MIXED COVERING ARRAYS ON GRAPHS AND TABU SEARCH ALGORITHMS

Latifa Zekaoui

September 2006

A Thesis submitted to the
Ottawa-Carleton Institute for Computer Science
at the University of Ottawa
in partial fulfillment of the requirements
for the degree of
Master's of Science (Computer Science)

Abstract

Mixed covering arrays on graphs and tabu search algorithms

Latifa Zekaoui, Master's 2006

Ottawa-Carleton Institute for Computer Science

University of Ottawa

Covering arrays are combinatorial objects that have been successfully applied in the design of test suits for testing software, networks and circuits. Mixed covering arrays on graphs are new generalizations of both mixed covering arrays and covering arrays on graphs.

In this thesis, we give new theoretical results and constructions for mixed covering arrays on graphs. First, we extend to the mixed case the work done by Meagher and Stevens [31], which uses graph homomorphisms for covering arrays on graphs. Second, we study covering arrays on special classes of graphs. In particular, we solve to optimality the case in which G is a tree, a cycle or a bipartite graph, as well as give results for wheels and cubic graphs. We also provide general graph operations that preserve the size of a balanced covering array.

In the second part of the thesis, we do a complete experimental study of two tabu search algorithms for covering array construction. POT is a variation of the algorithm given by Stardom [40] while PAT is an implementation of the algorithm proposed by Nurmela [35]. We conclude that they provide effective methods for constructing covering arrays of moderate size. In particular, POT and PAT improve upper bounds for 18 sets of parameters for covering arrays.

Acknowledgements

First and foremost, I would like to thank my supervisor, Lucia Moura, for her guidance, encouragement and support throughout the whole process of this research and for helping me shape my vague ideas into coherent research.
I would also like to thank the other members of the examination committee, Brett Stevens and Alan Williams, for their feedback and suggestions on this thesis.
I wish to extend my sincere gratitude to my parents, Farida and Abdelhamid Zekaoui, whose love and patience pushed me to be the best that I can and provided me with the necessary skills to achieve this goal.
I am grateful to my husband, Gihad Abdul-Rahim, for believing in me and for his continuous love and support throughout this journey.
Many thanks to Sofiane, Walid and Dyna for their encouragement and support.
Finally, I would like to thank the School of Information Technology and Engineering at the University of Ottawa for their financial support and also acknowledge the generous support of the National Science and Engineering Research Council.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Covering arrays have been extensively studied and have been the topic of interest and focus of many researchers. These interesting mathematical structures are generalizations of the well-known orthogonal arrays [25].

Let n, k, g be positive integers. A *covering array*, denoted by $CA(n, k, g)$, is an $n \times k$ array with entries from $\mathbb{Z}_g$ such that every pair of columns has every possible pair in $\mathbb{Z}_g \times \mathbb{Z}_g$ appearing in some row. The number of rows in such an array is called its *size*. Given k and g, the covering array number, denoted by $CAN(k, g)$, is the minimum n for which there exists a $CA(n, k, g)$. A $CA(n, k, g)$ of size $n = CAN(k, g)$ is called *optimal*. The question one would like an answer for is "What is the smallest n such that a $CA(n, k, g)$ exists?". Although, the question posed seems simple, it is a very complicated and difficult problem to construct covering arrays of optimal size.

Covering arrays have applications in many areas. They are used for testing software [6, 9, 16, 17], circuits [38] and networks [45]. They have also been applied to problems in other domains, such as material science [2], genomics [39] and statistical design of experiments [37].

Covering arrays are particularly useful in the design of test suites [6, 7, 20, 45, 46]. The testing application is based on the following translation. The system to be tested has k parameters, each of which can take one out of g possible values. We wish to build a test set that tests all pairwise interactions of parameters with the minimum number n of tests. Exhaustively testing every possible parameter combination would

be impossible due to time and cost constraints, as the number of possible combinations is exponential in k. Covering arrays provide compact test suites that guarantee pairwise coverage of parameters.

The main contributions of this thesis are a theoretical study of a new generalization of covering arrays and an experimental study of two tabu search algorithms for covering array construction. We outline each of these contributions in Sections 1.1 and 1.2, respectively, and give an overview of the thesis in Section 1.3.

1.1 Contributions on mixed covering arrays on graphs

Several generalizations of covering arrays have been proposed in order to address different requirements of the testing application (see [11, 24]). *Mixed covering arrays* are a generalization of covering arrays that allows for different alphabets in different columns. This meets the requirement that different parameters in the system may take a different number of possible values. Constructions for mixed covering arrays are given in [15, 33]. Another generalization of covering arrays are *covering arrays on graphs*. In these arrays, only specified pairs of columns need to satisfy the pairwise coverage requirements and these pairs are recorded in a graph structure [29, 31]. This is useful in situations in which some pairs of parameters do not interact; in these cases, we do not insist that these interactions be tested, which allows reductions in the number of required tests. This has been applied in the context of software testing by observing that we only need to test interactions between parameters that jointly affect one of the output values [4].

In the first part of the thesis, we study mixed covering arrays on graphs which generalize both mixed covering arrays and covering arrays on graphs. The addition of a graph structure to covering arrays makes it possible to use methods from graph theory to study these structures. Covering arrays on graphs were first studied by Serroussi and Bshouty [38], who showed that finding an optimal covering array on a graph is NP-hard for the binary case. Only more recently, covering arrays on general alphabets have been systematically studied. They were introduced in Steven's thesis [42] and Stevens and Meagher [29, 31] studied covering arrays on graphs in more

detail and gave some powerful results.

Let G be a graph with k vertices $v_1, v_2, \ldots, v_k$ with respective vertex weights $g_1 \leq g_2 \leq \cdots \leq g_k$. A *mixed covering array* on G, denoted by $CA(n, G, \prod_{i=1}^{k} g_i)$, is an $n \times k$ array such that column i corresponds to v_i, cells in column i are filled with elements from $\mathbb{Z}_{g_i}$ and every pair of columns i, j corresponding to an edge $\{v_i, v_j\}$ in G has every possible pair from $\mathbb{Z}_{g_i} \times \mathbb{Z}_{g_j}$ appearing in some row. The number of rows in such an array is called its *size*. Given a weighted graph G, a mixed covering array on G with minimum size is called *optimal*. In this thesis, we extend the work done by Meagher and Stevens [31] for covering arrays on graphs to the mixed case. We define a weight-restricted graph homomorphism and use it to give bounds on the mixed covering array number. We briefly define the special class of graphs called the mixed qualitative independence graphs. We use them to relate the existence of mixed covering arrays on graphs to the mixed qualitative independence graphs via weight-restricted graph homomorphisms, which allows us to derive general bounds for the mixed covering array number. In addition to generalizing the work in [31], we study covering arrays on special classes of graphs. We give graph operations that allow us to add vertices to a graph, while preserving the size of a balanced covering array on the graph. If G is n-colourable, for $n = 2, 3, 4, 5$, we show that in many cases the product of the largest two alphabets is an upper bound on the mixed covering array number. For the case in which G is a tree, a cycle or a bipartite graph, we get a stronger result, namely that its mixed covering array number is the largest product of alphabets connected by an edge. All the results mentioned up to this point appeared in our paper [30]. In addition, we determine the covering array numbers for wheels with uniform alphabet sizes and give an upper bound on the mixed covering array number for cubic graphs.

1.2 Contributions on tabu search algorithms for covering arrays

In the second part of the thesis, we study and implement tabu search algorithms for covering array construction. The tabu search algorithm is a metaheuristic method that was proposed by Glover [21]; a detailed account is given by Glover and Laguna [22]. Metaheuristic algorithms are capable of solving a wide range of combinatorial problems quickly and effectively, using generalized heuristics which can be tailored to suit the problem at hand. Heuristic search algorithms try to find a certain combinatorial structure or solve an optimization problem by the use of heuristics. A heuristic search is a method of performing a minor modification of a given solution in order to obtain a different solution. Some examples of heuristic search algorithms are hill climbing, simulated annealing and tabu search. Different variations of the tabu search algorithm have been proposed by Nurmela [35] and Stardom [40] in order to obtain upper bounds on the size of covering arrays.

We implement two tabu search algorithms which we call the point tabu search (POT) and the pair tabu search (PAT). POT algorithm is a variation of the tabu search algorithm proposed by Stardom [40]; we call it point tabu search because the basic tabu move is to switch a point in the array. PAT is an implementation of the algorithm proposed by Nurmela [35]; we call it pair tabu search because the basic tabu move aims at covering a new pair in the array. The main objective of this part of the thesis is to provide a detailed description of an efficient implementation of these algorithms, and to perform a complete experimental analysis of these methods.

We give a description of POT and PAT that includes data structures used and algorithm analysis which goes into much more detail than the original references. Our POT algorithm runs in $O(nk^2g)$ per tabu move, while PAT runs in $O(nk)$ per tabu move.

The objective of the experimental study is to examine the algorithms' behaviour individually, to try to improve on the implementation efficiency by tuning up parameters, to perform a detailed comparison between POT and PAT and to determine how they compare with other algorithms, and to find new upper bounds for covering

array numbers. We tune up the input parameters for both POT and PAT using a test bed of 30 test cases designed by us. We run our algorithms on a test bed acquired from Cohen's PhD thesis [9], which involves a comparison against many algorithms found in the literature that were used in the construction of fixed and mixed covering arrays. We also attempt to improve on upper bounds reported in recent tables by Colbourn [10] that summarize the best known bounds for covering arrays using all known constructions. We also report on a few new upper bounds for mixed covering arrays with parameter values close to those of an orthogonal array. We conclude that POT is comparable to PAT in terms of percentage success of finding covering arrays, although PAT is substantially faster than POT for most of our test cases. POT and PAT achieve most of the best known upper bounds reported in Cohen's test bed and even improve on 13 bounds from Colbourn's tables; in addition, PAT improves on 5 mixed covering array bounds.

1.3 Overview of the thesis organization

In Chapter 2, we give the necessary background on covering arrays and discuss some basic constructions as well as some generalizations. In particular, we discuss mixed covering arrays, and covering arrays on graphs and their relation to graph homomorphism. In Chapter 3, we study mixed covering arrays on graphs which generalize both mixed covering arrays and covering arrays on graphs. This chapter consists of original contributions towards constructions of these objects, which appeared in our paper [30]. As described previously, these contributions generalize the work of Meagher and Stevens [31], and provide additional optimal constructions for covering arrays on special classes of graphs. In Chapter 4, we discuss the two tabu search methods that we implemented, namely POT and PAT. We introduce tabu search in general, and give POT and PAT along with their pseudocode, data structures used and complexity analysis. In Chapter 5, we conduct a thorough experimental analysis of POT and PAT. We compare the effectiveness of each algorithm against one another and against some other algorithms for which data is provided in the literature. We also report on new improved upper bounds for covering array numbers found by our

algorithms. Finally, in Chapter 6, we conclude by outlining the main findings in this thesis and pointing out some future work. In the Appendix, we include tables of the best known upper bounds on the size of covering arrays, as reported by Colbourn [10].

Chapter 2

Background on Covering Arrays

In this chapter, we review covering array definitions and basic constructions as well as their associated generalizations.

2.1 Covering Arrays

Covering arrays have been studied extensively in the past few years [11] and their primary application is in software [6, 14, 16, 17], hardware [14], network [39] and circuit [33] testing.

Two length-n vectors x, y with entries from $\mathbb{Z}_g$ and $\mathbb{Z}_h$, respectively, are *qualitatively independent* if for any pair $(a, b) \in \mathbb{Z}_g \times \mathbb{Z}_h$, there exists $i \in \{1, 2, \ldots, n\}$ such that $(x_i, y_i) = (a, b)$. Covering Arrays are arrays whose columns are qualitatively independent.

Definition 2.1.1. *(Covering Arrays) Let n, k, g be positive integers. A* covering array*, denoted by $CA(n, k, g)$, is an $n \times k$ array A with entries from $\mathbb{Z}_g$ such that any two distinct columns of A are qualitatively independent.*

The parameter n is called the *size* of the array. Given k and g, the covering array number, denoted by $CAN(k, g)$, is the minimum n for which there exists a $CA(n, k, g)$. A $CA(n, k, g)$ of size $n = CAN(k, g)$ is called *optimal*. An obvious lower

bound for the size of a covering array is g^2 in order to guarantee that two of the columns be qualitatively independent.

Example 2.1.2. *The following array is an optimal* $CA(5, 4, 2)$*:*

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

Let us see a software application of covering arrays. The model most commonly used is that of a computer program or little snippet of code. The program has k input parameters and each can take one of g possible values. To perform a full coverage of the parameter space, one would have to test every possible combination of the input parameters. As g and k grow, the number of possible combinations g^k become infeasible to test. Cohen et al. [6] show by the use of empirical data that the vast majority of defects are due to the interaction between two parameters. Using covering arrays to build test suites ensures pairwise coverage and leads to quicker testing methodologies.

A lot of research has been developed and is being conducted to determine the minimum size of covering arrays for different parameter values as well as to develop constructions to build these interesting mathematical structures. Although, the problem of finding $CAN(k, g)$ for general k and g is not an easy problem, the determination of $CAN(k, 2)$ has been completely solved by Katona [26] and Kleitman and Spencer [27], independently. The construction is very simple. It specifies that given n, in order to build a $CA(n, k, 2)$ with maximum k, we form a matrix in which the columns consist of all distinct binary n-tuples of weight $\lceil \frac{n}{2} \rceil$ that have a 0 in the first position. The next theorem guarantees that this is in indeed a covering array, and gives a maximum k.

Theorem 2.1.3. *(Katona [26], Kleitman & Spencer [27]) Let* k *be a positive integer, then*

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

Another useful construction for covering arrays makes use of combinatorial designs called *orthogonal arrays*. An orthogonal array, denoted $OA(k, g)$, is a $g^2 \times k$ array, A, with entries from a $\mathbb{Z}_g$ such that any two distinct columns of A are qualitatively independent. Orthogonal arrays yield optimal covering arrays, since they have g^2 rows, which match the lower bound for $CAN(k, g)$.

Proposition 2.1.4. *If an $OA(k, g)$ exists then $CAN(k, g) = g^2$.*

The following theorem ensures the existence of orthogonal arrays with $g + 1$ columns when g is a prime power.

Theorem 2.1.5. *(Bush [1]) For a prime power g, there exists an $OA(g + 1, g)$.*

The maximum number of columns possible in an orthogonal array is $k = g + 1$. When covering arrays with larger k are needed, other constructions are required.

The *blocksize recursive construction* is a construction that concatenates copies of existing smaller covering arrays to build a larger covering array with the same alphabet size as can be seen in the next theorem.

Theorem 2.1.6. *(Cohen & Fredman [8], Poljak & Tuza [36], Stevens & Mendelsohn [41], Williams [46]) If there exists a $CA(m, j, g)$ and a $CA(n, k, g)$, then there exists a $CA(m + n, jk, g)$.*

Sketch of the proof. Let A be a $CA(m, j, g)$ and B be a $CA(n, k, g)$. We build C, a $CA(m+n, jk, g)$. In order to build the first k columns of C, the larger covering array, we need to repeat the first column of A, k times, and concatenate each of these columns with all the columns from the array B. Similarly, repeat each other column of A, k times, and concatenate to it a full copy of B. It can be verified that this constructs a $CA(m + n, jk, g)$. □

For fixed g, $CAN(k, g)$ has been asymptotically determined as k grows.

Theorem 2.1.7. *(Gargano, Korner & Vaccaro [20]) Let g be a positive integer. The following asymptotic result holds:*

$$\lim_{k\to\infty} CAN(k, g) = \frac{g}{2} \log_2 k$$

This result is not constructive, so it sheds no light on how to construct optimal covering arrays. Several generalizations of covering arrays have been proposed in order to address different requirements of the testing applications [11, 24]. In Section 2.2 and 2.3, we discuss two of these generalizations.

2.2 Mixed Covering Arrays

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

Definition 2.2.1. *(Mixed Covering Array) Let $n, k, g_1, \ldots, g_k$ be positive integers. A* mixed covering array*, denoted by $CA(n, \prod_{i=1}^{k} g_i)$, of type $\prod_{i=1}^{k} g_i$, is an $n \times k$ array A with entries from $\mathbb{Z}_{g_i}$ in column i, such that any two distinct columns of A are qualitatively independent.*

The parameter n is called the *size* of the array. Given k and g_i for $1 \leq i \leq k$, the covering array number, denoted by $CAN(\prod_{i=1}^{k} g_i)$, is the minimum n for which there exists a $CA(n, \prod_{i=1}^{k} g_i)$. A $CA(n, \prod_{i=1}^{k} g_i)$ of size $n = CAN(\prod_{i=1}^{k} g_i)$ is called *optimal*. An obvious lower bound for the size of a covering array is $g_i g_j$, where g_i and g_j are the largest two alphabets, in order to guarantee that the corresponding columns be qualitatively independent.

Example 2.2.2. *The following array is an optimal $CA(6, 4, 2^3 3^1)$.*

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

Constructions of mixed covering arrays are discussed in [15] and [33]. Next, we list some results that are used in the thesis.

Theorem 2.2.3. *(Moura et al. [33])(Increasing one of the alphabet sizes) Let* $e \geq 0$, $1 \leq i \leq k$, $g_1 \leq g_2 \leq \cdots \leq g_k$, *and* $k' = max\ \{j : 1 \leq j \leq k, j \neq i\}$. *Then,* $CAN(g_1 g_2 \ldots (g_i + e) \ldots g_k) \leq CAN(g_1 g_2 \ldots g_k) + e g_{k'}$.

Theorem 2.2.4. *(Moura et al. [33])(Decreasing one of the alphabet sizes) Let* $e \geq 0$ *and* $1 \leq i \leq k$. *Then,* $CAN(g_1 g_2 \ldots (g_i - e) \ldots g_k) \leq CAN(g_1 g_2 \ldots g_i \ldots g_k)$.

Theorem 2.2.5. *(Moura et al. [33]) Let* $g_1 \leq g_2 \leq g_3 \leq g_4 \leq g_5$ *and* $g_i \geq 2$. *For* $k \leq 4$, *we have* $CAN(\prod_{i=1}^{k} g_i) = g_{k-1} g_k$ *except in the following cases:* $CAN(2^4) = 5$ *and* $CAN(6^4) = 37$. *For* $k = 5$, *if* $g_4 \neq 6, 10$, *then* $CAN(\prod_{i=1}^{5} g_i) = g_4 g_5$ *except in the following cases:* $CAN(2^5) = 6$, $CAN(3^5) = 11$ *and* $CAN(2^1 3^4) = 10$.

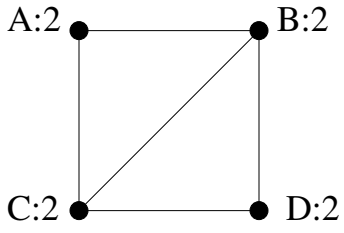
2.3 Covering Arrays on Graphs and Graph Homomorphism

Covering arrays on graphs are generalizations of covering arrays that only require that specified pairs of columns need to be qualitatively independent and these pairs are recorded in a graph structure [29, 31]. This is useful in situations in which some pairs of parameters do not interact; in these cases, we do not insist that these interactions be tested, which allows reductions in the number of required tests. This has been applied in the context of software testing by observing that we only need to test interactions between parameters that jointly affect one of the output values [4].

Definition 2.3.1. *(Covering Array on a Graph) Let* n *and* g *be positive integers and* G *be a graph with* k *vertices. A* covering array on G, *denoted by* $CA(n, G, g)$, *is an* $n \times k$ *array with the following properties: 1) the cells in column* i *are filled with elements from a* g*-ary alphabet, which is usually taken to be* $\mathbb{Z}_g$*; 2) column* i *corresponds to a vertex* $v_i \in V(G)$*; and 3) pairs of columns that correspond to adjacent vertices of* G *are qualitatively independent.*

Given a graph G and a positive integer g, the *covering array number* $CAN(G, g)$ is the minimum n for which there exists a $CA(n, G, g)$. A $CA(n, G, g)$ of size $n = CAN(G, g)$ is called *optimal*.

Example 2.3.2. *The following is an optimal* $CA(4, 4, 2^4)$ *for the graph below. Since vertices A and D are not adjacent in the graph, their corresponding columns need not be qualitatively independent.*



A	B	C	D
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Before we discuss known results for covering arrays on graphs, we need to review some graph theoretical definitions and results.

A mapping ϕ from V(G) to V(H) is a *graph homomorphism* from G to H (or simply a *homomorphism*) if for all $u, v \in V(G)$, the vertices $\phi(u)$ and $\phi(v)$ are adjacent in H whenever u and v are adjacent in G. For graphs G and H, if there exists a homomorphism from G to H we write $G \to H$.

The *complete graph* on n vertices, K_n, is the graph with n vertices and with an edge between any two distinct vertices. A *proper colouring* of G with n colours is a map from $V(G)$ to a set of n colours such that no two adjacent vertices are mapped to the same colour. A proper colouring of a graph G with n colours is equivalent to a homomorphism from G to K_n. The *chromatic number* of a graph G, denoted $\chi(G)$, is the smallest n such that $G \to K_n$. A *clique* in a graph G is a set C of vertices from $V(G)$ such that any two distinct vertices in C are adjacent in G. A clique of cardinality n in G is equivalent to a homomorphism from K_n to G. The *clique number* of a graph G, denoted by $\omega(G)$, is the largest n such that $K_n \to G$. For graphs G and H, if there is a homomorphism $G \to H$ then $\chi(G) \leq \chi(H)$ and $\omega(G) \leq \omega(H)$. Also, for all graphs G, $\omega(G) \leq \chi(G)$.

Next, we give some results by Meagher and Stevens [31] that relate graph homomorphisms with covering arrays on graphs. In section 3.2, we generalize these results to mixed covering arrays on graphs.

Theorem 2.3.3. *(Meagher and Stevens [31]) Let g be a positive integer and G and H be graphs. If there exists a graph homomorphism $\phi : G \to H$, then*

$$CAN(G, g) \leq CAN(H, g).$$

For any graph G, there are homomorphisms between the following graphs

$$K_{\omega(G)} \to G \to K_{\chi(G)}.$$

These homomorphisms can be used to find bounds on $CAN(G, g)$.

Corollary 2.3.4. *(Meagher and Stevens [31]) For all positive integers g and all graphs G,*

$$CAN(K_{\omega(G)}, g) \leq CAN(G, g) \leq CAN(K_{\chi(G)}, g).$$

A k-partition of an n-set is a set of k disjoint non-empty classes whose union is the n-set. Let P_k^n denote the set of all k-partitions of an n-set. We say that a vector $x \in \mathbb{Z}_g^n$ *corresponds* to a g-partition $P = \{P_1, P_2, \ldots, P_g\}$ of an n-set if

$$x_i = a \text{ if and only if } i \in P_a, \text{ for all } 1 \leq i \leq n, 1 \leq a \leq g.$$

Throughout this thesis, we denote by x_p the vector corresponding to partition P. We say that partitions P and Q are *qualitatively independent* if their corresponding vectors x_P and x_Q are qualitatively independent.

Definition 2.3.5. *(Qualitative Independence Graph) Let n and g be positive integers with $n \geq g^2$. Define the* qualitative independence graph*, $QI(n, g)$, to be the graph whose vertex set is the set of all g-partitions of an n-set with the property that every class of the partition has size at least g. Vertices are adjacent if and only if the corresponding partitions are qualitatively independent.*

The next theorem relates the existence of covering arrays on graphs to the mixed qualitative independence graphs via graph homomorphisms.

Theorem 2.3.6. *(Meagher and Stevens [31]) For a graph G and positive integers g and n, there exists a $CA(n, G, g)$ if and only if there exists a graph homomorphism $G \to QI(n, g)$.*

We can therefore rewrite the previous result in terms of covering array numbers.

Corollary 2.3.7. *(Meagher and Stevens [31]) For any graph G, and any positive integer g,*

$$CAN(G, g) = \min_{n \in \mathbb{N}}\{n \ : \ G \to QI(n, g)\}.$$

Since a homomorphism $G \to H$ implies $\chi(G) \leq \chi(H)$ and $\omega(G) \leq \omega(H)$, the following corollary is a direct consequence of Theorem 2.3.6.

Corollary 2.3.8. *(Meagher and Stevens [31]) Let G be a graph and n, g be positive integers. If there exists a $CA(n, G, g)$, then*

$$\chi(G) \leq \chi(QI(n, g)) \text{ and } \omega(G) \leq \omega(QI(n, g)).$$

Chapter 3

Mixed Covering Arrays on Graphs

In this chapter, we study mixed covering arrays on graphs, which generalize both covering arrays on graphs and mixed covering arrays.

We extend results given in Section 2.3 for covering arrays on graphs to the mixed case. We give graph operations that allow us to add vertices to a graph, while preserving the size of a balanced covering array on the graph. If G is n-colourable, for $n = 2, 3, 4, 5$, we show that in many cases the product of the largest two alphabets is an upper bound on the mixed covering array number. For the case in which G is a tree, a cycle or a bipartite graph, we get a stronger result, namely that its mixed covering array number is the largest product of alphabets connected by an edge. The contents of this chapter appear in our paper [30], except for extra results for wheels and cubic graphs.

3.1 Mixed covering arrays on graphs

Mixed covering arrays on graphs allow for different alphabets in different columns as well as ensure that only specified pairs of columns need to be qualitatively independent. The first property meets the requirement that different parameters in the system may take a different number of possible values, while the second property is useful in situations in which some pairs of parameters do not interact.

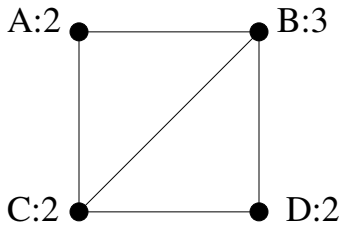
The parameters for mixed covering arrays on graphs are given by a vertex-weighted

graph. A *vertex-weighted graph* or simply a *weighted graph* is a graph with a weight assigned to each vertex. A weighted graph is given by a triple $(V(G), E(G), w_G(\cdot))$ where $V(G)$ is the vertex set, $E(G)$ is the edge set and $w_G : V(G) \to \mathbb{N}^+$ is the weight function. We assume that the vertices are labelled so that the weights are nondecreasing, that is, $w_G(v_i) \leq w_G(v_j)$, for all $1 \leq i \leq j \leq |V(G)|$. When there is no ambiguity on which graph we are using, we can write $w(\cdot)$ in place of $w_G(\cdot)$.

Definition 3.1.1. *(Mixed Covering Array on a Graph) Let G be a weighted graph with k vertices and weights $g_1 \leq g_2 \leq \cdots \leq g_k$, and let n be a positive integer. A* mixed covering array *on G, denoted by $CA(n, G, \prod_{i=1}^{k} g_i)$, is an $n \times k$ array with the following properties: 1) the cells in column i are filled with elements from a g_i-ary alphabet, which is usually taken to be $\mathbb{Z}_{g_i}$; 2) column i corresponds to a vertex $v_i \in V(G)$ with $w_G(v_i) = g_i$; and 3) pairs of columns that correspond to adjacent vertices of G are qualitatively independent. Given a weighted graph G with weights $g_1, g_2, \ldots, g_k$, the* mixed covering array number *on G, denoted by $CAN(G, \prod_{i=1}^{k} g_i)$, is the minimum n for which there exists a $CA(n, G, \prod_{i=1}^{k} g_i)$. A $CA(n, G, \prod_{i=1}^{k} g_i)$ of size $n = CAN(G, \prod_{i=1}^{k} g_i)$ is called* optimal.

A mixed covering array, denoted by $CA(n, \prod_{i=1}^{k} g_i)$, is a $CA(n, K_k, \prod_{i=1}^{k} g_i)$, where K_k is the complete graph on k vertices with weights g_i, for $1 \leq i \leq k$. A covering array on a graph, denoted by $CA(n, G, g)$, is a $CA(n, G, g^k)$ where $k = |V(G)|$.

Example 3.1.2. *The following is an optimal $CA(6, 4, 2^3 3^1)$ for the graph below.*



A	B	C	D
0	0	0	0
1	0	1	1
0	1	1	1
1	1	0	0
0	2	1	0
1	2	0	1

3.2 Weight-restricted graph homomorphisms

In this section, we generalize the results by Meagher and Stevens [31] given in section 2.3.

Let G and H be weighted graphs. A mapping ϕ from $V(G)$ to $V(H)$ is a *weight-restricted graph homomorphism* from G to H if ϕ is a graph homomorphism from G to H such that $w_G(v) \leq w_H(\phi(v))$, for all $v \in V(G)$. For weighted graphs G and H, if there exists a weight-restricted homomorphism from G to H then we write $G \xrightarrow{w} H$.

In the next proof, we use the concept of *dropping the alphabet size* of a particular column of a mixed covering array (from Theorem 2.2.4). Let $h \geq g$. To drop the alphabet size from h to g in a column of a covering array, we replace all symbols from $\mathbb{Z}_h \backslash \mathbb{Z}_g$ in the column by arbitrary symbols from $\mathbb{Z}_g$. Any pair of qualitatively independent columns of the covering array prior to the dropping operation will remain qualitatively independent.

Theorem 3.2.1. *Let G and H be weighted graphs with weights $g_1, g_2, \ldots, g_k$ and $h_1, h_2, \ldots, h_\ell$, respectively. If there exists a weight-restricted graph homomorphism $\phi : G \xrightarrow{w} H$ then $CAN(G, \prod_{i=1}^{k} g_i) \leq CAN(H, \prod_{j=1}^{\ell} h_j)$.*

Proof. Let C^H be a $CA(n, H, \prod_{j=1}^{\ell} h_j)$. The covering array C^H is used to construct C^G, a $CA(n, G, \prod_{i=1}^{k} g_i)$. Let $i \in \{1, 2, \ldots, k\}$ be the index of a column of C^G, and let v_i be the corresponding vertex in G. Column i of C^G is constructed from the column corresponding to $\phi(v_i)$ in C^H by dropping its alphabet from $w_H(\phi(v_i))$ to $w_G(v_i)$. Now, for any edge $\{v_i, v_j\}$ in G, the pair $\{\phi(v_i), \phi(v_j)\}$ is an edge in H and columns of C^H corresponding to $\phi(v_i)$ and $\phi(v_j)$ are qualitatively independent. Since dropping the alphabet size preserves qualitative independence, columns i and j of C^G are qualitatively independent. □

The weight-restricted homomorphism defined above gives more than just an upper bound on the size of a mixed covering array on a graph, it also describes a construction for the array.

The next corollary generalizes Corollary 2.3.4 to the mixed case. The *complete weighted graph* on n vertices $K_n(g_1, \ldots, g_n)$ is a complete graph on n vertices with

weights $g_1, \ldots, g_n$ on the vertices. For any graph G, there exist homomorphisms between the following graphs $K_{\omega(G)} \to G \to K_{\chi(G)}$. These homomorphisms can be extended to weight-restricted homomorphisms and we get the following lower and upper bounds on $CAN(G, \prod_{i=1}^{k} g_i)$.

Corollary 3.2.2. *Let G be a weighted graph with k vertices and $g_1 \leq g_2 \leq \cdots \leq g_k$ be positive weights. Then,*

$$CAN(K_{\omega(G)}, \prod_{i=1}^{\omega(G)} g_i) \leq CAN(G, \prod_{j=1}^{k} g_j) \leq CAN(K_{\chi(G)}, \prod_{\ell=k-\chi(G)+1}^{k} g_\ell).$$

Proof. Let G be a weighted graph with positive weights $g_1 \leq g_2 \leq \cdots \leq g_k$. From Section 2.3, we know there exists a graph homomorphism $\phi : K_{\omega(G)} \to G$. Assign weights $g_1, \ldots, g_{\omega(G)}$ to $K_{\omega(G)}$ so that $w_{K_{\omega(G)}}(u) \leq w_{K_{\omega(G)}}(v)$ whenever $w_G(\phi(u)) \leq w_G(\phi(v))$. Since the g_i's are ordered by weight, it is clear that ϕ is a weight-restricted homomorphism. Similarly, from Section 2.3, there exists a graph homomorphism $\varphi : G \to K_{\chi(G)}$. Assign weights $g_{k-\chi(G)+1}, \ldots, g_k$ to $K_{\chi(G)}$ so that $w_{K_{\chi(G)}}(\varphi(u)) \leq w_{K_{\chi(G)}}(\varphi(v))$ whenever $w_G(u) \leq w_G(v)$. It is clear that φ is a weight-restricted homomorphism. □

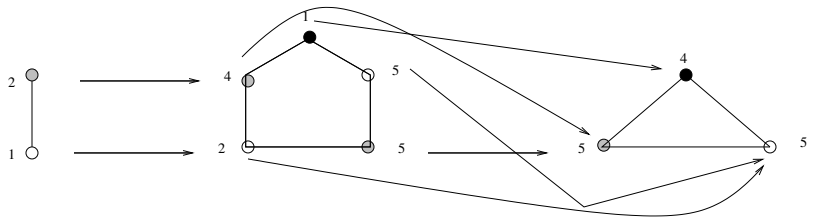


Figure 1: Illustrating the fact: $K_{\omega(G)}(g_1, \ldots, g_{\omega(G)}) \xrightarrow{w} G \xrightarrow{w} K_{\chi(G)}(g_{k-\chi(G)+1}, \ldots, g_k)$

Definition 3.2.3. *(Mixed Qualitative Independence Graph) Let n and $g_1 < \cdots < g_\ell$ be positive integers. The* mixed qualitative independence graph *$QI(n, \prod_{i=1}^{\ell} g_i)$ is the graph whose vertex set is $P_{g_1}^n \cup P_{g_2}^n \cup \cdots \cup P_{g_\ell}^n$ and two vertices are adjacent if and only if their corresponding partitions are qualitatively independent.*

Note that for all $i \in \{1, \ldots, \ell\}$, the graph $QI(n, g_i)$ is an induced subgraph of $QI(n, \prod_{i=1}^{\ell} g_i)$ and if $\ell = 1$ then $QI(n, \prod_{i=1}^{\ell} g_i) = QI(n, g_1)$.

Lemma 3.2.4. *Let n and $g_1 < g_2 < \cdots < g_\ell$ be positive integers and let $r_i = |P_{g_i}^n|$. Then, $CAN(QI(n, \prod_{i=1}^{\ell} g_i), \prod_{i=1}^{\ell} g_i^{r_i}) \leq n$.*

Proof. Build a $CA(n, QI(n, \prod_{i=1}^{\ell} g_i), \prod_{i=1}^{\ell} g_i^{r_i})$, by assigning to the column corresponding to vertex v (partition P_v), the vector x_{P_v}. □

The next theorem relates the existence of mixed covering arrays on graphs to the mixed qualitative independence graphs via weight-restricted graph homomorphisms.

Theorem 3.2.5. *For a weighted graph G and positive integers n and $g_1, g_2, \ldots, g_k$, there exists a $CA(n, G, \prod_{i=1}^{k} g_i)$ if and only if there exists a weight-restricted graph homomorphism $G \xrightarrow{w} QI(n, \prod_{i=1}^{k} g_i)$.*

Proof. Assume that there exists a $CA(n, G, \prod_{i=1}^{k} g_i)$; call it C. For any v in $V(G)$, denote by $x_v \in \mathbb{Z}_{g_i}^n$ the column in C corresponding to v and by P_v the partition corresponding to x_v. Consider a mapping $\phi : V(G) \to V(QI(n, \prod_{i=1}^{k} g_i))$ such that $\phi(v) = P_v$. The map ϕ is a weight-restricted homomorphism. To see this, let $v, u \in V(G)$ be adjacent vertices. Since C is a mixed covering array on G, columns x_v and x_u are qualitatively independent, thus the corresponding partitions P_v and P_u are qualitatively independent and adjacent in $QI(n, \prod_{i=1}^{k} g_i)$.

Conversely, assume there is a weight-restricted graph homomorphism $\phi : G \xrightarrow{w} QI(n, \prod_{i=1}^{k} g_i)$. We build C, a $CA(n, G, \prod_{i=1}^{k} g_i)$, as follows. For each $v \in V(G)$, $\phi(v)$ corresponds to a partition P_v. Take the column corresponding to v to be x_{P_v}, the vector corresponding to P_v. If the vertices $v, u \in V(G)$ are adjacent in G then the partitions $P_v = \phi(v), P_u = \phi(u)$ are qualitatively independent, thus the corresponding columns x_{P_v}, x_{P_u} are qualitatively independent. □

Corollary 3.2.6. *Let G be a weighted graph with distinct weights $g_1, g_2, \ldots, g_r$, repeated $s_1, s_2, \ldots, s_r$ times, respectively. Then,*

$$CAN(G, \prod_{i=1}^{r} g_i^{s_i}) = \min_{n \in \mathbb{N}} \{n : G \xrightarrow{w} QI(n, \prod_{i=1}^{r} g_i)\}.$$

The next corollary can be used to find a lower bound on the mixed covering array number. Specifically, we conclude that $CAN(G, \prod_{i=1}^{r} g_i^{s_i}) > n$ whenever $\omega(G) > \omega(QI(n, \prod_{i=1}^{r} g_i))$ or $\chi(G) > \chi(QI(n, \prod_{i=1}^{r} g_i))$.

Corollary 3.2.7. *Let n be a positive integer and let G be a weighted graph with distinct weights $g_1, g_2, \ldots, g_r$, repeated $s_1, s_2, \ldots, s_r$ times, respectively. Then, if there exists a $CA(n, G, \prod_{i=1}^{r} g_i^{s_i})$ then*

$$\chi(G) \leq \chi(QI(n, \prod_{i=1}^{r} g_i)) \text{ and } \omega(G) \leq \omega(QI(n, \prod_{i=1}^{r} g_i)).$$

Proof. By Theorem 3.2.5, there exists a weight-restricted homomorphism from G to $QI(n, \prod_{i=1}^{r} g_i)$. A weight-restricted homomorphism is a homomorphism, therefore we get that $\chi(G) \leq \chi(QI(n, \prod_{i=1}^{r} g_i))$ and $\omega(G) \leq \omega(QI(n, \prod_{i=1}^{r} g_i))$ by the properties of homomorphisms. □

3.3 Optimal Mixed Covering Arrays on Graphs

In this section, we give constructions of mixed covering arrays on graphs, several of which are optimal.

A length-n vector with alphabet size g is *balanced* if each symbol occurs $\lfloor n/g \rfloor$ or $\lceil n/g \rceil$ times. A *balanced covering array* is a covering array in which every row is balanced. Balanced covering arrays are of special interest, as Meagher [29] conjectures that there always exists an optimal covering array that is balanced and proves that this is true for $g = 2$.

Lemma 3.3.1. *For a balanced length-n vector x on $\mathbb{Z}_g$ and for any $h \leq \frac{n}{g}$ there exists a balanced length-n vector y on $\mathbb{Z}_h$ such that x and y are qualitatively independent.*

Proof. Let $n = gq + r$, with $0 \leq r < g$. Since x is balanced, we assume w.l.o.g. that the first $g - r$ symbols each occur q times in x and the last r symbols occur $q + 1$ times. Further, we assume w.l.o.g. that the letters occur in their natural order. That is,

$$x(i) = \begin{cases} \left\lfloor \frac{i}{q} \right\rfloor & \text{for } 0 \leq i < (g-r)q, \\ \left\lfloor \frac{i+(g-r)}{q+1} \right\rfloor & \text{for } (g-r)q \leq i < n. \end{cases}$$

Let $y \in \mathbb{Z}_g^n$ be the vector such that $y(i) = i \mod h$, for all $i \in [0, n-1]$. Clearly, y is balanced. We will now show that x and y are qualitatively independent. For $s \in \{0, 1, \ldots, g-1\}$, there exists some $j \in [0, n-1]$ such that $x(j) = x(j+1) = \ldots = x(j+q-1) = s$. Since $h \leq \frac{n}{g}$, we have $q \geq h$. Thus, all symbols in Z_h occur among $y(j), y(j+1), \ldots, y(j+q-1)$. Therefore, $\{(x(i), y(i)) : j \leq i \leq j+q-1\} = \{(s,t) : t \in \mathbb{Z}_h\}$. Since we can do this for all $s \in \{0, 1, \ldots, g-1\}$, all pairs from $\mathbb{Z}_g \times \mathbb{Z}_h$ are covered by some common coordinate of x and y. □

Lemma 3.3.2. *Let $x_1 \in \mathbb{Z}_{g_1}^n$ and $x_2 \in \mathbb{Z}_{g_2}^n$ be balanced vectors, then for any h such that $hg_1 \leq n$ and $hg_2 \leq n$, there exists a balanced vector $y \in \mathbb{Z}_h^n$ such that x_1 and y are qualitatively independent and x_2 and y are qualitatively independent.*

Proof. Define a bipartite multi-graph H as follows: H has g_1 vertices in the first part, $P \subseteq V(H)$ and g_2 vertices in the second part $Q \subseteq V(H)$. Let $P_a = \{i : x_1(i) = a\}$, for $a = 1, \ldots, g_1$, be the vertices of P, while $Q_b = \{i : x_2(i) = b\}$, for $b = 1, \ldots, g_2$, be the vertices of Q. We also have that $|P_a| \geq \left\lfloor \frac{n}{g_1} \right\rfloor$ and $|Q_b| \geq \left\lfloor \frac{n}{g_2} \right\rfloor$ since the vectors are balanced. For each $i = 1, \ldots, n$ there exists exactly one $P_a \in P$ with $i \in P_a$ and exactly one $Q_b \in Q$ with $i \in Q_b$. For each such i, add an edge between vertices corresponding to P_a and Q_b and label it i. Since $|P_a| \geq \left\lfloor \frac{n}{g_1} \right\rfloor$ and $|Q_b| \geq \left\lfloor \frac{n}{g_2} \right\rfloor$, the minimum degree of a vertex in H is $\delta = \min\{\left\lfloor \frac{n}{g_1} \right\rfloor, \left\lfloor \frac{n}{g_2} \right\rfloor\}$. By a dual of Konig's theorem, there are $\min\{\left\lfloor \frac{n}{g_1} \right\rfloor, \left\lfloor \frac{n}{g_2} \right\rfloor\}$ classes of edge-disjoint vertex covers. In particular, since $h \leq \min\{\left\lfloor \frac{n}{g_1} \right\rfloor, \left\lfloor \frac{n}{g_2} \right\rfloor\}$, we have h edge-disjoint vertex covers. These h edge-disjoint vertex covers form a partition R of $[1, n]$, which we use to build a balanced vector $y \in \mathbb{Z}_h^n$. Each edge-disjoint vertex cover corresponds to a symbol in $\mathbb{Z}_h$ and each edge corresponds to an index from [1,n]. For each edge i in an edge-disjoint vertex cover associated with $a \in \mathbb{Z}_h$, define $y(i) = a$. Fill the remaining positions in the vector y in such a way that y remains balanced. Next, we need to show that the partitions P and R are qualitatively independent. For any $c \in \mathbb{Z}_h$, the class R_c corresponds to an edge-disjoint vertex cover of H, this means that for

any $a \in \mathbb{Z}_{g_1}$, there exists an edge i in the edge-disjoint vertex cover incident to P_a. By the definition of R, we have $i \in R_c$. Since the edge labelled i is incident with the vertex corresponding to P_a, we also know that $i \in P_a$. This means that for any $a \in \mathbb{Z}_{g_1}, c \in \mathbb{Z}_h$, there exists an $i \in [1, n]$ such that $i \in P_a$ and $i \in R_c$, or in other words, $x_1(i) = a$ and $y(i) = c$. So, x_1 and y are qualitatively independent. Similarly, we can show that x_2 and y are qualitatively independent. □

3.3.1 Basic Graph Operations

Let G be a weighted multigraph with k vertices. Label the vertices $v_0, v_1, \ldots, v_{k-1}$ and for each vertex v_i the associated weight is denoted by $w_G(v_i)$. Let the *product weight* of G, denoted $PW(G)$, be $PW(G) = \max\{w_G(v_i)\, w_G(v_j) : \{v_i, v_j\} \in E(G)\}$. All the definitions for graphs extend naturally to multigraphs. Note that $CAN(G, \prod_{i=1}^{k} w_G(v_i)) \geq PW(G)$.

We define three graph operations: *one-vertex edge hooking*, *edge duplication* and *weight-restricted edge subdivision*. A *one-vertex edge hooking* in a multigraph G is the operation that inserts a new edge where one end is in $V(G)$ and the other is a new vertex. An *edge duplication* involves the creation of an edge that is parallel to an existing edge in G, that is, we are creating a multigraph with an existing edge appearing twice. An *edge subdivision* is the operation that replaces an edge by a path with two edges. A *weight-restricted edge subdivision* is an edge subdivision such that if v is the new vertex in G adjacent to vertices s and t then $w_G(v)w_G(s) \leq PW(G)$ and $w_G(v)w_G(t) \leq PW(G)$.

Proposition 3.3.3. *(One-vertex Edge Hooking) Let n be a positive integer and G be a weighted multigraph with k vertices. Let G' be the weighted multigraph obtained from G by a one-vertex edge hooking of a new vertex v with a new edge $\{u, v\}$, and $w(v)$ such that $w(u)w(v) \leq n$. Then, there exists a balanced $CA(n, G, \prod_{i=1}^{k} g_i)$ if and only if there exists a balanced $CA(n, G', w(v) \prod_{i=1}^{k} g_i)$.*

Proof. The direct implication is the only non-trivial one. Let C^G be a balanced $CA(n, G, \prod_{i=1}^{k} g_i)$. The balanced covering array C^G will be used to construct a $C^{G'}$, a balanced $CA(n, G', w(v) \prod_{i=1}^{k} g_i)$. Using Lemma 3.3.1, we know that we can build

a balanced length-n vector, call it x, corresponding to vertex v such that x is qualitatively independent with the length-n vector y corresponding to vertex u in G. The array $C^{G'}$ is built by appending column x to C^G. Since the only new edge is $\{u, v\}$, and x and y are qualitatively independent, $C^{G'}$ is a balanced mixed covering array on G'. □

Proposition 3.3.4. *(Edge Duplication) Let G be a weighted graph with k vertices and let G' be the weighted multigraph obtained from G by duplicating one of its edges. Then, there exists a balanced $CA(n, G, \prod_{i=1}^{k} g_i)$ if and only if there exists a balanced $CA(n, G', \prod_{i=1}^{k} g_i)$.*

Proof. The extension of the definition of mixed covering arrays on graphs to mixed covering arrays on multigraphs does not affect the size of a mixed covering array obtained by duplicating edges. The balanced mixed covering array on G is the same balanced mixed covering array on G'. □

Proposition 3.3.5. *(Weight-Restricted Edge Subdivision) Let G be a weighted multigraph with k vertices. Let G' be the weighted multigraph obtained from G by a weight-restricted edge subdivision creating the new vertex v that is adjacent to vertices s and t in G, with $w(v)$ such that $\max\{w(v)w(s), w(v)w(t)\} \leq n$. If there exists a balanced $CA(n, G, \prod_{i=1}^{k} g_i)$ then there exists a balanced $CA(n, G', w(v) \prod_{i=1}^{k} g_i)$.*

Proof. Let C^G be a balanced $CA(n, G, \prod_{i=1}^{k} g_i)$. The balanced covering array C^G is used to construct $C^{G'}$, a balanced $CA(n, G', w(v) \prod_{i=1}^{k} g_i)$. By Lemma 3.3.2, we know that we can build a balanced length-n vector, call it x, corresponding to vertex v such that x is qualitatively independent with the length-n vector y corresponding to vertex s in G and with the length-n vector z corresponding to vertex t in G. $C^{G'}$ is built by appending column x to C^G. Since $\{v, s\}$ and $\{v, t\}$ are the only new edges and x and y are qualitatively independent and x and z are qualitatively independent, $C^{G'}$ is a balanced mixed covering array on G'. □

The next theorem can be used to derive an algorithm that reverses the basic operations on a graph, obtaining a simpler graph to be used in a covering array construction.

Theorem 3.3.6. *Let G be a weighted multigraph and G' a weighted multigraph obtained from G via a sequence of weight-restricted edge subdivisions, one-vertex edge hooking and edge duplication. Let $v_{k+1}, v_{k+2}, \ldots, v_\ell$ be the vertices in $V(G')\backslash V(G)$ with weights $g_{k+1}, g_{k+2}, \ldots, g_\ell$, respectively. If there exists a balanced $CA(n, G, \prod_{i=1}^{k} g_i)$ then there exists a balanced $CA(n, G', \prod_{i=1}^{\ell} g_i)$.*

Proof. The result is derived by iterating the three previous propositions. □

3.3.2 n-chromatic graphs for $n = 2, 3, 4, 5$.

Combining Corollary 3.2.2 with Theorem 2.2.5, we get the following theorem.

Theorem 3.3.7. *Let G be a weighted graph with k vertices with weights $g_1 \leq g_2 \leq \ldots \leq g_k$. If one of the following holds:*
1) $\chi(G) = 2, 3$,
2) $\chi(G) = 4$ and $\prod_{i=k-3}^{k} g_i \notin \{2^4, 6^4\}$, or
3) $\chi(G) = 5$ and $\prod_{i=k-4}^{k} g_i \notin \{2^5, 3^5, 23^4\}$ and $g_{k-1} \notin \{4, 6, 10\}$,

then $CAN(G, \prod_{i=1}^{k} g_i) \leq g_{k-1}g_k$.

Proof. Let G be a weighted graph with k vertices. Then by Corollary 3.2.2, we have $CAN(G, \prod_{i=1}^{k} g_i) \leq CAN(K_{\chi(G)}, \prod_{l=k-\chi(G)+1}^{k} g_l)$. Moreover, by Theorem 2.2.5 for $p = 2, 3$; $p = 4$ and $\prod_{i=k-3}^{k} g_i \notin \{2^4, 6^4\}$; or $p = 5$ and $\prod_{i=k-4}^{k} g_i \notin \{2^5, 3^5, 23^4\}$ and $g_{k-1} \notin \{4, 6, 10\}$, we have $CAN(K_p, \prod_{l=k-p+1}^{k} g_l) = g_{k-1}g_k$. □

3.3.3 Trees

Theorem 3.3.8. *Let T be a weighted tree with k vertices with weights $g_1, g_2, \ldots, g_k$. Then, $CAN(T, \prod_{i=1}^{k} g_i) = PW(T)$.*

Proof. We can build T by starting with an edge $\{v_i, v_j\}$ such that $PW(T) = g_i \times g_j$, and by applying successive one-vertex edge hooking in the proper order so as to obtain T. Since, the covering array number on the first edge is $PW(T)$, and it continues to be the same at each new edge, we have $CAN(T, \prod_{i=1}^{k} g_i) = PW(T)$. □

3.3.4 Cycles

In this section, we solve the problem for cycles. Let C be a cycle with two largest weights g_{k-1} and g_k. Note that if $PW(C) = g_{k-1}g_k$, since cycles are 3-colourable, Theorem 3.3.7 solves the problem via the colouring construction (graph homomorphism to K_3). So, the non-trivial case we solve in this section is when $PW(C) < g_{k-1}g_k$.

We give two proofs. The first one uses the alternate construction and the second one uses graph operations introduced in section 3.3.1.

Theorem 3.3.9. *Let C be a weighted cycle of length k. There exists a balanced mixed covering array $CA(n, C, \prod_{i=1}^{k} w_C(v_i))$ with $n = PW(C)$. Moreover, this covering array is optimal.*

Proof. 1: Alternate construction
Let C be a weighted cycle with k vertices. Label the vertices $v_1, v_2, \ldots, v_k$ and for each vertex v_i the associated weight is denoted by $w_C(v_i)$. Set $n = PW(C)$. We will give a construction for A, a $CA(n, C, \prod_{i=1}^{k} w_C(v_i))$. Consider the path v_1 to v_{k-1}. Pick node v_1 to be the root. Let the column in C corresponding to v_1 to be a balanced vector of length n with weight $w_C(v_1)$. For $2 \leq i \leq k-1$, v_i is the adjacent vertex to v_{i-1}; then by Lemma 3.3.1, there exists a balanced length-n vector with weight $w_C(v_i)$ which is qualitatively independent with v_{i-1} since $w_C(v_i)w_C(v_{i-1}) \leq n$. This vector will be the column corresponding to v_i in A. Finally, from Lemma 3.3.2, it is possible to construct a vector which is qualitatively independent with both vectors v_1 and v_{k-1}. We use this vector to be the column corresponding to v_k in A. □

Proof. 2: Construction using graph operations
Let $\{v_{k-1}, v_k\}$ be an edge in C with $w(v_{k-1})w(v_k) = PW(C)$. Note that in this proof we do not assume that $w(v_{k-1})$ and $w(v_k)$ are the two largest weights. We build a balanced $CA(n, C, \prod_{i=1}^{k} w_C(v_i))$ with $n = PW(C)$ using the basic graph operations from Section 3.3.1. Let G_1 be the weighted graph with the single weighted edge $\{v_{k-1}, v_k\}$. From Lemma 3.3.1, there exists a balanced $CA(n, G_1, w(v_{k-1})w(v_k))$. Let G_2 be the graph obtained from G_1 by edge duplication. From Proposition 3.3.4 there exists a balanced $CA(n, G_2, \prod_{i=k-1}^{k} w_C(v_i))$. For $3 \leq j \leq k$, let G_j be the graph

obtained from G_{j-1} by applying a weight-restricted edge subdivision with vertex v_{j-2}, a minimum-weight vertex among the remaining vertices, and inserting v_{j-2} in the appropriate position based on the original cycle configuration. Note that $G_k = C$. From Proposition 3.3.5, for all $j = 3, \ldots, k$, if $PW(G_j) \leq PW(C)$, then there exists a balanced $CA(PW(C), G_j, w_C(v_{k-1})w_C(v_k)\prod_{i=1}^{j-2} w_C(v_i))$. In particular, there exists a balanced $CA(PW(C), C, \prod_{i=1}^{k} w_C(v_i))$. All we need to prove is that $PW(G_j) \leq PW(C)$ for all $j \in \{3, \ldots, k\}$.

To prove this we assume by contradiction that for some $j \in \{3, \ldots, k\}$, $PW(G_j) > PW(C)$. Thus, at some intermediate step, we have an edge $\{u, v\}$ in the cycle we are building with $w(u) = g$ and the $w(v) = h$ where $gh > PW(C)$. We know from the definition of $PW(C)$ that the vertices u, v are not adjacent in the original cycle. Let $up_1p_2 \ldots p_iv$ be the path in the original cycle that connects u to v and does not contain the edge $\{v_{k-1}, v_k\}$. Then, the vertex p_1 is not in the cycle G_j we are constructing. Since $gw(p_1) \leq PW(C) < gh$, we have $w(p_1) < h$. This is a contradiction since we have inserted the vertices with the smallest weight first.

This covering array is optimal since $PW(C)$ is a lower bound for the size of a covering array on C. □

3.3.5 Bipartite Graphs

In this section, we give a construction for optimal mixed covering arrays on bipartite graphs. A *bipartite* graph is one whose vertex set can be partitioned into two sets A and B, so that each edge has one end in A and one end in B. Note that if the two largest alphabets in the bipartite graph, denoted by g_k and g_{k-1}, are connected by an edge, then we can build an optimal mixed covering array of size g_kg_{k-1} via a weight-restricted homomorphism to K_2. The non-trivial case solved by the next theorem is when $PW(G) < g_kg_{k-1}$.

Theorem 3.3.10. *Let G be a weighted bipartite graph with k vertices. Then, there exists a balanced $CA(n, G, \prod_{i=1}^{k} w_G(v_i))$ with $n = PW(G)$. Moreover, this covering array is optimal.*

Proof. Let G be a bipartite graph with bipartitions A and B. Denote the vertices of

A by $a_1, a_2, \ldots, a_{|A|}$ and their corresponding weights by $w_G(a_i)$ where $i \in \{1, \ldots, |A|\}$. Similarly, denote the vertices of B by $b_1, b_2, \ldots, b_{|B|}$ and their corresponding weights by $w_G(b_i)$ where $i \in \{1, \ldots, |B|\}$. Set $n = PW(G)$. We give a construction for a $CA(n, G, \prod_{i=1}^{k} w_G(v_i))$ and call this covering array C.

For all vertices $a_i \in A$, where $i \in \{1, \ldots, |A|\}$, let x_{a_i} be the column in C corresponding to a_i. Set the j^{th} entry of x_{a_i} to be equivalent to $j \bmod w_G(a_i)$. For a vertex $b_i \in B$, where $i \in \{1, \ldots, |B|\}$, let x_{b_i} be the column in C corresponding to b_i. Set the j^{th} entry of x_{b_i} to be

$$x_{b_i}(j) = \begin{cases} \left\lfloor \frac{j}{\left\lfloor \frac{n}{w_G(b_i)} \right\rfloor} \right\rfloor & \text{if } 0 \leq j < w_G(b_i) \left\lfloor \frac{n}{w_G(b_i)} \right\rfloor, \\ j \mod w_G(b_i) & \text{otherwise.} \end{cases}$$

We will now show that for any two adjacent vertices a and b that the columns x_a and x_b in C corresponding to a and b are qualitatively independent. Since G is bipartite, we can assume that $a \in A$ and $b \in B$. Further, $w_G(a)w_G(b) \leq n$. Thus $\left\lfloor \frac{n}{w_G(b)} \right\rfloor \geq w_G(a)$. For $s \in \{0, 1, \ldots, w_G(b) - 1\}$, s occurs in positions $j, \ldots, j + \left\lfloor \frac{n}{w_G(b)} \right\rfloor - 1$ of x_b for some $j \in [0, w_G(b_i) \left\lfloor \frac{n}{w_G(b_i)} \right\rfloor]$. The entries of x_a in positions $\ell \in \{j, \ldots, j + \left\lfloor \frac{n}{w_G(b)} \right\rfloor - 1\}$ are $\ell \mod w_G(a)$. Since $\left\lfloor \frac{n}{w_G(b)} \right\rfloor \geq w_G(a)$, all symbols in the alphabet corresponding to a will occur in these $\left\lfloor \frac{n}{w_G(b)} \right\rfloor$ positions. This means we cover all possible pairs with s. Since we can do this for all $s \in \{0, 1, \ldots, w_G(b) - 1\}$, all pairs are covered between x_a and x_b.

This covering array is optimal, since $PW(G)$ is a lower bound on the size of a covering array on G. □

3.3.6 Wheels

In this section, we determine the covering array numbers for wheels with uniform alphabet sizes and give an upper bound in the mixed case. Let $t \geq 3$. A $t-$wheel, denoted by W_t, has t vertices forming a cycle in addition to one more vertex in the centre that is adjacent to every other vertex in the cycle. A $t-$wheel is odd if t is odd and even if t is even. It is easy to see that the $\chi(W_t) = 3$, for t even and $\chi(W_t) = 4$, for t odd.

We first look at the most difficult special cases of $g = 6$ and $g = 2$. Meagher and Stevens [31] found a $CA(36, W_5, 6^6)$ by computer, given next.

Example 3.3.11. *The following is the transpose of* $CA(36, W_5, 6^6)$

$$
\begin{array}{cccccccccccccccccccccccccccccccccccc}
0&0&0&0&0&0&1&1&1&1&1&1&2&2&2&2&2&2&3&3&3&3&3&3&4&4&4&4&4&4&5&5&5&5&5&5\\
0&1&2&3&4&5&0&1&2&3&4&5&0&1&2&3&4&5&0&1&2&3&4&5&0&1&2&3&4&5&0&1&2&3&4&5\\
0&5&2&1&3&4&4&2&5&3&1&0&1&4&3&2&0&5&3&0&1&4&5&2&2&1&0&5&4&3&5&3&4&0&2&1\\
0&2&1&4&3&5&2&5&1&4&0&3&2&0&5&3&1&4&1&2&5&3&0&4&0&1&5&3&4&2&5&0&1&4&2&3\\
0&2&4&1&3&5&4&3&0&2&1&5&1&4&0&2&5&3&2&5&1&0&3&4&2&3&4&1&5&0&2&5&1&0&3&4\\
0&4&1&5&2&3&2&5&3&0&1&4&4&3&5&2&0&1&5&1&2&4&3&0&1&0&4&3&5&2&3&2&0&1&4&5
\end{array}
$$

Lemma 3.3.12. *Let $t \geq 5$, t odd. Then, there exists a graph homomorphism $\phi : W_t \to W_5$.*

Proof. Label the middle vertex of W_5, w_0 and the vertices along the cycle $w_1, w_2, \ldots, w_5$. Also, label the middle vertex of W_t, v_0 and the vertices along the cycle $v_1, v_2, \ldots, v_t$. Define ϕ as follows:

$$\phi(v_i) = \begin{cases} w_i & \text{if } 0 \leq i \leq 5, \\ w_1 & \text{if } i \text{ even and } 6 \leq i \leq t, \\ w_2 & \text{if } i \text{ odd and } 6 \leq i \leq t. \end{cases}$$

It is easy to verify that ϕ is indeed a graph homomorphism. □

The next theorem solves the case of $g = 6$.

Theorem 3.3.13.

$$CAN(W_t, 6^{t+1}) = \begin{cases} 37 & \text{if } t = 3, \\ 36 & \text{if } t \geq 4. \end{cases}$$

Proof. For $t = 3$, $W_3 = K_4$, so $CAN(W_3, 6^4) = CAN(4, 6) = 37$ (see [33]). For $t \geq 4$ and even, $\chi(W_t) = 3$, so there exists a graph homomorphism to K_3. Then, $CAN(W_t, 6^{t+1}) \leq CAN(K_3, 6^3) = 36$. By Lemma 3.3.12 there exists a $\phi : W_t \to W_5$ graph homomorphism for $t \geq 5$ and odd and by Example 3.3.11, $CAN(W_5, 6^6) = 36$. Thus, for $t \geq 5$ and odd, $CAN(W_t, 6^{t+1}) \leq CAN(W_5, 6^6) = 36$. □

Next, we analyze the case for $g = 2$.

Theorem 3.3.14.

$$CAN(W_t, 2^{t+1}) = \begin{cases} 5 & \text{if } t \geq 3,\ t \text{ odd} \\ 4 & \text{if } t \geq 4,\ t \text{ even.} \end{cases}$$

Proof. We have that $QI(4,2)$ is the even wheel, W_2, which has $\chi(QI(4,2)) = 3$. We can thus deduce using Theorem 2.3.6 and Corollary 2.3.8 that there does not exist a $CA(4, W_t, 2^{t+1})$ for $t \geq 3$ and t odd since $\chi(W_t) = 4$ in this case and we can not have a graph homomorphism from W_t to $QI(4,2)$. Hence, $CAN(W_t, 2^{t+1}) > 4$ for $t \geq 3$ and t odd. For $t = 3$, $W_3 = K_4$, so $CAN(W_3, 2^4) = CAN(4,2) = 5$. There exists a graph homomorphism $\phi : W_t \to W_3$ for $t \geq 3$ and odd. Hence, $4 < CAN(W_t, 2^{t+1}) \leq CAN(W_3, 2^4) = 5$ for $t \geq 3$ and t odd. For $t \geq 4$ and t even, there exists a graph homomorphism $\phi : W_t \to K_3$. Therefore, $CAN(W_t, 2^{t+1}) \leq CAN(K_3, 2^3) = 4$ for $t \geq 4$ and t even. □

Theorem 3.3.15. *Let W_t be a weighted wheel with $t+1$ vertices with weights $1 < g_1 \leq g_2 \leq \cdots \leq g_{t+1}$. Then, $CAN(W_t, \prod_{i=1}^{t+1} g_i) \leq g_t g_{t+1}$ except for $\prod_{i=1}^{t+1} g_i \notin \{6^4\} \cup \{2^{t+1} : t \geq 3, t \text{ odd}\}$.*

Proof. We will look at five different cases.
Case 1: Let us look at the largest four alphabets, $g_{t+1}g_t g_{t-1}g_{t-2} \notin \{2^4, 6^4\}$. We have that if t is even then the wheel is $3-$chromatic and if t is odd then it is $4-$chromatic. By parts 1 and 2 of Theorem 3.3.7, we get that the $CAN(W_t, \prod_{i=1}^{t+1} g_i) \leq g_t g_{t+1}$.
Case 2: Let us look at the case when $\prod_{i=1}^{t+1} g_i \in \{6^{t+1} : t \geq 4\}$. By Theorem 3.3.13, we have that $CAN(W_t, 6^{t+1}) = 36$. Thus, $CAN(W_t, \prod_{i=1}^{t+1} g_i) \leq g_t g_{t+1}$.
Case: 3: Let us look at the case when $\prod_{i=1}^{t+1} g_i \in \{2^{t+1} : t \geq 4, t \text{ even}\}$. By Theorem 3.3.14, we have that $CAN(W_t, 2^{t+1}) = 4 \leq g_t g_{t+1}$.
Case 4: We will now discuss the first exception where $\prod_{i=1}^{t+1} g_i \in \{6^4\}$. The colouring construction of the $3-$wheel leads to a homomorphism to K_4. It is a well-known fact that $CAN(K_4, 6^4) = 37$. The covering array number can not be 36 because there exists no orthogonal latin squares of order 6. Thus, $CAN(W_3, 6^4) = 37$.
Case 5: Finally, we will discuss that second exception where $\prod_{i=1}^{t+1} g_i \in \{2^{t+1} : t \geq 3, t$

odd}. This is a direct result of Theorem 3.3.14. Thus, $CAN(W_t, 2^{t+1}) = 5$ for $t \geq 3$ and t odd. □

The next corollary summarizes the result for the wheel.

Corollary 3.3.16.

$$CAN(W_t, g^{t+1}) = \begin{cases} g^2 + 1, & \textit{if } (g = 6,\ t = 3) \textit{ or } (g = 2,\ t \textit{ odd}), \\ g^2, & \textit{otherwise} \end{cases}$$

3.3.7 Cubic Graphs

In this section, we give an upper bound on the covering array number for mixed cubic graphs. Cubic graphs are connected graphs having the property that each node has degree exactly three. We give a well-known colouring result for cubic graphs and then the corollary that results immediately from that theorem combined with Theorem 3.3.7.

Theorem 3.3.17. *[23] Every cubic graph is vertex colourable with three colors, except for the tetrahedron that requires four colors.*

Corollary 3.3.18. *Let G be a weighted cubic graph with k vertices with weights $1 < g_1 \leq g_2 \leq \cdots \leq g_k$. Then, $CAN(G, \prod_{i=1}^{k} g_i) \leq g_{k-1}g_k$ except for the tetrahedron when $\prod_{i=1}^{4} g_i \in \{2^4\} \cup \{6^4\}$. Moreover, $CAN(G, 2^4) = 5$ and $CAN(G, 6^4) = 37$.*

Chapter 4

Tabu Search Methods for Covering Arrays

In this section, we describe the general tabu search framework, as well as the point and pair tabu algorithms that we implemented to find covering arrays. These are implementations of methods by Stardom [40] and Nurmela [35], respectively. We provide a detailed description of the algorithm and data structures, as well as their asymptotic running times.

4.1 General Tabu Algorithm

The *tabu search algorithm* (TS), as originally proposed by Glover [21], is a meta-heuristic method belonging to the class of local search techniques. Tabu search is an algorithm that tries to find a certain combinatorial structure by the use of a heuristic. A heuristic method performs a minor modification, which for the tabu search we call a *tabu move*, of a given solution in order to obtain a different solution. The actual modifications involves a search of the neighborhood space. A fitness function is used to evaluate a solution and the objective is to explore the search space to find as fit a solution as possible. Nurmela [35] and Stardom [40] have both successfully implemented different variations of this algorithm in order to find new upper bounds for covering arrays.

Next, we define some notation to properly give an overview of the tabu search algorithm as described in [28]. Let $\mathcal{X}$ be a specified finite set, usually called the *universe*. An element $X \in \mathcal{X}$ is said to be a *feasible solution* if certain constraints are satisfied. If X is any object in the search space, then the set of states which can be obtained by applying a particular move from a given set of moves to the state X is called the *neighborhood*, $N(X)$, of X. By repeating this process for a succession of objects, a search algorithm is capable of examining a large number of states in the search space. We consider a minimization problem. Therefore, a fittest solution is a feasible solution X for which the cost, denoted $C(X)$, is as small as possible.

The algorithm GenericTabuSearch gives the main steps for the general TS algorithm. The heuristic of the TS algorithm is to replace X with an element $Y \in N(X)\backslash\{X\}$ such that Y is feasible, and $C(Y)$ is minimum among all feasible elements in $N(X)\backslash\{X\}$. In this way, $C(X)$ decreases throughout the search whenever possible, but the TS allows us to escape from a local minimum by selecting a less fit neighbour. However, this introduces the risk of cycling, for instance by going back to the local minimum in the next step. To avoid cycling recent moves are declared "tabu". The tabu list is used to ensure that a move that is performed is not undone at the next iterations in order to avoid cycling. Let us define the function $change(Y, X)$ which specifies the changes that are made to a feasible solution X in order to obtain a feasible solution Y. Thus, after any move $X \rightarrow Y$, $change(Y, X)$ is designated as forbidden and added to the tabu list. Changes that are in the tabu list remain prohibited for some time, known as the *tabu lifetime*. Therefore, the heuristic will only choose elements that are not in the tabu list. The algorithm will typically run until either a minimum solution has been reached or a predetermined maximum number of iterations has been exceeded or we can not move anymore because the neighborhood is empty.

The TS algorithm has 2 input parameters, I and L. The parameter I is used to specify the maximum number of iterations that are performed in the algorithm, while L is the tabu lifetime. Optionally, a lower bound, lb, on $C(.)$ could be provided, for detecting optimality. In GenericTabuSearch, we chose to provide a lower bound for the cost function as a third stopping criteria.

Algorithm: GENERICTABUSEARCH(I, L, lb)

$iter \leftarrow 1$
Randomly select a solution $X \in \mathcal{X}$.
$X_{best} \leftarrow X$
if $(C(X) == lb)$
  return $(X, 0)$
while $(iter \leq I)\{$
  $N \leftarrow (N(X) \setminus \{X\}) \setminus \{F : change(X, F) \in TabuList[d], iter - L \leq d \leq iter - 1\}$
  for each $(Y \in N)$
    if (Y is infeasible)
      $N \leftarrow N \setminus \{Y\}$
  if $(N = \emptyset)$
    return $(X_{best}, iter)$
  find $Y \in N$ such that $C(Y)$ is minimum among elements in N
  $TabuList[iter] \leftarrow change(Y, X)$
  $X \leftarrow Y$
  if $(C(X) < C(X_{best}))\{$
    $X_{best} \leftarrow X$
    if $(C(X) == lb)$
      return $(X, iter)$
  $\}$
  $iter{+}{+}$
$\}$
return $(X_{best}, iter)$

4.2 Data structures and basic procedures

In this section, we discuss the basic data structures that are common to both our implementations of the tabu algorithms that we later call POT and PAT. A covering array is represented by a $k \times n$ array X storing the transpose of the actual covering array. We store the alphabet size for each row of X in a one-dimensional array, denoted alphabet[r], where $0 \leq r \leq k - 1$. So, $X[r][c] \in \{0, 1, \ldots, alphabet[r] - 1\}$. The cost of X is the number of uncovered pairs in X. When the cost becomes zero, we have a covering array, since all the pairs are covered. We utilize a 4-dimensional array, *coverage*, to effectively represent the pair coverage of our current array X. The

element $coverage[i][j][a][b]$ specifies the number of times the pair (a, b) is covered at rows (i, j) for some column in X.

For PAT, we also have a list data structure that keeps track of all the pairs of the covering array that are still uncovered. The element (i, j, a, b) in this list indicates that the pair (a, b) at rows (i, j) is not covered in any column of X. This list is implemented as an array $UncoveredList$ of size $\binom{k}{2} \times g^2$ that stores elements of the form (i, j, a, b) in positions $0, 1, \ldots, u-1$, where u is the current number of uncovered pairs. Along with the uncovered list data structure, we also have a reverse index structure. The entry $indexOnUncoveredList[i][j][a][b]$ gives the position in $UncoveredList$ where (i, j, a, b) can be found or -1 if not uncovered. This data structure allows for addition and removal of an uncovered pair to be done in constant time.

In POT and PAT algorithms, $change(X, Y)$ corresponds to the unique position (r, c) such that $X[r][c] \neq Y[r][c]$. For the TS algorithm, we need to consult the tabu list at each step to ensure a change is not tabu. In order to efficiently use it, we have 2 data structures to represent the tabu list. The first is a linked list, $tabuList$, of size L, storing the pairs (r, c) of forbidden positions in the array. At each iteration, one more is added to the end of $tabuList$, and one is deleted from the beginning of $tabuList$ provided it has been in the list for L iterations. We also have a 2-dimensional array, $TABU[r][c]$, which is a $k \times n$ array. The element $TABU[r][c] = 1$, if (r, c) is in the $tabuList$, and 0 otherwise. This data structure allows checking whether a move is tabu or not to be done in constant time.

Next, we describe auxiliary procedures that use the data described in order to implement basic operations used in PAT and POT. Procedure **EvalChange**(i, c, a) evaluates the change in cost of changing a single element of the array $X[i][c] \leftarrow a$, while **EvalChangePair**(i, j, c, a, b) evaluates the change in cost of changing the pair $X[i][c] \leftarrow a$ and $X[j][c] \leftarrow b$. Uncovering a pair in our array yields a plus one change in the cost while covering a new pair yields a minus one change. Both procedures have running time $O(k)$.

Procedure 1: EvalChange(i, c, a)

global: $k, X, coverage$

```
effect = 0
for (int z = 0, z < k, z++)
    if (z! = i)
      if (coverage[z][i][X[z][c]][X[i][c]] == 1) ** Uncover a pair **
        effect++
      if (coverage[z][i][X[z][c]][a] == 0) ** Cover a new pair **
        effect- -
return effect
```

Procedure 2: EvalChangePair(i, j, c, a, b)

global: $k, X, coverage$

```
effect = 0
for (int z = 0, z < k, z++)
    if (z! = i) and (z! = j)
      if (coverage[z][i][X[z][c]][X[i][c]] == 1) ** Uncover a pair in row i **
        effect++
      if (coverage[z][j][X[z][c]][X[j][c]] == 1) ** Uncover a pair in row j **
        effect++
      if (coverage[z][i][X[z][c]][a] == 0) ** Cover a new pair using row i **
        effect- -
      if (coverage[z][j][X[z][c]][b] == 0) ** Cover a new pair using row j **
        effect- -
if (coverage[i][j][X[i][c]][X[j][c]] == 1) ** Uncover a pair in row i and j **
    effect++
if (coverage[i][j][a][b] == 0) ** Cover a new pair using rows i and j **
    effect- -
return effect
```

Procedure **MakeChange**(i, c, a) performs the change $X[i][c] \leftarrow a$, while procedure **MakeChangePair**(i, j, c, a, b) performs the changes $X[i][c] \leftarrow a$ and $X[j][c] \leftarrow b$. They also update the corresponding data structures, such as $coverage$, $UncoveredList$ and $indexOnUncoveredList$. Updates on the last two structures are done via calls to procedures **addUncovered** and **removeUncovered**; each of these procedures run in constant time. **MakeChange**(i, c, a) and **MakeChangePair**(i, j, c, a, b) run in $O(k)$. It is easy to check this, since each operation on their main loop runs in constant time.

Procedure 3: MakeChange(i, c, a)
global: $k, X, coverage$

```
for (int z = 0, z < k, z++){
    if (z! = i){
        if (coverage[z][i][X[z][c]][X[i][c]] == 1)
            addUncovered(z, i, X[z][c], X[i][c])
        coverage[z][i][X[z][c]][X[i][c]] - -
        coverage[z][i][X[z][c]][a] ++
        if (coverage[z][i][X[z][c]][a] == 1) {
            removeUncovered(z, i, X[z][c], a)
            }
    }
}
X[i][c] ← a
```

Procedure 4: MakeChangePair(i,j,c,a,b)
global: $k, X, coverage$

```
for( int z = 0, z < k, z++){
    if (z! = i) and (z! = j){
        if (coverage[z][i][X[z][c]][X[i][c]] == 1)
            addUncovered(z, i, X[z][c], X[i][c])
        if (coverage[z][j][X[z][c]][X[j][c]] == 1)
```

addUncovered$(z, j, X[z][c], X[j][c])$
$coverage[z][i][X[z][c]][X[i][c]]$ - -
$coverage[z][j][X[z][c]][X[j][c]]$ - -
$coverage[z][i][X[z][c]][a]$ ++
$coverage[z][j][X[z][c]][b]$ ++
if $(coverage[z][i][X[z][c]][a] == 1)$
removeUncovered$(z, i, X[z][c], a)$
if $(coverage[z][j][X[z][c]][b] == 1)$
removeUncovered$(z, j, X[z][c], b)$
}
}
if $(coverage[i][j][X[i][c]][X[j][c]] == 1)$
addUncovered$(i, j, X[i][c], X[j][c])$
$coverage[i][j][X[i][c]][X[j][c]]$ - -
$coverage[i][j][a][b]$ ++
if $(coverage[i][j][a][b] == 1)$
removeUncovered(i, j, a, b)
$X[i][c] \leftarrow a$
$X[j][c] \leftarrow b$

Procedure 5: addUncovered(x, y, a, b)
global: $UncoveredList, indexOnUncoveredList, u$

$UncoveredList[u] \leftarrow (x, y, a, b)$
$indexOnUncoveredList[x][y][a][b] \leftarrow u$
$u++$

Procedure 6: removeUncovered(x, y, a, b)
global: $UncoveredList, indexOnUncoveredList, u$

$p \leftarrow indexOnUncoveredList[x][y][a][b]$

$UncoveredList[p] \leftarrow UncoveredList[u-1]$
$(x', y', a', b') \leftarrow UncoveredList[p]$
$u--$
$indexOnUncoveredList[x][y][a][b] \leftarrow -1$
$indexOnUncoveredList[x'][y'][a'][b'] \leftarrow p$

The next two procedures perform the updates in the two tabu list data structures. They add the move to *tabuList* and remove the move that has been there more than L (tabu lifetime) updates. In the case that we make two simultaneous changes in the array, we must add two moves to the tabu list and therefore we randomly pick which move goes first to the list. Both **UpdateTabu**(r, c) and **UpdateTabuPair**(i, j, c) run in $O(1)$.

Procedure 7: UpdateTabu(r, c)
global: *tabuList*, TABU, L

Add (r, c) to the tail of *tabuList*
$TABU[r][c] \leftarrow 1$
if (Length(*tabuList*) $> L$)
  Remove (x, y) from the head of the *tabuList*
  $TABU[x][y] \leftarrow 0$

Procedure 8: UpdateTabuPair(i, j, c)
global: *tabuList*, $TABU, L$

Generate a random real value $a \in [0, 1]$
if ($a < 0.5$){
  Add (i, c) to the tail of *tabuList*
  Add (j, c) to the tail of *tabuList*
}
else {

Add (j, c) to the tail of $tabuList$
Add (i, c) to the tail of $tabuList$
}
if (Length($tabuList$) $> L$)
Remove (w, x) from the head of $tabuList$
if (Length($tabuList$) $> L$)
Remove (y,z) from the head of $tabuList$
$TABU[i][c] \leftarrow 1$
$TABU[j][c] \leftarrow 1$
$TABU[w][x] \leftarrow 0$
$TABU[y][z] \leftarrow 0$

4.3 Point Tabu Search (POT)

The point tabu search (POT) algorithm is a variation of the algorithm proposed by Stardom [40]. We call it *point* tabu search because the basic move is to switch a point in the array. Stardom's algorithm implements the tabu search algorithm as described in Section 4.1 with the neighborhood of a particular array being all possible arrays having one entry different. In effect, Stardom's algorithm exhaustively searches the neighborhood for the fittest solution, ensuring that the state chosen at each step is the best neighboring option possible. The algorithm terminates when it finds a covering array of cost zero or reaches the maximum number of iterations.

The POT algorithm (see pseudocode below) substitutes exhaustive neighborhood search by inspecting a random sample of size M of the neighborhood. In addition, our data structures and auxiliary procedures provide some optimization of Stardom's original implementation. POT algorithm's input parameters are the tabu lifetime, L, the maximum iteration, I, and the neighborhood size, M. At every iteration, we generate a list of M possible neighbors and pick the one with the lowest cost. If there are ties in the lowest cost, we break them randomly. The algorithm returns X_{best}, a covering array, or an array of smallest cost found.

Algorithm: POT(L, I, M)
globals: $TABU$
precondition: $M \leq (n \times k) - L$

$iter \leftarrow 1$
Randomly fill array X of size $k \times n$
$X_{best} \leftarrow X$
$Cost_{best} \leftarrow Cost(X)$
if ($Cost_{best} == 0$)
  return $(X, 0, 0)$
while ($iter \leq I$) {
    BEGIN: Get Point Tabu's Best in Neighborhood
    $B \leftarrow \emptyset$
    $m \leftarrow 1$
    effect$_{best} \leftarrow \infty$
    while ($m \leq M$) {
        Generate a random pair (r, c)
        if ($TABU[r][c] ==0$) {
            Generate a random value a from $\{0, \ldots, g-1\} \backslash \{X[r][c]\}$
            $m \leftarrow m + 1$
            effect = **EvalChange**(r, c, a)
            if (effect $<$ effect$_{best}$) {
                B $\leftarrow \{(r, c, a)\}$
                effect$_{best} \leftarrow$ effect
            }
            else if (effect == effect$_{best}$)
                $B \leftarrow B \cup \{(r, c, a)\}$
        }
    }
    END: Get Point Tabu's Best in Neighborhood
    $Cost \leftarrow Cost +$ effect$_{best}$

Pick (r, c, a) randomly from B
MakeChange(r, c, a)
UpdateTabu(r, c)
if $(Cost < Cost_{best})$
$X_{best} \leftarrow X$
$Cost_{best} \leftarrow Cost$
if $(Cost == 0)$
return $(X_{best}, 0, iter)$
$iter \leftarrow iter + 1$
}
return $(X_{best}, Cost_{best}, iter - 1)$

We now analyze the running time for a single POT Tabu move. Let us look inside the *while* loop where we generate a sample of M possible candidates. Generating a random pair is done in $O(1)$ and checking if the pair is tabu is also done in $O(1)$. Generating the value that will be placed at position (r, c) of our array has running time $O(1)$. We mentioned earlier that the evaluation of the change in cost using procedure **EvalChange**(r, c, a) has running time $O(k)$. Finally, checking if the effect of change in the cost is the best so far is done in $O(1)$. Therefore, the total running time for generating a neighbor inside the second while loop is $O(k)$. However, since we require that we generate exactly M neighbors, this process typically requires more than M iterations since we are keeping only the neighbors that are not tabu and trials are done at random. Indeed, since the probability of selecting a non-tabu pair (r, c) is $\frac{kn-L}{kn}$, the expected number of iterations until M non-tabu pairs are generated is $(\frac{kn-L}{kn})^{-1} \times M$. Since L is a constant, the expected number of iterations for the second while loop is $O(M)$. Therefore, the expected total running time for the first while loop is $O(Mk)$. After the while loop, the selection of best POT tabu move is done at random, which takes $O(1)$. Procedures **MakeChange**(r, c, a) and **UpdateTabu**(r, c) have running times $O(k)$ and O(1), respectively. Finally, updating the cost is done in $O(1)$. Overall, the average running time for a POT move is $O(Mk)$. In our implementation, we have chosen $M = cnk(g - 1)$, for some constant $c \leq 1$, a proportion c of the total number of neighbors. In this case, the average running time for a POT move is $O(nk^2g)$.

4.4 Pair Tabu Search (PAT)

The pair tabu search (PAT) algorithm is our implementation of the algorithm proposed by Nurmela [35]. We call it *pair* tabu search because the basic move aims at covering a new pair in array X. Nurmela's algorithm and PAT (see pseudocode below) attempt to find a covering array, $CA(n, k, g)$, by selecting an uncovered pair (i, j, a, b) at random from uncovered list at each iteration and trying to cover it. The neighborhood consists of all arrays that cover that particular pair, and is obtained by switching one or two array values. The algorithm gives priority to the neighbors that only require a single change such as either changing $X[i][c] = a$ or $X[j][d] = b$ for some columns c, d in the array. If a one element change is not possible, we consider the neighbors that require a change in two elements resulting in $X[i][c] = a$ and $X[j][c] = b$ for some column c. The change in cost corresponding to each move is calculated and the move leading to the minimum cost is selected, provided that the move is not tabu. In the case when we have ties, we break them randomly. The algorithm terminates when the cost is zero, i.e. all pairs are now covered, or when it reaches the maximum number of iterations, thus returning the best array it has found so far. The PAT algorithm's input parameters are the tabu lifetime, L, and the maximum iteration, I.

Algorithm: PAT(L, I)
global: $TABU$, $UncoveredList$, u

$iter \leftarrow 1$
Randomly fill array X of size $k \times n$
$X_{best} \leftarrow X$
$Cost_{best} \leftarrow Cost(X)$
if ($Cost_{best} == 0$)
  return $(X, 0, 0)$
while ($iter \leq I$){
    BEGIN: Get Pair Tabu's Best in Neighborhood
    $B \leftarrow \emptyset$

```
success ← false
while(! success) {
    effect_best ← ∞
    Select p randomly from {0, 1, ..., u − 1}
    (i, j, a, b) ← UncoveredList[p]
    for (int z = 0, z < n, z++){
        if (X[i][z] == a) and (TABU[j][z] == 0)
            effect ← EvalChange(j, z, b)
            if (effect < effect_best) {
                B ← {(j, z, b)}
                effect_best ← effect
            }
            else if (effect == effect_best)
                B ← B ∪ {(j, z, b)}
        if (X[j][z] == b) AND (TABU[i][z] == 0)
            effect ← EvalChange(i, z, a)
            if (effect < effect_best) {
                B ← {(i, z, a)}
                effect_best ← effect
            }
            else if (effect == effect_best)
                B ← B ∪ {(i, z, a)}
    }
    if (B ≠ ∅) {
        Cost ← Cost+effect_best
        Pick (r, c, a) randomly from B
        MakeChange(r, c, a)
        success ← true
        UpdateTabu(r, c)
    }
    else {
```

```
                effect_best ← ∞
                for (int z = 0,z < n, z++) {
                    if((TABU[i][z] == 0) AND (TABU[j][z] == 0))
                        effect ← EvalChangePair(i, j, z, a, b)
                        if (effect < effect_best){
                            B ← {(i, j, z, a, b)}
                            effect_best ← effect
                        }
                        else if (effect == effect_best)
                            B ← B ∪ {(i, j, z, a, b)}
                }
                if (B ≠ ∅) {
                    Cost ← Cost+effect_best
                    Pick (i, j, c, a, b) randomly from B
                    MakeChangePair(i, j, c, a, b)
                    success ← true
                    UpdateTabuPair(i, j, c)
                }
            }
        }
        END: Get Pair Tabu's Best in Neighborhood
        if (Cost < Cost_best) {
            X_best ← X
            Cost_best ← Cost
            if (Cost == 0)
                return (X_best, 0, iter)
        }
        iter ← iter + 1
}
return (X_best, Cost_best, iter − 1)
```

We now analyze the running time for a single PAT tabu move. Let us look at the piece of code inside the second while loop. The running time for checking whether the algorithm can perform the move with a single element change is $O(nk)$. This is because we call procedure **EvalChange** twice and it has a running time of $O(k)$ but we repeat this for the n columns in X. If a single move is possible then making the change is done in $O(k)$ due to calling procedure **MakeChange**. If a single element change is not possible, then we look at the possibility of making two changes and the running time for this process is $O(nk)$ since we call **EvalChangePair** for each of the n columns in X. Now, in order to perform the change, the running time is $O(k)$ since we are calling **MakeChangePair**. Therefore, one iteration of the second while loop takes $O(nk)$. We now analyze the expected number of iterations of the second while loop. Variable *success* is false at the end of the while loop if each one of the n columns of X has a tabu position in row i or j. An upper bound for the probability of failure is $\left(\frac{L}{nk}\right)^n$, so the expected number of iterations until a success is at most $\left(\frac{1}{1-\left(\frac{L}{nk}\right)^n}\right)$. Since L is a constant, the expected number of iterations in the second while loop is $O(1)$. So, the second while loop can be executed with an expected total time of $O(nk)$. After the second while loop, the cost updates take $O(1)$ steps. Therefore, the expected cost of a PAT tabu move is $O(nk)$.

Chapter 5

Experiments and Results for POT and PAT

In this chapter, we do an experimental comparison of POT and PAT tabu search algorithms described in the previous chapter. The objective of the experiments is not only to know which algorithm's performance prevails, but also to discover which set of input parameters optimizes the effectiveness of our code and how our implementation of POT and PAT compare with other algorithms from literature. Furthermore, we are interested in producing improvements on known upper bounds.

5.1 Experiment Description

In our experiments, both POT and PAT build fixed and mixed covering arrays. They are implemented in C and run on Solaris 5.9 (also known as Solaris 9) using a 900 MHz UltraSPARC III processors with 32 Gbytes of memory. The implementation that we report on here uses a linked list for the data structure that keeps track of uncovered pairs. This causes less efficient updates than the ones described in Section 4.2. However, as we will see in the conclusion this only affects the specific timing reports and not the relative comparison between POT and PAT.

Our experiment consists of three steps. The first step consists of a tune up of the input parameters for both POT and PAT as well as a study of their behaviour

individually and against one another. In order to do this, we design a test bed of 30 test cases upon which we do our analysis for different parameter variations of our algorithms. For the second step, we want to be able to compare the performance of our algorithms with previous algorithms implemented for finding covering arrays. In order to this, we require a test bed for which results have been reported in the past for different algorithms. Cohen [9], in her PhD thesis, has tables of data collected from a variety of algorithms which were implemented by various people including herself comparing the best upper bounds found for both fixed and mixed covering arrays. The algorithms implement a wide variety of greedy heuristics, hill climbing, simulated annealing and previous implementations of tabu search. We choose these tables to be used in order to compare POT and PAT with other approaches. We also compare our results with the best known bounds reported in tables by Colbourn [10]; with the permission of the author, we include these tables in the Appendix. In the final step, we pursue new upper bounds of two types. First, we check to see if our tabu implementations could improve on the current state-of-the-art bounds reported by Colbourn [10]. We look at Colbourn's tables and target those regions where the gaps are large. Those regions make ideal candidates for improvement on the bounds. Second, we attempt to find new upper bounds for small mixed covering arrays with parameters close to those of an orthogonal array.

5.2 Parameter tune up and algorithm behaviour study

In this section, we run several combinations of parameters for POT and PAT algorithms on the test bed given in Table 1. We select this test bed for the parameter tune up, in order to cover a good variation in g and k values. The first entry in each row of the table represents the largest orthogonal array known for the corresponding g, k values, while the last entry in each row is substantially larger than the orthogonal array size. This test bed is still too large for the large number of possible combinations of input parameters. So, we choose a subset of 5 tests to do a preliminary, more

extensive tune up. We call this subset of the test bed, marked in bold in Table 1, the *reduced test bed*. We chose one test for each of the smallest g's that had a "reasonable" size. The values $8 \leq g \leq 10$ where excluded from the reduced test bed as preliminary tests showed they were very hard. The input parameters for both POT and PAT are the tabu lifetime L and the maximum number of iterations I, as well as the neighborhood size M, for POT. To effectively determine the optimal choices for L and I, L is initially fixed while I is varied, and then the reverse experiment is performed. In the tune up experiments, we fix the upper bound for the size of the desired covering array and the algorithms are run for a fixed number of times, R, with different random seeds. The number of covering arrays found, the total time and the total number of iterations the algorithms spend are recorded and tabulated in the following sections. Once we decide on the best values for L and I, we run the algorithms on the entire test bed and vary the neighborhood size for POT. In Table 2, we give a legend for the headings of the tables used in the next sections. We mark in bold face the winning rows in each table to facilitate the analysis of the data. The criteria by which we decide that a particular row is a winner is described in each section. We also use the short hand form 50K, 100K, 200K and 500K to denote 50000, 100000, 200000 and 500000 iterations, respectively.

Table 1: Test bed for tune up with reduced test bed in bold

$g = 3$	$CA(9, 4, 3), CA(11, 5, 3), CA(14, 10, 3)$, $\mathbf{CA(15, 20, 3)}$, $CA(20, 43, 3)$
$g = 4$	$CA(16, 5, 4), CA(19, 6, 4), CA(24, 10, 4), \mathbf{CA(27, 14, 4)}$
$g = 5$	$CA(25, 6, 5), CA(29, 7, 5)$, $\mathbf{CA(33, 8, 5)}$, $CA(44, 16, 5)$
$g = 6$	$CA(36, 3, 6), CA(37, 4, 6), CA(55, 11, 6), \mathbf{CA(63, 16, 6)}$
$g = 7$	$CA(49, 8, 7), CA(61, 10, 7)$, $\mathbf{CA(85, 16, 7)}$, $CA(89, 18, 7)$
$g = 8$	$CA(64, 9, 8), CA(78, 11, 8), CA(111, 17, 8)$
$g = 9$	$CA(81, 10, 9), CA(105, 13, 9), CA(129, 16, 9)$
$g = 10$	$CA(100, 4, 10), CA(101, 6, 10), CA(136, 15, 10)$

Table 2: Legend for the tables that follow

Symbol	Definition
CA(n, k, g)	Covering array parameters
I	Maximum number of iterations allowed
sM	Neighborhood size setting for POT
L	Tabu lifetime
R	Total number of replications
No. CA	Number of runs in which a CA was found
% CA	Percentage of runs in which a CA was found
Avg. Iter/run	Total number of iterations used / R
Avg. Time/run(sec)	Total time used / R

5.2.1 Parameter tune up for POT

In this section, we tune up POT's input parameter values. The input parameter values are the tabu lifetime (L), the maximum number of iterations (I) and the neighborhood size (M). The maximum number of iterations is a stopping criteria to ensure that the tabu search does not look for a covering array forever, especially if one does not exist for that particular upper bound. When running the algorithm, we specify the size of the covering array we want to find and record the data for covering array success, time and iterations spent.

POT Maximum Number of Iterations Tune Up

Based on Table 3, we study the trade off between the maximum number of iterations and the number of replications on the reduced test bed. This was done to answer questions such as "Is it better to run the algorithm for many iterations with fewer replications?" or " Is it better to run the algorithm for fewer iterations with more replications?" So, we ran POT with I = 50K, 100K and 200K and with R = 40, 20 and 10, respectively. In this case the tabu lifetime was fixed to 5. In Table 3, we bold faced the entries that are the winners in our opinion. We decide that an entry is a winner if it yields the highest percentage of covering arrays found. If we have ties in the number of covering arrays found then we looked at the average time per

Table 3: POT Maximum number of iterations tune up with fixed L = 5

CA(n,k,g)	I	R	No. CA	% CA	Avg. Iter/run.	Avg. Time/run(sec)
CA(15,20,3)	50K	40	23	58	35207	114
CA(15,20,3)	100K	20	15	75	49908	161
CA(15,20,3)	**200K**	**10**	**9**	**90**	**55015**	**176**
CA(27,14,4)	50K	40	10	25	44367	180
CA(27,14,4)	**100K**	**20**	**10**	**50**	**69719**	**283**
CA(27,14,4)	200K	10	5	**50**	119804	487
CA(33,8,5)	50K	40	17	43	37043	91
CA(33,8,5)	100K	20	13	65	60745	148
CA(33,8,5)	**200K**	**10**	**8**	**80**	**74462**	**182**
CA(63,16,6)	50K	40	36	90	24207	430
CA(63,16,6)	100K	20	20	**100**	27756	493
CA(63,16,6)	**200K**	**10**	**10**	**100**	**24691**	**438**
CA(85,16,7)	50K	40	13	33	44786	1265
CA(85,16,7)	100K	20	15	75	62563	1789
CA(85,16,7)	**200K**	**10**	**10**	**100**	**77342**	**2211**

run and chose the entry with the smallest time. The data collected below suggests that running POT with the largest number of iterations and fewer replications yields a higher rate of success in terms of finding more covering arrays. We could have continued the analysis with other values by increasing I and reducing R further but we stopped since $I = 200$K and $R = 10$ was adequate in achieving a high percentage of success especially for the harder problems.

POT Tabu Lifetime Tune Up

In Table 4, we fix $I = 200$K and study the effect of varying the tabu lifetime on the reduced test bed. We vary $L = 3, 4, 5, 6, 7$, use $R = 10$ and collect the associated data for the reduced test bed. In the table, we mark in bold face the entries that are the winners according to the criteria described before: a winner is an entry with largest rate of success, using time for breaking ties. The winners are in order: $L = 4, 3, 3, 5, 7$. Since no value of L stands out as an overall winner, we choose the one that gives 100%

success for most problems in the test bed. In this case, we observe that $L = 4$ gives 100% success for all the problems, although not always in the best time. For instance, finding 100% success for $CA(33, 8, 5)$ with $L = 3$ is faster than with $L = 4$, but $L = 3$ does not give us 100% success for finding a $CA(85, 16, 7)$. Therefore, we use $L = 4$ for the rest of POT experiments.

POT Neighborhood Size Tune Up

After settling for a tabu lifetime value of L=4 and the maximum number of iterations of I=200K, we explore variations in the neighborhood size (M) using the whole test bed in Table 1. The neighborhood of a particular array X in POT consists of all the possible arrays having one entry different when compared with X. Since we are only looking at a random sample of these neighbors, the parameter M in POT was introduced to specify the size of this sample. We decided to compare the strategies of using fixed percentages of the maximum neighborhood size $\overline{M} = nk(g-1)$. We used $M = \overline{M},\ 0.75\overline{M}$ and $0.5\overline{M}$, which we denote as settings sM = 1,2,3, respectively, in the tables.

In Tables 5 and 6, we once again bold face the row with winning M parameter value, based on the highest percentage of success, with best time used for breaking ties. We have entries that were found very quickly and are denoted by time 0 since they only took milliseconds; in that case, we compare the average number of iterations per run and pick the smallest value. The algorithm was not very successful at finding covering arrays when $g = 8, 9, 10$ from our test bed and that is why no results are being reported.

We notice from the tables that there is no one M setting that works for all test cases, but rather it is dependent on the input parameters. When $g = 3$ and $k \leq 20$, having a larger neighborhood search space is more beneficial, while when $g = 3$ and $k = 43$ having $sM = 3$ is better. For $g = 4$, $sM = 1$ is the winner for 3 out of 4 of the cases but $sM = 2$ seems a to be a good compromise choice for the last 2 cases. For $g = 5$, the best choice is $sM = 2$ for smaller k and $sM = 3$ for larger k. Finally, it appears that $sM = 3$ works well for $g = 6, 7$. Therefore, when choosing the neighborhood size, one has to examine the input parameters in order to try to

Table 4: POT Tabu Lifetime tune up

CA(n,k,g)	I	R	L	No. CA	% CA	Avg.Iter/run.	Avg Time/run(sec)
CA(15,20,3)	200K	10	3	8	80	75162	242
CA(15,20,3)	200K	10	**4**	**10**	**100**	**32174**	**103**
CA(15,20,3)	200K	10	5	9	90	55015	176
CA(15,20,3)	200K	10	6	10	**100**	69494	224
CA(15,20,3)	200K	10	7	8	80	72474	234
CA(27,14,4)	200K	10	**3**	**10**	**100**	**25559**	**103**
CA(27,14,4)	200K	10	4	10	**100**	58241	236
CA(27,14,4)	200K	10	5	5	50	119804	483
CA(27,14,4)	200K	10	6	3	30	146854	597
CA(27,14,4)	200K	10	7	2	20	179523	730
CA(33,8,5)	200K	10	**3**	**10**	**100**	**20430**	**49**
CA(33,8,5)	200K	10	4	10	**100**	33198	80
CA(33,8,5)	200K	10	5	8	80	74462	180
CA(33,8,5)	200K	10	6	4	40	137165	335
CA(33,8,5)	200K	10	7	2	20	176577	431
CA(63,16,6)	200K	10	3	10	**100**	34620	613
CA(63,16,6)	200K	10	4	10	**100**	29359	520
CA(63,16,6)	200K	10	**5**	**10**	**100**	**24691**	**438**
CA(63,16,6)	200K	10	6	10	**100**	25122	445
CA(63,16,6)	200K	10	7	10	**100**	28046	497
CA(85,16,7)	200K	10	3	4	40	176563	5026
CA(85,16,7)	200K	10	4	10	**100**	92468	2637
CA(85,16,7)	200K	10	5	10	**100**	77342	2211
CA(85,16,7)	200K	10	6	10	**100**	62616	1788
CA(85,16,7)	200K	10	**7**	**10**	**100**	**55248**	**1579**

efficiently pick an appropriate value of M using the above classification as a guideline.

5.2.2 Parameter tune up for PAT

In this section, we tune up PAT's input parameter values. The input parameter values are the tabu lifetime (L) and the maximum number of iterations (I). When running the algorithm, we specify the size of the covering array we want to find and record the data for covering array success, time and iterations spent.

PAT Maximum number of Iterations and Tabu Lifetime Tune Up

In Table 7 and 8, we study the effect of varying the tabu lifetime and maximum number of iterations simultaneously on the reduced test bed. So, we choose five values for the tabu lifetime, $L = 1, 2, 3, 4, 5$ and two values for the maximum number of iterations, I= 200K and 500K. We display in bold face the row with winning input parameter values based on the one that yields the highest percentage of success. If we have ties, then we break them by picking the entry that requires the least amount of time per run. By looking at the data in Tables 7 and 8, one can see that running PAT for the longer number of iterations, I = 500K, yields more covering arrays. For example, for $L = 2$, finding a $CA(27, 14, 4)$ using I = 500K instead of I = 200K increases the percentage of success from 70% to 100%. According to the data, tabu lifetime of 2 achieves the most success especially for the harder problems. For instance, we see that an increase in L from 2 to 3, reduces the percentage of success from 100% to 20% when I =200K and from 100% to 40% when I = 500K for a $CA(33, 8, 5)$. PAT appears to be very sensitive to the tabu life time parameter.

PAT Tabu Lifetime Tune Up using the full test bed

In this section, we continue to run PAT on the entire test bed cases with I=500K but with L=2,3,4, just to verify if the best choice for this data remains to be $L = 2$. In Tables 9 and 10, you can see the test bed results. Here, we display in bold face the rows that correspond to the tabu lifetime value that yields the highest percentage of success. In the case when we have ties, we break them by picking the row that

Table 5: POT Test Bed with I=200K & L=4 & R=10 (Part I)

CA(n,k,g)	sM	No. CA	% CA	Avg.Iter/run.	Avg Time/run(sec)
CA(9,4,3)	**1**	**10**	**100**	**12**	**0**
CA(9,4,3)	2	10	**100**	17	0
CA(9,4,3)	3	10	**100**	22	0
CA(11,5,3)	**1**	**10**	**100**	**22**	**0**
CA(11,5,3)	2	10	**100**	24	0
CA(11,5,3)	**3**	**10**	**100**	**22**	**0**
CA(14,10,3)	**1**	**10**	**100**	**29032**	**26**
CA(14,10,3)	2	8	80	93538	65
CA(14,10,3)	3	6	60	130359	62
CA(15,20,3)	**1**	**10**	**100**	**32175**	**103**
CA(15,20,3)	2	8	80	73447	179
CA(15,20,3)	3	8	80	99293	166
CA(20,43,3)	1	4	40	176625	3373
CA(20,43,3)	2	7	70	127089	1835
CA(20,43,3)	**3**	**9**	**90**	**114166**	**1127**
CA(16,5,4)	**1**	**10**	**100**	**51**	**0**
CA(16,5,4)	2	10	**100**	50	0
CA(16,5,4)	3	10	**100**	58	0
CA(19,6,4)	**1**	**10**	**100**	**192**	**0**
CA(19,6,4)	2	10	**100**	697	0
CA(19,6,4)	3	10	**100**	968	0
CA(24,10,4)	1	3	30	166272	341
CA(24,10,4)	**2**	**7**	**70**	**156831**	**243**
CA(24,10,4)	3	1	10	194275	204
CA(27,14,4)	**1**	**10**	**100**	**58241**	**236**
CA(27,14,4)	2	9	90	69706	213
CA(27,14,4)	3	7	70	134588	280

Table 6: POT Test Bed with I=200K & L=4 & R=10 (Part II)

CA(n,k,g)	sM	No. CA	% CA	Avg.Iter/run.	Avg Time/run(sec)
CA(25,6,5)	1	10	**100**	351	0
CA(25,6,5)	2	10	**100**	213	0
CA(25,6,5)	**3**	**10**	**100**	**163**	**0**
CA(29,7,5)	1	10	**100**	1875	3
CA(29,7,5)	**2**	**10**	**100**	**1804**	**2**
CA(29,7,5)	3	10	**100**	10751	9
CA(33,8,5)	1	10	**100**	33198	80
CA(33,8,5)	**2**	**10**	**100**	**36562**	**67**
CA(33,8,5)	3	8	80	77388	96
CA(44,16,5)	1	10	**100**	16978	175
CA(44,16,5)	2	10	**100**	16713	130
CA(44,16,5)	**3**	**10**	**100**	**14909**	**78**
CA(36,3,6)	1	10	**100**	61	0
CA(36,3,6)	**2**	**10**	**100**	**56**	**0**
CA(36,3,6)	3	10	**100**	67	0
CA(37,4,6)	**1**	**10**	**100**	**904**	**1**
CA(37,4,6)	2	10	**100**	1176	1
CA(37,4,6)	3	10	**100**	2248	1
CA(55,11,6)	1	10	**100**	26113	213
CA(55,11,6)	2	10	**100**	23006	141
CA(55,11,6)	**3**	**10**	**100**	**33041**	**137**
CA(63,16,6)	1	10	**100**	29359	520
CA(63,16,6)	2	10	**100**	47675	636
CA(63,16,6)	**3**	**10**	**100**	**24635**	**221**
CA(49,8,7)	1	0	0	200000	1003
CA(49,8,7)	2	0	0	200000	757
CA(49,8,7)	3	0	0	200000	512
CA(61,10,7)	1	0	0	200000	1802
CA(61,10,7)	2	0	0	200000	1358
CA(61,10,7)	3	0	0	200000	915
CA(85,16,7)	1	10	**100**	9246	2636
CA(85,16,7)	2	10	**100**	10282	2707
CA(85,16,7)	**3**	**10**	**100**	**9279**	**1339**
CA(89,18,7)	1	9	90	93401	3382
CA(89,18,7)	2	10	**100**	59185	1613
CA(89,18,7)	**3**	**10**	**100**	**62642**	**1146**

Table 7: PAT Maximum Iterations & Tabu Lifetime Tune Up (Part I)

CA(n,k,g)	I	R	L	No. CA	% CA	Avg. Iter/R	Avg. Time/R
CA(15,20,3)	200K	10	1	6	60	125372	50
CA(15,20,3)	200K	10	**2**	**8**	**80**	**121979**	**48**
CA(15,20,3)	200K	10	3	3	30	157582	63
CA(15,20,3)	200K	10	4	0	0	200000	80
CA(15,20,3)	200K	10	5	0	0	200000	80
CA(15,20,3)	500K	10	1	6	60	245372	97
CA(15,20,3)	500K	10	**2**	**10**	**100**	**144139**	**57**
CA(15,20,3)	500K	10	3	5	50	337376	135
CA(15,20,3)	500K	10	4	0	0	500000	200
CA(15,20,3)	500K	10	5	0	0	500000	200
CA(27,14,4)	200K	10	1	0	0	200000	65
CA(27,14,4)	200K	10	**2**	**7**	**70**	**118206**	**37**
CA(27,14,4)	200K	10	3	3	30	169687	54
CA(27,14,4)	200K	10	4	2	20	179123	56
CA(27,14,4)	200K	10	5	0	0	200000	63
CA(27,14,4)	500K	10	1	0	0	500000	162
CA(27,14,4)	500K	10	**2**	**10**	**100**	**176610**	**55**
CA(27,14,4)	500K	10	3	7	70	344087	109
CA(27,14,4)	500K	10	4	2	20	419122	132
CA(27,14,4)	500K	10	5	0	0	500000	158
CA(33,8,5)	200K	10	1	0	0	200000	31
CA(33,8,5)	200K	10	**2**	**10**	**100**	**54978**	**8**
CA(33,8,5)	200K	10	3	2	20	167953	25
CA(33,8,5)	200K	10	4	0	0	200000	30
CA(33,8,5)	200K	10	5	0	0	500000	30
CA(33,8,5)	500K	10	1	0	0	500000	78
CA(33,8,5)	500K	10	**2**	**10**	**100**	**54978**	**8**
CA(33,8,5)	500K	10	3	4	40	379386	57
CA(33,8,5)	500K	10	4	4	40	446918	67
CA(33,8,5)	500K	10	5	2	20	466485	240

Table 8: PAT Maximum Iterations & Tabu Lifetime Tune Up (Part II)

CA(n,k,g)	I	R	L	No. CA	% CA	Avg. Iter/R	Avg. Time/R
CA(63,16,6)	200K	10	1	0	0	200000	163
CA(63,16,6)	200K	10	**2**	**10**	**100**	**50763**	**40**
CA(63,16,6)	200K	10	3	10	**100**	62670	50
CA(63,16,6)	200K	10	4	4	40	156611	125
CA(63,16,6)	200K	10	5	0	0	200000	160
CA(63,16,6)	500K	10	1	0	0	500000	407
CA(63,16,6)	500K	10	**2**	**10**	**100**	**50763**	**40**
CA(63,16,6)	500K	10	3	10	**100**	62670	50
CA(63,16,6)	500K	10	4	7	70	265957	213
CA(63,16,6)	500K	10	5	2	20	484272	388
CA(85,16,7)	200K	10	1	0	0	200000	221
CA(85,16,7)	200K	10	**2**	**7**	**70**	**122074**	**133**
CA(85,16,7)	200K	10	3	6	60	134862	146
CA(85,16,7)	200K	10	4	3	30	162083	178
CA(85,16,7)	200K	10	5	1	10	183282	199
CA(85,16,7)	500K	10	1	0	0	500000	555
CA(85,16,7)	500K	10	**2**	**9**	**90**	**181345**	**198**
CA(85,16,7)	500K	10	3	9	90	229159	248
CA(85,16,7)	500K	10	4	4	40	354749	385
CA(85,16,7)	500K	10	5	1	10	453282	491

represents the least amount of time. If it is not clear from the average time per run which is faster, since some entries only take a few milliseconds, we compare the average number of iterations per run. For example, we have a tie in percentage of success for a $CA(29, 7, 5)$ with $L = 2$ and $L = 3$. If we look at the time, we also have a tie of 1 second. In this case, we go ahead and compare the average iterations per run and deduce that $L = 2$ is better since it requires fewer number of iterations per run on average. Indeed, L=2 is the best choice for the PAT algorithm on our test bed according to the winners from Tables 9 and 10 for most cases. Once again, as was the case with POT, PAT was not very successful at finding covering arrays when $g = 8, 9, 10$, the harder problems from our test bed and that is why no results are reported.

5.2.3 Tabu Search Behaviour on the Test Bed

In this section, we study the behaviour of POT versus PAT in terms of time and percentage of success. In Table 11, we record the best results for the entire test bed for POT and PAT. When we have ties, we choose the first entry that appears in the table. We omit the entries for $g \geq 8$ since both POT and PAT are not successful at obtaining those cases in our test bed. The parameters for PAT are I = 500K and L =2, while for POT, we have I = 200K, L = 4 and M varies according to the settings sM in Table 11. We record the percentage of success of finding covering arrays with $R = 10$ and the average time per run. We omit the average iterations per run since in this case, POT and PAT are different implementations and comparing iterations counts is not meaningful. We also add a column that specifies the ratio of POT's average time per run to PAT's average time per run to analyze which algorithm is faster. From the data in Table 11, we see that POT and PAT have comparable percentages of success for finding covering arrays but it is clear from the time ratio that PAT is much faster than POT. For instance, both POT and PAT are 100% successful at finding a $CA(89, 18, 7)$ but in this case PAT is about 19 times faster than POT. We also observe that they tend to behave similarly with respect to an increase in k. As, k increases for a fixed g, both algorithms experience an increase in

Table 9: PAT Test Bed with I=500K & R=10 (Part I)

CA(n,k,g)	L	No. CA	% CA	Avg.Iter/run.	Avg Time/run(sec)
CA(9,4,3)	**2**	**10**	**100**	**13**	**0**
CA(9,4,3)	**3**	**10**	**100**	**13**	**0**
CA(9,4,3)	4	10	**100**	14	0
CA(11,5,3)	**2**	**10**	**100**	**24**	**0**
CA(11,5,3)	**3**	**10**	**100**	**24**	**0**
CA(11,5,3)	**4**	**10**	**100**	**24**	**0**
CA(14,10,3)	2	9	90	71099	7
CA(14,10,3)	**3**	**10**	**100**	**72118**	**7**
CA(14,10,3)	4	10	**100**	120113	12
CA(15,20,3)	**2**	**10**	**100**	**144139**	**57**
CA(15,20,3)	3	5	50	337376	135
CA(15,20,3)	4	0	0	500000	200
CA(20,43,3)	**2**	**6**	**60**	**333678**	**622**
CA(20,43,3)	3	0	0	500000	934
CA(20,43,3)	4	0	0	500000	925
CA(16,5,4)	**2**	**10**	**100**	**66**	**0**
CA(16,5,4)	3	10	**100**	68	0
CA(16,5,4)	4	10	**100**	68	0
CA(19,6,4)	**2**	**10**	**100**	**219**	**0**
CA(19,6,4)	3	10	**100**	540	0
CA(19,6,4)	4	10	**100**	669	0
CA(24,10,4)	**2**	**10**	**100**	**118995**	**19**
CA(24,10,4)	3	4	40	408933	67
CA(24,10,4)	4	0	0	500000	82
CA(27,14,4)	**2**	**10**	**100**	**176610**	**55**
CA(27,14,4)	3	7	70	344087	109
CA(27,14,4)	4	2	20	419122	132

Table 10: PAT Test Bed with I=500K & R=10 (Part II)

CA(n,k,g)	L	No. CA	% CA	Avg.Iter/run.	Avg Time/run(sec)
CA(25,6,5)	**2**	**10**	**100**	**396**	**0**
CA(25,6,5)	3	10	**100**	407	0
CA(25,6,5)	4	10	**100**	501	0
CA(29,7,5)	**2**	**10**	**100**	**7312**	**1**
CA(29,7,5)	3	10	**100**	11275	1
CA(29,7,5)	4	10	**100**	16820	2
CA(33,8,5)	**2**	**10**	**100**	**549778**	**8**
CA(33,8,5)	3	4	40	379386	57
CA(33,8,5)	4	4	40	446918	67
CA(44,16,5)	**2**	**10**	**100**	**14157**	**8**
CA(44,16,5)	3	10	**100**	41096	24
CA(44,16,5)	4	6	60	284528	168
CA(36,3,6)	2	10	**100**	49	0
CA(36,3,6)	**3**	**10**	**100**	**46**	**0**
CA(36,3,6)	4	10	**100**	51	0
CA(37,4,6)	2	8	80	100614	5
CA(37,4,6)	3	9	90	51022	2
CA(37,4,6)	**4**	**10**	**100**	**841**	**0**
CA(55,11,6)	**2**	**10**	**100**	**43176**	**16**
CA(55,11,6)	3	10	**100**	123360	46
CA(55,11,6)	4	7	70	361801	137
CA(63,16,6)	**2**	**10**	**100**	**50763**	**40**
CA(63,16,6)	3	10	**100**	62670	50
CA(63,16,6)	4	7	70	265957	213
CA(49,8,7)	2	0	0	500000	153
CA(49,8,7)	3	0	0	500000	153
CA(49,8,7)	4	0	0	500000	153
CA(61,10,7)	2	0	0	500000	220
CA(61,10,7)	3	0	0	500000	220
CA(61,10,7)	4	0	0	500000	221
CA(85,16,7)	**2**	**9**	**90**	**181345**	**198**
CA(85,16,7)	3	9	**90**	229159	248
CA(85,16,7)	4	4	40	354749	385
CA(89,18,7)	**2**	**10**	**100**	**44761**	**61**
CA(89,18,7)	3	10	**100**	65864	89
CA(89,18,7)	4	10	**100**	105844	144

Table 11: POT vs PAT with R=10

CA(n,k,g)	sM	%POT	%PAT	POT time/R	PAT time/R	POT:PAT
CA(9,4,3)	1	100	100	0	0	-
CA(11,5,3)	1	100	100	0	0	-
CA(14,10,3)	1	100	100	26	7	3.7
CA(15,20,3)	1	100	100	103	57	1.8
CA(20,43,3)	3	90	60	1127	622	1.8
CA(16,5,4)	1	100	100	0	0	-
CA(19,6,4)	1	100	100	0	0	-
CA(24,10,4)	2	70	100	243	19	12.8
CA(27,14,4)	1	100	100	236	55	4.3
CA(25,6,5)	3	100	100	0	0	-
CA(29,7,5)	2	100	100	2	1	2
CA(33,8,5)	2	100	100	67	8	8.4
CA(44,16,5)	3	100	100	78	8	9.8
CA(36,3,6)	2	100	100	1000	0	-
CA(37,4,6)	1	100	100	1	0	-
CA(55,11,6)	3	100	100	137	16	8.5
CA(63,16,6)	3	100	100	221	40	5.5
CA(49,8,7)	1	0	0	1003	153	6.5
CA(61,10,7)	1	0	0	1802	220	8.2
CA(85,16,7)	3	100	90	1339	198	6.8
CA(89,18,7)	3	100	100	1146	61	18.9

time as well. Overall, PAT is better than POT for this test bed.

One might think the reason that POT's running time is poor compared to PAT's is due to its higher iteration complexity caused by our choices of large neighborhood sizes, which are proportional to $\overline{M}$. An experimental study is conducted to show that decreasing POT's neighborhood size in such a way as to make POT and PAT more comparable in terms of their complexity per tabu move increases POT's total running time drastically and causes POT to converge more slowly towards a covering array. We verify this using the next table and using test bed data where $g = 5, 6$. For each $CA(n, k, g,)$, we run POT with $M = \overline{M}$, $0.5\overline{M}$ and $\frac{\overline{M}}{ng}$, which we denote as settings $sM = 1, 3, 4$, respectively. We show in bold the data for $M = \frac{\overline{M}}{ng}$, which

is the entry for which POT's neighborhood size is of the same order as PAT's. The other parameters are fixed as $I = 200K$, $L = 4$ and $R = 10$. We conclude that not only is PAT better than POT in its complexity per iteration but also at reducing the number of iterations.

Negative effect of decreasing POT's neighborhood size to PAT's size

CA(n,k,g)	sM	No. CA	% CA	Avg.Iter/run.	Avg Time/run(sec)
CA(25,6,5)	1	10	100	351	0
CA(25,6,5)	3	10	100	163	0
CA(25,6,5)	**4**	**10**	**100**	**15395**	**2**
CA(29,7,5)	1	10	100	1875	3
CA(29,7,5)	3	10	100	10751	9
CA(29,7,5)	**4**	**5**	**50**	**110392**	**32**
CA(33,8,5)	1	10	100	33198	80
CA(33,8,5)	3	8	80	77388	96
CA(33,8,5)	**4**	**0**	**0**	**20000**	**82**
CA(44,16,5)	1	10	100	16978	175
CA(44,16,5)	3	10	100	14909	78
CA(44,16,5)	**4**	**1**	**10**	**181507**	**217**
CA(36,3,6)	1	10	100	61	0
CA(36,3,6)	3	10	100	67	0
CA(36,3,6)	**4**	**10**	**100**	**929**	**0**
CA(37,4,6)	1	10	100	904	1
CA(37,4,6)	3	10	100	2248	1
CA(37,4,6)	**4**	**10**	**100**	**67146**	**14**
CA(55,11,6)	1	10	100	26113	213
CA(55,11,6)	3	10	100	33041	137
CA(55,11,6)	**4**	**0**	**0**	**20000**	**157**
CA(63,16,6)	1	10	100	29359	520
CA(63,16,6)	3	10	100	24635	221
CA(63,16,6)	**4**	**2**	**20**	**188428**	**320**

5.3 Algorithm Comparison

In this section, we compare upper bounds on the covering array number found by POT and PAT with upper bounds published in the literature. We need a test bed that compares a wide variety of algorithms for both fixed and mixed covering arrays. Cohen [9], in her PhD thesis, implements four algorithms and performs a comparison of her results against other published results for fixed and mixed cases for a variety of algorithms. Therefore, we choose the same data set used by Cohen to do our comparison of POT and PAT with other algorithms. The implementations used in the comparison are Automatic Efficient Test Generator (AETG) [6, 7], commercial AETG product (AP), Combinatorial Test Services (CTS) [24], In Parameter Order (IPO) [44, 48], Test Case Generator (TCG) [49], TConfig [47], Stardom's simulated annealing (SSA) [40], Nurmela's tabu search algorithm(TS) [35] and M. Cohen's greedy algorithms TCG and AETG (mTCG, mAETG), simulated annealing (SA) and hill climbing (HC) [9]. In Tables 12, 14 and 16, columns 2-8, 2-11 and 2-6 respectively, are original columns from Cohen's tables while column 7, labelled COL, in Table 16 represents the best upper bound found reported by Colbourn [10]. We add the last two columns in all the tables which represent the covering array sizes found using POT and PAT algorithms. The bold face entries in each table highlight the best upper bounds.

For this section, we run POT and PAT algorithms in *optimization mode*. We provide the algorithms with an upper and lower bound (ub, lb) for the covering array size as well as the number of replications (R). In optimization mode, the algorithm attempts to find covering arrays of size n, where n starts at the upper bound and is decreased by one, each time the search is successful, until it hopefully reaches the lower bound. For each n, the experiment is attempted at most R times with different random seeds. Whenever a covering array of size n is found, it considers a success and moves to the next smallest n; if no covering array was found after R replications for this n, the whole algorithm stops. We give the pseudocode for this algorithm next.

Algorithm: optimizationMode(ub, lb, R, k, g)

```
n ← ub
success ← true
while((n >= lb) and (success)) {
      replications ← 0
      success ← false
      while ((replications < R) and (! success)) {
            Try to find a CA(n, k, g) using tabu search (POT or PAT)
            replications++
            if (found) {
              record n, total time and iterations over all replications for the same n
              success ← true
              n − −
            }
      }
}
```

In our tests, we run the algorithm in optimization mode picking upper and lower bounds based on data reported in previous experiments. More specifically, if b is the best upper bound found by previous results reported in a line of Table 12, 14 or 16, we use $ub = b + 2$ and $lb = b - 2$. We do this in order to ensure that we are able to find a covering array even if it is not the best known so far and we try to improve on the best known bound once a covering array of the best known size is found. In certain cases, we had to increase the upper bound to $b + 8$ for harder problems such as finding a $CA(n, 20, 10)$ and $CA(n, 100, 4)$. When running our algorithms in optimization mode, we are primarily interested in the smallest value of n obtained by each algorithm, which we report in Tables 12, 14 and 16 under the headings POT and PAT. In addition, we tabulate in Tables 13, 15 and 17 all covering array sizes found along with their corresponding times and total iterations in order to do a further comparison between POT and PAT.

The parameters for POT and PAT were set at their default values, based on the

findings in Section 5.2. More specifically, for POT we used $L = 4$ and $I = 200$K, while for PAT we used $L = 2$ and $I = 500$K. The choice of M for POT was done in an adhoc manner, inspired by the findings from Section 5.2. This parameter is reported in the column labelled sM in Tables 13,15, and 17.

Looking at Table 12, we have a test set that consists of 4 mixed covering arrays. Cohen's implementation of simulated annealing achieves the best bounds in the table for all four entries and so does our PAT algorithm. Our POT algorithm is successful in achieving 3 out of the 4 best known bounds but only finds a $CA(22, 5^1 4^4 3^{11} 2^5)$. We see in Table 13, that PAT is much faster at find covering arrays than POT at every step, except in one of the cases; POT only requires 66 seconds to find a $\mathrm{CA}(42, 7^1 6^1 5^1 4^5 3^8 2^3)$ while PAT needs 478 seconds. In most cases, PAT only requires a few milliseconds to find the covering arrays.

In Table 14, we have 4 fixed covering arrays and 2 mixed cases. The last two covering arrays in the table are relatively harder problems and we are not successful at finding the best upper bound reported by other algorithms. We are able to improve on the best bound reported for a $CA(n, 4^{15} 3^{17} 2^{29})$; the best bound was $n = 29$ found by Nurmela's tabu search algorithm, but we improved on that with our implementation of the same algorithm and achieved a size $n = 28$. PAT appears to be better than POT for these particular cases as it achieves better bounds in most cases, and in less time, as it can be seen in Table 15.

Based on Table 16, we see three things. First, that POT achieves two new bounds, a $CA(84, 16, 7)$ and $CA(110, 16, 8)$ and PAT one new bound, a $CA(110, 16, 8)$. We also notice that for this test set, POT is as good or more successful than PAT for finding bounds, although it requires a little more time. The fourth column in Table 16 reports on results by Cohen's simulated annealing [9] which achieved most of the best upper bounds prior to POT, but we see that POT performs better for this set of data by improving on 3 out of 4 bounds.

Table 12: Algorithm Comparison 1: Best upper bounds

	TCG	AETG	AP	mTCG	mAETG	HC	SA	POT	PAT
CA($n, 5^1 3^8 2^2$)	20	19	21	18	20	16	**15**	**15**	**15**
CA($n, 7^1 6^1 5^1 4^5 3^8 2^3$)	45	45	53	**42**	44	**42**	**42**	**42**	**42**
CA($n, 5^1 4^4 3^{11} 2^5$)	30	30	33	25	28	23	**21**	22	**21**
CA($n, 6^1 5^1 4^6 3^8 2^3$)	33	34	39	32	35	**30**	**30**	**30**	**30**

Table 13: Algorithm Comparison 1: Time for POT vs PAT

	sM	N	POT time	POT iter	PAT time	PAT iter
CA(n, $5^1 3^8 2^2$)	1	17	0	218	0	120
		16	1	832	0	258
		15	264	206631	0	447
CA(n, $7^1 6^1 5^1 4^5 3^8 2^3$)	3	44	14	1067	0	120
		43	44	3436	0	226
		42	66	5265	478	1000302
CA(n, $5^1 4^4 3^{11} 2^5$)	1	23	220	32791	0	568
		22	2311	366184	0	2062
		21	-	-	7	18832
CA(n, $6^1 5^1 4^6 3^8 2^3$)	2	32	34	3927	0	176
		31	108	12912	0	296
		30	941	123744	0	497

Table 14: Algorithm Comparison 2: Best upper bounds

	IPO	AETG	TConfig	CTS	AP	TS	mTCG	mAETG	HC	SA	POT	PAT
CA(n,4,3)	**9**	**9**	**9**	**9**	10	NA	**9**	**9**	**9**	**9**	**9**	**9**
CA(n,13,3)	17	**15**	**15**	**15**	22	NA	17	17	16	16	**15**	**15**
CA(n,$4^{15} 3^{17} 2^{29}$)	34	41	40	39	41	29	34	37	30	30	30	**28**
CA(n,$4^1 3^{39} 2^{35}$)	26	28	30	29	30	**21**	26	27	**21**	**21**	22	**21**
CA(n,100,4)	53	NA	**43**	**43**	52	NA	56	56	47	45	46	45
CA(n,20,10)	212	**180**	231	210	230	NA	213	198	189	183	188	185

Table 15: Algorithm Comparison 2: Time for POT vs PAT

	sM	N	POT time	POT iter	PAT time	PAT iter
CA(n,4,3)	1	11	0	15	0	24
		10	0	24	0	34
		9	0	34	0	43
CA(n,13,13)	1	17	0	47	0	89
		16	0	89	0	141
		15	4	2790	0	300
CA(n,$4^{15}3^{17}2^{29}$)	2	31	5970	38194	1	438
		30	48582	328483	3	964
		29	-	-	15	5112
		28	-	-	49	160878
CA(n,$4^{1}3^{39}2^{35}$)	3	23	305	3083	1	292
		22	2128	22777	2	605
		21	-	-	5	1348
CA(n,100,4)	2	46	10524	28933	-	-
		45	-	-	949	57868
CA(n,20,10)	3	188	31202	213845	-	-
		185	-	-	53220	262067

Table 16: Algorithm Comparison 3: best upper bounds

CA(n,k,g)	SSA	mAETG	SA	CTS	AP	COL	POT	PAT
CA(n,16,6)	65	70	**62**	88	80	63	**62**	63
CA(n,16,7)	88	94	87	91	110	85	**84**	85
CA(n,16,8)	113	120	112	120	128	111	**110**	**110**
CA(n,17,8)	116	123	114	120	128	**111**	112	112

Table 17: Algorithm Comparison 3: Time for POT vs PAT

CA(n,k,g)	sM	N	POT time	POT iter	PAT time	PAT iter
CA(n,16,6)	1	64	141	7989	7	9673
		63	584	33062	74	92701
		62	2168	122728	-	-
CA(n,16,7)	3	87	131	9002	7	6657
		86	707	48518	20	18815
		85	2683	183783	458	421618
		84	18889	1293895	-	-
CA(n,16,8)	1	113	371	17540	9	6689
		112	1158	27957	40	28192
		111	5040	121637	76	53695
		110	20223	488151	1911	1367757
CA(n,17,8)	1	113	18159	378797	41	26008
		112	25587	533800	524	329939

5.4 New Results

In this section, we provide tables with new upper bounds for covering arrays using POT and PAT algorithms. In this step, we look at the two types of problems. First, we investigate if our tabu implementations can improve on upper bounds for fixed covering arrays reported by Colbourn [10]. We look at Colbourn's tables and target those regions where the gaps are large. More specifically, when these tables have consecutive entries: $CA(n_1, k_1, g)$ and $CA(n_2, k_2, g)$, we try to find a $CA(n_2 - 1, k_1 + 1, g)$ where the difference between n_1 and n_2 or k_1 and k_2 is more than one. These regions make ideal candidates for improvement on the bounds. Second, we attempt to discover new upper bounds for small mixed covering arrays with parameters close to those of an orthogonal array.

In Tables 18 and 19, we give the the best known upper bound extracted from [10] denoted by *ub* and provide the upper bound that POT and PAT algorithms found, denoted POT n and PAT n, respectively. We also provide the iteration and time at which the covering array was found. The parameters used in the tests are given

Table 18: New results for POT with I=500K, L=5, sM=1, R=1

CA(n,k,g)	ub	POT n	iter	time(sec)
CA(n,21,3)	17	16	267621	990
CA(n,16,5)	44	43	292455	5443
CA(n,17,5)	45	44	373390	4291
CA(n,17,6)	64	63	422371	8268
CA(n,18,6)	**70**	**65**	**145072**	**3244**
CA(n,19,6)	**70**	**66**	**62301**	**3041**
CA(n,14,7)	**81**	**80**	**164746**	**3387**
CA(n,15,7)	**83**	**82**	**153248**	**3657**
CA(n,16,7)	**85**	**84**	**755264**	**20838**
CA(n,17,7)	**87**	**86**	**279963**	**8007**
CA(n,18,7)	**89**	**88**	**134951**	**4811**
CA(n,19,7)	**91**	**90**	**140122**	**5694**
CA(n,16,8)	**111**	**110**	**488151**	**20223**

in the caption of the tables. For the POT results in Table 18, we attempt manual testing by providing n for the algorithm and checking to see if it finds it, in which case we attempted a lower n. For PAT results in Table 19, we used the algorithm in optimization mode, as described in Section 5.3. We run this algorithm starting with an upper bound which is one less than the best known (ub), the lower bound for the algorithm is two less than the best known upper bound, and the number of replications is 10.

In most cases for POT in Table 18, the improved bound is one less than the best bound reported in [10], except for $CA(65, 18, 6)$ and $CA(66, 19, 6)$ where the bound was reduced by 5 and 4, respectively. We also observe that for the common covering arrays in Tables 18 and 19, which are marked in bold in each table, PAT is much faster at finding the improved bounds than POT. For example, POT took 20223 seconds to find a $CA(110, 16, 8)$ while PAT only needs a mere 371 seconds, PAT being roughly 54 times faster.

In the second type of experiments, we try to find new upper bounds for small mixed covering arrays that have parameters close to those of orthogonal arrays, using PAT

Table 19: New results for PAT with I=10^6, L=2 and $R = 10$

CA(n,k,g)	ub	PAT n	iter	time(sec)
CA(n,18,6)	**70**	**65**	**769098**	**777**
CA(n,19,6)	**70**	**66**	**585686**	**660**
		65	3889078	4390
CA(n,14,7)	**81**	**80**	**6709736**	**5508**
CA(n,15,7)	**83**	**82**	**517620**	**488**
CA(n,16,7)	**85**	**84**	**1554019**	**1685**
CA(n,17,7)	**87**	**86**	**807431**	**983**
CA(n,18,7)	**89**	**88**	**318665**	**435**
		87	6903745	9423
CA(n,19,7)	**91**	**90**	**685056**	**1051**
		89	169310	260
CA(n,16,8)	**111**	**110**	**265331**	**371**

algorithm. We have two classes of mixed arrays that we look at: CA$(n, g^{g+1}(g-1))$ and CA$(n, g^{g+1}(g+1))$, for g a prime power. For these covering arrays, we attempt a different approach of optimization mode, which we call *lower optimization mode*. We provide a lower bound and the algorithm tries to achieve that lower bound a fixed number of times and if it is successful then it records the statistics and quits, otherwise, it increases the lower bound by one and repeats the process. In lower optimization mode, the algorithm keeps running until it finds a covering array for that set of input parameter values.

For the first class of covering arrays, we decrease one of the alphabet sizes by one from a CA(n, g^{g+2}), so we provide a lower bound of g^2 and check if PAT achieves this bound, we also report the upper bound $CAN(g^{g+2})$ for comparison purposes. For the second class of covering arrays, we are increasing one of that alphabet sizes by one in a CA(n, g^{g+2}). In this case, we provide a lower bound which is the product of the two largest alphabets and we report on the upper bound given by Theorem 2.2.3, a construction that we use for increasing the largest alphabet size in the best known CA(n, g^{g+2}). In Tables 20 and 21, we see the lower bound (lb) provided for the lower optimization mode, upper bound (ub) and the actual bound (PATn) that PAT

Table 20: Class 1: PAT with I=500K, L=2, R=5

	lb	ub	PATn	iter	time(sec)
CA(n,$3^1 4^5$)	16	19	**18**	5000061	289
CA(n,$4^1 5^6$)	25	29	**28**	7501257	827
CA(n,$6^1 7^8$)	49	61	66	3469155	1206
CA(n,$7^1 8^9$)	64	78	90	26287408	14555

Table 21: Class 2: PAT with I=500K, L=2, R=5

	lb	ub	PATn	iter	time(sec)
CA(n,$4^5 5^1$)	20	23	**20**	355	0
CA(n,$5^6 6^1$)	30	35	**32**	5007050	612

achieves as well as the total time and iteration. We bold face the entries in tables where PAT returns a bound that is better than the upper bound reported. We see from Table 20 that we improve the upper bound for 2 out of the 4 entries and improve on the 2 entries from Table 21.

5.5 Experimental Conclusion

In this section, we give a brief overview of the entire chapter and summarize the results obtained. Before doing that, we observe that the conclusions drawn from this experiment would be the same if the more efficient implementation of $UncoveredList$ in Section 4.2 were used. Indeed, such an implementation would only affect the times reported, while the number of iterations and percentage of success would remain the same. In addition, PAT would have larger relative savings than POT, so we only expect a possible increase in the time ratio by which PAT is faster than POT.

Based on our designed test bed of 30 cases for the parameter tune up of POT and PAT, we deduce that POT is comparable to PAT in terms of the percentage of success of finding covering arrays but PAT tends to be much faster with a rate varying from roughly 2$\sim$19 times faster.

Table 22: Summary of results found by POT or PAT

CA(n,k,g)	Previous ub	Algorithm
CA(16,21,3)	17	POT
CA(43,16,5)	44	POT
CA(44,17,5)	45	POT
CA(63,17,6)	64	POT
CA(65,18,6)	70	POT/PAT
CA(65,19,6)	70	PAT
CA(80,14,7)	81	POT/PAT
CA(82,15,7)	83	POT/PAT
CA(84,16,7)	85	POT/PAT
CA(86,17,7)	87	POT/PAT
CA(87,18,7)	89	PAT
CA(89,19,7)	91	PAT
CA(110,16,8)	111	POT/PAT
CA(28,$4^{15}3^{17}2^{29}$)	29	PAT
CA(18,$3^{1}4^{5}$)	19	PAT
CA(28,$4^{1}5^{6}$)	29	PAT
CA(20,$4^{5}5^{1}$)	23	PAT
CA(32,$5^{6}6^{1}$)	35	PAT

In analyzing Cohen's tables with appended POT and PAT results, we observe one more time that PAT is much faster than POT and that PAT is more successful in the optimization mode at achieving better bounds than POT for the mixed case. An exception is Table 16, where POT gives similar or better bounds than PAT for $CA(16, g)$ for $g = 6, 7, 8$. Our results are most comparable to Cohen's simulated annealing results. Out of the 14 test cases, PAT matches 7 of the simulated annealing results, improves on 5 of them and does worse on 2 of them. Similar counts for POT, gives 6 matches, 4 improvements and 4 worsenings, when compared to Cohen's simulated annealing. In Table 22, we summarize the new results found. We give the covering arrays found, previous best known bound and specify the algorithm that achieves that bound.

Chapter 6

Conclusion

In this thesis, we explore a new generalization of both mixed covering arrays and covering arrays on graphs known as the mixed covering arrays on graphs. We also examine two variations of the tabu search method to try and improve on covering array bounds for both fixed and mixed covering arrays.

Constructions of **mixed covering arrays on graphs** is our first area of contribution. We generalize results for covering arrays on graphs by Meagher and Stevens [31] to the mixed case and also provide constructions for some special classes of graphs. We give a construction of mixed covering arrays on graphs based on weight-restricted graph homomorphism between two graphs which also provide bounds on the mixed covering array number. For any graph G, there exist homomorphisms between the following graphs $K_{\omega(G)} \to G \to K_{\chi(G)}$. We extend these homomorphisms to weight-restricted homomorphisms which allow us to give lower and upper bounds on $CAN(G, \prod_{i=1}^{k} g_i)$. In particular, let G be a weighted graph with k vertices and $g_1 \leq g_2 \leq \cdots \leq g_k$ be positive weights; we prove that the $CAN(K_{\omega(G)}, \prod_{i=1}^{\omega(G)} g_i) \leq CAN(G, \prod_{j=1}^{k} g_j) \leq CAN(K_{\chi(G)}, \prod_{\ell=k-\chi(G)+1}^{k} g_\ell)$. We also define a special class of graphs, which we call the mixed qualitative independence graphs, which is particularly useful for studying covering arrays. We prove that for a weighted graph G and positive integers $n, g_1, g_2, \ldots, g_k$, a $CA(n, k, \prod_{i=1}^{k} g_i)$ exists if and only if there is a weight-restricted graph homomorphism from G to $QI(n, \prod_{i=1}^{k} g_i)$. In addition to

giving a result for the existence of mixed covering arrays on graphs, the mixed qualitative independence graph allows us to define a lower bound on the mixed covering array number on a graph based on the knowledge of both graphs' clique or chromatic numbers.

Defining the appropriate weight-restricted homomorphism is not always easy and it has been shown that finding an optimal covering array on a general graph is NP-hard for the binary case [38]. This motivates us to try to first construct mixed covering arrays on special classes of graphs. In order to do this, we define some basic graph operations: one-vertex edge hooking, edge duplication and weight-restricted edge subdivision. When these operations are performed to a weighted graph, they preserve the size of a balanced covering array on the graph. We use these operations to give constructions of optimal mixed covering arrays for trees and cycles. Independently from graph operations, we construct optimal mixed covering arrays for bipartite graphs and provide an alternate construction for cycles. Moreover, if G is n-colourable, for $n = 2, 3, 4, 5$, we show that in many cases the product of the largest two alphabets is an upper bound on the mixed covering array number.

There are many areas that one can study and expand on, with regards to mixed covering arrays on graphs. It is a new generalization and there is room for a lot of work to be done in this area. An interesting problem to extend the work done in this thesis would be to find optimal mixed covering arrays on cubic graphs and wheels with non-uniform alphabet sizes. Another problem is to find other classes of graphs for which their mixed covering array number is $PW(G)$. A more ambitious direction to explore is the design of recursive constructions of covering arrays on graphs based on graph decompositions.

The second main contribution is in the area of **tabu search algorithms** for covering array construction. We implement a variation of Stardom's tabu algorithm [40] which we call POT, since the basic move in the algorithm is to switch a point in the array. The variation is in the neighborhood selection. Stardom exhaustively searches the whole neighborhood at every iteration, while in POT we select a random sample from the neighborhood. In several cases, such a reduction on the neighborhood size proved to be advantageous. The second algorithm we implement is proposed

by Nurmela [35] and we call it PAT, since the basic move aims at covering a new pair in the array. We give efficient data structures and detailed pseudocode for our implementations of POT and PAT and a thorough complexity analysis. POT has an expected running time of $O(nk^2g)$ per tabu move while PAT has an expected running time of $O(nk)$ per tabu move.

In our experimental study, we provide an extensive comparison between POT and PAT and we are able to improve on existing upper bounds for fixed and mixed covering arrays. We also perform a comparison of POT and PAT against many algorithms from the literature comparing the bounds each algorithm achieves. Based on our experiments, we conclude that POT is comparable to PAT in terms of the percentage of success of finding covering arrays, but PAT tends to be much faster. This is partially due to the fact that the neighborhood space for PAT is much smaller than POT and therefore one iteration for POT takes much longer than that for PAT. However, we also observe that the total number of iterations for PAT tends to be smaller than POT, which indicates that PAT's moves are more effective at arriving faster at a covering array. This notion is further supported by evidence that the total time for POT increases when its neighborhood size is made comparable to PAT's. PAT also appears to be better than POT at finding better bounds for mixed covering arrays. POT and PAT are comparable to algorithms in literature since they achieve most of the best known upper bounds reported. Finally, POT and PAT are also successful at finding improved upper bounds: 13 new results are found for fixed covering arrays and 5 new ones for the mixed case.

We conclude this chapter with future work on the area of tabu search algorithms. A possible extension of our tabu search implementations is to find covering arrays on graphs and covering arrays with the disjoint row property specified. This can be done by initializing the data structures as if some pairs had already been covered. Another interesting variation on the tabu search method for covering arrays is the test tabu search algorithm (TET). TET is a new technique partially explored by us that makes a more drastic move at each iteration. The basic move is to add or remove an entire test suite (row). The difficulty arises in that we try to simultaneously deal with conflicting objectives, namely increasing coverage and decreasing the number of

tests. Nonetheless, this is a promising direction for further research that could be explored.

Appendix A

Colbourn's tables of CA upper bounds

In this section, we include tables produced by Colbourn [10], which give the best known upper bounds for covering arrays. The author has authorized us to reproduce the tables here; starting on the next paragraph until the end of the appendix we extract the rest word by word from [10]. Note that the new results obtained by our algorithms have not been incorporated here.

To determine the best known bound on $CAN(k, g)$, look in the table for the specified value of g. Entries in the table are of the form "$\underline{k}, n^{\alpha}$. Select the smallest $\underline{k}$ for which $\underline{k} \geq k$, and let n be the size tabulated for $\underline{k}$. Then, $CAN(k, g) \leq n$, and the construction to establish this is specified by the authority α.

2	Theorem 2.2 [15]	3	Theorem 2.3 [15]
a	1-rotational [29, 32]	b	1-rotational [15] and Colbourn (unpublished)
o	orthogonal array [25]	p	projection [10]
s	simulated annealing [9]	t	tabu search [35]
z	composition	$\downarrow$	symbol identification
d	derivation		

Covering array bounds for $g = 3$.

4	9^o	5	11^b	7	12^s
9	13^s	10	14^s	20	15^t
24	17^t	30	18^t	36	19^t
43	20^t	74	21^3	94	23^2
134	24^3	174	25^3	194	26^2
394	27^3	474	29^3	594	30^3
714	31^3	854	32^3	1474	33^3
1796	35^2	2364	36^2	3030	37^2
3766	38^2	6836	39^2	8238	41^2
10296	42^2	12372	43^2	14794	44^2
20000	45^2				

Covering array bounds for $g = 4$.

5	16^o	6	19^a	7	21^s
8	22^s	10	24^s	11	25^s
12	26^s	14	27^s	24	28^3
29	31^3	30	32^3	34	33^3
38	34^3	40	35^3	49	36^3
54	37^3	59	38^3	69	39^3
116	40^3	140	43^3	144	44^3
164	45^3	184	46^3	192	47^3
236	48^3	260	49^3	284	50^3
332	51^3	560	52^3	676	55^3
696	56^3	792	57^3	888	58^3
928	59^3	1140	60^3	1256	61^3

Covering array bounds for $g = 5$.

6	25^o	7	29^b	8	33^b
9	35^s	10	37^s	11	38^s
12	40^s	13	41^s	14	42^s
15	43^s	16	44^s	35	45^3
41	49^3	42	52^3	48	53^2
52	55^3	54	56^3	59	57^3
65	58^3	71	60^3	77	61^3
83	62^3	90	63^3	95	64^3
205	65^3	240	69^3	245	72^3
281	73^2	305	75^3	315	76^3
345	77^3	380	78^3	415	80^3
450	81^3	485	82^3	525	83^3

Covering array bounds for $g = 6$.

3	36^o	4	37^s	5	39^s
6	41^t	8	42^t	9	46^b
10	51^b	11	55^s	12	56^s
13	58^s	14	60^s	15	61^s
16	63^s	17	64^s	19	70^3
20	71^2	24	72^3	26	73^3
32	74^2	34	75^3	40	76^2
42	77^2	48	78^2	50	79^2
56	80^2	66	82^2	72	84^2
80	86^2	81	88^2	89	91^2
90	92^2	96	93^2	98	94^2
99	95^2	107	96^2	114	97^2

Covering array bounds for $g = 7$.

8	49^o	10	61^b	11	67^b
12	73^b	13	78^s	14	81^s
15	83^s	16	85^s	17	87^s
18	89^s	63	91^3	64	97^3
79	103^3	80	105^2	87	109^3
99	115^2	100	117^2	103	120^3
109	121^2	112	123^3	119	125^3
120	126^3	127	127^3	135	129^3
143	131^3	497	133^3	504	139^3
623	145^3	640	147^3	686	151^3
781	157^2	800	159^2	812	162^3
860	163^2	882	165^3	938	167^3

Covering array bounds for $g = 8$.

9	64^o	11	78^a	12	85^b
13	92^b	14	99^b	15	106^b
17	111^s	18	115^s	19	117^s
80	120^3	81	127^3	90	134^3
99	136^2	107	141^3	116	148^3
121	150^2	125	155^3	132	157^2
143	162^2	151	167^3	155	169^2
160	171^3	170	173^3	712	176^3
720	183^3	808	190^3	891	192^3
952	197^3	1032	204^3	1089	206^3
1112	211^3	1188	213^2	1273	218^2
1331	220^2	1344	223^3	1380	225^2

Covering array bounds for $g = 9$.

10	81^o	13	105^b	14	113^b
15	121^b	16	129^b	17	137^b
18	145^b	99	153^3	100	161^3
129	177^3	139	185^3	140	191^2
149	193^3	168	201^2	181	209^2
182	215^2	195	217^2	196	223^2
981	225^3	990	233^3	1000	241^3
1278	249^3	1377	257^3	1400	263^3
1476	265^3	1665	273^2	1794	281^2
1820	287^2	1933	289^2	1960	295^2
9720	297^3	9810	305^3	9900	313^3
12663	321^3	13644	329^3	13860	335^3

Covering array bounds for $g = 10$.

4	100^o	6	101^s	12	118^o
13	120^p	15	136^b	16	145^b
17	154^b	18	163^b	19	172^b
20	174^u	21	190^b	24	191^2
35	192^3	36	201^2	48	208^2
71	209^3	78	211^2	143	226^2
156	228^2	169	230^2	179	244^2
195	246^2	208	255^2	224	262^2
239	271^2	255	280^2	260	284^2
271	289^2	288	298^2	415	300^3
468	302^3	572	316^2	841	317^2
936	319^2	1014	321^2	1694	334^2

Covering array bounds for $g = 11$.

12	121^o	16	161^b	17	171^b
18	181^b	19	191^b	20	201^b
21	211^b	22	221^b	143	231^3
144	241^3	191	271^3	192	277^2
203	281^3	215	291^3	227	301^3
255	311^2	256	317^2	271	321^2
272	327^2	288	331^2	1705	341^3
1716	351^3	2277	381^3	2304	387^3
2420	391^3	2563	401^3	2706	411^3
3041	421^2	3072	427^2	3232	431^2
3264	437^2	3435	441^2	3456	447^2
20000	451^3				

Covering array bounds for $g = 12$.

7	144^o	14	166^o	15	168^p
18	199^b	19	210^b	20	221^b
21	232^b	22	243^b	23	254^b
24	265^b	48	276^3	49	287^2
97	298^3	105	300^2	195	320^2
210	322^2	225	324^2	251	353^2
270	355^2	285	366^2	300	377^2
323	386^2	341	397^2	360	408^2
379	419^2	666	430^3	735	432^3
1345	452^2	1470	454^2	1575	456^2
2704	474^2	2940	476^2	3150	478^2
3375	480^2	3484	507^2	3780	509^2

Covering array bounds for $g = 13$.

14	169^{o}	17	$249^{\downarrow}$	18	$251^{\downarrow}$
19	$252^{\downarrow}$	21	253^{b}	22	265^{b}
23	277^{b}	24	289^{b}	25	301^{b}
26	313^{b}	195	325^{3}	196	337^{3}
237	405^{3}	252	407^{3}	266	408^{3}
293	409^{3}	307	421^{3}	321	433^{3}
335	445^{3}	349	457^{3}	363	469^{3}
2717	481^{3}	2730	493^{3}	3302	561^{3}
3510	563^{3}	3705	564^{3}	4082	565^{3}
4277	577^{3}	4472	589^{3}	4667	601^{3}
4862	613^{3}	5057	625^{3}	20000	637^{3}

Covering array bounds for $g = 14$.

5	196^{o}	6	$222^{\downarrow}$	17	251^{o}
18	$253^{\downarrow}$	19	254^{p}	20	$285^{\downarrow}$
21	286^{p}	23	300^{b}	24	313^{b}
25	326^{b}	26	339^{b}	27	352^{b}
28	365^{b}	29	378^{b}	30	391^{b}
31	404^{b}	32	417^{b}	36	430^{2}
85	433^{2}	90	435^{2}	95	436^{2}
102	459^{2}	108	461^{2}	114	462^{2}
115	482^{2}	288	488^{2}	306	490^{2}
323	491^{2}	324	492^{2}	342	493^{2}
361	494^{2}	380	525^{2}	399	526^{2}
414	539^{2}	437	540^{2}	456	553^{2}

Covering array bounds for $g = 15$.

6	225^{o}	17	253^{o}	18	255^{p}
19	$286^{\downarrow}$	20	287^{p}	21	$354^{\downarrow}$
22	$355^{\downarrow}$	23	$356^{\downarrow}$	24	357^{p}
26	365^{b}	27	379^{b}	28	393^{b}
29	407^{b}	30	421^{b}	36	435^{2}
102	463^{2}	108	465^{2}	288	491^{2}
306	493^{2}	324	495^{2}	340	525^{2}
342	526^{2}	360	527^{2}	361	557^{2}
380	558^{2}	400	559^{2}	408	595^{2}
414	596^{2}	432	597^{2}	441	603^{2}
468	605^{2}	486	619^{2}	504	633^{2}
520	637^{2}	522	647^{2}	540	651^{2}

Covering array bounds for $g = 16$.

17	256^{o}	18	286^{o}	19	288^{p}
20	354^{o}	21	$356^{\downarrow}$	22	$357^{\downarrow}$
23	358^{p}	28	421^{b}	29	436^{b}
30	451^{b}	31	466^{b}	32	481^{b}
288	496^{3}	289	511^{3}	305	526^{3}
323	528^{2}	342	558^{2}	361	560^{2}
374	597^{3}	391	598^{2}	396	627^{2}
414	628^{2}	418	629^{2}	437	630^{2}
475	661^{3}	492	676^{3}	509	691^{3}
532	693^{2}	551	708^{2}	570	723^{2}
4880	736^{3}	4896	751^{3}	5168	766^{3}
5491	768^{3}	5814	798^{2}	6137	800^{2}

Covering array bounds for $g = 17$.

18	289^{o}	20	356^{o}	21	$358^{\downarrow}$
22	359^{p}	31	497^{b}	32	513^{b}
33	529^{b}	34	545^{b}	323	561^{3}
324	577^{3}	359	628^{3}	378	630^{3}
396	631^{2}	399	695^{2}	420	697^{2}
440	698^{2}	441	699^{2}	462	700^{2}
484	701^{2}	557	769^{3}	575	785^{3}
593	801^{3}	611	817^{3}	5797	833^{3}
5814	849^{3}	6443	900^{3}	6783	902^{3}
7128	903^{3}	7161	967^{2}	7539	969^{3}
7920	970^{2}	7938	971^{3}	8316	972^{2}
8712	973^{2}	8800	1037^{2}	8820	1038^{2}

Covering array bounds for $g = 18$.

5	324^{o}	20	358^{o}	21	360^{p}
24	518^{o}	25	$520^{\downarrow}$	26	$521^{\downarrow}$
27	$522^{\downarrow}$	28	$523^{\downarrow}$	29	524^{p}
33	562^{b}	34	579^{b}	35	596^{b}
36	613^{b}	37	630^{b}	38	647^{b}
100	664^{2}	105	666^{2}	399	698^{2}
420	700^{2}	441	702^{2}	479	858^{2}
504	860^{2}	520	861^{2}	540	862^{2}
560	863^{2}	580	864^{2}	588	865^{2}
609	866^{2}	659	902^{2}	693	904^{2}
714	921^{2}	735	938^{2}	756	955^{2}
777	972^{2}	798	989^{2}	1995	1004^{2}

Covering array bounds for $g = 19$.

20	361^{o}	24	520^{o}	25	$522^{\downarrow}$
26	$523^{\downarrow}$	27	$524^{\downarrow}$	28	525^{p}
29	$616^{\downarrow}$	30	$617^{\downarrow}$	31	$618^{\downarrow}$
32	619^{p}	36	649^{b}	37	667^{b}
38	685^{b}	399	703^{3}	400	721^{3}
479	862^{3}	500	864^{3}	520	865^{3}
540	866^{3}	560	867^{2}	580	958^{3}
600	959^{3}	620	960^{3}	640	961^{2}
719	991^{3}	739	1009^{3}	759	1027^{3}
784	1031^{2}	7961	1045^{3}	7980	1063^{3}
9557	1204^{3}	9975	1206^{3}	10374	1207^{3}
10773	1208^{3}	11200	1209^{3}	11571	1300^{3}

Covering array bounds for $g = 20$.

6	400^{o}	7	$438^{\downarrow}$	8	513^{d}
24	522^{o}	25	$524^{\downarrow}$	26	$525^{\downarrow}$
27	526^{p}	28	$617^{\downarrow}$	29	$618^{\downarrow}$
30	$619^{\downarrow}$	31	620^{p}	32	$719^{\downarrow}$
33	$720^{\downarrow}$	34	$721^{\downarrow}$	35	722^{p}
39	742^{b}	40	761^{b}	41	780^{b}
42	799^{b}	43	818^{b}	44	837^{b}
48	856^{2}	49	877^{2}	144	902^{2}
150	904^{2}	156	905^{2}	162	906^{2}
167	940^{2}	175	942^{2}	182	943^{2}
189	944^{2}	200	1017^{2}	208	1018^{2}
216	1019^{2}	575	1024^{2}	600	1026^{2}

Covering array bounds for $g = 21$.

7	441^{o}	24	524^{o}	25	$526^{\downarrow}$
26	527^{p}	27	$618^{\downarrow}$	28	$619^{\downarrow}$
29	$620^{\downarrow}$	30	621^{p}	31	$720^{\downarrow}$
32	$721^{\downarrow}$	33	$722^{\downarrow}$	34	723^{p}
35	$830^{\downarrow}$	36	$831^{\downarrow}$	37	$832^{\downarrow}$
38	833^{p}	42	841^{b}	48	861^{3}
49	881^{2}	167	944^{3}	175	946^{2}
182	947^{2}	575	1027^{2}	600	1029^{2}
624	1030^{2}	625	1031^{2}	650	1032^{2}
676	1033^{2}	696	1123^{2}	720	1124^{2}
728	1125^{2}	754	1126^{2}	780	1127^{2}
784	1217^{2}	812	1218^{2}	841	1219^{2}

Covering array bounds for $g = 22$.

5	484^{o}	24	526^{o}	25	528^{p}
26	618^{o}	27	$620^{\downarrow}$	28	$621^{\downarrow}$
29	622^{p}	30	$721^{\downarrow}$	31	$722^{\downarrow}$
32	$723^{\downarrow}$	33	724^{p}	34	$831^{\downarrow}$
35	$832^{\downarrow}$	36	$833^{\downarrow}$	37	834^{p}
45	946^{b}	46	967^{b}	120	988^{2}
125	990^{2}	575	1030^{2}	600	1032^{2}
625	1034^{2}	650	1124^{2}	672	1125^{2}
696	1126^{2}	700	1127^{2}	725	1128^{2}
728	1217^{2}	754	1218^{2}	756	1219^{2}
784	1220^{2}	812	1221^{2}	841	1222^{2}
858	1320^{2}	870	1321^{2}	899	1322^{2}

Covering array bounds for $g = 23$.

24	529^{o}	26	620^{o}	27	$622^{\downarrow}$
28	623^{p}	30	$723^{\downarrow}$	31	$724^{\downarrow}$
32	725^{p}	34	$833^{\downarrow}$	35	$834^{\downarrow}$
36	835^{p}	38	$951^{\downarrow}$	39	$952^{\downarrow}$
40	953^{p}	41	$1014^{\downarrow}$	42	1015^{p}
575	1035^{3}	576	1057^{3}	623	1126^{3}
648	1128^{3}	672	1129^{2}	675	1217^{2}
702	1219^{2}	728	1220^{2}	729	1221^{2}
756	1222^{2}	784	1223^{2}	806	1321^{2}
832	1322^{2}	840	1323^{2}	868	1324^{2}
896	1325^{2}	900	1423^{2}	930	1424^{2}
961	1425^{2}	992	1426^{2}	1024	1427^{2}

Covering array bounds for $g = 24$.

9	576^{o}	26	622^{o}	27	624^{p}
30	$725^{\downarrow}$	31	726^{p}	34	$835^{\downarrow}$
35	836^{p}	38	$953^{\downarrow}$	39	954^{p}
40	$1015^{\downarrow}$	41	1016^{p}	80	1128^{3}
81	1151^{2}	233	1174^{3}	243	1176^{2}
675	1220^{2}	702	1222^{2}	729	1224^{2}
780	1323^{2}	806	1324^{2}	810	1325^{2}
837	1326^{2}	900	1426^{2}	930	1427^{2}
961	1428^{2}	1020	1536^{2}	1054	1537^{2}
1085	1538^{2}	1107	1616^{2}	1156	1646^{2}
1190	1647^{2}	1225	1648^{2}	1240	1717^{2}
1271	1718^{2}	2072	1726^{3}	2187	1728^{3}

Covering array bounds for $g = 25$.

26	625^o	30	727^p	34	837^p
38	955^p	40	1017^p	675	1225^3
676	1249^3	780	1327^2	900	1429^2
1020	1539^2	1040	1617^2	1156	1649^2
1200	1719^2	1292	1767^2	17525	1825^3
17550	1849^3	20000	1927^2		

Bibliography

[1] K. Bush. "Orthogonal Arrays of Index Unity", Annals of Mathematical Statistics, 1952, 23: p. 426-434.

[2] J.N. Cawse (ed.). Experimental Design for Combinatorial and High Throughput Materials Development, John Wiley & Sons, New York, 2003.

[3] M. Chateauneuf and D.L. Kreher. "On the state of strength three covering arrays." J. Comb. Designs, 2002, 10(4), p.217-238.

[4] C. Cheng, A. Dumitrescu and P. Schroeder. "Generating small combinatorial test suites to cover input-output relationships", Proc. of the Third Intern. Conf. on Quality Software (QSIC 03), 2003, p. 76-82.

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

[6] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. "The AETG system: an approach to testing based on combinatorial design", IEEE Trans. on Software Engineering, 1997, 23, p. 437-444.

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

[8] D.M. Cohen and M.L. Fredman. "New techniques for designing qualitatively independent systems", J. Combin. Des., 1998, 6(6): 411-16.

[9] M.B. Cohen. "Designing test suites for software interaction testing", PhD thesis, University of Auckland, 2004.

[10] C.J. Colbourn. "Strength two covering arrays: Existence tables and Projection", preprint (July 2006).

[11] C.J. Colbourn. "Combinatorial aspects of covering arrays", Le Matematiche (Catania), 2004, 58, p. 121-167.

[12] C.J. Colbourn and J.H. Dinitz, "Making the MOLS table", In: Computational and Constructive Design Theory (W.D.Wallis, ed.) Kluwer Academic Press, 1996, p. 67-134.

[13] C.J. Colbourn and J.H. Dinitz. "The CRC handbook of combinatorial designs", CRC Press, Boca Raton, 1996.

[14] C. J. Colbourn, J.H. Dinitz, D.R.Stinson, "Applications of Combinatorial Designs to Communications, Cryptography, and Networking", Surveys in Combinatorics, 1999, p. 37-100.

[15] C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. E. Shasha, G. B. Sherwood, and J. L. Yucas, " Products of mixed covering arrays of strength two", J. Combin. Des., 2006, 14, p. 124-138.

[16] S.R. Dalal, A. Jain, G.C. Patton, M. Rathi, and P. Seymour. "AETG Web: A web-based service for automatic efficient test generation from functional requirements", Proc. 2nd IEEE Worksh. on Industrial Strength Formal Specif. Techn., IEEE Press, 1998, p. 84-85.

[17] S.R. Dalal and C.L. Mallows. "Factor-covering designs for testing software", Technometrics, 1998, 40, p. 234-243.

[18] P. Erdos, C. Ko and R. Rado. "Intersection theorems for systems of finite sets", Quart. J. Math, Oxford Series, 1961, 12(2), p.313-320.

[19] L. Gargano, J. Korner and U. Vaccaro. "Sperner Capacities", Graphs Combin., 1993, 9(1), p. 31-46.

[20] L. Gargano, J. Korner and U. Vaccaro. "Capacities: from information to extremal set theory," J. Combin. Theory, 1994, (A)68: p. 296-316.

[21] F. Glover. "Tabu Search", ORSA, J. Comput., 1989, 1, p.190-206.

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

[23] R. Greenlaw and R. Petreschi. "Cubic Graphs", ACM Computing Surveys,1995, 27(4): 471-495.

[24] A. Hartman and L. Raskin. "Problems and algorithms for covering arrays", Discrete Math., 2004, 284, p. 149-156.

[25] A.S. Hedayat, N.J.A. Sloane and J. Stufken. "Orthogonal Arrays", Theory and Applications, Springer, 1999.

[26] G. Katona. "Two applications(for search theory and truth functions) of Sperner type theorems. Periodica Math., 1973, 3, p. 19-26.

[27] D. Kleitman and J. Spencer. "Families of k-independent sets", Discrete Math, 1973, 6, p. 255-262.

[28] D.L. Kreher and D.R. Stinson. "Combinatorial algorithms: Generation, Enumeration and Search", CRC Press, Boca Raton, 2000.

[29] K. Meagher, "Covering Arrays on Graphs: Qualitative Independence Graphs and Extremal Set Partition Theory", PhD thesis, University of Ottawa, Ottawa, 2005.

[30] K. Meagher, L. Moura and L. Zekaoui, "Mixed Covering Arrays on Graphs", J. Combin. Des., to appear (submitted Nov./2005).

[31] K. Meagher and B. Stevens, "Covering Arrays on Graphs," J. Combin. Theory. Ser., 2005, B 95, p. 134-151.

[32] K. Meagher and B. Stevens. "Group Construction of covering arrays", J. Combin. Des., 2005, 13, p.70-77.

[33] L. Moura, J. Stardom, B. Stevens and A. Williams, "Covering arrays with mixed alphabet sizes," J. Combin. Des., 2003, 11, p. 413-432.

[34] L. Moura, B. Stevens, E. Mendelsohn. "Lower bounds for transversal covers", Design Codes and Cryptography, 1998, 15(3), p.279-299.

[35] K.J. Nurmela. "Upper Bounds for covering arrays by Tabu Search", Discrete Applied Math., 2004, 138, p.143-152.

[36] S. Poljak and Z. Tuza. "On the maximum number of qualitatively independent partitions and related problems," J. Combin. Theory, 1989,(A)51, p. 111-116.

[37] R.K. Roy. "Design of Experiments using the taguchi approach", John Wiley & sons, Inc, New York, 2001.

[38] G. Seroussi and N.H. Bshouty. "Vector sets for exhaustive testing of logic circuits", IEEE Trans. on Infor. Theory, 1988, 34, p. 513-522.

[39] D.E. Shasha, A.Y. Kouranov, L.V. Lejay, M.F. Chou, and G.M. Coruzzi. "Using combinatorial design to study regulation by multiple input signals: a tool for parsimony in the post-genomics era", Plant Physiology, 2001, 127, p. 1590-1594.

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

[41] B. Stevens and E. Mendelsohn. "New recursive methods for transversal covers", J. Combin. Des., 1999, 7(3): 185-203.

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

[43] D. Stinson. "Combinatorial Designs: Constructions and Analysis", Springer Verlag, 2003.

[44] K.C Tai, and L. Yu. "A test generation strategy for pairwise testing", IEEE transactions of Software Engineering, 2002, 28, p.109-111.

[45] A.W. Williams and R.L. Probert. "A practical strategy for testing pair-wise coverage of network interfaces", Proc. Seventh Intern. Symp. on Software Reliability Engineering, 1996, p. 246-54.

[46] A.W. Williams. "Determination of test configurations for pair-wise interaction coverage", In thirteenth Int. Conf. Testing Communication Systems, 2000, p.57-74.

[47] A.W. Williams. "Software Component Interaction Testing", PhD thesis, University of Ottawa, Ottawa, 2002.

[48] L. Yu and K.C. Tai. "In parameter-order: a test generation strategy for pairwise testing", In Proc Third IEEE Intl. High-Assurance Systems Engineering Symp, 1998, p.254-261.

[49] T. Yu-wen, and W.S. Aldwin. "Automating test case generation for the new generation mission software system", In Proc. IEEE Aerospace Conf., 2000, p.431-437.