where $d \neq b \times c$; and

2. when $a = 0$, include all permutations with $c = 1$ and $b \neq 0$.

For each 4-tuple generated we make a test configuration in the following way. $Q \cup \{\infty\}$ indexes the components of the test configuration. For component x, we set the value ,m, using these rules:

1. $m = a/b$ if $x = \infty$, and $b \neq 0$;

2. $m = \infty$ if $x = \infty$ and $b = 0$;

3. $m = \infty$ if $bx + d = 0$; and

4. $m = (ax + c)/(bx + d)$ for all other cases.

In order to do the arithmetic for both of these steps, the set Q needs some structure. The easiest case is when q is a prime. Then Q is the set of integers modulo q. Multiplying and adding are the same as usual but the result is reduced

$\times$	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	1	6	8	7	3	5	4
3	0	3	6	7	1	4	5	8	2
4	0	4	8	1	5	6	2	3	7
5	0	5	7	4	6	2	8	1	3
6	0	6	3	5	2	8	7	4	1
7	0	7	5	8	3	1	4	2	6
8	0	8	4	2	7	3	1	6	5

Table 6.2: GF(9) multiplication table

0	1	2	3	4	5	6	7	8
NA	1	2	4	3	7	8	5	6

Table 6.3: GF(9) inverses

modulo q. To divide we just multiply by the inverse, i.e. $a/b = a \times b^{-1}$. We can use the Extended Euclidean Algorithm to find inverses modulo q when q is prime [64].

When q is a prime power, we must use a finite field to multiply, add and find inverses. We do not attempt to describe the construction of a finite field here, but provide the addition, multiplication and inverse tables used for $q = 9$ in Tables 6.1-6.3. In these tables GF(9) stands for Galois Field of order 9. We can use any of a number of algebraic computer packages to produce a finite field, or alternatively, we can create one by hand (see [64]).

The second problem is that we must be able to determine the existence of two covering arrays with disjoint rows and then construct these. There is no general proof of the existence of these arrays. We have instead used simulated annealing to construct these and present them in Tables 6.4-6.5 on page 119.

6.4 Augmented Annealing Process

In the previous section we described a construction using an ordered design, which can produce covering arrays smaller than those obtained from the best known algebraic constructions. There are two problems that have been mentioned. Each

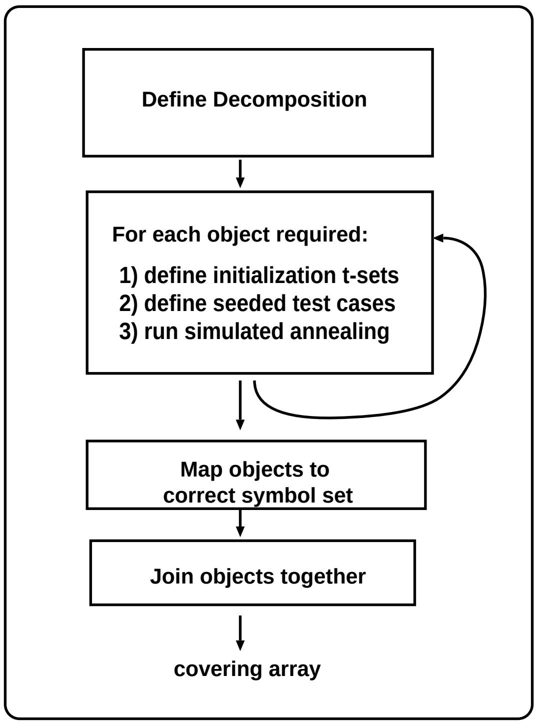

Define Decomposition
For each object required:
1) define initialization t-sets
2) define seeded test cases
3) run simulated annealing
Map objects to correct symbol set
Join objects together
covering array

Figure 6.1: Augmented annealing process

may be handled differently by a software tester and a mathematician. The software tester must actually construct these arrays, not just prove they exist. In this case, the construction of an ordered design requires a separate algorithm (see Section 6.3.2) and may require the use of finite field arithmetic. The second problem, which is imperative for proof of existence is probably less worrisome for the software tester but imperative for the mathematician.

To solve both of these problems one can use simulated annealing, "augmented" with pre-processing and post-processing phases. This provides the facility to decompose the problem and then build each object required by the constructions using only one algorithm. Furthermore it does not require proofs of existence. We can use this process to build *partial* covering arrays or *seeded* covering arrays.

The augmented annealing process (shown in Figure 6.1) begins by defining the decomposition for the object to be constructed. Each required object is then built using simulated annealing. As part of the process we can build objects where some

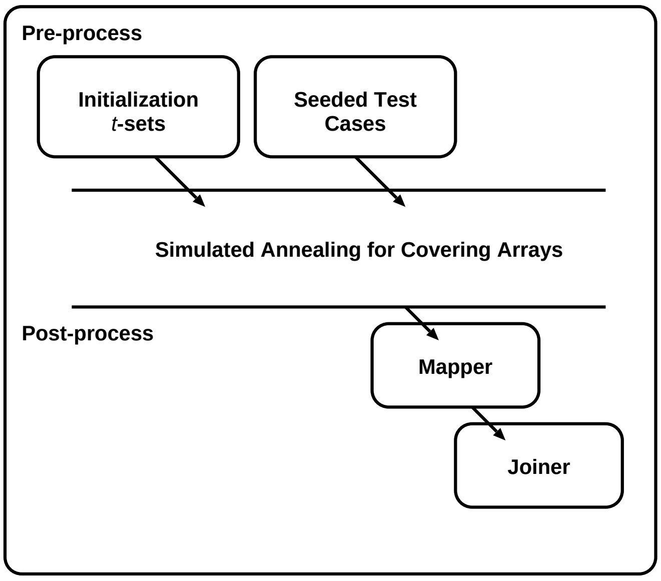


Figure 6.2: Augmented annealing modules

portion is fixed (seeding), or define t-sets that are not required to be covered by the annealing process (initialization t-sets). After each object is created the pieces are combined together in a mapping and joining phase. To achieve augmented annealing we have added several modules to the simulated annealing program (see Figures 6.2 and 6.3).

We present a method below to avoid the ordered construction using augmented annealing. For some of the smaller cases such as $CA(3, 6, 6)$ this method works very well. On larger problems using the actual construction certainly is preferred if the test suite size is of importance, but if the only toolkit the tester has is simulated annealing then this method is one which can be employed.

In the second case proving that we have a $CA(3, 6, 2)$ with two disjoint rows can be done using heuristic search. If the search produces an array of slightly larger size, then we can nonetheless get a nearly optimal test suite that appears to be smaller than the one built from straight annealing. We address the specifics of each module and how to augment the simulated annealing program to handle these two problems next. In addition we provide some constructions using the augmented annealing

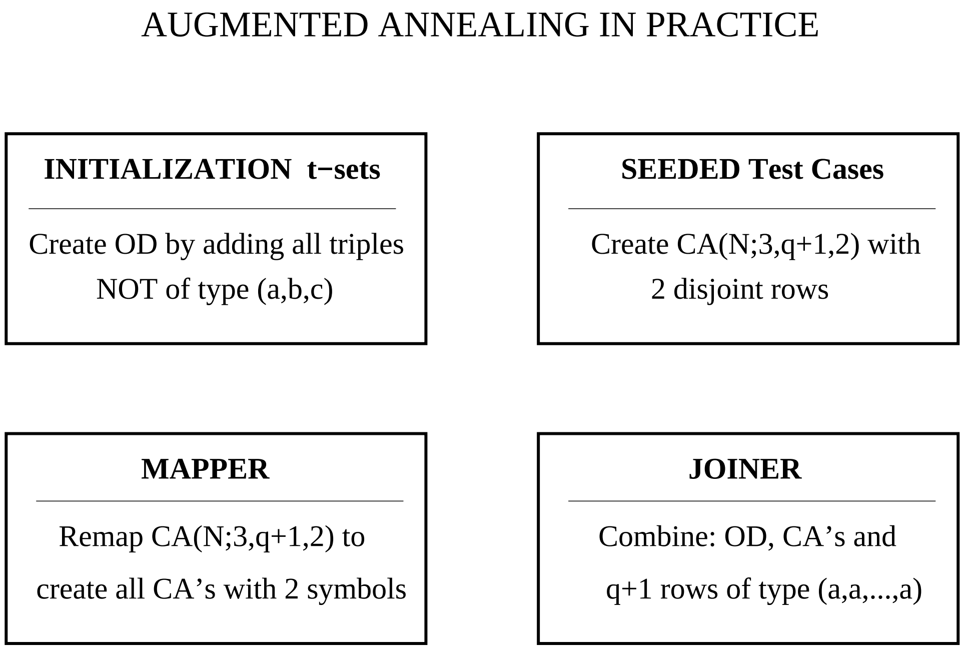

AUGMENTED ANNEALING IN PRACTICE
INITIALIZATION t−sets
Create OD by adding all triples NOT of type (a,b,c)
SEEDED Test Cases
Create CA(N;3,q+1,2) with 2 disjoint rows
MAPPER
Remap CA(N;3,q+1,2) to create all CA's with 2 symbols
JOINER
Combine: OD, CA's and q+1 rows of type (a,a,...,a)

Figure 6.3: Applying augmented annealing to construction 3

algorithm to improve further upon the bounds given above.

6.4.1 Modules

The initialization method reads in a subset of t-sets and counts these as covered. The annealing proceeds to build a potentially incomplete covering array since it believes these initialization t-sets are not needed. Therefore a move to a feasible solution that adds one of these t-sets will not improve our solution and is rarely chosen. We do not explicitly exclude these from being covered, but see this as a potential further enhancement. We can use this to build an ordered design of a small size, by initializing it with all triples which have repeated symbols, i.e. (a, a, b) and to build partial arrays that cover all triples excluding those of type (a, b, c) found in the ordered design. In our experimentation we have found that the second problem is easier for annealing than the first where fewer solutions exist in the search space. We believe this can be used in other cut-and-paste constructions allowing us to build individual partial arrays of other types.

The AETG system includes the ability to add seeded test configurations to a test suite [23]. These are test configurations that the tester wants to run each time, regardless of coverage. In any real test situation, one should have the ability to choose a set of tests that must be run. We have included the ability to add seeded test configurations to our program. It counts these as part of the covering array to be built, but it does not alter them. The seeds are *fixed*, i.e. no changes can be made to their values during annealing. We can use seeded test configurations that span entire rows of the array or partial rows of the array. In this case the non-fixed part of the test configurations can be changed. The covered t-sets are counted, but the program must do all of its annealing excluding these positions. We use this module, for example, to seed the arrays with disjoint rows.

The symbol mapping for a covering array is arbitrary and can be re-mapped as long as we use v unique symbols for each column of the covering array. We see this in Figure 2.4 on page 23. When we build the smaller covering arrays with disjoint rows, we may want to use the same array repeatedly, with different symbol mappings. Therefore, a mapper is used to translate arrays from one symbol set to another.

Lastly, when building test suites using cut and paste techniques, such as those presented here, we end up with pieces that must be merged together. A joiner module that appends test suites together both horizontally and vertically has been added for this purpose.

6.4.2 Roux-type Constructions

In [89], a theorem is presented from Gilbert Roux's Ph.D. dissertation. Figure 6.4 is an illustration of this construction.

Theorem 6.4.1 *$CAN(3, 2k, 2) \leq CAN(3, k, 2) + CAN(2, k, 2)$.*

Proof. To construct a $CA(3, 2k, 2)$, we begin by placing two $CA(N_3, 3, k, 2)$s side by side. We now have a $N_3 \times 2k$ array. If one chooses any three columns whose

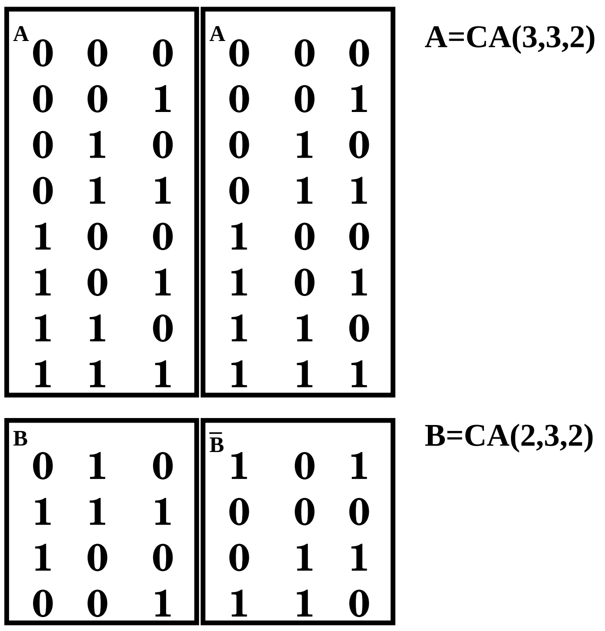


Figure 6.4: Original Roux construction

indices are distinct modulo k, then all triples are covered. The other type of selection contains a column x from among the first k, its copy, x', from among the second k, and a further column y. The triples that need to be covered are of the form $(a, a, a), (b, b, b), (a, a, b), (b, b, a), (a, b, a), (b, a, b), (a, b, b), (b, a, a)$. We can assume without loss of generality that the column index of x is less than the column index of y and that y is selected from the second array. All pairs of symbols will occur between x and y. The first four triples occur because the first two columns are identical. To handle the rest, we append two $CA(N_2, 2, k, 2)$s side by side, the second being the bit complement of the first. All pairs of symbols between x and y occur. The first two columns are bit complements of each other, which covers that last four triples. This is a covering array of size $N_2 + N_3$. ∎

Chateauneuf *et al.* [17] prove a generalization, which we repeat here.

Theorem 6.4.2 $CAN(3, 2k, v) \leq CAN(3, k, v) + (v - 1)CAN(2, k, v)$.

Proof. Begin as in Theorem 6.4.1 by placing two $CA(N_3; 3, k, v)$s side by side. Let

C be a $CA(N_2; 2, k, v)$. Let π be a cyclic permutation of the v symbols. Then for $1 \leq i \leq v-1$, we append N_2 rows consisting of C and $\pi^i(C)$ placed side-by-side. The verification is as for Theorem 6.4.1. ■

Martirosyan and Trung [69] generalize this for cases when $t > 3$. We now develop a substantial generalization to permit the number of components to be multiplied by $\ell \geq 2$ rather than two; this is the *multi-Roux construction* [1]. To carry this out, we require another combinatorial object. Let Γ be a group of order v, with $\odot$ as its binary operation.

Definition 6.4.1 *A* difference covering array $D = (d_{ij})$ *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.*

When Γ is abelian, additive notation is used, explaining the "difference" terminology. We shall only employ the case when $\Gamma = \mathbb{Z}_v$, and omit it from the notation. We denote by $DCAN(2, k, v)$ the minimum N for which a $DCA(N, \mathbb{Z}_v; 2, k, v)$ exists.

Theorem 6.4.3 *$CAN(3, k\ell, v) \leq CAN(3, k, v) + CAN(3, \ell, v) + CAN(2, \ell, v) \times DCAN(2, k, v)$.*

Proof. We suppose that the following all exist:

1. a CA$(N; 3, \ell, v)$ A;

2. a CA$(M; 3, k, v)$ B;

3. a CA$(R; 2, \ell, v)$ F; and

4. a DCA$(Q; 2, k, v)$ D.

[1]This is called k-ary Roux in [28].

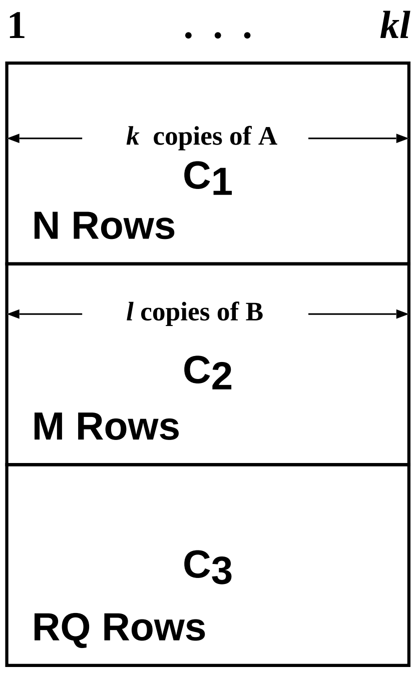


Figure 6.5: multi-Roux construction

We produce a CA$(N + M + QR; 3, k\ell, v)$ C (see Figure 6.5). For convenience, we index the $k\ell$ columns of C by ordered pairs from $\{1, \ldots, k\} \times \{1, \ldots, \ell\}$. C is formed by vertically juxtaposing three arrays, C_1 of size $N \times k\ell$, C_2 of size $M \times k\ell$, and C_3 of size $QR \times k\ell$. We describe the construction for each in turn.

C_1 is produced as follows. In row r and column (i, j) of C_1 we place the entry in cell (r, j) of A. Thus C_1 consists of k copies of A placed side by side. This is illustrated in Figure 6.6.

C_2 is produced as follows. In row r and column (i, j) of C_2 we place the entry in cell (r, i) of B. Thus C_2 consists of ℓ copies of the first column of B, then ℓ copies of the second column, and so on (see Figure 6.7).

To construct C_3 (see Figure 6.8), let $D = (d_{ij} : i = 1, \ldots, Q; j = 1 \ldots, k)$ and $F = (f_{rs} : r = 1, \ldots, R; s = 1, \ldots, \ell)$. Let π be the cyclic permutation $(0, 1, 2, \ldots, v - 1)$ on the v symbols of F. Then in row $(i - 1)R + r$ and column (j, s) of C_3 place the entry $\pi^{d_{ij}}(f_{rs})$, where $\pi^{d_{ij}}(f_{rs}) = f_{rs} + d_{i,j} \mod v$.

We verify that C is indeed a CA$(N + M + QR; 3, k\ell, v)$. The only issue is to

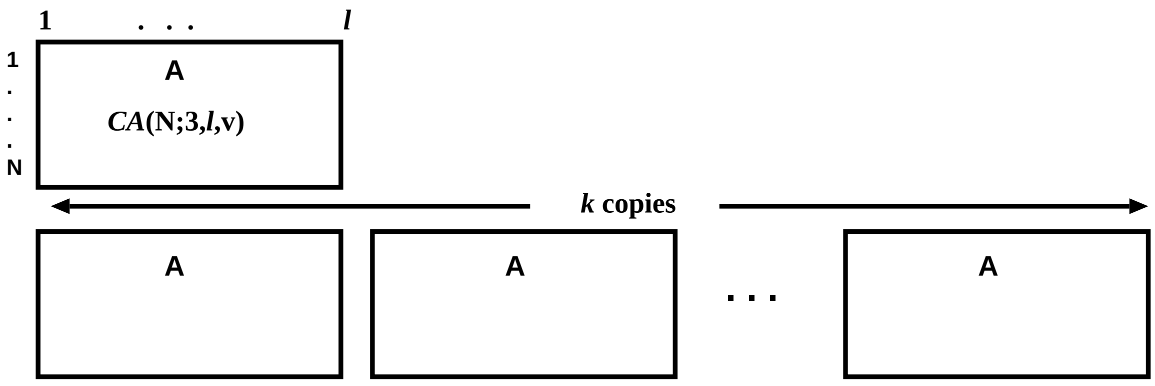


Figure 6.6: Construction of C_1

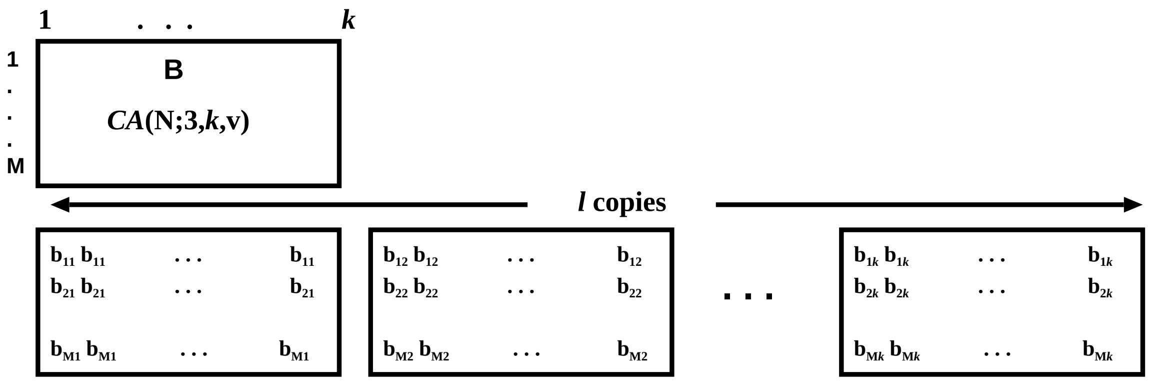


Figure 6.7: Construction of C_2

ensure that every 3 columns of C cover each of the v^3 3-tuples. Select three columns (i_1, j_1), (i_2, j_2), and (i_3, j_3) of C. If j_1, j_2 and j_3 are all distinct, then these three columns restricted to C_1 arise from three different columns of A, and hence all 3-tuples are covered. Similarly, if i_1, i_2, and i_3 are all distinct, then restricting the three columns to C_2, they arise from three distinct columns of B and hence again all 3-tuples are covered.

So we suppose without loss of generality that $i_1 = i_2 \neq i_3$ and $j_1 \neq j_2 = j_3$. The structure of C_3 consists of a $Q \times k$ block matrix in which each copy is a permuted version of F (under a permutation that is a power of π). That $i_1 = i_2$ indicates

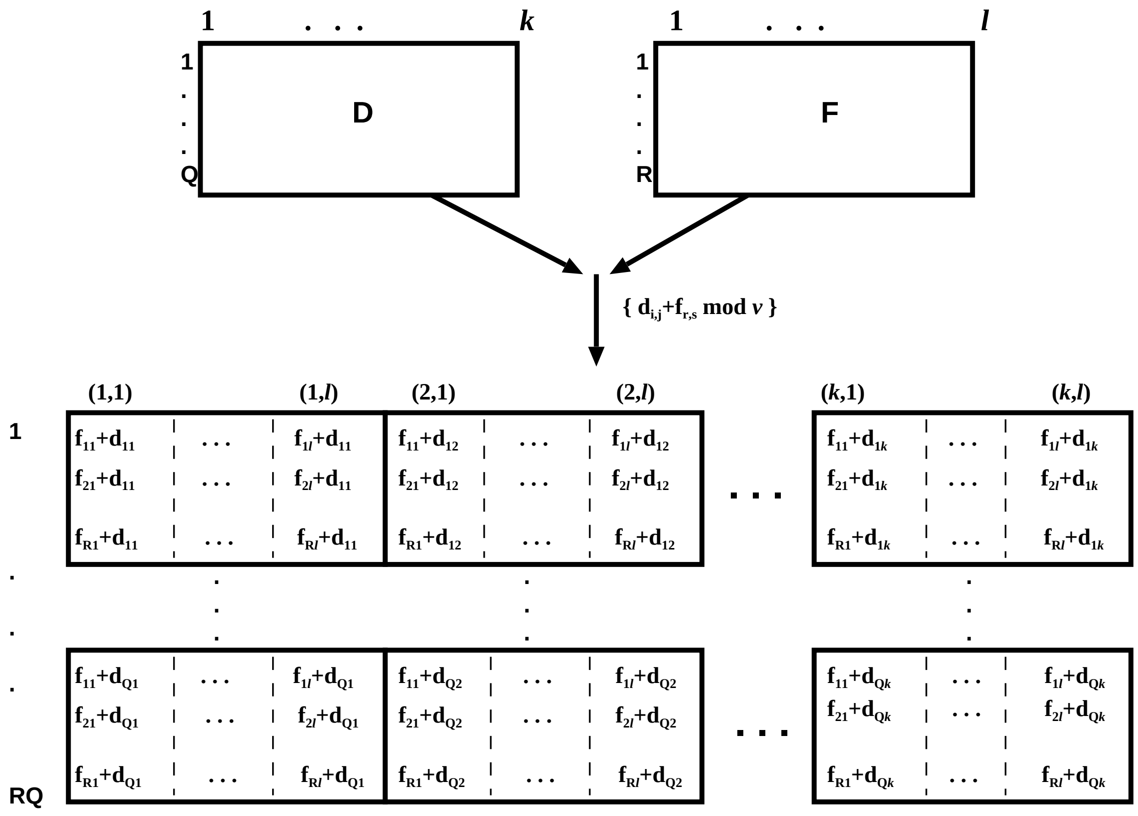

1 . . . k
1 . . . Q
D
1 . . . l
1 . . . R
F
{ d_{i,j}+f_{r,s} mod v }
(1,1) (1,l) (2,1) (2,l) (k,1) (k,l)
1
RQ

Figure 6.8: Construction of C_3

that two columns are selected from one column of this block matrix, and that i_3 is different means that the third column is selected from a different column of the block matrix. Now consider a selection $(\sigma_1, \sigma_2, \sigma_3)$ of symbols in the three chosen columns of C (actually, of C_3). Each selection of (σ_1, σ_2) appears in each block of the Q permuted versions of F appearing in the indicated column of the block matrix. Now suppose that $\sigma_3 = \pi^i(\sigma_2)$; since π is a v-cycle, some power of π satisfies this equality. Considering the permuted versions of F appearing in the columns corresponding to i_3, we observe that since D is an array covering all differences modulo v, in at least one row of the block matrix, we find that the block X in column i_3 and the block Y in column i_2 satisfy $Y = \pi^i(X)$. Hence every choice for σ_3 appears with the specified pair (σ_1, σ_2). ∎

This can be improved upon: we do not need to cover 3-tuples when $\sigma_3 = \sigma_2$ since these are covered in C_1. Nor do we need to cover 3-tuples when $\sigma_1 = \sigma_2$, since these are covered in C_2. So we can eliminate some rows from F as we do not need to cover pairs whose symbols are equal in F. This modification further improves on the bounds.

6.4.3 Construction Using Generalized Hadamard Matrices

Augmented annealing affords the opportunity to develop "constructions" when some of the "ingredients" are not known at all. We illustrate this next. The basic plan is to construct a large portion of a covering array to use as a seed. Consider an $OA_\lambda(2, k, v)$. Each 2-tuple is covered exactly λ times. Some 3-tuples are also covered. Indeed, among the v 3-tuples containing a specified 2-tuple, at least one and at most $\min(v, \lambda)$ are covered. If λ of the v are covered for every 2-tuple, then the orthogonal array is *supersimple*. Little is known about supersimple orthogonal arrays except when k is small. However our concern is only that "relatively many" triples are covered using "relatively few" rows. This is intentionally vague, since our intent is only to use the rows of the orthogonal array as a seed for a strength three covering array. A natural family of orthogonal arrays to consider arise from generalized Hadamard matrices (see [32]). We have no assurance that the resulting orthogonal arrays are supersimple, but instead choose generalized Hadamard matrices since they provide a means to cover many of the triples to be covered by the covering array. Although orthogonal arrays in general may be useful in constructions here, those from generalized Hadamard matrices appear frequently to cover either only one, or all v, of the triples containing a specified pair; this regularity appears to be beneficial. In the next section, we report computational results using these as seeds in annealing. The most important remark here is that, given such a generalized Hadamard matrix, it is not at all clear what "ingredients" are needed to complete it to a covering array in general, despite the fact that in any specific case we can easily enumerate the triples left uncovered.

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

Table 6.4: $TOCA(12; 3, 6, 2; 2)$

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

Table 6.5: $TOCA(13; 3, 10, 2; 2)$

6.5 Computational Results

6.5.1 Constructions Using Ordered Designs

We presented constructions for $CA(3, 6, 6)$s earlier. Construction 1 gave an array of size 300, while Construction 2, requiring covering arrays with two disjoint rows, gave a size of 276. Table 6.4 gives a $CA(3, 6, 2)$ with two disjoint rows.

We can create variations on this construction using augmented annealing. We can construct a $TOCA(30; 3, 6, 3; 0)$. The bound for a $CA(3, 6, 3)$ is 33 so we have saved three rows by using augmented annealing. We can use this to cover $\binom{3}{2} = 3$ of the 15 pairs of symbols. There are still 12 remaining. We can cover these using 12 $TOCA(12; 3, 6, 2; 2)$s. Each of these are of size 10 once constant rows are removed. Lastly we add back in three rows of type (a, a, a, a, a, a) (we can exclude the three symbols covered by the $TOCA(30; 3, 6, 3; 0)$) and join these together. This gives us a covering array of size $120 + 30 + (12 \times 10) + 3 = 273$. This is smaller than the constructions given. Using instead two $TOCA(30; 3, 6, 3; 0)$s reduces the bound further to 270. There are 6 combinations of pairs handled by the two $TOCA(30; 3, 6, 3; 0)$s. With careful mapping we can use all six symbols. Therefore all triples of the type (a, a, a, a, a, a) have been accounted for in these 60 rows. There are still 9 uncovered pairs. This requires 9 $TOCA(12; 3, 6, 2, 2)s$. We can remove all of the constant rows. This gives us $120 + (2 \times 30) + (9 \times 10) = 270$. Other linear spaces in Construction 3 can be employed. In the case of $CA(3, 6, 6)$ we found the best bound using only two building blocks. We used annealing to create an ordered design of size 120 and annealing to create a $TOCA(140; 3, 6, 6; 0)$. This gives us a $CA(260; 3, 6, 6)$, improving considerably on the constructions given above.

These applications of Construction 3 can be used in all of the cases outlined below. Tables 6.8-6.11 show the smallest sizes of (2,1)-covering arrays found by simulated annealing. The first column of each gives the size with v constant disjoint rows, and the second with no rows specified.

Table 6.6 shows the smallest covering arrays found using two augmented methods and provides the smallest numbers we have obtained using straight annealing as well as known bounds published in [17, 27]. The first method, labeled A, uses an ordered design combined with a $TOCA(3, k, k; 0)$ found by annealing. The second method uses an ordered design and combines it with $\binom{k}{2}$ $TOCA(3, k, 2; 2)$s. The best values we have found for these arrays are also given in Table 6.6. The ordered design for $CA(3, 6, 6)$ was created using annealing. All of the other ordered designs were created using the definition of $PSL(2, q)$ (see [12]). Values in bold font are new upper bounds for these arrays.

In the cases of $CA(3, 8, 8)$ and $CA(3, 9, 9)$, the collection of all triples can be covered exactly, i.e. every triple is covered precisely once (this is an orthogonal array of strength three). We therefore cannot improve on the best known result since it is optimal. Nevertheless, these cases illustrate improvement from augmented annealing over straight annealing. The smallest array we have found using simulated annealing in a reasonable amount of computational time for the $CA(3, 8, 8)$ has 918 rows. This result required almost three hours to run, illustrating the severity of the difficulty with naive computational search. We can instead create an $OD(3, 8, 8)$ of size 336 in significantly less time and anneal a (2,1)-covering array of size 280 in approximately five minutes. This provides us with a $CA(3, 8, 8)$ of size 616 which is smaller and computationally less expensive than using just annealing.

For $CA(3, 9, 9)$ similar results are found. In this case, however, using either $TOCA(3, 9, 2; 2)$s or a $TOCA(3, 9, 9; 0)$ does not fare as well as using Construction 3 with a linear space consisting of twelve blocks of size three; then $CAN(3, 9, 9) \leq 900$ is obtained. Perhaps this serves well to illustrate a general conclusion. An optimal

$CA(t,k,v)$	Augmented Annealing			Simulated Annealing	Smallest Reported Array Size
	A	B	TOCA		[17, 27, 53]
$CA(3,6,6)$	**260**	276	12	300	300
$CA(3,8,8)$	616	624	12	918	512
$CA(3,9,9)$	906	909	13	1,490	729
$CA(3,10,10)$	**1,219**	1,225	13	2,163	1,330
$CA(3,12,12)$	2,339	**2,190**	15	4,422	2,196
$CA(3,14,14)$	4,134	**3,654**	18	8,092	4,094

Table 6.6: Sizes for covering arrays using augmented annealing
Method A = $TOCA(3,k,k;0)$
Method B = $TOCA(3,k,2;2)$s
The column headed "TOCA" gives the size of the $TOCA(3,k,2;2)$s used.

solution has 729 rows, while annealing alone takes substantial time to obtain a bound of 1490. Augmented annealing yields a bound of 900 quickly, and applies more generally than the existence of an orthogonal array.

For the $CA(3,10,10)$ we can use the ordered design construction to generate the first part of this array, giving us 720 rows. We can build 45 $TOCA(13;3,10,2;2)$s and add back in 10 rows of type $(a,a,..,a)$. If we do this then we have an array of size $720 + (45 \times 11) + 10 = 1225$ which improves upon the published bound of 1331 [17]. We can also build a $TOCA(499;3,10,10;0)$ using annealing. When combined with the ordered design, the size of the covering array is 1219. The smallest array we have built with straight annealing for a $CA(3,10,10)$ is of size 2163. Again using Construction 3 with a suitably chosen linear space yields the best known result. A linear space with three lines of size four and nine of size three gives $CAN(3,10,10) \leq 1215$. The 3 $TOCA(66;3,10,4;0)$s account for 18 of the pairs of 10 symbols. The 9 $TOCA(33;3,10,3;0)$s account for the other 27 pairs. Therefore we have $720 + (9 \times 33) + (3 \times 66) = 1215$. It appears that the $TOCA(3,10,10;0)$ is not yielding as strong a result in part because it has, in some sense, become a "large" ingredient and annealing is not as effective.

$TOCA(3, q+1, q+1; 0)$ t, k, v	Size	Ordered Design
$3, 6, 6$	140	120
$3, 8, 8$	280	336
$3, 9, 9$	402	504
$3, 10, 10$	499	720
$3, 12, 12$	1,019	1,320
$3, 14, 14$	1,950	2,184

Table 6.7: Sizes for $TOCA(3, q+1, q+1; 0)$s and ordered designs

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 6, 2$	12	12
$3, 6, 3$	33	30
$3, 6, 4$	60	56
$3, 6, 5$	99	94
$3, 6, 6$	145	140

Table 6.8: Sizes for TOCAs with $k = 6$

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 8, 2$	12	12
$3, 8, 3$	33	30
$3, 8, 4$	64	60
$3, 8, 5$	105	100
$3, 8, 6$	156	150
$3, 8, 7$	217	210
$3, 8, 8$	288	280

Table 6.9: Sizes for TOCAs with $k = 8$

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 9, 2$	13	12
$3, 9, 3$	36	33
$3, 9, 4$	70	67
$3, 9, 5$	116	110
$3, 9, 6$	171	166
$3, 9, 7$	239	233
$3, 9, 8$	316	308
$3, 9, 9$	416	402

Table 6.10: Sizes for TOCAs with $k = 9$

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 10, 2$	13	12
$3, 10, 3$	36	33
$3, 10, 4$	70	66
$3, 10, 5$	115	111
$3, 10, 6$	172	165
$3, 10, 7$	239	232
$3, 10, 8$	322	310
$3, 10, 9$	409	401
$3, 10, 10$	506	499

Table 6.11: Sizes for TOCAs with $k = 10$

CA	Size	Previous Bound [17]	OA,Size	Percent triples covered by OA
$CA(3, 9, 3)$	**50**	51	$OA_3(2, 9, 3)$,27	90.5
$CA(3, 25, 5)$	**371**	465	$OA_5(2, 25, 5)$,125	89.6
$CA(3, 27, 3)$	118	99	$OA_9(2, 27, 3)$,81	97.3
$CA(3, 16, 4)$	174	159	$OA_4(2, 16, 4)$,64	78.6
$CA(3, 17, 4)$	**180**	184	$OA_4(2, 17, 4)$,64	77.9
$CA(3, 10, 5)$	266	185	$OA_2(2, 10, 5)$,50	40.0

Table 6.12: Sizes for CAs built with OAs of higher index

$CA(t, k\ell, v)$	Size	Previous Bound [17]	$CA(3, k, v)$,Size	$CA(3, \ell, v)$,Size	Size of D	Size of F
$CA(3, 25, 4)$	**188**	229	$CA(3, 5, 4)$,64	$CA(3, 5, 4)$,64	4	15
$CA(3, 30, 4)$	**203**	238	$CA(3, 5, 4)$,64	$CA(3, 6, 4)$,64	5	15
$CA(3, 24, 6)$	**692**	795	$CA(3, 6, 6)$, 260	$CA(3, 4, 6)$,216	6	36
$CA(3, 36, 3)$	109	?	$CA(3, 4, 3)$,27	$CA(3, 9, 3)$,50	4	8

Table 6.13: multi-Roux

6.5.2 Constructions from Generalized Hadamard Matrices

Table 6.12 gives some results for building covering arrays from strength two orthogonal arrays of index higher than one, obtained from generalized Hadamard matrices. In each of these cases we seed the annealing program with the OA of higher index and then anneal the rest of the array. We include in the table the percentage of triples covered by and the size of the OA prior to annealing. In our experience if too many triples are covered before annealing occurs, then the best bound is not found. There seems to be a trade-off in the tightness of the structure used for seeding and the final covering array. We have listed the size of the strength two orthogonal array and the percentage of triples that are covered in this subset.

6.5.3 Constructions Using multi-Roux

We applied the multi-Roux construction to some covering arrays for sizes of $\ell > 2$. Table 6.13 gives some of these results. This construction appears to do well when the two smaller building blocks are themselves optimal. In the first two entries we have used orthogonal arrays as ingredients. In each of these entries we do not have to handle triples when $\sigma_1 = \sigma_2$. We have used the augmented annealing program to build D and F by initializing them with these triples. The sizes we found for these are listed in the table. For instance, we can build a difference covering array $DCA(N; 2, 5, 4)$ of size 4 instead of 5 if we do not care about covering the zero differences. And we can create a $CA(2, 6, 4)$ of size 15 if we do not care about pairs with equal entries. This saves us 15 rows in the final covering array.

Table 6.14 gives results of computations using simulated annealing for the existence of difference covering arrays. Two entries are given. The first is for a DCA that (in addition) covers the zero difference, while the second does not require the zero difference to be covered.

Sizes for DCA's with,without Zero Differences								
k/q	3	4	5	6	7	8	9	10
3	3,2	5,4	5,4	7,6	7,6	9,8	9,8	11,10
4	4,3	5,4	5,4	7,6	7,6	9,8	10,9	11,10
5	5,4	5,4	5,4	8,7	7,6	9,8	10,9	12,11
6	5,4	6,5	7,6	8,7	7,6	10,9	11,10	12,11
7	5,4	6,5	7,6	8,7	7,6	10,9	11,11	13,12
8	5,4	6,5	8,7	8,7	9,8	11,9	12,11	13,12
9	5,4	7,6	8,7	9,8	9,8	11,11	12,12	14,13
10	5,4	7,6	8,7	9,8	10,9	12,11	13,12	15,14

Table 6.14: Table of difference covering arrays with and without zero differences

6.5.4 Extending Augmented Annealing

We close with some examples that are not handled by the constructions from this chapter because one or more of the required objects cannot be constructed. The intent is to see if we can still use the given methods to build covering arrays for these cases. The first example selected is a $CA(3, 7, 7)$. An orthogonal array of size 343 can be constructed for this example [51, 53]. We would like to use our ordered design construction to try and build this array, however there is no ordered design for this problem since 6 is not a prime power. We have instead used annealing to create (2,1)-covering arrays (Table 6.15) and *analogs* of ordered designs. We do not approach the size of the orthogonal array and improve only slightly on the best bound found for this array obtained from straight annealing, but this approach improves the time that is required to solve this problem over that of straight annealing.

The second example is an $MCA(3, 6^6 4^2 2^2)$. This array contains a $CA(3, 6, 6)$ but has four additional columns. We have tried several techniques to build this array. When we use straight annealing we found an array of size 317, which is much larger than the best bound we have found for the sub-array $CA(3, 6, 6)$. Based on the experience reported in [26] we believe that the hardest problem, that of the $CA(3, 6, 6)$, dictates the size of this array. When we used two partial covering arrays as in Method B, the best bound we found was 313. We have therefore tried

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 7, 2$	12	12
$3, 7, 3$	33	30
$3, 7, 4$	64	60
$3, 7, 5$	105	100
$3, 7, 6$	156	150
$3, 7, 7$	217	210

Table 6.15: Sizes for $TOCA(3, k, v)$s with $k = 7$

Covering Array	Method	Size	Published Bound[53]
$CA(3, 7, 7)$	Straight Annealing	552	343
$CA(3, 7, 7)$	Partial Arrays	545	
$MCA(3, 6^6 4^2 2^2)$	Straight Annealing	317	NA
$MCA(3, 6^6 4^2 2^2)$	Partial Arrays	313	
$MCA(3, 6^6 4^2 2^2)$	Seeded with $OD(3, 6, 6)$	283	
$MCA(3, 6^6 4^2 2^2)$	Seeded with $CA(3, 6, 6)$	272	

Table 6.16: Extending augmented annealing

seeding this array with solutions for subproblems already found. We seed either the $OD(3, 6, 6)$ of size 120 or the $CA(3, 6, 6)$ of size 263 and then anneal to complete the structure. Both of these improve markedly upon the first two methods as shown in Table 6.16. The smallest test suite we found used the $CA(3, 6, 6)$ as a seed. This added fewer than 10 rows to complete the missing coverage. This highlights the need for the software tester to have knowledge (or a tool) to determine which method is best for which problem.

6.6 Summary

The construction of strength three covering arrays is a challenging combinatorial and computational problem. The real and potential applications in the design of software test suites necessitate reasonably fast and accurate techniques. Computational search techniques, while general, degrade in speed and accuracy as problem size increases. Combinatorial techniques suffer lack of generality despite offering the promise of fast and accurate solutions in specific instances.

We have therefore proposed a framework for combining combinatorial constructions with meta-heuristic search, and examined a specific instantiation of this, augmented annealing. The covering arrays produced illustrate the potential of this approach, demonstrating that a combinatorial construction can be used as a master to decompose a search problem so that much smaller ingredient designs can be found.

Perhaps what distinguishes this from the majority of existing recursive constructions is that we are not concerned primarily with finding a master for which the ingredients needed are themselves well understood combinatorial objects. Augmented annealing can be viewed as a first step in designing a tool to exploit combinatorial constructions along with heuristic search to produce covering arrays for the variety of parameters arising in practice.

Chapter 7

Conclusions

In this thesis we have examined the problem of building test suites for software interaction testing. We have made contributions in the fields of software engineering, heuristic search and combinatorial design theory. The results presented in this thesis provide foundations for our ultimate goal, an interaction testing toolkit. We summarize our findings next and follow this with a presentation of future directions.

7.1 Summary

In heuristic search we have developed a guided meta-heuristic search technique, augmented annealing. This technique uses a general purpose simulated annealing program as the engine to build other objects and support cut-and-paste constructions. It includes two pre-processing stages, *initialization* and *seeding* and two post-processing steps, *mapping* and *joining*. By using this process we can build other objects such as partial covering arrays, difference covering arrays and covering arrays with constraints (i.e. those with x disjoint rows). We can leverage the power of computational search and relax some of our mathematical constructions to build small covering arrays for software testing. In addition it moves us closer to a real test environment, by allowing us to seed test configurations, and has the potential to add avoids, aggregates and other constraints.

We have also examined several computational methods for finding fixed and mixed level covering arrays of strength 2 and 3. These are the first results using meta-heuristic search for mixed level arrays and the first using meta-heuristic search to find covering arrays with $t > 2$. We have shown that simulated annealing builds covering arrays closer to the optimal size for N than greedy methods, and often does as well as algebraic constructions.

In software engineering we have developed a model for the *variable strength* covering array and have provided some initial bounds and methods for constructing these. We have shown that we can often use this type of model to gain a stronger interaction test suite without increasing the number of test configurations.

In combinatorial mathematics we have developed some new cut-and-paste constructions for covering arrays and have provided some new bounds for strength three covering arrays. The multi-Roux construction is a generalization of [17]. We have also explored using higher index orthogonal arrays as seeds.

Although our work has been directed toward traditional software interaction testing, there are many emerging applications for these types of combinatorial designs. As we have seen, the use of these for fault characterization in a large software configuration space is emerging [108]. Covering arrays have been used in logic testing [61] and for database testing [19]. And they are emerging as important designs outside computer science. Recent studies use them to describe inputs for biological systems [86] and for chemical interaction testing [15]. The techniques developed, therefore, extend beyond the application of software testing.

7.2 Future Work

This research on constructing test suites for interaction testing has highlighted many future avenues to pursue. There are four main directions that we outline here. We discuss the first three next. The last section of this thesis is devoted to the fourth, our long term goal : *an integrated approach.*

In software engineering, more studies on the variable strength array and its usefulness in real test environments are needed. Empirical studies are required to quantify the additional code coverage and benefits (if any) of this approach. Modeling test environments is useful to understand how the variable strength array can best be applied. The empirical results to date suggest that higher strength arrays are needed for good code coverage and better fault characterization [44, 66, 108]. Therefore it is natural to explore ways to increase the strength of sub-arrays when possible and to test this hypothesis further.

In the area of computational search there are still many unanswered questions as to the effectiveness and performance of a variety of search techniques. We have presented a general framework for a set of greedy algorithms, but have only explored these in the context of the known/published algorithms. An examination of the generality of this framework and experimentation with different combinations of options can tell us more about what the important decisions are when it comes to time, precision and repeatability.

We have not yet explored the use of genetic algorithms or extended Nurmela's work on tabu search, and have only made an initial attempt at examining other meta-heuristic techniques such as the great deluge algorithm. In addition, we have not explored the landscape of covering arrays. A visualization of the landscape may provide us with more information about which search techniques are best for particular values of t, k and v.

In the area of combinatorial design theory, what has evolved is an appreciation for the complexity and difficulty of the problem of constructing strength three arrays. There is still much room for exploration and development of new constructions. For instance, Matirosyan *et al.* [69] have generalized the Roux construction recently in t. The combination of techniques to extend Roux in both t and k is an interesting problem.

Only recently have constructions appeared for mixed level arrays [73]. This is another avenue to explore. There are currently no known constructions for variable strength arrays. And one cannot ignore arrays of even higher strength. The literature contains very little on covering arrays beyond strength 3.

Up until this point we have examined our contributions and research in terms of three separate disciplines. But the ultimate goal of our work is to develop a toolkit for software testers that combines the best from all of these areas. This leads us to the ideas presented in the next section.

7.2.1 An Integrated Approach

One of the main themes of this thesis is the multitude of approaches for constructing covering arrays for software interaction testing. We have examined a few computational methods as well as some algebraic ones. We have explored issues related to a practical test environment. We are left with the difficulty of merging these areas of research to approach our ultimate goal, useful test generation tools for the design of interaction test suites.

Given the computational efficiency of an algebraic construction, this is the best method to find a covering array when one is known to exist for the given parameters t, k and v. Indeed this idea has been suggested by Stevens *et al.* [94], who point out that this may not be a simple task. In order to use an algebraic construction we often use smaller objects which must also be constructed. This was illustrated in Chapter 6. But the recursive nature of this makes the existence question alone here quite difficult (see [30]). If we can determine that the smaller objects exist, then we first need to build these, and the information on how to do so must be stored somewhere. In addition, we may also need to store many small starter objects, for which space constraints may become a problem. A further issue is that there are many different types of constructions and sometimes multiple ways to arrive at the same object. This perhaps explains why commercial test generators do not always utilize the best known constructions, but instead search each time from scratch.

By combining several of these techniques we expect to be able to find a large range of arrays that can be expanded or reduced as necessary. We could, for in-

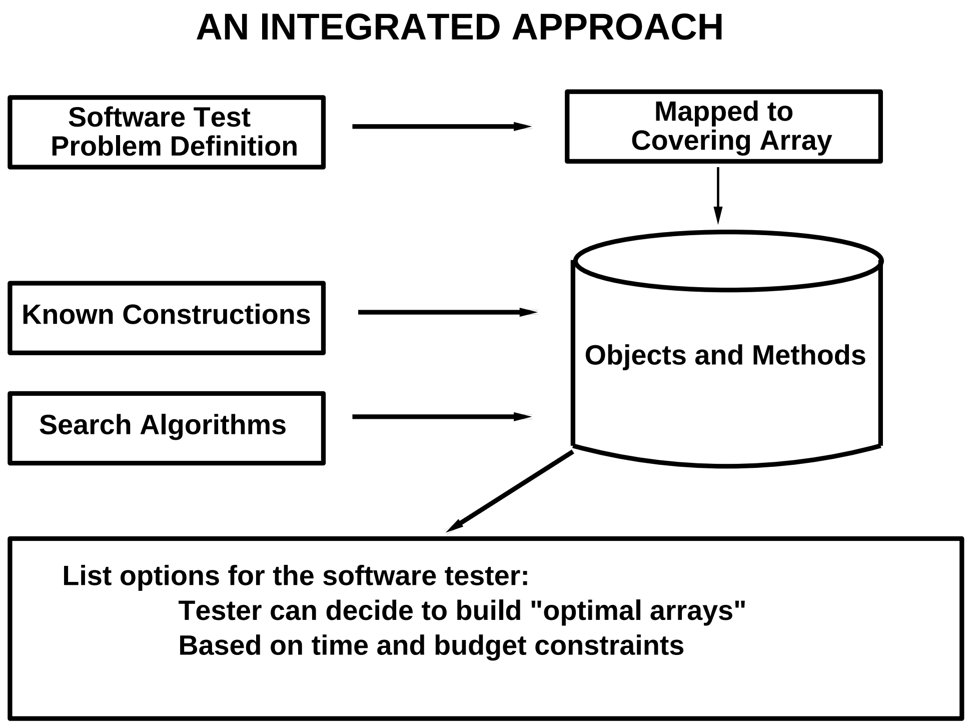

AN INTEGRATED APPROACH
Software Test Problem Definition
Mapped to Covering Array
Known Constructions
Search Algorithms
Objects and Methods
List options for the software tester:
Tester can decide to build "optimal arrays"
Based on time and budget constraints

Figure 7.1: Using an integrated approach

stance, begin with a less costly algorithm, such as one from our greedy framework, and define a critical point in our test suite where we make a switch to a more computationally expensive algorithm. Another possibility is to simply build a starter test suite deterministically, and then use simulated annealing to reduce its size. An example of this technique is described in [22]. In Chapter 6 we have had some success in seeding with test configurations that have tight structures and building upon these. To this end we suggest a middle road.

Figure 7.1 shows our view of an integrated approach for our software interaction testing. This model suggests a knowledge base of the best known covering arrays, as well as starter objects and computational as well as algebraic methods which can be used to build bigger arrays. This tool may not always provide the best known covering array, but must offer options to balance the task at hand. For instance if a software tester wanted to build a $CA(N; 3, 6, 6)$ they might be presented with an option of building it very quickly using a deterministic method (i.e. a construction that gives us an array with $N = 300$), or for slightly more computational power they could arrive at 260 which is the best known bound at this time. If on the other hand they were interested in building an $MCA(2, 3^{20}2^{10})$, then the tool might suggest that they instead build a $VCA(2, 3^{20}2^{10}, (CA(3, 3^{20})))$ since this can be created without adding any extra test configurations.

With this model, testers ideally would input a set of parameters and requirements. They would be presented with options which include time trade-offs for building the test suites as well as the level of strength required for testing and the need for an optimal test suite. In some cases the best method for a tester is a greedy approach that is fast and "good-enough". At other times when the testing and cost of checking results is expensive, the need for optimal test suites may indicate that long run times to find a test suite is needed.

Using this model as our goal for the future, the next steps are (a) finding more effective computational methods and constructions, (b) developing a comprehensive compilation of data containing the best known covering array bounds and associated construction methods, and (c) more comprehensive theoretical explorations of variable strength arrays.

Bibliography

[1] ANDERSON, I. *Combinatorial Designs and Tournaments.* Oxford University Press, New York, 1997.

[2] APACHE SOFTWARE FOUNDATION.
apache core features. http://www.apache.org/, 2004.

[3] BACH, J. Satisfice, Inc. ALLPAIRS test generation tool, Version 1.2.1. http://www.satisfice.com/tools.shtml, 2004.

[4] BACH, J. James Bach on risk-based testing. *STQE Magazine* (Nov/Dec 1999).

[5] BEIZER, B. *Software Testing Techniques, Second Edition.* International Thompson Computer Press, Boston, 1990.

[6] BLUM, C., AND ROLI, A. Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Computing Surveys 35*, 3 (September 2003), 268–308.

[7] BREYFOGLE, F. *Implementing six sigma: smarter solutions using statistical methods.* Wiley, Hoboken, 2003.

[8] BROWNLIE, R., PROWSE, J., AND PADKE, M. S. Robust testing of AT&T PMX/StarMAIL using OATS. *AT& T Technical Journal 71*, 3 (1992), 41–47.

[9] BROWNSWORD, L., OBERNDORF, T., AND SLEDGE, C. Developing new processes for COTS-based systems. *IEEE Software 17*, 4 (2000), 48–55.

[10] Burr, K., and Young, W. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review* (1998).

[11] Burroughs, K., Jain, A., and Erickson, R. L. Improved quality of protocol testing through techniques of experimental design. In *Supercomm/IC, IEEE International Conference on Communications* (1994), pp. 745 – 752.

[12] Cameron, P. Notes on classical groups. Queen Mary, University of London, School of Mathematical Sciences, http://www.maths.qmul.ac.uk/ pjc/books.html, 2000.

[13] Carroll, C. T. The cost of poor testing: A U.S. government study (part 1). *EDPACS, The EDP Audit, Control and Security Newsletter 31*, 1 (2003), 1–17.

[14] Carroll, C. T. The cost of poor testing: A U.S. government study (part 2). *EDPACS, The EDP Audit, Control and Security Newsletter 31*, 2 (2003), 1–16.

[15] Cawse, J. N., Ed. *Experimental Design for Combinatorial and High Throughput Materials Development.* John Wiley & Sons, Inc., New York, 2003.

[16] Chateauneuf, M. *Covering Arrays.* PhD dissertation, Michigan Technological University, Department of Mathematical Sciences, 2000.

[17] Chateauneuf, M., and Kreher, D. On the state of strength-three covering arrays. *Journal of Combinatorial Designs 10*, 4 (2002), 217–238.

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

[19] Chays, D., Dan, S., Deng, Y., Vokolos, F. I., Frankl, P. G., and Weyuker, E. J. AGENDA: A test case generator for relational database applications. Tech. rep., Polytechnic University, 2002.

[20] Chays, D., Deng, Y., Frankl, P., Dan, S., Vokolos, F., and Weyuker, E. An AGENDA for testing relational database applications. *Journal of Software Testing, Verification, and Reliability 14*, 10 (2004), 1–29.

[21] Cheng, C., Dumitrescu, A., and Schroeder, P. Generating small combinatorial test suites to cover input-output relationships. In *Proc. of the 3rd Quality Software International Conference (QSIC)* (November 2003), pp. 76–82.

[22] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. Method and system for automatically generating efficient test cases for systems having interacting elements. United States Patent, Number 5,542,043, 1996.

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

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

[25] Cohen, D. M., and Fredman, M. L. New techniques for designing qualitatively independent systems. *Journal of Combinatorial Designs 6*, 6 (1998), 411–416.

[26] Cohen, M. B., Colbourn, C. J., Collofello, J., Gibbons, P. B., and Mugridge, W. B. Variable strength interaction testing of components. In *Proc. of 27th Intl. Computer Software and Applications Conference (COMPSAC)* (November 2003), pp. 413–418.

[27] Cohen, M. B., Colbourn, C. J., Gibbons, P. B., and Mugridge, W. B. Constructing test suites for interaction testing. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)* (May 2003), pp. 38–48.

[28] Cohen, M. B., Colbourn, C. J., and Ling, A. C. H. Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics*. To appear accepted(Oct 03).

[29] Cohen, M. B., Colbourn, C. J., and Ling, A. C. H. Augmenting simulated annealing to build interaction test suites. In *14th IEEE Intl. Symp. on Software Reliability Engineering (ISSRE)* (November 2003), pp. 394–405.

[30] Colbourn, C., and Dinitz, J. Making the MOLS table. In *Computational and Constructive Design Theory*, W. D. Wallis, Ed. Kluwer Academic Press, 1996, pp. 67–134.

[31] Colbourn, C. J., Cohen, M. B., and Turban, R. C. A deterministic density algorithm for pairwise interaction coverage. In *IASTED Proc. of the Intl. Conference on Software Engineering* (February 2004), pp. 345–352.

[32] Colbourn, C. J., and Dinitz, J. H., Eds. *The CRC Handbook of Combinatorial Designs.* CRC Press, Boca Raton, 1996.

[33] Confidential. Company proprietary information.

[34] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Second Edition.* The MIT Press, Cambridge, 2001.

[35] Daich, G. T. New spreadsheet tool helps determine minimal set of test parameter combinations. *CrossTalk* (August 2003).

[36] Daich, G. T. Testing combinations of parameters made easy. In *Proc. of the IEEE Systems Readiness Technology Conference (AUTOTESTCON)* (September 2003), pp. 379–384.

[37] Dalal, S. R., Horgan, J. R., and R.Kettenring, J. Reliable software and communication: Software quality, reliability and safety. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)* (May 1993), pp. 425–435.

[38] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., and Lott, C. M. Model-based testing of a highly programmable system. In *Proc. of the Intl. Symp. on Software Reliability Engineering* (1998), pp. 174–179.

[39] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. Model-based testing in practice. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE)* (1999), pp. 285–294.

[40] Dalal, S. R., Jain, A., Patton, G., Rathi, M., and Seymour, P. Aetg^{SM}web: a Web based service for automatic efficient test generation from functional requirements. In *Proc. IEEE Workshop on Industrial Strength Formal Specification Techniques* (Oct 1998), pp. 84–85.

[41] Dalal, S. R., and Mallows, C. L. Factor-covering designs for testing software. *Technometrics 40*, 3 (August 1998), 234–243.

[42] Demko, E. Commercial-off-the shelf(COTS): A challenge to military equipment reliability. In *Proc. Intl. Symp. on Software Reliability Engineering, (ISSRE)* (1996), pp. 7–12.

[43] Dueck, G. New optimization heuristics - the great deluge algorithm and the record-to-record travel. *Journal of Computational Physics 104* (1993), 86–92.

[44] Dunietz, I. S., Ehrlich, W. K., Szablak, B. D., Mallows, C. L., and Iannino, A. Applying design of experiments to software testing. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE)* (1997), pp. 205–215.

[45] Gargano, L., Körner, J., and Vaccaro, U. Capacities: from information theory to extremal set theory. *Journal of Combinatorial Theory Series A 68*, 2 (1994), 296–316.

[46] Glover, F., and Kochenberger, G. A., Eds. *Handbook of Metaheuristics.* Kluwer Academic Publishers, Boston, 2003.

[47] Glover, F., and Laguna, M. *Tabu Search.* Kluwer Academic Publishers, Boston, 1997.

[48] GNU Free Software Foundation.
GCC compiler options. http://gcc.gnu.org/, 2004.

[49] Harman, M., and Jones, B. F. Search-based software engineering. *Information and Software Technology 43*, 14 (2001), 833–839.

[50] Harrold, M. J. Testing: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (May 2000), pp. 61 – 72.

[51] Hartman, A. Software and hardware testing using combinatorial covering suites. In *Haifa Workshop on Interdisciplinary Applications and Graph Theory, Combinatorics and Algorithms* (June 2002).

[52] Hartman, A. Personal communication. IBM, Haifa, 2003.

[53] Hartman, A., and Raskin, L. Problems and algorithms for covering arrays. *Discrete Math 284* (2004), 149 – 156.

[54] Hedayat, A., Sloane, N., and Stufken, J. *Orthogonal Arrays.* Springer-Verlag, New York, 1999.

[55] Horgan, J. R., London, S., and Lyu, M. R. Achieving software quality with testing coverage measures. *IEEE Software* (1994), 60–69.

[56] Huller, J. Reducing time to market with combinatorial design method testing. In *Proceedings of the 2000 International Council on Systems Engineering (INCOSE) Conference* (2000).

[57] Ido, N., and Kikuno, T. Lower bounds estimation of factor-covering design sizes. *Journal of Combinatorial Designs 11* (2003), 89–99.

[58] Kaner, C., Bach, J., and Pettichord, B. *Lessons Learned in Software Testing.* John Wiley & Sons, New York, 2001.

[59] Katona, G. Two applications (for search theory and truth functions) of Sperner type theorems. *Periodica Mathematica 3* (1973), 19–26.

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

[61] Kobayashi, N., Tsuchiya, T., and Kikuno, T. Applicability of non-specification-based approaches to logic testing for software. In *Proc. of the Intl. Conf. on Dependable Systems and Networks* (July 2001), pp. 337–346.

[62] Kobayashi, N., Tsuchiya, T., and Kikuno, T. A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters 81* (2002), 85–91.

[63] Körner, J., and Lucertini, M. Compressing inconsistent data. *IEEE Transactions on Information Theory 40*, 3 (May 1994), 706 – 715.

[64] Kreher, D. L., and Stinson, D. R. *Combinatorial Algorithms, Generation, Enumeration and Search.* CRC Press, Boca Raton, 1999.

[65] Kuhfeld, W. F. Marketing research methods in SAS. Tech. rep., SAS Institute, Inc., May 2003. http://support.sas.com/techsup/technote/ts689.pdf.

[66] Kuhn, D., and Reilly, M. An investigation of the applicability of design of experiments to software testing. In *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop* (2002), pp. 91–95.

[67] Kuhn, D., Wallace, D. R., and Gallo, A. M. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering 30*, 6 (2004), 418–421.

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

[69] Martirosyan, S., and van Trung, T. On t-covering arrays. *Designs, Codes and Cryptography 32*, 1, 323–339.

[70] Memon, A., Porter, A., Yilmaz, C., Nagarajan, A., Schmidt, D., and Natarajan, B. Skoll: Distributed continuous quality assurance. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)* (May 2004), pp. 459 – 468.

[71] Meyers, G. J. *The Art of Software Testing.* John Wiley & Sons, Inc., New York, 1979.

[72] Microsoft Corporation. Administering SQL Server: Setting configuration options. http://msdn.microsoft.com/, 2004.

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

[74] Nair, V., James, D., Ehrlich, W., and Zevallos, J. A statistical assessment of some software testing strategies and application of experimental design techniques. *Statistica Sinica 8*, 1 (1998), 165 – 184.

[75] Nguyen, N. A note on the construction of near-orthogonal arrays with mixed levels and economic run size. *Technometrics 38*, 3 (August 1996), 279–283.

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

[77] Nurmela, K., and Östergård, P. R. J. Constructing covering designs by simulated annealing. Tech. rep., Digital Systems Laboratory, Helsinki Univ. of Technology, 1993.

[78] Oracle. Oracle technology network. http://www.oracle.com/, 2004.

[79] Östergård, P. R. J. Constructions of mixed covering codes. Tech. rep., Digital Systems Laboratory, Helsinki Univ. of Technology, 1991.

[80] Phadke, M. S. *Quality Engineering Using Robust Design.* Prentice-Hall, Inc., New Jersey, 1989.

[81] Poljak, S., Pultr, A., and Rödl, V. On qualitatively independent partitions and related problems. *Discrete Applied Math 6*, 2 (1983), 193–205.

[82] Poljak, S., and Tuza, Z. On the maximum number of qualitatively independent partitions. *Journal of Combinatorial Theory Series A 51*, 1 (1989), 111–116.

[83] Réyni, A. *Foundations of Probability.* Wiley, New York, 1971.

[84] Roux, G. *k-Propriétés dans des Tableaux de n Colonnes; Gas Particulier de la k-Surjectivité et de la k-Permutivité.* PhD dissertation, University of Paris, Department of Mathematics, 1987.

[85] Roy, R. K. *Design of Experiments Using the Taguchi Approach.* John Wiley & Sons, Inc., New York, 2001.

[86] Sasha, D. E., Kouranov, A. Y., Lejay, L. V., Chou, M. F., and Coruzzi, G. M. Using combinatorial design to study regulation by multiple input signals. a tool for parsimony in the post-genomics era. *Plant Physiology 127* (December 2001), 1590–1594.

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

[88] Sherwood, G. On the construction of orthogonal arrays and covering arrays using permutation groups. http://home.att.net/ gsherwood/cover.htm.

[89] Sloane, N. Covering arrays and intersecting codes. *Journal of Combinatorial Designs 1*, 1 (1993), 51–63.

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

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

[92] Stevens, B. *Transversal Covers and Packings.* PhD dissertation, University of Toronto, Department of Mathematics, 1998.

[93] Stevens, B., Ling, A. C. H., and Mendelsohn, E. A direct construction of transversal covers using group divisible designs. *Ars Combinatoria 63* (2002), 145–159.

[94] Stevens, B., and Mendelsohn, E. Efficient software testing protocols. In *Proc of Center for Advanced Studies Conf.(CASCON)* (1998), pp. 270–293.

[95] Stevens, B., and Mendelsohn, E. New recursive methods for transversal covers. *Journal of Combinatorial Designs 7*, 3 (1999), 185–203.

[96] Stevens, B., Moura, L., and Mendelsohn, E. Lower bounds for transversal covers. *Designs Codes and Cryptography 15*, 3 (1998), 279–299.

[97] Taguchi, G. *System of Experimental Design.* UNIPUBKraus International Publications, New York, 1987.

[98] Tai, K. C., and Yu, L. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering 28*, 1 (2002), 109–111.

[99] Tatsumi, K., Watanabe, S., Takeuchi, Y., and Shimokawa, H. Conceptual support for test case design. In *Proc. of 11th Intl. Computer Software and Applications Conference (COMPSAC)* (October 1987), pp. 285–290.

[100] WEYUKER, E. J. Testing component-based software: A cautionary tale. *IEEE Software 15*, 5 (1998), 54–59.

[101] WHITE, L. J. Regression testing of GUI event interactions. In *Intl. Conf. on Software Maintenance* (1996), pp. 350 – 358.

[102] WILLIAMS, A. W. Determination of test configurations for pair-wise interaction coverage. In *Thirteenth Int. Conf. Testing Communication Systems* (2000), pp. 57–74.

[103] WILLIAMS, A. W. *Software Component Interaction Testing*. PhD dissertation, University of Ottawa, School of Information Technology and Engineering, 2002.

[104] WILLIAMS, A. W., AND PROBERT, R. L. A practical strategy for testing pair-wise coverage of network interfaces. In *Proc. Intl. Symp. on Software Reliability Engineering, (ISSRE)* (1996), pp. 246–54.

[105] WILLIAMS, A. W., AND PROBERT, R. L. A measure for component interaction test coverage. In *Proc. ACS/IEEE Intl. Conf. on Computer Systems and Applications* (2001), pp. 301–311.

[106] WILLIAMS, A. W., AND PROBERT, R. L. Formulation of the interaction test coverage problem as an integer program. In *Fourteenth International Conference on Testing of Communicating Systems (TestCom)* (March 2002), pp. 283–298.

[107] XU, H. An algorithm for constructing orthogonal and nearly-orthogonal arrays with mixed levels and small runs. *Technometrics 44*, 4 (November 2002), 356–368.

[108] YILMAZ, C., COHEN, M. B., AND PORTER, A. Covering arrays for efficient fault characterization in complex configuration spaces. In *Intl. Symp. on Software Testing and Analysis (ISSTA)* (July 2004), pp. 45–54.

[109] Yu, L., and Tai, K. C. In-parameter-order: a test generation strategy for pairwise testing. In *Proc. Third IEEE Intl. High-Assurance Systems Engineering Symp* (1998), pp. 254–261.

[110] Yu-Wen, T., and Aldiwan, W. S. Automating test case generation for the new generation mission software system. In *Proc. IEEE Aerospace Conf.* (2000), pp. 431–437.

Appendix

This appendix contains a sample of covering arrays, $(2,1)$-covering arrays and difference covering arrays from Chapters 4, 5 and 6. All of the covering arrays and $(2,1)$ covering arrays represent new bounds. The set of difference covering arrays without zero differences provide the first bounds for objects of this type. We have included "small" examples only. A more comprehensive repository is being constructed as future work. It will be available for download via the Internet.

All of the objects are represented as $N \times k$ arrays. For all objects (except the difference covering arrays) each column of the array uses a unique set of integers to represent its values. The difference covering arrays use the same set of integers in each of its columns. All covering arrays are represented in the same order as in the notation. Suppose we have an $MCA(3, 2^3 4^3)$. This will contain six columns. The first column will use the values 0 and 1, the second column will use 2 and 3 and the third will use 4 and 5. The last three columns of this array will each contain four possible values. In the variable strength covering arrays the vector of higher strength covering arrays are disjoint and presented in the order from left to right that matches the notation.

The objects are presented in the order they appear in this thesis. References are made to the tables that contain each one.

Mixed level covering arrays from Table 4.1 (page 80)

$MCA(2, 5^1 3^8 2^2), N = 15$										
0	7	9	13	14	17	20	23	26	30	32
1	7	8	12	14	19	20	25	28	30	32
4	6	10	12	14	17	21	25	26	29	31
4	5	9	13	15	18	20	24	28	29	31
3	5	8	11	14	17	22	24	26	29	32
1	5	10	13	15	17	22	23	27	29	31
3	6	9	13	16	19	20	25	27	30	31
4	7	8	11	16	19	22	23	27	30	32
1	6	9	11	16	18	21	24	26	30	32
0	5	10	12	16	19	21	24	27	29	31
2	6	9	12	15	19	22	23	26	30	32
0	6	8	11	15	18	22	25	28	30	31
2	7	10	11	14	18	20	24	27	29	31
3	7	10	12	15	18	21	23	28	30	32
2	5	8	13	16	17	21	25	28	30	32

$MCA(2, 5^1 4^4 3^{11} 2^5), N = 21$																				
4	7	12	15	17	23	26	28	30	35	36	40	42	47	49	51	55	57	59	61	62
0	5	9	14	17	23	25	29	31	33	37	40	42	46	48	51	54	57	58	61	63
3	8	11	15	18	21	25	29	31	35	37	40	42	45	48	51	54	56	58	60	62
1	6	9	15	20	21	24	27	30	35	38	41	42	45	49	53	55	56	58	60	62
4	8	9	16	18	22	24	27	32	35	37	39	43	46	50	52	55	56	58	60	63
0	5	10	15	20	22	25	28	32	35	38	40	44	46	50	51	54	56	58	60	63
2	5	10	15	18	21	24	29	32	33	38	39	42	47	50	52	55	57	58	61	62
1	8	10	13	17	23	26	28	32	33	37	39	44	45	48	52	54	56	59	60	63
1	5	12	16	19	22	25	29	31	34	37	40	42	47	50	53	54	57	59	61	62
2	7	9	13	19	22	25	27	31	35	37	40	43	45	49	51	55	57	59	60	62
3	5	10	14	19	22	26	27	30	33	36	39	43	46	49	53	55	57	59	60	63
4	5	11	13	20	23	25	29	31	33	37	41	43	45	48	53	55	56	59	60	63
0	6	12	13	18	23	25	29	30	34	36	39	42	46	48	52	55	56	59	61	62
3	7	9	16	20	23	26	28	31	34	36	39	44	47	48	52	54	57	58	61	63
2	8	12	14	20	23	26	28	30	34	36	41	44	45	50	53	54	57	58	60	63
0	8	11	15	19	23	24	27	32	34	38	41	43	47	49	52	54	57	58	61	62
0	7	10	16	18	21	26	29	30	33	38	41	44	45	49	53	55	56	58	61	63
2	6	11	16	17	22	26	27	30	34	37	41	44	46	48	51	55	56	58	61	62
4	6	10	14	19	21	24	28	31	34	38	40	44	47	48	52	54	56	59	60	63
3	6	12	13	17	21	24	27	32	33	38	41	43	47	50	53	55	57	58	61	63
1	7	11	14	18	21	24	28	32	35	36	39	43	46	50	51	54	57	59	61	62

Mixed level covering array from Table 4.1 (page 80)

$MCA(2, 6^1 5^1 4^6 3^8 2^3), N = 30$																		
0	6	14	17	20	26	28	31	36	40	42	46	48	51	55	58	60	62	64
5	7	13	17	22	23	30	32	36	40	43	44	47	52	55	57	59	62	63
0	7	12	18	19	24	28	33	36	40	41	45	49	50	53	57	60	61	64
1	9	13	17	21	24	30	34	35	40	43	46	48	51	53	56	60	61	64
2	9	11	18	22	24	27	33	36	39	43	46	49	51	54	57	60	62	64
5	6	12	15	21	25	30	34	35	39	41	46	49	51	53	58	59	62	63
3	7	11	15	22	26	27	34	37	39	42	46	48	50	55	57	60	61	64
2	8	13	15	19	26	28	34	36	39	41	45	47	51	54	58	59	61	63
4	10	11	18	19	24	29	31	37	40	43	45	48	50	55	58	59	61	63
0	10	11	16	19	23	30	34	35	40	43	44	47	52	55	58	59	61	63
0	8	14	15	22	24	29	32	37	38	41	44	48	50	54	56	59	61	63
2	10	14	17	21	25	30	31	37	38	41	45	48	52	54	57	60	62	63
4	8	11	17	20	25	30	33	35	39	41	45	47	51	53	57	60	62	63
3	6	13	18	20	24	30	33	37	38	43	46	47	52	54	56	60	61	64
1	6	11	16	19	25	27	32	37	39	42	45	49	52	54	58	60	61	64
4	6	13	16	22	24	29	31	36	38	42	44	49	52	55	58	60	62	64
5	9	14	16	20	26	28	33	37	39	42	45	48	50	53	57	60	61	64
5	8	11	18	21	26	29	34	35	38	42	45	47	51	54	57	59	62	64
5	10	12	17	19	24	27	31	36	38	42	46	47	51	53	56	60	62	63
1	10	13	15	20	26	30	33	36	40	42	44	49	50	55	57	59	62	64
1	8	12	16	22	23	29	32	37	39	41	46	47	51	54	57	59	62	63
4	9	12	15	21	26	28	32	37	40	43	46	48	52	53	56	60	62	63
1	7	14	18	21	23	28	31	35	38	42	44	47	51	53	58	60	61	63
2	7	12	16	21	25	29	33	36	40	43	44	49	52	55	56	60	61	64
4	7	14	15	20	23	27	34	37	40	41	45	48	51	54	56	59	61	64
2	6	12	18	20	23	29	32	35	39	42	44	47	50	53	57	59	61	64
3	10	11	18	22	25	28	32	35	39	41	45	47	51	53	56	59	61	63
0	9	13	15	21	25	27	31	35	39	41	45	47	50	55	56	59	62	63
3	9	14	17	19	23	29	33	36	38	43	44	49	50	54	58	59	61	63
3	8	12	16	21	23	27	31	35	40	43	44	49	52	55	58	59	62	64

Variable strength covering arrays from Table 5.5 (page 97)

$VCA(2, 3^{15}, (CA(3, 3^3))), N = 27$														
0	5	7	11	12	15	19	23	25	27	31	33	38	40	43
2	5	6	10	13	17	18	21	26	28	31	35	37	39	42
2	3	8	10	12	15	19	23	24	27	30	34	36	39	44
0	4	8	9	14	15	19	22	25	28	31	35	37	40	43
1	3	7	9	12	16	18	23	24	29	30	35	38	40	42
1	4	6	10	14	16	18	21	26	28	30	33	36	41	43
1	5	6	9	14	16	19	21	26	29	32	34	38	40	43
1	3	6	10	12	15	19	21	24	29	30	33	37	40	43
2	4	8	9	13	17	18	22	24	29	32	34	38	41	42
0	3	6	9	12	15	19	22	24	29	32	35	36	40	44
1	4	7	9	12	15	19	22	24	29	32	33	36	39	44
2	4	7	11	14	16	20	22	25	27	32	34	37	40	42
1	3	8	9	13	17	19	23	26	27	31	33	37	39	42
0	5	6	10	14	15	19	23	25	28	32	33	36	40	44
2	5	8	10	14	15	20	23	24	27	30	33	37	41	44
2	5	7	11	13	16	20	22	26	29	31	34	37	39	44
0	4	7	11	12	15	19	23	26	28	31	34	37	39	43
1	5	7	10	14	17	20	21	25	28	30	34	37	40	44
0	3	7	10	14	15	18	22	26	27	30	35	38	41	42
2	3	6	11	12	17	20	23	24	28	32	35	36	40	43
0	3	8	9	13	15	20	21	25	27	32	34	36	40	42
2	3	7	11	12	15	20	21	24	27	31	33	36	41	44
0	4	6	10	12	16	19	23	25	27	30	35	38	41	43
2	4	6	11	14	15	20	21	25	29	32	33	37	39	42
0	5	8	11	14	17	20	21	24	28	31	34	38	39	44
1	5	8	11	13	16	20	23	25	29	30	34	38	41	43
1	4	8	11	13	16	18	21	25	29	32	34	38	39	44

$VCA(2, 3^{15}, (CA(3, 3^3), CA(3, 3^3))), N = 27$														
2	3	8	9	13	15	20	21	26	29	31	35	37	41	42
0	3	8	11	13	16	19	21	24	29	30	33	36	41	43
1	4	8	9	14	15	18	21	24	28	30	35	38	40	44
2	5	7	10	12	17	20	21	25	29	32	35	38	41	43
0	4	7	9	12	15	19	23	24	27	31	33	38	39	44
2	3	6	10	12	16	18	22	24	28	30	34	36	39	44
0	5	6	10	13	15	18	22	25	27	31	34	38	40	43
0	5	8	10	14	16	20	23	26	27	31	35	36	41	44
2	5	6	10	13	17	18	22	24	28	31	33	36	41	42
1	4	6	11	12	17	20	21	26	28	31	34	36	40	44
0	4	6	9	12	17	18	23	25	29	32	34	36	41	43
2	4	7	9	13	17	19	21	24	29	32	33	36	39	44
0	4	8	9	12	16	19	21	25	28	32	34	38	39	42
1	4	7	10	14	15	20	21	26	28	30	33	36	39	43
0	3	6	10	12	15	19	23	26	28	32	34	37	41	44
1	3	6	11	13	15	18	23	25	27	32	35	37	39	43
1	5	7	11	14	15	19	22	26	29	32	33	36	39	43
1	5	6	10	14	17	18	23	25	29	31	35	37	39	42
0	5	7	10	13	16	19	23	25	27	30	33	37	40	43
2	5	8	11	14	16	20	22	25	27	31	33	38	41	44
1	5	8	9	14	17	20	23	25	27	31	34	36	39	42
2	3	7	9	13	16	18	22	26	29	32	34	38	40	43
2	4	8	11	13	17	20	23	24	29	30	35	37	40	43
0	3	7	11	14	17	18	21	24	27	30	35	36	39	42
1	3	8	11	12	15	19	22	26	29	31	35	37	41	42
1	3	7	9	14	16	18	23	24	29	32	33	37	39	42
2	4	6	11	12	16	18	22	24	29	32	34	38	40	42

Variable strength covering arrays from Table 5.5 (page 97)

$VCA(2, 3^{15}, (CA(3, 3^3), CA(3, 3^3), CA(3, 3^3))), N = 27$														
2	5	8	9	13	17	19	23	26	27	32	35	38	41	42
1	3	6	10	13	17	18	21	24	27	30	35	37	40	42
0	4	7	11	13	16	20	21	26	28	32	33	37	39	44
0	4	6	9	14	17	19	22	25	29	31	35	36	40	43
0	5	8	10	13	16	20	22	25	29	32	33	36	40	43
0	5	7	10	14	15	19	21	24	28	30	34	36	39	42
1	4	7	10	12	15	20	22	24	28	32	34	38	40	43
2	4	6	10	13	15	20	23	25	27	30	35	36	39	42
1	5	6	10	12	17	19	23	24	29	30	33	36	39	42
0	5	6	9	12	16	19	22	24	27	32	34	38	41	44
2	5	6	11	14	17	18	22	26	27	30	35	38	39	42
0	4	8	11	13	17	20	23	24	27	31	35	38	40	44
1	3	8	11	14	15	19	23	25	28	30	35	36	41	43
2	3	6	9	12	15	19	22	26	29	30	35	37	40	44
2	3	7	9	12	17	19	21	26	27	31	35	37	39	43
2	4	7	10	14	17	18	22	24	29	30	34	37	41	42
2	5	7	11	12	16	18	23	25	29	30	33	37	40	42
1	4	6	10	14	16	20	22	26	27	30	34	37	39	44
0	3	7	11	12	17	18	22	25	28	32	34	38	39	42
1	4	8	9	14	16	18	23	24	29	30	35	38	40	42
1	3	7	9	13	16	18	21	25	29	31	34	36	39	43
2	3	8	11	12	15	20	23	26	27	30	34	36	41	43
0	3	8	9	13	15	18	21	26	29	31	33	38	41	44
1	5	8	11	13	15	20	21	24	27	31	33	37	39	43
2	4	8	11	14	16	19	21	25	28	32	33	37	40	43
0	3	6	9	14	15	20	21	25	28	32	33	36	41	44
1	5	7	10	12	16	18	23	26	28	31	34	38	39	42

$VCA(2, 3^{15}, (CA(3, 3^4))), N = 27$														
2	5	6	10	14	16	18	23	24	29	31	33	38	39	43
1	3	7	11	14	17	19	23	24	27	32	34	38	39	42
0	5	7	10	12	17	19	22	24	29	32	35	36	39	43
2	3	8	11	12	16	18	23	26	27	31	33	38	41	44
0	3	6	11	13	15	18	22	25	29	32	34	37	40	44
1	4	7	10	14	15	20	22	25	27	30	35	36	41	44
1	5	8	10	12	16	20	21	25	28	30	35	36	40	43
2	4	6	11	12	15	19	23	25	27	31	33	36	39	44
0	3	8	10	14	16	19	21	25	28	32	34	37	41	44
1	4	8	11	12	16	18	21	26	27	31	33	37	41	44
0	4	7	11	13	17	20	23	24	28	32	35	37	40	44
0	5	8	11	12	16	19	22	24	29	31	35	38	39	42
0	3	7	9	14	15	19	23	25	28	30	34	36	41	44
2	3	7	10	14	17	18	22	26	28	30	33	37	39	42
1	5	7	9	13	16	20	22	26	27	31	33	38	41	44
0	5	6	9	14	15	18	22	24	27	30	33	36	40	42
2	3	6	9	12	15	20	21	25	27	32	33	38	41	43
0	4	8	9	13	15	19	21	26	29	32	35	36	41	43
1	4	6	9	14	17	18	22	24	27	31	34	38	40	43
2	4	8	10	12	17	19	21	25	27	32	34	38	40	42
0	4	6	10	13	15	20	22	26	28	31	34	38	39	44
2	4	7	9	12	15	19	21	25	29	30	35	37	41	43
1	3	6	10	14	16	20	22	24	28	30	35	38	40	44
2	5	8	9	14	15	19	23	25	29	30	34	37	39	44
2	5	7	11	13	17	18	23	24	29	31	35	36	41	42
1	5	6	11	13	17	19	21	24	27	30	33	38	39	43
1	3	8	9	12	17	20	22	26	29	31	35	38	40	42

Variable strength covering array from Table 5.5 (page 97)

$VCA(2, 3^{15}, (CA(3, 3^5))), N = 33$														
2	3	6	9	12	16	19	22	24	28	30	35	37	39	44
0	4	7	10	13	17	19	22	26	29	31	34	38	41	44
2	3	8	10	12	16	18	23	26	29	30	33	36	40	42
1	3	7	9	14	17	20	23	25	27	32	34	38	39	42
1	3	6	10	13	15	20	21	25	27	32	34	38	41	43
1	5	7	10	12	17	18	21	24	28	31	34	38	40	43
2	4	8	10	13	15	18	22	25	27	30	34	37	40	43
1	5	6	9	14	16	20	21	26	29	32	34	38	40	44
2	4	6	11	14	16	18	22	26	27	31	35	38	39	43
0	4	8	11	12	15	18	21	25	28	31	33	37	41	43
1	3	6	11	12	15	20	23	24	27	30	35	38	40	42
2	3	7	10	14	15	20	21	24	29	30	35	37	39	44
2	5	8	11	12	16	18	23	26	29	30	34	37	41	44
0	5	6	11	13	17	20	23	25	27	30	34	36	41	44
0	5	8	10	14	17	19	21	25	27	32	33	36	41	43
1	3	8	11	14	15	18	22	24	29	32	35	37	39	44
0	3	8	9	13	16	18	23	25	28	32	34	37	39	43
0	5	7	11	14	16	19	22	25	29	32	34	36	40	44
2	4	7	9	12	15	18	21	26	29	32	33	36	40	43
0	5	6	9	12	15	19	23	25	28	31	35	36	41	42
1	5	8	11	13	15	20	23	25	28	32	33	37	41	42
0	4	7	9	14	17	18	23	25	27	30	33	36	39	43
0	3	6	10	14	16	19	22	24	27	31	34	38	41	44
2	5	7	9	13	16	18	23	26	29	30	35	38	41	44
1	4	8	9	12	17	20	21	25	27	31	34	38	41	44
1	4	7	11	13	17	19	21	26	28	31	35	38	40	42
1	4	6	9	13	17	20	21	24	28	31	33	38	39	44
0	4	6	10	12	16	19	22	24	28	30	33	38	40	42
0	3	7	11	12	16	19	23	26	28	30	35	37	41	43
2	5	8	9	14	17	20	21	24	29	31	34	37	40	42
2	3	7	11	13	17	20	22	24	28	32	33	38	40	44
1	4	8	10	14	15	20	23	24	28	32	33	36	41	43
2	5	6	10	13	15	20	22	24	29	32	34	38	39	44

Variable strength covering array from Table 5.5 (page 97)

$VCA(2, 3^{15}, (CA(3, 3^4), CA(3, 3^5), CA(3, 3^6))), N = 33$														
2	5	8	10	14	15	18	23	26	27	32	34	38	40	42
1	4	7	11	14	16	18	21	24	28	30	35	38	41	42
1	5	8	9	13	15	20	23	25	27	32	33	37	39	43
2	4	8	11	13	15	18	22	24	29	31	35	38	40	44
2	3	7	10	12	16	18	23	24	28	30	35	37	39	44
0	3	8	10	12	15	18	23	25	29	30	34	38	39	43
2	5	6	9	13	17	19	21	24	28	32	33	38	39	44
1	4	6	9	14	15	19	23	24	27	31	34	36	40	43
2	3	8	9	13	15	18	21	26	27	30	33	38	40	44
0	3	6	9	12	15	20	21	24	29	32	34	37	40	44
1	4	8	9	14	16	20	21	26	28	32	34	37	41	43
0	4	7	10	12	16	20	22	25	28	30	34	36	40	44
0	4	7	10	14	16	19	22	24	27	32	35	36	41	44
1	3	8	11	12	16	19	21	25	29	32	33	38	41	42
0	4	8	9	13	16	19	23	26	29	31	35	37	41	43
2	4	7	9	13	17	20	22	26	27	30	34	37	41	42
1	5	7	10	13	17	20	21	25	29	31	33	36	39	44
0	5	8	11	13	17	18	23	24	27	30	33	36	39	42
0	5	7	9	12	15	19	22	26	29	31	34	36	41	42
0	3	7	11	13	15	19	22	25	27	31	34	37	39	44
2	3	6	11	14	16	20	23	25	29	30	35	36	40	42
2	4	7	9	14	15	20	22	26	28	30	33	36	41	43
0	5	6	10	13	16	20	22	24	29	30	33	37	41	44
2	3	6	10	12	17	19	22	24	27	31	35	38	39	42
1	4	8	10	14	17	18	22	25	28	31	33	37	40	42
2	4	6	10	12	16	18	22	26	28	32	35	38	40	43
1	3	7	9	14	17	19	21	26	27	30	35	37	40	43
1	5	6	11	12	17	20	23	26	28	31	35	36	39	43
0	4	6	11	12	17	18	21	26	29	32	35	37	39	42
2	5	7	11	14	17	20	23	24	27	31	33	38	41	43
1	3	6	10	12	17	19	23	25	29	32	33	36	40	43
0	5	8	9	13	16	18	21	25	28	31	34	38	41	44
0	5	7	9	14	15	19	21	25	28	32	34	36	39	42

Variable strength covering array from Table 5.5 (page 97)

$VCA(2,3^{15},(CA(3,3^6))), N=33$														
1	5	8	10	14	17	19	22	26	29	30	35	37	39	44
2	3	8	11	14	15	20	21	26	27	32	33	36	41	42
0	5	6	11	14	16	20	22	25	28	31	34	36	39	42
0	4	7	11	13	15	18	23	26	27	30	33	38	41	44
1	5	7	9	13	16	18	22	26	29	30	33	38	41	43
0	3	7	9	14	17	19	23	24	27	32	34	38	41	43
0	3	6	9	13	15	19	21	26	29	31	34	37	40	43
2	4	7	10	14	16	20	21	26	27	30	34	38	40	43
2	5	8	10	12	16	19	22	24	27	31	34	38	40	42
2	5	7	11	12	17	19	21	26	29	32	33	38	39	44
2	5	6	10	13	15	20	21	25	29	32	35	36	40	43
2	3	7	11	13	16	19	21	25	29	30	35	37	41	43
1	3	7	10	12	15	19	21	26	29	30	34	38	39	44
1	3	8	9	14	16	19	23	24	27	31	33	37	39	42
0	4	7	9	12	16	19	23	26	29	30	34	38	39	42
0	3	8	10	13	16	18	22	25	28	30	34	36	40	42
1	4	7	11	14	17	18	21	24	28	32	35	38	40	44
0	3	8	11	12	17	18	21	24	28	31	35	37	41	43
1	5	8	11	13	15	20	23	24	29	31	34	36	41	42
2	5	7	9	14	15	18	21	25	28	30	35	36	39	42
1	4	6	10	13	16	19	23	24	28	32	34	38	39	42
0	5	8	9	12	15	19	23	25	27	32	33	36	41	44
2	4	8	9	13	17	18	21	26	29	31	34	38	40	44
0	5	7	10	13	17	20	23	25	27	31	34	38	40	44
2	3	6	9	12	16	20	23	26	28	31	34	36	40	43
2	4	6	11	12	15	18	22	26	28	32	34	37	39	42
0	4	8	10	14	15	19	22	25	27	31	35	36	40	42
1	4	8	11	12	16	18	23	24	29	30	35	38	40	44
0	4	6	10	12	17	18	21	25	29	31	33	36	40	44
2	3	6	10	14	17	19	23	26	29	31	35	38	41	42
1	4	6	9	14	15	20	22	24	28	32	33	36	39	44
1	3	6	11	13	17	19	21	25	28	30	34	37	39	43
1	5	6	9	12	17	20	21	24	27	31	34	37	40	44

Variable strength covering array from Table 5.5 (page 97)

$VCA(2,3^{15},(CA(3,3^7))), N=41$														
1	4	8	11	13	16	20	21	26	29	31	33	37	40	42
1	3	6	10	12	16	19	22	25	29	31	35	36	40	43
0	5	6	9	14	17	18	23	24	29	32	33	37	41	43
0	3	8	9	12	16	20	23	24	28	30	35	37	41	43
0	4	6	9	14	16	19	21	24	28	32	33	37	40	42
0	5	6	10	12	15	20	23	24	29	31	34	36	39	42
2	3	6	10	14	17	18	21	26	27	31	34	38	39	43
0	4	7	11	14	16	18	21	25	29	30	35	38	41	42
2	3	7	9	14	15	20	22	26	28	32	33	38	41	43
1	4	7	9	12	16	18	23	24	29	30	34	38	39	42
2	4	7	10	13	16	19	21	26	27	31	34	37	41	44
0	3	7	10	14	15	19	21	24	27	30	35	36	40	44
2	3	8	11	14	16	19	22	25	29	30	35	37	41	42
2	4	8	9	13	17	18	21	25	27	31	33	38	39	44
2	5	7	9	12	17	19	23	26	28	30	33	36	41	42
1	3	7	10	13	17	20	21	25	28	30	35	38	41	44
0	4	8	11	12	15	19	22	26	28	31	34	36	39	44
1	3	8	11	12	17	18	23	26	27	31	34	38	39	42
2	3	6	9	12	15	18	21	26	27	30	33	38	39	44
1	5	8	10	13	17	19	21	26	28	32	34	37	40	42
1	5	6	11	12	16	18	22	26	27	31	35	37	41	42
1	5	7	10	14	16	20	21	24	28	32	33	37	41	43
2	3	7	9	13	16	18	21	24	29	32	35	38	39	42
0	5	8	11	14	17	20	23	26	29	31	33	38	41	42
1	3	7	11	12	15	20	23	26	27	32	33	36	39	43
2	5	8	10	12	16	18	23	26	29	30	35	36	39	44
1	4	7	11	14	17	19	21	26	29	30	33	38	41	42
0	3	8	10	13	15	18	23	25	29	32	35	37	40	43
2	4	6	11	12	17	20	23	26	28	30	35	36	40	44
2	5	6	11	14	15	19	23	24	28	31	35	38	39	43
1	4	6	10	13	15	18	21	24	27	32	34	37	41	44
2	4	8	10	14	15	20	23	24	28	32	35	38	41	42
2	5	6	9	13	16	20	21	25	27	31	35	36	39	44
0	4	7	9	13	15	20	21	26	28	31	34	37	39	43
1	5	8	9	14	15	18	21	24	28	32	34	36	41	44
1	3	6	9	14	17	20	23	24	29	32	34	38	40	42
0	5	7	10	13	16	19	22	25	28	32	34	36	41	43
1	3	8	9	13	15	19	22	24	27	30	35	36	39	44
0	3	6	11	13	17	19	22	26	28	32	35	38	41	42
2	5	7	11	13	15	18	23	26	29	32	33	36	39	44
0	4	7	10	12	17	18	21	26	29	32	33	38	39	43

Variable strength covering array from Table 5.5 (page 97)

$VCA(2,3^{15},(CA(3,3^9))), N=50$														
2	5	7	9	14	17	20	23	25	29	32	33	37	39	44
2	3	7	11	12	15	20	22	26	27	31	34	37	40	44
2	4	7	9	13	15	18	22	25	27	32	35	36	40	44
2	5	8	9	13	16	19	22	26	27	31	34	36	39	42
1	3	8	9	13	17	19	23	24	29	31	33	38	41	44
0	5	6	11	12	16	20	21	24	28	32	35	38	41	42
1	4	8	9	14	15	20	23	24	28	32	33	36	41	42
2	4	7	10	13	16	20	21	24	29	30	34	37	41	43
1	5	6	9	13	15	20	21	26	28	32	34	37	40	43
0	5	7	10	13	15	18	23	24	28	31	35	36	40	42
0	4	6	9	13	17	19	21	25	27	32	35	38	39	42
1	3	7	10	12	17	18	21	25	27	30	34	36	40	42
2	5	7	11	14	15	19	21	24	29	32	35	36	39	44
0	5	6	10	14	15	20	22	25	27	30	35	37	40	43
0	3	7	9	13	16	20	23	25	28	30	33	36	39	42
2	4	6	10	14	17	19	22	24	27	30	35	38	41	44
1	3	8	11	13	16	18	21	26	29	30	34	37	39	43
0	4	6	11	13	15	20	23	26	29	31	34	38	41	43
1	4	7	10	12	15	19	21	26	27	30	34	36	40	44
2	5	6	10	12	17	18	21	25	29	31	33	38	40	42
1	5	7	10	14	16	18	23	26	27	32	33	37	41	44
0	3	8	10	14	16	20	22	26	28	31	35	38	41	43
0	3	6	11	13	15	18	22	24	28	30	34	36	41	43
1	3	7	9	14	15	19	22	25	28	30	33	36	40	42
1	5	7	9	12	16	20	22	24	29	30	34	37	40	44
1	4	6	9	12	16	18	22	26	27	30	34	36	41	43
2	3	6	11	12	15	19	23	25	28	30	35	38	41	43
2	3	8	10	14	15	20	21	25	27	31	33	36	40	42
2	3	8	9	12	16	19	23	26	28	32	33	37	40	44
0	5	7	11	12	17	19	23	26	28	31	35	37	40	44
2	3	6	9	14	16	18	23	24	29	31	34	38	39	43
1	5	6	10	13	16	19	23	25	27	31	33	36	40	43
2	3	7	9	13	16	19	21	26	27	32	34	37	40	44
1	4	7	11	13	17	19	23	25	27	32	33	36	40	43
1	5	8	10	13	17	18	22	24	27	32	33	38	41	42
0	3	8	9	12	15	18	21	24	27	31	35	38	40	43
0	4	8	9	12	17	20	22	25	27	31	35	38	41	42
0	4	8	10	13	15	19	21	25	27	30	34	38	39	43
1	3	6	11	14	17	20	21	24	27	31	33	38	39	44
0	4	8	11	14	16	19	23	24	29	32	35	37	39	42
2	3	6	10	13	17	20	23	26	27	30	33	36	39	44
2	4	8	10	14	15	18	22	26	27	31	35	37	40	42
1	5	8	11	12	15	18	23	25	27	31	34	37	39	43
2	4	8	11	12	17	18	23	24	27	31	33	38	40	44
0	4	7	11	14	16	18	21	25	28	32	35	38	39	43
1	4	8	10	12	16	20	23	25	27	30	33	38	39	42
0	5	8	9	14	17	18	21	26	29	31	33	38	39	44
1	4	6	11	14	17	19	22	26	28	31	35	38	39	43
2	5	8	11	13	16	20	22	25	27	32	35	38	40	42
0	3	7	10	12	17	19	22	24	27	32	35	36	40	44

Variable strength covering array from Table 5.5 (page 97)

$VCA(2, 4^3 5^3 6^2, (CA(3, 4^3))), N = 64$							
3	6	11	13	19	22	27	38
1	6	9	14	21	24	29	36
0	5	8	16	18	23	29	33
2	5	9	16	19	24	27	38
3	4	9	16	20	23	29	33
1	6	10	16	17	23	30	35
3	6	8	14	17	23	27	33
2	5	8	14	21	24	30	34
2	6	8	12	19	23	32	38
2	7	9	14	21	22	29	33
1	4	11	12	21	22	30	33
3	4	8	12	21	24	27	36
3	7	9	15	19	24	32	33
3	4	10	15	19	22	31	35
3	4	11	15	21	26	32	35
1	4	8	13	20	23	28	33
1	6	11	14	18	25	28	34
1	5	8	14	21	22	32	36
1	5	11	15	18	22	31	35
2	7	8	16	20	26	30	38
0	5	11	13	17	26	32	37
0	6	9	15	21	22	27	34
2	6	11	15	18	22	28	35
0	7	10	16	18	23	30	36
1	6	8	15	17	25	27	38
0	4	9	16	21	23	31	38
0	4	8	16	17	26	32	36
3	6	10	16	20	25	30	34
0	6	8	16	19	25	31	34
3	7	10	12	17	25	29	34
1	4	10	14	20	26	27	38
1	5	9	13	20	26	29	34
1	5	10	16	20	24	32	38
2	7	10	13	21	23	31	34
2	7	11	15	17	23	28	36
2	4	9	13	17	25	31	37
0	5	9	15	19	26	30	34
2	4	11	16	17	22	31	34
0	6	10	12	19	24	28	34
0	4	10	13	18	26	29	33
0	7	8	14	17	24	27	37
0	4	11	13	20	25	30	33
2	4	8	12	20	22	27	34
1	7	11	13	20	26	31	36
3	5	9	12	21	25	27	36
1	4	9	14	17	25	31	35
2	5	11	12	20	26	29	38
1	7	10	12	18	25	32	34
2	5	10	14	18	22	31	33
3	5	10	12	21	24	30	37
2	6	10	13	21	22	28	38
3	7	8	12	17	23	30	37
2	4	10	14	20	24	28	34
3	6	9	14	19	25	29	36
2	6	9	12	18	26	31	33
0	7	11	12	20	26	28	37
3	7	11	16	18	24	27	38
0	7	9	15	20	24	27	37
0	6	11	15	18	26	29	37
1	7	9	16	19	22	28	37
3	5	8	14	20	22	28	37
3	5	11	13	21	24	31	35
0	5	10	12	20	25	27	35
1	7	8	16	19	23	29	35

Variable strength covering array from Table 5.5 (page 97)

$VCA(2, 4^3 5^3 6^2, (CA(3, 4^3 5^2))), N = 100$							
3	4	8	14	20	24	31	34
2	4	8	15	21	25	32	33
3	4	11	15	17	26	28	37
1	6	8	14	17	25	31	36
0	6	11	16	20	22	30	36
0	7	9	16	17	26	27	37
3	5	9	16	20	22	32	35
3	5	8	15	18	22	31	35
3	4	9	13	17	23	32	33
0	6	10	13	17	25	32	35
0	4	9	13	19	22	31	36
0	6	8	14	21	26	27	36
3	5	10	14	21	23	30	38
3	7	9	14	19	26	31	38
0	7	8	15	19	25	28	37
2	7	8	16	20	24	27	37
3	6	10	16	17	25	27	38
1	7	11	13	19	23	29	35
1	7	11	16	19	23	28	34
1	4	10	13	18	24	29	37
0	6	9	12	19	22	32	38
3	7	8	12	17	23	29	38
2	5	8	14	19	25	30	37
0	7	10	13	21	25	28	38
0	5	9	14	18	24	28	33
0	7	9	16	21	23	32	37
1	5	10	12	18	26	27	37
1	7	8	13	20	22	27	37
3	5	11	13	21	26	28	38
2	7	8	13	17	26	27	34
1	4	9	14	21	26	27	38
0	4	9	14	17	22	30	37
0	6	11	12	17	22	32	36
2	5	9	13	20	22	30	35
0	4	11	14	19	22	28	38
1	6	9	13	21	23	32	37
2	7	11	14	21	25	31	38
2	7	11	14	17	22	32	34
1	4	8	12	19	26	31	38
1	5	9	15	19	24	30	36
2	5	11	16	17	25	29	38
0	5	8	15	17	25	32	38
3	6	9	15	21	25	29	34
2	4	11	12	20	22	28	36
3	6	8	13	19	25	28	38
0	6	10	15	18	24	30	35
2	4	9	16	19	22	28	36
1	4	10	16	20	24	29	34
2	6	9	15	17	24	27	34
3	7	9	13	18	23	28	35
2	4	10	12	17	26	28	33
1	4	11	15	18	25	32	36
1	6	9	16	18	26	30	34
2	6	10	16	21	23	29	35
0	7	11	12	18	23	27	37
1	7	9	12	20	26	32	35
1	6	8	15	20	24	28	36
3	7	8	12	21	25	32	36
1	4	8	16	17	25	27	33
1	5	8	16	21	25	32	38
3	4	11	16	21	24	28	34
3	5	10	14	17	24	28	34
3	6	10	12	20	23	32	34
0	5	11	15	21	26	27	33
0	5	8	13	18	23	31	38
1	5	11	13	17	22	31	33
1	7	8	14	18	23	29	36
3	7	10	16	18	22	30	35
3	6	8	16	19	26	27	35
3	6	10	13	20	22	28	38
2	6	11	13	18	22	28	37
2	4	10	14	18	26	28	35
3	6	11	14	18	23	30	38
0	4	9	15	20	25	30	37
1	6	10	14	19	26	29	33
3	4	10	15	19	22	29	38
1	5	11	14	20	25	29	37
0	4	11	13	20	25	29	34
1	7	10	15	21	22	28	36
0	5	8	12	20	26	31	38
2	5	11	16	18	26	31	38
3	5	11	12	19	23	31	38
2	7	10	12	19	23	28	37
2	7	9	15	18	24	32	33
2	4	8	13	21	25	32	35
0	4	10	12	21	22	31	33
0	4	8	16	18	23	31	38
3	7	11	15	20	23	28	37
2	6	11	15	19	22	30	38
2	5	10	13	19	26	30	35
0	7	10	14	20	26	32	38
2	5	10	15	20	23	30	33
2	6	8	12	18	22	27	36
1	7	10	15	17	26	27	33
1	6	11	12	21	23	28	38
1	5	9	12	17	24	29	38
3	4	9	12	18	23	28	37
0	5	10	16	19	24	31	37
2	6	9	14	20	26	27	38
2	5	9	12	21	22	30	33

Variable strength covering array from Table 5.5 (page 97)

$VCA(2, 4^3 5^3 6^2, (CA(3, 5^3))), N = 125$							
1	7	12	18	21	26	29	34
2	5	10	15	20	25	30	38
1	6	14	17	19	26	27	35
0	5	12	17	22	24	31	33
0	7	10	18	19	24	28	35
1	9	14	15	21	23	31	38
1	7	10	17	22	24	31	34
3	6	13	16	21	24	29	33
3	5	13	18	19	25	31	36
0	5	13	17	20	24	31	37
3	8	11	17	22	25	29	35
1	6	13	16	20	24	28	33
4	8	13	17	20	26	28	36
1	6	10	17	21	23	29	35
3	6	10	18	20	24	28	35
0	5	11	17	21	24	32	38
4	9	11	17	22	26	27	33
3	9	12	15	20	26	31	37
0	6	12	16	19	26	29	38
3	5	11	15	19	26	32	37
3	9	10	16	22	25	30	35
2	7	12	17	19	23	32	38
3	8	12	16	19	24	30	33
4	8	10	15	21	25	27	38
2	8	11	18	19	24	27	38
0	8	12	16	22	26	31	36
4	8	12	15	20	24	29	37
4	6	11	18	20	25	28	33
4	8	14	15	21	23	27	36
3	8	14	17	19	25	28	36
4	5	10	17	20	24	29	34
4	5	12	17	19	26	32	36
1	6	12	17	21	25	31	35
4	7	14	17	21	24	27	38
0	9	13	16	21	23	29	37
2	6	13	15	21	24	29	36
1	5	11	16	20	26	32	34
3	7	10	15	22	24	29	33
4	9	14	17	22	25	30	37
2	8	14	15	22	26	32	34
1	5	10	16	21	25	30	36
0	7	13	15	21	23	30	38
0	8	14	15	21	23	28	34
2	7	13	17	19	25	27	34
0	9	12	15	22	26	28	35
3	5	14	15	22	23	27	38
0	8	11	18	19	25	29	35
1	8	11	16	19	23	29	36
2	7	11	16	19	25	27	37
2	9	12	17	20	26	31	34
3	7	13	15	22	24	32	33
4	6	13	15	22	26	29	37
4	5	13	16	21	26	30	33
1	6	11	16	20	26	32	34
0	8	13	15	20	26	29	34
4	7	11	17	22	24	31	34
0	6	11	18	21	25	29	38
2	6	10	15	22	25	32	36
0	5	14	16	20	23	28	33
0	9	11	17	22	24	28	36
3	6	12	18	22	24	27	35
2	9	13	17	22	24	27	35
2	9	11	16	19	23	29	35
2	5	12	17	21	23	30	34
4	6	12	16	19	25	31	37
4	5	14	17	22	24	30	36
3	7	11	16	22	25	32	35
1	8	14	15	19	26	28	36
3	8	10	18	20	26	31	36
4	6	10	17	20	24	29	33
2	5	13	17	20	26	30	35
1	7	11	15	21	23	31	37
3	6	14	18	21	24	29	35
2	9	14	17	21	24	28	33
1	5	13	17	22	25	30	36
0	9	14	15	21	25	32	38
3	8	13	18	19	26	29	37
3	5	10	17	21	25	30	36
4	8	11	18	21	25	27	38
2	6	11	18	20	24	31	34
0	5	10	18	19	26	27	34
0	7	11	18	22	24	27	35
3	7	14	18	22	23	27	33
1	9	13	15	22	23	32	34
3	9	11	17	19	26	30	35
3	7	12	15	22	26	27	34
1	8	13	18	20	23	30	35
4	5	11	15	19	25	27	35
4	9	12	17	22	24	27	35
2	6	12	18	20	26	28	33
2	8	10	15	22	26	31	34
0	7	12	17	21	23	28	33
0	8	10	16	21	25	31	38
4	9	13	18	22	23	31	38
2	8	13	15	20	23	29	34
0	7	14	17	19	23	27	36
3	9	14	16	19	24	28	36
1	8	12	16	19	25	29	37
1	9	12	18	22	23	27	36
3	5	12	18	20	23	31	35
4	9	10	18	22	23	29	37
2	7	10	17	19	24	28	34
3	9	13	18	19	25	30	35
0	6	13	18	21	26	28	38
2	5	14	16	20	23	27	34
2	5	11	18	20	25	31	36
1	9	11	17	22	23	32	35
2	7	14	15	22	23	31	35
1	8	10	18	20	24	31	38
3	6	11	18	19	26	29	33
2	8	12	15	20	23	28	35
0	6	10	18	19	26	32	33
1	9	10	17	20	23	27	33
0	9	10	18	19	24	30	33
4	6	14	15	19	24	30	34
0	6	14	18	21	23	31	34
2	6	14	17	19	26	30	37
1	7	14	17	21	25	28	35
1	7	13	18	22	23	32	35
4	7	10	17	20	24	31	35
2	9	10	16	19	25	27	38
4	7	13	17	20	26	27	38
1	5	12	17	22	25	31	33
4	7	12	15	20	26	32	36
1	5	14	15	20	26	28	37

Variable strength covering array from Table 5.5 (page 97)

$VCA(2, 4^3 5^3 6^2, (CA(3, 4^3), CA(3, 5^3)))$ $N = 125$							
0	6	11	16	21	24	29	37
1	6	10	15	19	23	27	34
1	6	11	12	21	22	30	34
0	5	10	16	18	23	32	33
2	4	11	15	19	25	27	37
2	6	9	12	17	25	29	38
3	7	8	15	19	26	28	37
0	4	11	16	17	23	30	35
0	6	10	15	20	24	31	33
3	6	10	14	18	22	32	37
2	5	8	15	18	26	30	33
1	7	9	16	19	22	32	38
3	7	9	15	17	22	29	34
0	6	10	14	17	26	30	38
1	7	11	12	21	24	31	37
1	5	10	15	19	22	27	35
2	7	9	13	21	25	28	37
0	4	8	14	21	23	30	37
3	7	8	13	17	22	28	36
1	7	10	15	21	23	30	33
0	5	11	15	18	23	28	37
2	7	10	13	20	25	32	35
0	5	11	16	17	26	30	33
3	4	9	13	19	24	29	33
3	5	8	12	17	22	27	37
1	7	8	14	21	24	32	36
3	7	11	15	20	23	27	36
1	4	8	12	20	23	32	37
2	6	8	15	17	23	30	37
2	6	11	14	17	22	31	33
0	5	9	15	17	25	32	36
2	4	10	15	17	24	28	38
1	4	11	16	21	22	32	34
2	7	8	13	19	25	31	34
2	5	8	13	21	23	29	34
1	5	9	13	19	26	29	38
3	4	10	15	18	22	28	33
1	6	9	13	19	23	30	37
3	5	10	13	18	25	28	34
0	4	11	16	17	25	31	36
1	7	10	14	21	25	31	36
3	7	10	16	18	22	29	35
1	7	11	16	18	25	30	38
1	5	8	15	21	26	32	38
1	7	11	16	20	26	29	36
3	5	8	12	17	26	31	38
2	6	10	15	18	25	32	38
2	6	8	12	20	24	32	38
1	6	11	13	20	23	31	38
0	4	8	15	20	26	27	34
3	7	9	14	20	25	32	37
2	5	9	13	21	22	32	34
3	4	11	12	20	22	28	36
3	7	10	13	17	26	31	33
0	6	9	14	20	22	28	38
0	6	11	16	19	25	31	34
3	4	9	12	20	25	30	37
0	6	8	13	17	23	32	36
2	5	11	14	19	23	28	34
2	5	10	13	19	22	30	33
0	6	10	12	17	24	28	37
2	5	9	14	21	22	32	38
3	4	10	15	21	25	31	34
0	6	11	15	19	24	30	34
0	4	9	14	20	26	32	37
3	7	11	14	18	26	29	33
2	6	10	12	18	25	32	36
1	7	8	13	18	23	32	38
2	5	10	16	19	23	28	33
3	6	11	12	18	23	30	37
0	7	9	13	21	24	31	35
0	7	11	13	18	22	32	35
0	5	8	16	20	22	29	33
3	5	11	12	21	23	27	33
0	5	9	12	19	23	28	36
0	4	11	14	18	24	27	38
0	5	11	14	21	26	27	38
1	4	11	12	17	23	27	34
1	5	8	14	19	22	27	33
1	5	11	14	19	26	30	36
1	4	9	12	18	22	29	36
2	4	8	15	21	22	29	37
1	7	11	13	21	26	30	36
0	5	11	16	20	24	28	38
3	6	11	16	18	24	27	33
3	5	9	12	20	26	27	33
1	5	10	13	18	24	32	38
1	5	8	14	19	25	28	35
2	4	10	13	20	24	30	38
2	5	11	13	20	22	31	33
3	4	8	16	20	25	31	35
3	4	11	12	21	26	29	36
0	6	9	14	19	24	28	36
1	6	11	14	17	25	29	34
0	7	8	16	20	23	32	37
1	5	11	15	21	24	31	38
2	7	11	14	17	24	30	37
0	5	11	16	21	26	28	35
3	6	9	13	17	25	32	33
1	5	8	12	21	25	28	34
3	5	9	15	20	22	28	35
2	7	11	16	19	26	31	33
1	5	10	16	21	23	29	36
1	4	10	13	20	26	28	37
3	6	11	12	18	26	30	36
0	6	10	12	19	24	32	36
2	7	8	16	17	24	28	38
1	5	9	16	18	26	29	33
2	7	11	13	18	26	29	37
1	7	10	14	17	23	28	35
0	5	11	16	17	22	31	33
2	4	9	15	17	26	29	35
1	6	9	14	18	23	31	35
3	6	8	12	19	22	29	36
0	6	11	12	19	25	32	35
3	7	10	14	20	24	29	36
0	4	10	12	19	26	30	35
1	6	9	16	21	25	31	35
3	7	8	16	19	24	32	38
1	6	8	15	20	25	29	36
1	7	11	14	18	25	31	36
2	6	10	12	18	24	28	37
3	7	8	13	17	24	27	36
0	7	10	15	18	24	29	36
1	5	10	14	20	23	32	35

Ordered design from Table 6.7 (page 122)

$OD(3,6,6), N=120$					
4	11	12	20	27	31
5	7	12	22	26	33
4	11	14	21	25	30
3	10	12	23	26	31
4	9	13	20	29	30
5	9	16	18	26	31
4	7	15	23	26	30
0	8	16	23	27	31
2	6	16	19	27	35
2	10	17	18	27	31
0	7	17	22	27	32
1	8	12	22	27	35
3	11	13	20	24	34
5	7	16	20	27	30
1	8	16	21	29	30
2	10	15	19	29	30
4	9	17	18	25	32
3	7	16	23	24	32
3	6	16	20	29	31
1	10	12	20	29	33
1	9	17	22	26	30
3	7	14	22	29	30
4	6	13	23	27	32
4	8	15	18	29	31
2	11	12	19	28	33
4	6	15	20	25	35
3	8	17	22	24	31
2	9	13	18	28	35
3	6	14	23	25	34
1	9	14	18	29	34
0	10	13	20	27	35
5	9	13	22	24	32
1	9	16	20	24	35
0	11	16	20	25	33
0	7	16	21	26	35
4	7	12	21	29	32
5	10	13	21	26	30
0	10	14	21	29	31
3	6	17	19	28	32
1	8	17	18	28	33
2	11	13	22	27	30
5	8	13	18	27	34
3	6	13	22	26	35
5	8	12	21	28	31
2	9	12	22	29	31
3	10	13	18	29	32
0	7	15	20	29	34
4	11	15	19	24	32
1	10	17	21	24	32
0	11	14	19	27	34
5	7	14	21	24	34
5	8	16	19	24	33
2	7	16	18	29	33
2	11	15	18	25	34
1	11	12	21	26	34
3	7	17	18	26	34
0	11	15	22	26	31
4	9	12	19	26	35
0	7	14	23	28	33
3	8	12	19	29	34
2	10	13	23	24	33
4	11	13	18	26	33
1	6	15	22	29	32
1	10	14	23	27	30
5	6	16	21	25	32
1	10	15	18	26	35
3	11	12	22	25	32
0	11	13	21	28	32
0	10	17	19	26	33
3	10	14	19	24	35
4	7	17	20	24	33
3	8	13	23	28	30
3	11	16	19	26	30
0	8	15	19	28	35
1	11	16	18	27	32
5	7	15	18	28	32
4	7	14	18	27	35
0	9	14	22	25	35
4	9	14	23	24	31
2	7	15	22	24	35
5	8	15	22	25	30
2	7	12	23	27	34
5	10	12	19	27	32
5	6	13	20	28	33
3	11	14	18	28	31
2	6	17	22	25	33
0	9	13	23	26	34
4	6	17	21	26	31
2	6	15	23	28	31
5	10	15	20	24	31
0	9	17	20	28	31
5	6	15	19	26	34
5	9	12	20	25	34
4	8	13	21	24	35
0	8	17	21	25	34
4	8	12	23	25	33
4	8	17	19	27	30
5	9	14	19	28	30
0	8	13	22	29	33
2	9	16	23	25	30
1	11	15	20	28	30
1	11	14	22	24	33
5	10	14	18	25	33
3	8	16	18	25	35
2	7	17	21	28	30
1	9	12	23	28	32
2	9	17	19	24	34
2	11	16	21	24	31
4	6	14	19	29	33
1	6	17	20	27	34
2	10	12	21	25	35
1	6	14	21	28	35
3	10	17	20	25	30
2	6	13	21	29	34
5	6	14	22	27	31
0	9	16	19	29	32
1	6	16	23	26	33
1	8	15	23	24	34
3	7	12	20	28	35
0	10	15	23	25	32

(2, 1)-covering array from Table 6.7 (page 122)

$TOCA(N; 3, 6, 6; 0), N = 140$					
5	11	12	22	24	33
4	7	16	19	25	34
3	9	15	18	27	30
1	8	14	20	25	32
0	6	14	20	26	30
2	10	16	20	28	32
4	10	15	19	27	32
3	10	14	21	28	31
2	7	14	20	25	31
0	6	16	18	28	34
3	7	13	21	25	33
2	9	15	20	26	32
4	10	13	22	25	31
2	11	17	23	26	35
4	8	16	22	26	32
3	10	15	22	28	33
5	9	16	18	24	35
5	6	12	23	29	35
0	8	14	18	24	30
2	9	14	21	27	32
2	8	14	22	28	34
4	8	13	21	27	34
2	11	14	19	25	30
2	9	16	22	25	33
4	10	16	21	27	33
3	9	15	19	25	31
3	6	12	21	27	33
3	11	12	18	28	35
5	10	16	22	29	34
3	6	15	18	24	33
5	8	17	20	26	35
0	6	14	19	26	33
5	7	13	19	24	31
4	9	15	22	27	34
5	7	13	23	29	35
5	9	17	23	27	33
1	9	15	21	26	33
3	9	13	22	28	32
0	6	15	21	27	30
0	10	16	22	24	34
4	6	16	22	24	30
1	7	17	23	29	31
3	6	17	23	28	30
0	7	13	18	25	30
0	6	12	20	24	32
1	7	14	18	26	35
5	10	17	22	28	35
4	6	12	22	28	34
2	8	12	20	24	30
4	10	14	20	26	34
0	7	12	18	24	31
1	8	14	19	26	31
5	11	13	23	25	31
0	9	16	23	29	30
5	8	14	20	24	32
5	6	17	18	29	30
2	6	13	20	25	35
3	9	14	20	26	33
5	11	14	20	29	35
1	6	12	18	25	30
5	9	15	21	29	33
4	10	12	18	24	34
5	11	14	23	26	32
4	11	16	22	29	35
2	7	13	19	26	31
1	9	13	19	27	33
3	8	14	20	27	32
2	8	14	21	26	33
2	10	14	22	26	32
5	10	15	23	24	30
2	8	15	20	27	33
0	11	17	23	24	30
5	11	16	23	28	34
2	6	14	18	24	32
4	10	16	23	28	35
3	11	17	21	27	33
5	11	17	19	29	31
4	11	12	23	27	30
5	7	17	19	25	35
1	10	13	19	28	34
1	11	17	23	25	35
2	6	12	18	26	30
0	6	12	23	24	35
0	7	14	19	29	32
1	11	13	19	29	35
0	6	17	22	29	33
2	8	13	19	25	32
3	10	16	22	27	34
1	6	17	19	26	32
4	6	17	18	27	35
1	7	15	21	27	31
2	8	14	23	29	35
0	6	13	19	24	30
0	10	12	22	28	30
0	11	12	21	29	34
2	10	15	21	25	34
5	8	17	23	29	32
0	11	17	18	27	35
3	9	15	23	27	35
4	7	13	19	28	31
4	11	17	22	28	34
2	7	12	23	25	32
4	8	14	20	28	32
2	11	17	20	29	32
1	7	13	20	26	32
1	7	13	22	25	34
1	7	16	22	28	31
4	10	16	18	28	30
5	6	17	21	24	34
1	10	16	22	26	34
0	9	12	18	27	33
4	9	15	21	28	33
0	11	17	18	24	35
2	8	17	18	25	31
3	11	15	23	29	33
3	7	15	20	28	34
1	8	13	23	26	30
3	9	16	21	28	34
5	11	15	21	27	35
2	8	16	20	26	34
3	8	16	19	28	33
1	7	12	19	24	30
3	7	13	19	27	31
1	11	12	20	26	31
0	8	13	20	29	31
1	10	16	19	25	31
3	9	12	21	24	30
0	8	12	18	26	32
0	9	15	21	24	33
0	10	15	18	29	35
0	6	12	19	25	31
3	8	15	21	26	32
4	10	17	23	29	34
3	9	17	21	29	35
1	7	15	19	25	33
5	11	12	18	29	30
1	9	13	21	25	31
4	7	14	22	27	33
4	9	16	20	27	31
1	6	13	18	24	31

$(2,1)$-covering arrays from Table 6.8 (page 122)

$TOCA(N;3,6,2;2), N = 12$					
0	2	4	6	8	10
1	3	5	7	9	11
0	3	4	6	9	10
0	3	5	7	8	10
1	3	4	6	8	11
0	2	5	6	9	11
0	3	4	7	8	11
0	2	5	7	9	10
1	2	4	7	9	10
1	2	5	7	8	11
1	3	5	6	8	10
1	2	4	6	9	11

$TOCA(N;3,6,2;0), N = 12$					
1	2	4	7	8	11
0	3	4	6	9	11
1	3	5	6	8	10
1	2	5	6	9	11
1	2	5	7	8	10
0	3	4	7	8	10
0	2	5	7	9	10
0	3	5	6	8	11
1	3	5	7	9	11
0	2	4	6	8	10
1	3	4	6	9	10
0	2	4	7	9	11

$TOCA(N;3,6,3;3), N = 33$					
0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17
0	3	8	11	12	17
2	4	8	10	13	16
1	4	8	10	14	17
0	4	7	10	12	15
1	3	7	9	13	15
2	3	6	11	14	17
2	3	8	9	14	15
2	5	6	9	14	15
2	3	8	11	12	15
1	3	6	10	13	15
0	3	7	10	12	16
2	4	7	11	13	17
2	5	7	10	13	17
1	4	6	9	13	16
0	5	6	11	12	17
0	5	6	11	14	15
0	4	7	9	13	15
2	5	6	9	12	17
1	5	7	11	13	16
1	4	6	10	12	15
1	5	8	11	13	17
0	3	8	9	14	17
1	5	8	10	14	16
0	4	6	9	12	16
0	5	8	9	12	15
2	5	7	10	14	16
1	4	7	11	14	17
2	4	8	11	14	16
0	3	6	10	13	16
1	3	7	9	12	16

$TOCA(N;3,6,3;0), N = 30$					
0	5	6	11	14	17
1	3	7	10	12	15
1	3	6	9	12	16
1	5	8	10	13	17
2	3	6	9	14	17
0	4	7	9	12	15
1	4	8	11	14	17
2	5	6	9	14	15
0	4	6	10	12	16
1	4	7	9	12	16
1	4	7	10	14	17
0	3	6	10	13	15
2	5	7	10	13	17
1	5	8	10	14	16
0	5	8	11	12	15
2	5	7	11	14	16
2	5	8	9	12	17
0	4	6	9	13	16
2	4	8	10	14	16
0	3	8	11	12	17
1	5	7	11	13	16
2	4	8	11	13	16
1	4	6	10	13	15
2	3	8	11	14	15
2	4	7	11	13	17
1	3	7	9	13	15
2	3	6	11	12	15
0	3	7	10	13	16
0	3	8	9	14	15
0	5	6	9	12	17

(2, 1)-covering arrays from Table 6.9 (page 122)

$TOCA(N;3,8,2;2), N=12$							
0	2	4	6	8	10	12	14
1	3	5	7	9	11	13	15
0	3	5	7	8	10	12	15
0	2	4	7	9	11	12	15
1	2	4	6	9	11	13	14
1	2	5	6	8	11	12	15
1	3	4	6	8	10	13	15
0	2	5	6	9	10	13	15
0	3	4	7	8	11	13	14
0	3	5	6	9	11	12	14
1	2	5	7	8	10	13	14
1	3	4	7	9	10	12	14

$TOCA(N;3,8,2;0), N=12$							
1	2	4	6	9	10	13	14
0	3	4	6	9	11	13	15
0	3	4	7	8	10	13	14
0	2	5	6	8	11	13	14
1	3	5	7	8	11	13	15
0	2	5	7	9	10	13	15
1	3	4	7	9	10	12	15
1	3	5	6	8	10	12	14
1	2	5	6	9	11	12	15
0	2	4	6	8	10	12	15
1	2	4	7	8	11	12	14
0	3	5	7	9	11	12	14

$TOCA(N;3,8,3;3), N=33$							
0	3	6	9	12	15	18	21
1	4	7	10	13	16	19	22
2	5	8	11	14	17	20	23
0	4	6	10	12	16	18	22
1	4	8	10	14	17	19	23
0	3	6	10	13	16	19	21
0	3	8	11	14	17	18	21
2	5	8	10	13	16	19	23
2	4	7	11	14	17	19	22
1	3	6	9	13	15	19	22
1	4	7	9	12	15	18	22
1	5	7	10	14	16	20	23
1	4	6	9	13	16	18	21
2	3	8	11	12	15	20	21
0	3	7	9	12	16	19	22
2	5	6	11	14	15	18	21
0	3	7	10	13	15	18	22
0	5	6	11	12	17	18	23
0	5	8	9	12	15	20	23
1	4	8	11	13	17	20	22
2	4	7	11	13	16	20	23
0	3	6	11	14	15	20	23
1	5	7	11	13	17	19	23
1	5	8	11	14	16	19	22
0	5	6	9	14	17	20	21
2	3	8	9	14	15	18	23
2	4	8	10	14	16	20	22
2	5	7	10	13	17	20	22
1	3	7	10	12	16	18	21
2	5	8	9	12	17	18	21
0	4	7	9	13	15	19	21
2	3	6	9	12	17	20	23
1	4	6	10	12	15	19	21

$TOCA(N;3,8,3;0), N=30$							
0	3	6	9	14	17	20	23
0	5	8	9	12	17	18	23
2	5	7	10	13	17	20	22
1	4	8	11	14	17	19	22
1	4	6	9	12	16	19	21
0	4	7	9	13	15	19	21
0	3	8	11	14	15	18	23
2	5	7	10	14	16	19	23
1	5	7	11	13	17	19	23
1	3	7	9	12	15	19	22
2	4	8	10	13	17	19	23
1	5	8	10	14	16	20	22
2	3	8	9	14	15	20	21
0	3	8	11	12	17	20	21
2	3	6	11	12	17	18	23
1	3	7	10	12	16	18	21
2	5	6	9	14	17	18	21
2	4	7	11	14	16	20	22
1	4	6	9	13	15	18	22
2	5	6	9	12	15	20	23
0	3	6	10	12	16	19	22
1	3	6	10	13	15	19	21
0	4	6	10	13	16	18	21
0	4	7	10	12	15	18	22
0	5	6	11	14	15	20	21
0	3	7	9	13	16	18	22
2	5	8	11	13	16	19	22
1	4	8	11	13	16	20	23
2	5	8	11	12	15	18	21
1	4	7	10	14	17	20	23

(2, 1)-covering arrays from Table 6.10 (page 122)

$TOCA(N; 3, 9, 2; 2), N = 13$								
0	2	4	6	8	10	12	14	16
1	3	5	7	9	11	13	15	17
0	3	4	6	8	10	13	15	16
0	2	4	7	9	11	12	15	16
0	2	4	6	9	11	13	14	17
0	3	5	6	9	10	12	14	17
0	3	5	7	8	11	12	14	16
1	2	4	7	8	10	12	14	17
1	3	4	7	9	10	13	14	16
1	3	4	6	8	11	12	15	17
0	2	5	7	8	10	13	15	17
1	2	5	6	9	10	12	15	16
1	2	5	6	8	11	13	14	16

$TOCA(N; 3, 9, 2; 0), N = 12$								
1	2	5	7	9	10	12	14	16
1	3	4	7	8	10	13	14	16
1	2	5	6	8	11	12	14	17
1	2	4	7	9	11	13	15	17
0	2	5	7	8	10	13	15	17
0	2	4	6	8	11	13	14	16
0	3	5	7	8	11	12	15	16
1	3	4	6	8	10	12	15	17
0	3	5	6	9	10	13	14	17
1	3	5	6	9	11	13	15	16
0	3	4	7	9	11	12	14	17
0	2	4	6	9	10	12	15	16

$TOCA(N; 3, 9, 3; 3), N = 36$								
0	3	6	9	12	15	18	21	24
1	4	7	10	13	16	19	22	25
2	5	8	11	14	17	20	23	26
0	3	7	10	12	16	19	21	25
2	5	6	11	14	15	18	21	24
1	4	7	10	12	16	18	21	24
0	4	6	9	13	16	18	21	25
0	4	7	9	12	15	19	22	25
1	3	6	10	13	15	19	21	25
1	4	6	10	12	15	18	22	25
2	3	8	9	14	15	20	23	24
1	5	8	10	14	16	20	22	26
0	5	8	9	12	15	20	21	26
1	5	8	11	13	16	19	23	25
1	3	7	9	13	16	18	22	25
0	3	6	9	14	17	18	23	26
1	4	7	10	14	16	20	23	25
1	4	7	11	14	17	19	22	26
0	3	7	10	13	15	18	22	24
2	5	6	9	12	17	20	23	24
2	5	7	10	14	17	19	23	25
0	3	8	11	14	17	20	21	24
0	5	6	11	14	15	20	23	26
0	4	6	10	13	16	19	22	24
2	5	8	9	14	17	18	21	26
2	4	7	11	13	16	20	23	26
1	3	6	9	12	16	19	22	24
2	4	8	11	14	16	19	22	25
2	3	6	11	12	17	20	21	26
0	5	8	11	12	17	18	23	24
2	3	8	11	12	15	18	23	26
2	4	8	10	13	17	20	22	25
1	4	8	10	13	17	19	23	26
1	5	7	11	13	17	20	22	25
2	5	7	10	13	16	19	22	26
1	4	7	9	13	15	19	21	24

$TOCA(N; 3, 9, 3; 0), N = 33$								
1	4	6	9	12	16	18	22	25
2	4	7	10	14	17	19	23	26
2	5	7	11	13	16	19	23	25
2	5	8	11	14	15	18	21	24
1	5	8	10	13	17	20	23	25
1	3	7	9	12	15	19	21	25
0	4	6	10	13	15	19	21	25
0	3	8	9	14	17	18	21	26
1	3	6	10	13	16	18	21	24
1	4	7	9	13	16	19	21	24
2	3	8	9	12	17	18	23	24
1	3	7	10	13	15	18	22	25
1	4	7	11	14	17	20	23	25
1	4	6	10	12	15	19	22	24
0	5	6	9	14	15	18	23	26
2	4	8	10	14	16	20	22	25
0	5	8	9	12	15	20	21	24
0	3	8	11	12	15	20	23	26
0	3	7	10	12	16	19	22	24
2	5	7	10	13	17	20	22	26
1	5	7	11	14	16	20	22	26
2	3	6	9	14	15	20	23	24
0	3	6	9	13	16	19	22	25
2	4	8	11	13	16	20	23	26
0	4	7	9	13	15	18	22	24
1	4	8	11	13	17	19	22	26
1	5	8	10	14	16	19	23	26
0	4	7	10	12	16	18	21	25
2	5	6	9	12	17	20	21	26
2	5	8	11	14	17	19	22	25
0	3	6	11	14	17	20	21	24
0	5	6	11	12	17	18	23	24
2	3	6	11	12	15	18	21	26

$(2,1)$-covering arrays from Table 6.11 (page 123)

$TOCA(N;3,10,2;2), N = 13$

0	2	4	6	8	10	12	14	16	18
1	3	5	7	9	11	13	15	17	19
0	3	5	7	8	11	12	15	17	18
0	3	4	7	9	11	13	14	16	19
1	2	5	6	9	11	13	15	16	18
1	3	4	6	8	11	13	14	17	18
0	2	5	6	9	11	12	14	17	19
1	2	5	7	8	10	13	14	17	19
1	2	4	7	8	11	12	15	16	19
1	3	4	6	9	10	12	15	17	19
0	2	4	7	9	10	13	15	17	18
1	3	5	7	9	10	12	14	16	18
0	3	5	6	8	10	13	15	16	19

$TOCA(N;3,10,2;0), N = 12$

1	3	5	7	8	10	13	15	16	18
0	2	5	7	8	11	12	14	16	18
1	3	4	7	8	11	12	15	17	19
1	2	5	6	9	11	13	15	17	18
0	3	4	6	8	10	13	14	17	18
0	3	4	6	9	11	12	15	16	18
1	3	5	6	9	10	12	14	16	19
0	2	5	6	8	10	12	15	17	19
1	2	4	6	8	11	13	14	16	19
0	3	5	7	9	11	13	14	17	19
0	2	4	7	9	10	13	15	16	19
1	2	4	7	9	10	12	14	17	18

$TOCA(N;3,10,3;3), N = 36$

0	3	6	9	12	15	18	21	24	27
1	4	7	10	13	16	19	22	25	28
2	5	8	11	14	17	20	23	26	29
0	3	8	11	12	17	18	23	26	29
1	3	6	9	13	15	19	22	25	28
2	4	8	10	13	16	20	22	26	29
1	4	8	11	14	17	19	22	25	29
0	4	6	10	12	15	19	22	24	28
2	3	6	9	14	17	18	23	26	27
1	5	7	11	14	16	20	22	26	28
0	3	7	10	12	16	19	21	25	28
0	5	8	9	14	15	18	23	24	29
2	5	6	9	12	15	20	23	26	29
1	4	6	10	13	16	19	21	24	27
2	5	8	11	13	17	20	22	25	28
1	4	7	10	12	15	18	22	25	27
0	3	6	11	14	15	20	21	26	29
1	4	7	11	13	17	20	23	26	29
0	4	7	9	13	15	19	21	25	27
2	5	7	11	13	16	19	23	25	29
2	5	7	10	14	17	19	22	26	29
2	3	8	11	12	15	20	23	24	27
0	3	6	10	13	16	18	22	25	27
1	4	6	9	12	16	18	21	25	28
2	4	7	10	14	17	20	23	25	28
0	5	6	11	14	17	20	23	24	27
2	5	6	11	12	17	18	21	24	29
0	4	7	9	13	16	18	22	24	28
0	5	8	9	12	17	20	21	26	27
2	5	8	11	14	15	18	21	26	27
2	4	8	11	14	16	19	23	26	28
1	3	7	10	13	15	18	21	24	28
1	3	7	9	12	16	19	22	24	27
2	3	8	9	14	17	20	21	24	29
1	5	8	10	13	17	19	23	26	28
1	5	8	10	14	16	20	23	25	29

$TOCA(N;3,10,3;0), N = 33$

2	3	8	9	14	15	20	21	26	27
1	4	7	11	13	17	19	23	25	29
1	5	8	11	14	16	19	22	25	28
2	5	6	11	14	17	18	21	24	27
2	3	8	11	14	15	18	23	24	29
1	3	7	9	13	16	19	21	24	27
2	3	6	11	12	17	20	23	26	27
1	4	7	11	14	16	20	23	26	28
1	4	6	9	12	16	18	22	24	28
0	3	6	10	12	16	19	22	25	27
1	3	6	10	13	15	18	21	25	28
0	3	6	9	14	17	20	23	24	29
0	4	6	9	13	16	18	21	25	27
0	5	6	11	14	15	20	21	26	29
0	5	8	9	14	17	18	23	26	27
0	3	6	9	13	15	19	22	24	28
2	5	7	10	13	16	19	23	26	28
1	4	8	10	13	16	20	22	26	29
2	4	8	11	13	17	19	22	26	28
0	5	8	11	12	15	20	23	24	27
1	3	7	9	12	15	18	22	25	27
0	4	7	10	13	15	18	22	24	27
2	4	7	10	14	17	20	22	25	28
2	4	8	10	14	16	19	23	25	29
0	3	7	10	12	16	18	21	24	28
2	5	6	9	12	15	18	23	26	29
0	3	8	11	12	17	18	21	26	29
2	5	8	9	12	17	20	21	24	29
1	5	8	10	13	17	20	23	25	28
1	4	6	10	12	15	19	21	24	27
1	5	7	10	14	17	19	22	26	29
0	4	7	9	12	15	19	21	25	28
2	5	7	11	13	16	20	22	25	29

(2, 1)-covering array from Table 6.11 (page 123)

$TOCA(N; 3, 10, 4; 0), N = 66$

2	6	11	15	19	22	26	31	35	39
2	4	8	14	16	22	26	30	34	36
1	6	9	14	17	21	26	30	33	38
0	4	8	12	17	21	24	29	33	37
2	6	10	14	17	22	26	29	33	37
3	4	8	15	16	20	27	28	35	39
2	5	10	14	17	21	25	30	34	38
1	5	11	15	17	21	27	29	35	37
1	5	9	15	19	23	25	29	33	39
3	7	11	15	18	23	26	31	34	38
3	7	11	12	16	20	24	28	32	39
2	7	11	15	19	22	27	30	34	38
1	5	10	13	17	22	26	30	34	37
3	7	9	15	19	21	25	29	35	37
2	6	9	13	17	22	25	29	34	38
0	4	10	12	16	22	26	30	32	38
0	7	8	15	16	23	24	31	32	39
0	4	9	13	16	21	25	29	32	36
2	7	10	14	19	23	27	31	35	38
3	7	11	13	17	23	25	29	33	37
2	4	10	12	18	20	24	30	34	38
2	6	11	14	18	23	27	31	34	39
3	5	11	15	17	21	25	31	33	39
2	6	9	13	18	21	26	30	34	37
3	7	10	14	19	22	26	31	34	39
1	4	9	12	16	21	24	28	33	36
0	6	8	12	18	22	24	30	34	36
3	6	10	15	18	22	27	31	35	38
0	4	11	12	19	23	24	28	35	39
2	5	9	14	18	22	25	30	33	37
2	5	10	13	18	21	26	29	33	38
2	7	10	15	18	23	26	30	35	39
0	4	8	12	19	20	27	31	32	39
0	4	10	14	18	20	26	28	34	36
0	7	11	15	19	20	27	28	32	36
1	5	8	13	16	20	24	29	33	36
1	7	11	13	19	21	27	29	33	39
1	5	11	13	19	23	25	31	35	37
1	5	9	14	18	22	26	29	34	38
2	6	8	12	16	20	26	28	34	38
0	5	9	12	16	20	24	29	32	37
0	4	8	13	16	20	25	28	33	37
1	7	9	15	17	23	27	31	33	37
3	5	9	13	17	23	27	29	35	39
0	7	8	12	16	23	27	28	35	36
2	6	10	12	18	22	26	28	32	36
1	6	10	13	18	22	25	30	33	38
2	4	8	14	18	22	24	28	32	38
3	7	11	14	18	22	27	30	35	39
2	6	10	14	16	20	24	30	32	36
1	6	10	14	18	21	25	29	34	37
3	6	11	14	19	23	26	30	35	38
3	7	8	12	19	20	24	31	35	36
1	4	9	13	17	20	24	28	32	37
3	5	9	13	19	21	27	31	33	37
3	4	8	15	19	23	24	28	32	36
0	5	9	12	17	20	25	28	33	36
3	4	11	12	16	23	27	31	32	36
1	5	8	12	16	21	25	28	32	37
0	6	8	14	18	20	26	30	32	38
1	7	9	13	17	21	25	31	35	39
1	4	8	12	17	20	25	29	32	36
3	6	10	15	19	23	27	30	34	39
0	4	11	15	16	20	24	31	35	36
0	5	8	13	17	21	24	28	32	36
0	6	10	14	16	22	24	28	34	38

Covering array built using an $OA_3(2,9,3)$ as a seed: from Table 6.12 (page 123)

$CA(3,9,3), N = 50$								
0	3	6	9	12	15	18	21	24
1	4	7	10	13	16	19	22	25
2	5	8	11	14	17	20	23	26
0	4	8	9	13	17	18	22	26
1	5	6	10	14	15	19	23	24
2	3	7	11	12	16	20	21	25
0	5	7	9	14	16	18	23	25
1	3	8	10	12	17	19	21	26
2	4	6	11	13	15	20	22	24
0	3	6	11	14	17	19	22	25
1	4	7	9	12	15	20	23	26
2	5	8	10	13	16	18	21	24
0	4	8	11	12	16	19	23	24
1	5	6	9	13	17	20	21	25
2	3	7	10	14	15	18	22	26
0	5	7	11	13	15	19	21	26
1	3	8	9	14	16	20	22	24
2	4	6	10	12	17	18	23	25
0	3	6	10	13	16	20	23	26
1	4	7	11	14	17	18	21	24
2	5	8	9	12	15	19	22	25
0	4	8	10	14	15	20	21	25
1	5	6	11	12	16	18	22	26
2	3	7	9	13	17	19	23	24
0	5	7	10	12	17	20	22	24
1	3	8	11	13	15	18	23	25
2	4	6	9	14	16	19	21	26
2	5	7	9	12	16	20	23	24
1	4	6	9	13	16	18	23	24
2	5	6	9	13	17	19	22	26
2	3	8	10	13	15	20	23	24
1	5	7	10	13	17	18	23	26
0	3	7	9	14	15	19	23	26
0	5	6	10	12	16	19	21	25
2	3	6	11	14	16	19	21	24
1	3	7	9	12	17	18	22	25
2	4	8	11	12	17	18	21	26
1	4	7	10	14	16	20	21	26
2	4	7	9	13	15	18	21	25
0	5	8	10	12	15	20	22	26
1	5	8	11	14	15	18	21	24
0	4	7	11	13	17	20	23	25
1	3	6	11	12	15	20	23	26
0	3	8	11	13	16	18	22	26
0	4	6	10	14	17	18	22	24
1	4	7	11	12	15	19	22	24
1	4	8	9	14	17	19	23	25
0	3	8	9	13	17	20	21	24
2	3	8	10	13	16	19	22	25
2	5	6	11	14	15	20	22	25

Difference covering arrays with and without zero differences
(from Table 6.14)
used to build covering arrays in Table 6.13 (page 123)

$DCA(N;2,5,5)$ $N=5$				
2	2	2	2	4
0	1	3	4	4
1	3	2	4	2
3	2	0	4	3
2	0	1	4	0

$DCA(N;2,5,6)$ $N=8$				
1	2	0	1	2
4	0	2	3	2
3	1	0	2	5
3	3	4	1	2
1	5	5	5	0
3	3	3	0	0
3	0	1	5	3
4	3	0	5	1

$DCA(N;2,5,6)$ $N=7$,without all zero differences				
5	0	2	4	1
2	5	0	4	3
1	5	3	2	4
3	5	4	1	2
4	3	3	4	5
3	3	0	0	1
0	2	4	1	5

$DCA(N;2,9,4)$ $N=7$								
1	3	1	3	1	2	1	3	2
2	2	1	2	3	2	1	0	3
3	2	1	2	2	2	3	2	0
3	2	2	2	1	3	1	0	1
1	0	3	2	3	2	0	1	0
3	3	0	0	2	0	0	1	3
0	1	0	0	1	2	0	0	0

$DCA(N;2,9,4)$ $N=6$, without all zero differences								
1	1	2	0	3	2	2	0	3
0	2	2	3	3	1	3	2	1
3	2	1	0	1	2	3	0	2
0	3	3	3	1	1	2	0	0
1	2	0	3	3	0	0	1	2
1	1	0	0	2	3	2	1	0

$DCA(N;2,4,9)$ $N=10$			
1	4	1	7
8	7	4	4
8	7	0	3
4	2	6	2
2	4	5	3
5	0	0	8
2	3	8	1
8	5	4	6
0	0	7	2
5	1	4	5

$DCA(N;2,4,9)$ $N=9$, without all zero differences			
8	3	6	2
3	8	5	7
1	2	7	8
4	3	7	6
6	8	7	3
6	0	2	5
8	5	3	7
5	3	4	1
0	0	3	1

$DCA(N;2,4,6)$ $N=7$			
2	1	2	0
3	1	5	1
1	2	4	3
0	0	5	0
0	1	4	3
3	0	4	4
3	5	5	2

$DCA(N;2,4,6)$ $N=6$, without all zero differences			
5	2	4	3
2	3	0	4
1	3	4	0
5	4	3	0
2	1	3	5
0	4	2	3

$DCA(N;2,5,5)$ $N=4$, without all zero differences				
3	1	2	0	4
1	0	3	2	4
2	3	0	1	4
3	0	4	1	2

$DCA(N;2,6,4)$ $N=6$					
1	0	0	2	1	1
3	3	0	3	3	0
2	0	0	0	0	0
1	2	0	0	0	2
2	3	2	0	2	1
1	2	0	1	2	1

$DCA(N;2,6,4)$ $N=5$, without all zero differences					
0	2	2	0	1	0
1	2	3	2	3	0
2	1	0	2	1	3
0	3	1	2	2	0
0	3	3	3	1	2

$DCA(N;2,6,5)$ $N=7$					
3	1	3	3	2	3
4	3	3	4	2	3
4	4	2	1	3	3
2	4	4	4	4	2
1	3	2	4	0	4
3	0	1	4	3	0
0	1	1	4	1	1

$DCA(N;2,6,5)$ $N=6$, without all zero differences					
1	4	2	1	0	3
0	4	0	3	4	3
2	1	2	4	4	3
0	4	3	4	2	0
3	0	2	1	4	1
3	4	0	4	1	2

Index of Terms