Metaheuristics and the Search for Covering and Packing Arrays

by

John Stardom

B.Sc. Hons., Trent University, 1998

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Mathematics

SIMON FRASER UNIVERSITY
May 16, 2001

Abstract

A strength 2 covering array (packing array) is an array with k columns and entries from a g-ary alphabet such that given any two columns i and j and any ordered pairs of elements (g_1, g_2) from the g-ary alphabet, there exists at least (at most) one row r such that $a_{ri} = g_1$ and $a_{rj} = g_2$. The problem of interest is to determine the minimum (maximum) number b of rows for which a $b \times k$ covering (packing) array with entries from the g-ary alphabet exists, where k and g are given.

Upper bounds on the size of covering arrays are constantly being improved through new constructions. Randomized searches can also be used to find new arrays through the use of metaheuristics, which are widely applicable to a number of combinatorial problems. It is possible that some randomized search algorithms are more suited to finding better bounds on the sizes of covering and packing designs than others.

We search for covering and packing arrays using simulated annealing, tabu and genetic algorithms. We compare both the differences between these three search techniques, as well as their effectiveness in constructing covering and packing arrays. We determine that the algorithms best used to find quality covering and packing arrays are the simulated annealing algorithm and the tabu search algorithm. We give tables of the best known bounds on the sizes of covering and packing arrays, including those bounds improved through our own metaheuristic searches.

The primary application of covering arrays is that of software and network testing. In order to construct arrays which are fully applicable to software and network tests of any flavour, it is necessary to consider arrays where each column possesses its own alphabet size. We present some initial results for this problem, including a complete solution for the case $k = 4$.

Acknowledgements

While my name is the only one appearing on the cover of this thesis, it was truly written by many hands. Firstly, I would like to thank Brett Stevens, whose mentorship and friendship were both invaluable throughout the writing of this thesis. His stubborn optimism, enthusiasm, support and regard for my sanity made this whole process a lot less painful than it could have been. I believe he will be a valued educator.

I would also like to acknowledge the love and support I have had the honour of receiving from my wife and best friend, Holly Morrison. Her encouragement was instrumental in my being able to finish on time. I promise I will never wed while writing a thesis again.

I also need to thank my family, Eleanor, Richard and Murray Stardom, Ed Rea and Karan Moore. Their love, patience and guidance throughout this crazy adventure have saved me many times. An added special thanks to Kerry Colpitts, who put out innumerable fires over the past three years. Thank you.

I would also like to thank Luis Goddyn, for introducing me to programming, and Karen Meagher, for the countless chats and helping me get my feet off the *C++* ground.

In writing this thesis, I had to communicate with many people to check facts and explore ideas. I would like to thank Mark Chateauneuf, Peter Gibbons and Kari Nurmela for their assistance. I would also like to thank Lucia Moura for going out of her way to be my external supervisor.

I would also like to extend my thanks to all of the educators who have nurtured and stimulated my passion for mathematics. In particular, I would like to thank David Poole (you were right, design theory is beautiful), Stefan Bilaniuk and Stephen Khan.

Finally, I would like to acknowledge the support of NSERC. Without the assistance of NSERC, my time at Simon Fraser University would have been much less enriching.

Contents

Chapter 1

Introduction

There has been much research over the past decade regarding the existence of covering arrays and, in particular, bounds on the size of said arrays. Many constructions, both recursive and otherwise, have been devised to produce these design-like objects. While many of these constructions have been used to produce arrays with small parameter values, fairly little is known with regards to the strength of the bounds on the size of arrays with slightly larger parameter values. A recent trend in searching for these covering arrays has been to implement searching algorithms which would build the arrays by a stochastic process. In this thesis, we implement three randomized search algorithms in the hope of finding better bounds on arrays with larger parameter sizes.

Another similar problem is that of determining bounds on the size of packing arrays. Packing arrays are a sort of complementary structure to covering arrays. While a lower bound exists on the number of times pairs of elements can occur in a covering array, a packing array places an upper bound on this quantity. As the applications of packing arrays are subtler than those of covering arrays, these structures have been studied less actively. Consequently, fewer good bounds on the size of these arrays are known, and less constructions of packing arrays have been discovered.

1.1 Definitions and Notation

In order to define the objects of study, we must first present some other definitions and notation. Henceforth, when reference is made to a g-set or an alphabet of size g, it shall be assumed that this set is Z_g, the set of integers $\{0, 1, \ldots, g-1\}$. Furthermore, unless specified otherwise, all variables mentioned shall only take integer values.

The majority of tools used to construct optimal covering and packing arrays, as well as establish results about them stem from the study of designs. We first define some basic design theoretic structures and then generalize these structures into covering and packing arrays. The definitions presented can be found in [5, 24].

Definition 1.1. *A* **Latin square of side** g *is a* $g \times g$ *array in which each cell contains a single element from a* g*-set such that each element of the set occurs exactly once in each row and exactly once in each column. The entry in row* a *and column* b *of Latin square* L *is denoted* $L(a, b)$*. A Latin square of side* g *is said to be* **idempotent** *if for all* $0 \leq i \leq g-1$*, cell* (i, i) *of the Latin square contains the symbol* i*.*

Definition 1.2. *A pair of Latin squares of side* g*,* L_0 *and* L_1 *are said to be* **orthogonal** *if* $L_0(a, b) = L_0(c, d)$ *and* $L_1(a, b) = L_1(c, d)$ *implies* $a = c$ *and* $b = d$*. In other words,* $L_0 = (a_{i,j})$ *and* $L_1 = (b_{i,j})$ *are orthogonal if every element in* $Z_g \times Z_g$ *occurs exactly once among the* g^2 *pairs* $(a_{i,j}), (b_{i,j})$ *for all* $0 \leq i, j \leq g-1$*. A set of Latin squares* $L_1, L_2, \ldots, L_m$ *is a set of* **mutually orthogonal** *Latin squares, or a set of* **MOLS** *if for every* $1 \leq i < j \leq m$*,* L_i *and* L_j *are orthogonal. We let* $N(g)$ *denote the maximum number of Latin squares in a set of MOLS of side* g*.*

Definition 1.3. *An* **orthogonal array** $OA(k, g)$ *is a* $g^2 \times k$ *array with entries from a* g*-set having the property that in any two columns, each ordered pair of symbols from the* g*-set occurs exactly once.*

Orthogonal arrays and sets of mutually orthogonal Latin squares are closely related. It is well known that a set of $k-2$ MOLS of side g is equivalent to an $OA(k, g)$. Consider the set of $k-2$ MOLS $\{L_1, L_2, \ldots, L_{k-2}\}$ of side g. The $g^2 \times k$ array formed by the g^2 rows $(i, j, L_1(i, j), L_2(i, j), \ldots, L_{k-2}(i, j))$ for $0 \leq i, j \leq g-1$ is an orthogonal array. Figure 1.1 shows a pair of orthogonal Latin squares of side 3 and the

0	1	2
1	2	0
2	0	1

0	1	2
2	0	1
1	2	0

0000
0111
0222
1012
1120
1201
2021
2102
2210

Figure 1.1: A pair of orthogonal Latin squares of side 3 and the corresponding $OA(4,3)$.

corresponding orthogonal array formed by the above construction.

If the set of MOLS used to build an $OA(k,g)$ in this way are all idempotent, the orthogonal array will have g disjoint rows. In Chapter 4, it shall be shown how orthogonal arrays are used to produce good bounds for covering arrays. Orthogonal arrays will also be used in Chapter 5 to construct portions of covering arrays having special alphabet restrictions.

Definition 1.4. *A* **transversal design of order g and blocksize k**, *denoted* $TD(k,g)$, *is a triple* (V,G,B), *where*

- V *is a set of* kg *elements;*
- G *is a partition of* V *into* k *groups, each of size* g*;*
- B *is a collection of* k*-subsets of* V *called blocks;*
- *each block intersects each group* G_i *in exactly one point; and*
- *each pair of points not in the same group occurs in exactly one block.*

A $TD(k,g)$ is also equivalent to a set of $k-2$ MOLS of side g and can be constructed from an $OA(k,g)$ as follows. Let A be an $OA(k,g)$ on the symbol set Z_g. Taking V to be the set $Z_g \times Z_k$ let B be the set of blocks $\{(a_{i,j},j) : 0 \leq j \leq k-1\}$ for all $i \in Z_{g^2}$,

where $a_{i,j}$ is the element in row i and column j of A. Then, if G is the partition of V whose classes are $\{Z_g \times \{j\} : 0 \leq j \leq k-1\}$, the triple (V, G, B) is a transversal design. The blocks and groups of a transversal design derived in this way from the $OA(4,3)$ in Figure 1.1 are presented in Figure 1.2.

Blocks:	$\{00,01,02,03\}$	$\{00,11,12,13\}$	$\{00,21,22,23\}$
	$\{10,01,12,23\}$	$\{10,11,22,03\}$	$\{10,21,02,13\}$
	$\{20,01,22,13\}$	$\{20,11,02,23\}$	$\{20,21,12,03\}$
Groups:	$\{00,10,20\}$	$\{01,11,21\}$	$\{02,12,22\}$

Figure 1.2: A $TD(4,3)$ derived from an $OA(4,3)$.

Definition 1.5. *An* **incomplete transversal design**, *denoted* $ITD(k, g; b_1, b_2, \ldots, b_s)$ *with* $b_i \geq 0, \sum_{i=1}^{s} b_i \leq g$ *is a quadruple* (V, G, H, B) *where*

- V, G *and* B *are as defined in Definition 1.4;*
- H *is a set of disjoint subsets* $H_1, H_2, \ldots, H_s$ *of* V *called* **holes** *with the property that for each* $1 \leq i \leq s$ *and each group* $G' \in G$, $|G' \cap H_i| = b_i$;
- *every unordered pair of elements from* V *is contained in either a hole or a group and contained in no blocks, or contained in exactly one block, but not in any group or hole.*

A construction mentioned in Chapter 5 uses ITDs to build a very specific class of covering arrays.

1.2 Covering and Packing Arrays

We shall now define covering arrays and packing arrays by first presenting some equivalent objects.

Definition 1.6. *A* **strength-t transversal cover**, *denoted* $t\text{-}TC(k, g : n)$ *is a triple* (V, G, B) *where*

Groups:	$\{0,1\}$	Covering Array		
	$\{2,3\}$			
	$\{4,5\}$			
Blocks:	$\{0,2,4\}$	0	0	0
	$\{0,3,5\}$	0	1	1
	$\{1,2,4\}$	1	0	0
	$\{1,2,5\}$	1	0	1
	$\{1,3,4\}$	1	1	0

Figure 1.3: A $2-TC(3,2)$ with $V = Z_6$ and the corresponding $CA(3,2:2)$.

- *V, G and B are as defined in Definition 1.4;*
- *each block intersects each group G_i in exactly one point;*
- *each t-set of points with no two in the same G_i occurs in* **at least** *one block; and*
- *there is a set of n pairwise disjoint blocks in B.*

A transversal cover is essentially a transversal design with a relaxed pair requirement. A transversal cover with $k = 3$ and $g = 2$ is presented in Figure 1.3. These design-like structures can be formulated in a much more aesthetic way by letting the blocks be the rows of a $b \times k$ array, where $|B| = b$. Then, by replacing the contents of each group by the same g-ary alphabet we have an array with entries from a uniform alphabet. This new structure is called a covering array. In defining covering and packing arrays, we first define the concept of disjoint rows.

Definition 1.7. *In any covering or packing array, any set of rows which pairwise differ in each column is a set of* **disjoint rows**.

Definition 1.8. *A* **strength-t covering array**, *denoted t-$CA(k,g:n)$ is a $b \times k$ array with entries from a g-ary alphabet such that given any t columns, $c_1, c_2, \ldots, c_t$, and for all ordered t-sets of elements $(g_1, g_2, \ldots, g_t)$ from the alphabet, there exists a row r such that $a_{r,c_i} = g_i$ for all $1 \leq i \leq t$. Furthermore, there is a set of at least*

n disjoint rows. The smallest number of rows possible is denoted t-ca$(k, g : n)$. For $n = 1$, this quantity may simply be denoted by t-ca(k, g).

The covering property of a covering array is implied by that of a transversal cover. The abovementioned covering array construction can also be reversed to transform a covering array into a transversal cover.

Definition 1.9. *A* **strength-t transversal packing**, *denoted t-TP$(k, g : n)$ is a triple (V, G, B) where all entities are as defined in Definition 1.6, except that each each t-set of points, no two of which are in the same G_i, occurs in no more than one block.*

Definition 1.10. *A* **strength-t packing array**, *denoted t-PA$(k, g : n)$ is a $b \times k$ array with entries from a g-ary alphabet (typically Z_g) such that given any t columns, $c_1, c_2, \ldots, c_t$, and for all ordered t-sets of elements $(g_1, g_2, \ldots, g_t)$ from the alphabet, there exists at most one row r such that $a_{r,c_i} = g_i$ for all $1 \leq i \leq t$. Furthermore, there is a set of at least n disjoint rows. The largest number of rows possible is denoted t-pa$(k, g : n)$. For $n = 1$, this quantity may be denoted t-pa(k, g).*

Throughout this thesis, we will only be concerned with structures covering pairs of elements, or t-sets of elements with $t = 2$. As a result, when reference is made to a packing or covering array throughout this thesis, it shall be assumed that $t = 2$ and the t shall be dropped from all notation unless specified otherwise.

Finally, we present a third structure equivalent to a covering arrray.

Definition 1.11. *Let a g-partition of a b-set B denote a partition of B into g classes. A family of k g-partitions, $P_1, P_2, \ldots P_k$, of a b-set B are* **qualitatively t-independent** *if whenever one selects t distinct partitions from the family and one part from each, then the intersection of these parts is nonempty.*

Consider a covering array. Let the set of rows be our b-set, labelled $\{0, 1, \ldots, b-1\}$. Then each column partitions this set into g classes, where all rows with entry i are in class i, for $0 \leq i \leq g - 1$. If we select any t columns from the array and one class from each, as defined above, then the t-set-covering condition guarantees that

the intersection of these parts is non-empty. Therefore, a strength-t covering array is equivalent to a set of t-independent g-partitions of a b-set.

1.3 Applications

The primary application of covering arrays is that of the design of experiments. If the rows of a covering array correspond to experiment trials and its columns to the test subjects, then a covering array is equivalent to a test suite in which the interaction between each pair of test subjects may be observed. Recently, this method of testing has been applied in testing software and networks [3, 4, 6, 23, 24, 29]. It is reported in [6] that anywhere from a third to a half of the total cost of software development is due to the software testing. Not only is there a financial cost in performing an extensive amount of testing, but completing the testing may also require a great deal of time. When testing anything, it is desirable to approximate a balance between the quality of the test and the possible costs incurred.

For software testing, the model most often used is that of a computer program having k variables, which are all discrete and can each take one of g values at any time. To ensure that the code is fully functional in an exhaustive fashion, one would want to test every possible ordered set of values as input for the variables. Unfortunately, as g and k grow even moderately large, this process quickly becomes infeasible as the exhaustive set of g^k tests could take a lot of time and money to complete.

In [3], Cohen et al. claim from empirical data that the majority of code errors are due to either the interaction between a pair of values, or faults found in a single parameter. This finding suggests that to test software or a network sufficiently well, one needs only ensure that between any two variables in the code or nodes in the network, all ordered pairs of possible states are tested. In order to get a more comprehensive test of the system, test suites that cover every t-set of variables for $t \geq 3$ may be used.

Consider the case of employees in a call centre who are to call alumni for support in an upcoming funding drive. A computer program can be used to manage the distribution of phone numbers prior to presenting them to the caller and may even adapt the script the caller must read, according to some criteria. For example, the

caller might want to know if the respondent has contributed money in the past, if the respondent has been contacted before or even if they have only recently graduated. If these are the only three variables of concern, then a test for the program requires that between any two pairs of columns, every possible ordered pair of values is present. In Figure 1.4, we present a sample set of runs which may serve as a sufficiently good test for this program.

Previously contributed?	Contacted this year?	Graduated within last year?
No	No	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No

Figure 1.4: A sample set of test cases for the call centre problem.

Notice that this set of test cases is isomorphic to the covering array and transversal cover of Figure 1.3. Presumably, applying this set of test cases should determine any existing errors in the code. However, the primary question with regards to covering arrays is to determine the minimum number of rows required to form a covering array given the integers k and g. Such covering arrays would generate the minimum number of test cases required to test an instance of software or a network sufficiently well for failure. It is known that for $k = 3$ and $g = 2$, a covering array with four rows exists. Our test set in Figure 1.4 can be made optimal by removing the third row, which is entirely redundant with regards to pair coverage.

For small k and g, current bounds on the size of a covering array are quite good and are fairly difficult to meet, let alone beat, by simply constructing an array by hand. One concern is that many computer programs and networks require that k and g be quite large. Many of the constructions designed to construct covering arrays could generate a corresponding $CA(k, g)$ with a reasonably small, but not minimal, number of rows. It is for this reason that we had hoped that randomized search methods could be used to obtain even better bounds for these values of k and g.

Another concern that shall be addressed in Chapter 5 is that it is rarely the case that each variable in a program has the same number of possible states. The concept of a covering array can be generalized to handle these cases and even less is known about bounds on these special objects.

Another application of the covering array is in the compression of inconsistent or contradictory data. In [14], Körner and Lucertini claim that Shannon's contributions to the field of information theory cannot deal with many real world situations. In particular, they propose that the results in Shannon theory regarding multiterminal information sources cannot interpret data inconsistency mathematically. They consider the case of simultaneous observations of some occurrence or system such that the observations from any one location are fragmentary and may contradict either other observations or what is already known about the system. The goal is to obtain a count of the minimal number of full and consistent descriptions of the system such that each fragmentary observation contributes to at least one of them. This quantity is referred to as the maximum achievable compression of the data.

The model of Körner and Lucertini considers a finite set X as the set of attributes of the system which can be observed. Then, they assign to each observer a function whose domain is X. They further define a function f as being **extendable** if there is some function g such that the domain of f is a subset of the domain of g and both functions coincide on the domain of f. Therefore, two observers reporting consistent data have corresponding functions which have a common extension. If F represents the family of functions corresponding to the observations made, then the problem is to determine the minimum cardinality of a family of functions G such that for every function in F, there is some function in G which extends it. Stevens [24] presents a simplified version of the problem in which the observation fragments are all possible t-sets of observation values. Let the set of observation points correspond to the columns of the array. Then an optimal strength t covering array would yield the desired minimal set of observations in which every set of fragmentary observations corresponding to the functions of F occurs, regardless of their consistency.

Covering arrays are used in many other capacities. Katona [12] applies covering arrays in the guise of qualitatively independent sets to multivariate truth functions

and search theory, while Sloane [23] links strength 3 covering arrays to intersecting codes. Chateauneuf and Kreher [2] and Stevens [24] list references to other uses, such as computer architecture design, drug screening, block ciphers, and zero-error noisy channel communication.

The applications of packing arrays are not as obvious as those of covering arrays. Stevens [24] writes that the rows of an optimal packing array "form the maximal set of words from a partial maximum distance separable code (MDS code) with minimum spanning distance $k-1$". These codes have been studied and Stevens notes that Abdel-Ghaffar and Abbadi use packing arrays to store large files across multiple hard disk systems so that the files can be retrieved in the shortest time possible. Stevens also presents references to some of the known upper bounds, such as the Plotkin, Elias and Hamming bounds. A weak version of the Plotkin bound is

$$pa(k, g : 1) \leq \frac{g(k-1)}{k-g}, \text{ where } k > g$$

We use this bound during the randomized searches as an upper bound on the binary search process, as discussed in the second section of Chapter 2.

The dual of packing arrays are highly structured resolvable block designs. In particular, the dual of a $PA(k, g : 1)$ with no pair of disjoint rows is a **class-uniformly resolvable design**, which are used to construct round-robin tournaments and design experiments efficiently [7, 16]. Packing arrays, seen as MDS codes, can also be used to detect and correct transmission errors.

1.4 Outline of Thesis

In the first part of this thesis, we present a survey of metaheuristic search techniques and select three such algorithms to attempt to improve on some of the best known bounds on the size of covering and packing arrays. In Chapter 2, a brief survey of heuristics is provided, together with a complete description of the code implemented, including considerations from the literature and a qualitative comparison of the algorithms.

In Chapter 3, we conduct preliminary tests and use the results to select appropriate input parameters for the algorithms. We also compare the effectiveness of each algorithm, based on the results of these tests.

Once we ascertain the capability of each algorithm to find good covering and packing arrays, we use the more effective of the three algorithms to improve some of the bounds on these objects. Tables of results are also presented.

A brief history of the problem is presented at the beginning of Chapter 4. Theoretical and computational methods are presented and compared to the techniques employed in this thesis. All covering and packing array bounds improved by the implemented metaheuristics are presented in sections 4.2 and 4.3, respectively.

Finally, in Chapter 5, we consider the problem of determining the minimum number of rows possible for a covering array in which the alphabet size can differ between columns. We present a solution for the case $k = 4$ and an initial set of results for the case $k = 5$.

Chapter 2

Metaheuristics and Designs

The vast majority of upper bounds established on the value of $ca(k, g : 1)$ for any k and g have been determined by recursive constructions based on smaller covering arrays. Typically, these small covering arrays are either produced by hand or derived from tables of existing transversal designs. For smaller values of b, k and g, the amount of time required to manually construct these $b \times k$ arrays with entries from Z_g while continuously ensuring that all required ordered pairs are present may be reasonable. However, for larger values, it becomes infeasible to perform these constructions by hand. Designed to perform repeated operations quickly, computers are the natural means by which this expense of time can be avoided.

2.1 Algorithms and Heuristics

Given a certain set of parameters and constraints, the set of all objects, or **states**, which can be formed based on this information constitutes a **search space** in which a feasible object is sought. With covering and packing arrays, every state in a search space for fixed b, k and g is a $b \times k$ array with entries from Z_g. A state is **feasible** if it satisfies the appropriate covering or packing criteria. The algorithms one can use to propagate through a search space are divided into two categories: exhaustive search algorithms and randomized search algorithms. Algorithms which cover a search space exhaustively, such as backtracking and branch-and-bound algorithms, are the

only algorithms which are sure to return either a feasible state or a proof of its nonexistence. However, the time needed to examine every object in larger search spaces may increase to a point which renders an exhaustive search infeasible. For these larger search spaces, randomized search techniques become useful [15].

Randomized search algorithms proceed through a given search space in a random fashion, examining only as many objects as specified by the user. This enables the user to explore the space as thoroughly as desired, but offers no proof regarding the nonexistence of a solution. Clearly, a purely random examination of the states in any space will only return a feasible state with probability corresponding to the percentage of states in the search space possessing the desired criteria. This probability can be increased by supplying the algorithm with decision rules, or **heuristics**, to help it search through the space more efficiently. Typically, given a state in the search space, a randomized search algorithm first makes a small modification, or a **move**, to the object being examined to proceed to an adjacent state in the space. The heuristic is used to help the algorithm choose exactly which adjacent state is chosen as the new state of study. 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 **neighbourhood** of x, denoted by $N(x)$. By repeating this process for a succession of objects, a randomized search algorithm is capable of examining a vast number of states in the space.

In the case of covering and packing arrays, if positive integers b, k and g are provided, then the set of all $b \times k$ arrays with entries from Z_g makes up the search space of interest in which one attempts to find a feasible array with b rows. The natural move one can use is to select one row and one column at random from the array being examined, say row i and column j, and then to randomly change entry (i, j) to a different element of Z_g. Thus, the neighborhood of any array A is the set of all arrays that can be obtained from A by randomly changing some entry a to an element of $Z_g \setminus \{a\}$. Equipped with this move, the only ingredient missing in order to assemble a randomized search algorithm is a decision rule for moving from object to object in the search space.

Search algorithms, both randomized and exhaustive, typically run until either an

optimal state has been located, or a predetermined bound M on the number of state changes has been exceeded. As it is more desirable to implement faster algorithms, it is important to be able to locate any existing optimal states as quickly as possible. There exists a class of algorithms which employ an evaluation function to assign value or **fitness** to the objects they encounter, in order to converge to an optimal state in the search space more rapidly. For a potential covering array, fitness is measured by a count of the number of ordered pairs absent from the set of all possible pairs of columns in the array. Therefore, a covering array will have no ordered pairs mising from any selected pair of columns, or a fitness value of zero. The corresponding fitness of a potential packing array is measured by the count of the number of pairs occurring more than once in any possible pair of columns. Therefore, a covering array will have no pair of columns in which any ordered pair occurs more than once.

The heuristic typically adopted for these fitness-oriented algorithms at any given state is one which searches through the neighborhood of the current state by applying a move to the object [15]. The object obtained is then selected as the new state if it is more fit than the current object. If one visualizes the search space to be a landscape, where the elevation at any state corresponds to the state's fitness, then moving from state to state using this heuristic gives the impression of always climbing in elevation. As a result, this class of greedy algorithms is referred to as **hill-climbing algorithms**. It should be noted that in the case of packing and covering arrays, this is slightly counter-intuitive as the more desirable arrays have lower scores. This can of course be remedied by simply making the fitness equal to the number of pairs covered.

At any step of a hill-climbing algorithm, the neighbourhood of a state can be searched randomly or exhaustively, each of which leads to a different walk through the search space. Hill-climbing algorithms which search exhaustively through a state's neighbourhood at every step are referred to as **steepest ascent algorithms**, as they converge to the most fit state accessible by greedy progression from the starting state. As a result, the success of steepest ascent algorithms is determined solely by the state chosen as the starting point for the search. Ideally, one might want to use a heuristic which is less restrictive with regards to fitness acceptability in order to escape from local optima which are not global optima. Such hill-climbing algorithms are very

useful in that they search a wider segment of the space before eventually converging at an optimal state. Gibbons presents a thorough survey of randomized hill-climbing algorithms in [5].

The simulated annealing, tabu search and genetic 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. For this reason, these three algorithms are often referred to as **metaheuristics** [8]. These three algorithms were employed to locate the smallest covering arrays possible, for various values of k and g. The implementation details of the main program as well as the three chosen metaheuristics are outlined below. Considerations made to implementation suggestions found in literature are discussed and a qualitative comparison of the algorithms is made. All three of the algorithms written were based on skeletal pseudocode presented in [15].

2.2 Implementation Details

The search code was written to take in a series of input parameters and then to use a binary search method to attempt to locate covering or packing arrays of the proposed dimensions. The user must supply the name of the metaheuristic to be employed, either ANNEAL, TABU or GENETIC, followed by a suggested upper bound on the number of rows in the array, b_0, the number of columns in the array, k, and the size of the alphabet, g, used to fill the array. The parameter b_0 acts as a lower bound on the number of rows in the array when searching for a packing array. Next, the user may input a lower bound on the number, n, of disjoint rows to be fixed in the sought array. One construction mentioned in Chapter 4.1 requires that the number of disjoint rows in an array is known. It should be noted that for $n > 1$, the size of any array's neighbourhood is effectively decreased as the fixed disjoint rows are left unaffected by moves. Then, an upper bound, M, on the number of times an algorithm can run through its main loop is provided, in order to let the algorithm exit if it is unable to locate a design. Finally, the user must supply all input parameters required by the metaheuristic being called upon to perform the search.

When searching for covering arrays, the specified metaheuristic starts by looking for a $CA(k, g : n)$ with b rows, where b is the integer closest to the halfway point between proposed upper bound b_0 and the guaranteed lower bound, g^2. Should a covering array be found, the current value of b becomes the new upper bound and the algorithm begins to search for a $CA(k, g : n)$ with a number of rows equal to the midpoint of b and g^2. Otherwise, b becomes the new lower bound and the algorithm begins to search for a $CA(k, g : n)$ with a number of rows equal to the midpoint of b and b_0. The binary search continues until the new value chosen for b is equal to the value for which a search just concluded. When searching for packing arrays, the same process is performed with the roles of the upper and lower bounds being reversed. Also, the original upper bound on the number of rows in a packing array is taken to be the appropriate value of the Plotkin Bound (see Chapter 1.3) for g strictly less than k, and is taken to be g^2 otherwise.

Given integers k and g, it was uncertain to what degree the existing bounds on the number of rows in the corresponding covering or packing array could be improved upon. The binary search method allows an efficient exploration of search spaces for many values of b, thereby reducing the amount of time spent searching for arrays that are more difficult to find or do not exist at all.

In writing the program, it was necessary to construct a data structure which would take up little memory and allow fast access and computation. For reasons of experience and familiarity, the program was written in the *C++* programming language. The class structures prevalent in *C++* programming provided a simple way to describe the potential covering and packing arrays as objects. A class entitled *BKG_Array* was created such that each *BKG_Array* object would represent a potential array of either type. Also, each object carries with it a set of member functions for the purposes of quick evaluation and modification. Each *BKG_Array* object contains the global parameters k and g, as well as the current row count, b. The $b \times k$ array of entries from Z_g is stored as a one-dimensional array of entries from Z_g, called the **blocks** array. Each object contains a set of one-dimensional score arrays which are used to evaluate the fitness of the object. The most important score array is *missing_pair_array*, which has a cell counting the number of missing ordered pairs

for each possible pair of columns. Another score array, *pair_in_columns_array*, has a cell for each possible ordered pair (a, b) in $Z_g \times Z_g$ and each possible pair of columns, (p, q). Each cell counts the number of occurrences of ordered pair (a, b) in pair of columns (p, q). These score arrays for ordered pairs have single element counterparts, *missing_element_array* and *element_in_column_array*, which were intended to be used in conjunction with the algorithms' heuristics.

The most important member functions are *evaluate*, *swap_single_entry*, *anneal_swap_entry*, *tabu_swap_entry*, and *update_arrays_single_swap*. The *evaluate* function takes the sum of the values over all cells in *missing_pair_array*, returning the number of ordered pairs required to complete the covering array. This number represents the fitness of the object. The *swap_single_entry* function selects an entry at random and then switches it for another element of the alphabet. The *anneal_swap_entry* function swaps an entry at random and applies the annealing decision rule, while the *tabu_swap_entry* function works exhaustively through the object, switching each entry for each possible member of the alphabet not forbidden during a tabu search. The simulated annealing decision rule and forbidden moves will be discussed later in the chapter. The fourth function updates all score arrays in a local manner following each modification.

Another class entitled *Move_List* was created to keep a record of the modifications performed on any given object. This structure proves to be quite useful, as it could keep an online list of those moves which are forbidden during a tabu search.

At the completion of any step in the binary search, the program ouputs either the array found or, if none was located, the array located with the most covered pairs and its score. The program also returns the number of moves required to find the array (M, if none was found) and the amount of time required to complete this number of moves.

2.3 The Simulated Annealing Algorithm

The first metaheuristic chosen is the simulated annealing, or **SA**, algorithm. The SA algorithm itself is modelled after the effect of a slow cooling process on the molecules of

a metallic substance [8, 20]. Just as cooling brings these molecules to an optimal rest energy, this algorithm slowly converges the state being examined toward an optimal state. Prior to running, an initial positive temperature t_0 and a decimal decrement factor δ_t lying strictly between 0 and 1 must be provided as input. The algorithm's simulated cooling schedule is determined entirely by these two parameters, as t_0 is multiplied successively by δ_t after each pass through the main loop.

At each step, the algorithm randomly selects one array from the neighbourhood of the current state and evaluates its fitness. For our objects of study, the neighbourhood of any potential array is the set of arrays that can be obtained by switching a single element in the current array with a different legal member of the alphabet. If the neighbouring array is more fit than the current array, then the neighbouring array becomes the new state. However, should the selected neighbouring array be less fit than the current array, the SA heuristic is employed. This heuristic is the main feature of the SA algorithm and guarantees that at each step, there is a non-zero probability of moving to a state which is less fit than the current state.

Let the fitness of the current array and the neighbouring array be denoted by F_0 and F, respectively. Taking t to be the current temperature, a random decimal number r between 0 and 1 is generated and compared to the quantity $e^{(F_0-F)/t}$. The neighbouring array is accepted as the new state if r is less than or equal to this quantity and rejected otherwise. After accepting or rejecting the neighbouring array, the temperature is multiplied by a factor of δ_t. Therefore, in the early stages of the algorithm, when t is still fairly close to t_0, there is a greater probability of accepting a less fit neighbouring array as the new state and hence a good chance to explore many regions of the search space before settling in one which may contain a global optimum. In the later stages of the algorithm, it is less probable that a state with a poor fitness is selected forcing the algorithm to converge toward the local optimum of the region in which its current state lies. The point at which it becomes difficult to move to a less fit state is referred to as the **focal point**, as it is at this point where it is most likely that the algorithm will simply proceed to a local optimum, rather than continuing its search through a large cross-section of the search space. Some versions of this algorithm accept the new state only if the random number r is

strictly less than the quantity $e^{(F_0-F)/t}$, but it was determined that after executing a large number of moves, the value of t was rounded down to 0. Considering that r is nonnegative, this rounding down prevents the algorithm from accepting any less fit arrays. Therefore, to maintain a minimal amount of effectiveness, we chose to accept new states for $r \leq e^{(F_0-F)/t}$.

There are many schools of thought regarding the use of the chosen input parameters. Kreher and Stinson [15] simply state that the initial temperature should be some value greater than zero, and that the decrement factor δ_t is required to be some positive number less than one. Presumably, one could slow down the cooling process by only multiplying t by δ_t every n^{th} pass through the main loop of the algorithm, for some integer n greater than one. Also, one could use an oscillating temperature function in order to occasionally release the algorithm from a regional search where no global optima exist. However, this may also lead the algorithm to free itself prematurely from searching in a region where an optimal solution does exist. After reviewing some of the literature regarding annealing algorithms in [15, 20], it was determined that the monotonic decreasing function described in detail above was the function of choice. As a result, alternative cooling schedules were not implemented.

2.4 The Tabu Search Algorithm

The second metaheuristic selected to assist in locating covering arrays is the tabu search, or **TS**, algorithm. As with the SA algorithm, the heuristic involved in the TS algorithm allows the selection of a new state which is less fit than the current state. However, each neighbourhood search is exhaustive, rather than random, ensuring that the state chosen at each step is the best neighbouring option possible. Unlike the SA algorithm, the TS algorithm has only one input parameter, called the **tabu lifetime** and denoted by L.

At each step of the search, the TS algorithm evaluates the fitness of every array in the neighbourhood of the current state. The array in the neighbourhood which is the most fit is selected as the new state, regardless of whether it is more or less fit than the current state.

One concern in such an algorithm is that it is entirely possible for it to be caught in an infinite loop and to stop propagating through the search space altogether [10]. Consider the following situation. Let the algorithm decide to move from state A to state B in step i, where state B is less fit than state A. It is entirely possible that in step $i+1$, the algorithm finds that the fittest object in $N(B)$ is the object A. For the remainder of the algorithm's run time, the current state will alternate between A and B with no chance of escape. To avoid this situation, the TS algorithm uses a list of forbidden moves, called the **tabu list**. At any step, this list contains a history of the L most recent moves, L being the specified tabu lifetime. Prior to deciding which neighbouring array shall become the new state, the algorithm verifies that the move resulting in the most fit array is not contained in the tabu list. If the move is not forbidden, the fittest array is selected as the new state. Otherwise, the algorithm considers the move resulting in the next fittest array, and then the next until the move in question is not contained in the tabu list. If every locally available move is forbidden, then the algorithm stops and moves on to the next possible value of b.

In the program written, a move in the *Move_list* object acting as a tabu list has four parameters: a row r, a column c, an old entry i and a new entry j, which corresponds to entry (r, c) having been changed from i to j. In order to make it very hard for the algorithm to revisit certain objects multiple times, it was decided that a move having parameters (r', c', o', n') was forbidden if for some move in the last L moves of the tabu list having parameters (r, c, i, j), it was true that $r = r'$, $c = c'$, and either $i = o'$ and $j = n'$ or $i = n'$ and $j = o'$. Therefore, a move is forbidden if it is either the same as or the reverse of one of the most recent L accepted moves, as documented in the tabu list. This makes it impossible for the TA algorithm to either retrace or repeat any recent moves.

The main difficulty in fine-tuning a TS algorithm is in attempting to balance computation time and the possibility of endless looping. By reducing the tabu lifetime, less time is spent comparing moves to those in the tabu list while the chance that the algorithm will be caught in a loop increases. On the other hand, the possibility of looping can be reduced by increasing the tabu lifetime, which in turn causes the

algorithm to spend more time passing through the tabu list at each step. In attempting to locate a middle ground, many different tabu lifetime values were tested and the empirical data presented in Chapter 3 heavily influenced the tabu lifetime value chosen. The advice of Falkenauer [8] was also taken into account: he claims that the size of a tabu list should be small, rarely exceeding a dozen moves. Aside from using the tabu list, there is another way to reduce the chance of looping. If there is a set of many neighbouring arrays all having the same locally optimal fitness at any particular step, then the next state is randomly selected from those locally optimal arrays which are not forbidden.

We also considered changing the definition of a forbidden move. For example, rather than keeping track of moves as ordered quadruples as described above, we could simply keep track of the coordinates of the cell that has been changed, or even just the row or column in which it appears. We decided, however, that these alternate definitions would be too restrictive on the size of the neighbourhood and that for even relatively small values of L, too many good neighbouring arrays may be deemed forbidden.

2.5 The Genetic Algorithm

The third and final metaheuristic used is the genetic, or **GA**, algorithm. Like the TS algorithm, the GA algorithm requires only one input value, called the population size, S. Other than this, the GA algorithm is structurally quite different from the other two metaheuristics previously mentioned. The genetic algorithm does not adopt the standard approach of the hill-climbing algorithm. In fact, the model for the genetic algorithm is based on the general theory behind evolution and speciation [8, 11]. The theory is that given an initial population of organisms, the fittest of the group will be most likely to survive from generation to generation. Through reproduction, certain traits of fitness can be passed from parents to their children. This, and the possibility of random mutation, can result in the evolution of the population into a fitter form.

Unlike the other two metaheuristics which consider a single array at a time, the GA algorithm starts with a population of potential arrays or **genes** equal in number to

the value supplied by the user. Then, in a process referred to as the **recombination** stage, these genes are grouped into pairs of **parent** genes and combined with one another, with each pair of parent genes producing two other genes. These genes are often referred to as **offspring** genes and each one will possess some characteristics of each of its parents. After their creation, the same move used in the other two algorithms is applied to each of the offspring genes to simulate mutation. Through a chosen culling step, the population is reduced to its original size and the process begins anew.

The primary goal is to create and maintain a population of reasonably fit genes, the theory being that the more fit members of the population should help to produce fit offspring genes. There are many ways to accomplish this goal. One may use a strategy in which genes are allowed to mate many times in any one generation and mates are chosen at random from the population. To be sure that more fit genes mate more frequently, the selection can be performed in a weighted fashion, where genes that are more fit have a higher probability of being selected as a mate. Yet another way to maintain a fit population would be to reject the S weakest members of the $2S$-element population at the end of every generation. Unfortunately, both of these methods could lead to one particularly fit gene having batches of similarly fit offspring over the course of many mating phases. In subsequent generations, these offspring genes may inbreed to create a homogeneous population which would be limited to evolution through mutation alone.

Ashlock [1] suggests that a remedy to this situation is to use a **tournament selection** heuristic to choose the mates. If the population size, S, is a multiple of four then the process can occur as follows. The population of genes is first randomly partitioned into groups of four. Within each group, the two most fit genes are chosen as the parents and their offspring replace the other two genes in the group at the end of the generation. This heuristic ensures that the fittest genes remain in the population, but restricts the amount of times they can reproduce to once per generation. Also, at the end of each generation, half of the population is turned over, ensuring a wide coverage of the search space through successive mating. The repeated introduction of less fit offspring increases the chance of a less fit gene being involved in the recombination

phase, thus maintaining diversity in the population.

An alternative to this tournament selection is a method suggested in [15] that we refer to as the **quick convergence** method. In this method, the original population is partitioned randomly into two groups (male and female) of size $S/2$. The members of each group are ordered randomly and then the i^{th} arrays from each group are mated with each other, for $1 \leq i \leq S/2$. The S offspring are mutated and then the most fit S members of the male, female and offspring populations combined are selected as the new population. This is essentially the steepest ascent version of the genetic algorithm: the population heavily favours fit arrays and moves rapidly to a very fit state. Unfortunately, these fit arrays dominate the population very quickly and if the local optimum toward which the population moves is not a global optimum, it may take many mutations to locate an optimal covering or packing array.

During the recombination phase, selected parents can be mated in many ways. The primary method of merging two parents is the **crossover** method of mating. In this method, a subset of coordinates in a $b \times k$ array is selected (call this set E). Each child is formed by filling in the coordinates corresponding to those in E with the entries of one parent and those coordinates not covered by E with the entries of the other parent. Three natural crossover methods are to take E to be the set of rows $0, 1, \ldots, i$ in their entirety, the set of columns $0, 1, \ldots, j$ in their entirety, or even all the coordinates of the $b \times k$ array, reading left to right and top to bottom, up to some coordinate (i, j). In each such method, we require that i and j be integers such that i lies between 0 and b and j lies between 0 and k. These methods shall be referred to as **row crossover**, **column crossover** and **point crossover**, respectively. The region of coordinates represented by the set E does not necessarily have to be contiguous, but for ease of computation, the implemented algorithm does not consider any non-contiguous cases. It was not decided, though, whether a choice of necessarily contiguous sets of points could contribute to rapid homogenization of the population. Finally, in order to avoid producing clones, the implemented algorithm has been configured to ensure that neither E nor its complement is an empty set.

The effectiveness of a GA algorithm may also be altered at the mutation stage. Mutation can be forced to occur with a fixed fractional probability, instead of having

all offspring mutate at every step. Also, in order to decrease the chances of population homogeneity, mutation can be made to occur more than once per step. In implementing the GA algorithm, it was decided that mutation should always occur in order to avoid homogeneity in the population. However, none of the literature suggested multiple mutations per mating phase and therefore, the concept was abandoned.

The implemented GA algorithm using the method of tournament selection starts by randomly generating S arrays, which are then partitioned and mated according to the method described above. The offspring are formed by a crossover method and then are all mutated by a random entry swap. Finally, the parent arrays and offspring arrays are merged into one S-element population before the next iteration begins.

The quick convergence version of the GA algorithm was implemented in a different fashion. At first, S arrays are randomly generated and then randomly ordered and mated as described above. As with the other version of the GA algorithm, the S offspring are all mutated by a random entry swap. After mutation, the median value of fitness across the population is determined and the T arrays which have a fitness value strictly less than this are chosen to be members of the next population. Finally, the remaining $S - T$ spaces in the next population are filled by randomly selected arrays with fitness equal to the old population's median fitness value.

2.6 A Comparison of Algorithmic Complexity

In light of the differences between these metaheuristics, it is possible that any shortcoming one algorithm might have in attempting to locate a particular $CA(k, g : n)$ might be overcome by one of the others. Therefore, it was originally thought that all three algorithms should be used in series when searching for a particular $CA(k, g : n)$. Through testing, though, it became apparent that the most effective searches were achieved using only two of the algorithms. The results of these tests are discussed later in Chapter 3.

The main distinguishing feature of each algorithm is the complexity of its inner loop structure. In order to be able to discuss algorithmic complexity, we assumed that any basic mathematical operation such as addition or multiplication occurred

with constant complexity $O(1)$. The complexity of some of the *BKG_Array* class member functions must be evaluated prior to assessing the total complexity of each of the three algorithms.

The most basic member function in the *BKG_Array* class is the *fill_all_arrays* function, which is used to initialize the score arrays within any particular array object. This function is broken down into two subfunctions, *fill_pair_arrays*, which initializes the two arrays which track the presence of the ordered pairs across columns, and *fill_element_arrays*, which initializes the other arrays used to count the occurrences of each element of the alphabet in each column. As the pair arrays are indexed by pairs of columns and ordered pairs of alphabet entries, this process requires $O(k^2b)$ operations to examine the contents of the *BKG_Array* object's cells and another $O(k^2g^2)$ operations to enter this information into the tabulation array's cells. Unlike the pair arrays, the single element tabulation arrays are only indexed by column and by alphabet element. Therefore, the same process costs only $O(kb)$ operations to examine the object's cells and another $O(kg)$ operations to tabulate this information. Hence, the *fill_all_arrays* function performs $O(k^2(b+g^2))$ operations every time it is called.

The initialization of a *BKG_Array* object starts with the allocation of memory for the *blocks* array, which contains the row and column entries of the object, and the four score arrays. This was all assumed to require $O(1)$ operations. Filling the blocks array clearly requires $O(kb)$ operations while filling the score arrays, as mentioned above, requires $O(k^2(b+g^2))$ operations. The latter of these costs becomes the cost of initializing an object. The cost of initializing an object with random entries is exactly the same. Copying an object is in fact cheaper. We still require $O(k^2g^2)$ operations to copy over the contents of the *pair_in_columns* array, but the cost of copying over the other three score arrays is dominated by this quantity. The only additional cost is the $O(kb)$ operations required to copy the *blocks* array, so the total cost of making a copy is $O(kb+k^2g^2)$ operations.

Whenever an entry of one of the objects is changed, a local update is performed to correct the score arrays in a quick manner. The algorithm examines only the score array cells pertaining to the precise pairs of columns and ordered pairs arising from

the entry change and increases or decreases the quantities in the corresponding cells as is necessary. As only one row is involved, one sweep through the columns is all that is required to update the score arrays and therefore the cost of a local update is only $O(k)$ operations.

The final basic *BKG_Array* class member functions are the *evaluate* function, which returns the fitness of the object and the *is_taboo* function, which determines if a particular move is taboo. The *evaluate* function runs through the *missing_pair_array* score array and sums up the entries of all of its k^2 cells to obtain the object's fitness. Clearly, the cost of evaluating the fitness of any *BKG_Array* object is $O(k^2)$ operations. The *is_taboo* function runs exhaustively through the L cells of the current list of taboo moves, comparing the contents of each cell with the current move. These comparisons are assumed to have a negligible cost and so the total cost of a taboo check is simply $O(L)$ operations.

As outlined in Section 2.2, there are three types of moves or **swap functions** implemented in our code. First is the simple swap, which chooses a row and column at random, changes the entry in that cell to a different legal member of the alphabet and then updates the object's score arrays in a local fashion. Selecting random numbers and changing the cell entry have a negligible cost and hence the cost of a simple swap is dominated by the cost of the local update function, which is $O(k)$. The second swap function is the anneal swap function. This function initially evaluates the current object's fitness and then performs a simple swap, for a total of $O(k^2)$ operations. The function then applies the SA heuristic and checks if the fitness has increased or not. If so, nothing more is done. Otherwise, a random number is generated and compared to the quantity . If the random number test fails then the simple swap is reversed. In any case, the application of this heuristic costs another $O(k^2)$ operations, which allows the anneal swap to be called for a cost of only $O(k^2)$ operations. The third and final swap function is the tabu swap function. The function is given a row and a column in the object in which an entry swap is to be performed. The function notes the entry in this cell and for every other alphabet member that would not create a taboo swap, the function swaps the entry for this new entry, evaluates the fitness of the new array and then reverses the swap. This process uses a taboo check, an object

evaluation and two local updates for each other member of the alphabet, yielding a total cost of $O(g(L + k^2))$ operations. If the fitness of the new object is at least as good as the best fitness found so far in the current round of tabu swaps, then the row, column, old entry and new entry values are noted.

With these functions in hand, we can describe the complexity of the three algorithms. The SA algorithm is by far the least complex of the three. In each pass through its main loop, the algorithm calls the anneal swap function and evaluate once apiece. If the new object is more fit than any found previously, it is copied. Finally, the temperature is decreased and the loop counter is augmented. Therefore, , if M is the maximum number of temperature decrements allowed before declaring a failed search and C is the number of times a more fit array is copied and stored as described above, then the complexity of the SA algorithm is of $O(Mk^2 + C(k^2g^2 + kb))$.

The TS algorithm executes $O(kbg - L)$ simple swaps per run through the loop, where the L subtrand reflects the number of swaps which may be taboo at any step. As we rarely used a value of L greater than $O(k)$, we assume that in each pass through its main loop, the TS algorithm calls the tabu swap function $O(kb)$ times, once for each row and column in the *BKG_Array* object. After constructing a list of the optimal fitnesses which can be obtained by a single entry swap, the TS algorithm selects one move from list list, executes it and updates the object's score arrays locally. As with the SA algorithm, the object is evaluated and if it is more fit than any other found previously by the algorithm, it is copied as a record-keeping measure. Therefore, if M and C are as above, then this algorithm performs $O(Mkbg(k^2 + L) + C(k^2g^2 + kb))$ operations per step in the binary search. If we assume that L is significantly smaller than k^2, this reduces to a complexity of $O(Mk^3bg)$. Noting that $b \geq g^2$ and that for large k, the value of b typically exceeds that of g^2 by a great deal, the TS algorithm seems to be far more complex than the SA algorithm. However, aside from an apparent increase in run time for the TS algorithm, it would seem that an exhaustive search through a state's neighbourhood would be more likely to locate good neighbouring candidates than by randomly selecting only one array at a time from the neighbourhood.

The genetic metaheuristic is sufficiently different from both of the other algorithms

used, and so it is unclear what net advantages might be held over the SA and TS algorithms. In terms of complexity, there is no difference between the GA algorithm which uses tournament selection and that which adopts the method of quick convergence. Below, we outline the complexity computation for the GA algorithm which employs the method of tournament selection, as it is the algorithm which was used most to construct new designs.

For the method of tournament selection, we take S to be the population size input by the user and assume that S is divisible by four. The main loop of the GA algorithm starts by initializing an index array and then randomizing the indices inside it. This costs $O(S)$ operations in total. The randomized sequence inside the index array is used to shuffle the initial population of S randomly generated *BKG_Array* objects. The shuffled population array is then divided into subarrays, each containing only four *BKG_Array* objects and a simple bubble sort algorithm sorts each set of four objects so that the fittest two in each foursome can be removed for mating. Due to the copying involved during the sort, this part of the loop performs $O(S(kb + k^2g^2))$ operations. Regardless of the crossover method selected, the algorithm mates by reading each entry in the parents' designs, deciding which entry goes to which offspring and then placing that entry in the appropriate *blocks* array. After the offspring *blocks* arrays are completed, *fill_all_arrays* is used to fill in their respective score arrays. Due to this non-local score array completion, this portion of the loop processes $O(Sk^2(b + g^2))$ operations. All of the S objects in the population are mutated by a simple swap, which costs $O(Sk)$ operations. Finally, the S objects which are at this time held in two separate population arrays, parent and offspring, are merged back into a single population, requiring S more copies. Therefore, again taking M to be the threshold on the number of times the algorithm will run through its main loop, the algorithmic complexity of the GA algorithm is $O(SMk^2(b + g^2))$.

Regardless of the high amount of loop complexity, the GA algorithm has S feasible arrays at its disposal at every step, at least half of which are relatively fit. Together with a tournament selection method in place, these arrays can spread out and cover a wide crossection of the search space in relatively few runs through the loop. Unfortunately, as with the SA algorithm, if there is an optimal state in the neighbourhood

of one of the arrays in the current population, it is unlikely that it will be located quickly. Ashlock [1] claims from experimental experience that it is more effective to conduct many runs in parallel with a smaller population size, rather than a single run with a larger initial family of feasible arrays.

As a final comment, it would seem that as the values of b, k, g and M vary, no one set of input parameters can be supplied to maintain optimal efficiency in locating covering arrays. For example, a TS algorithm with large L searching for smaller objects might be more likely to arrive at a state from which all moves are forbidden. The same algorithm with the same value L, but searching for much larger objects might even be caught in a loop, as cycles for larger objects can contain many moves. Another possibility is that, for elevated values of M, the temperature value in the later stages of an SA algorithm could get so small that the probability of accepting a less fit neighbouring array as the new state may become negligible. The next chapter features a much more detailed quantitative analysis of the different algorithms.

2.7 Algorithmic Enhancements

It was realized after the thesis was submitted [17] that some enhancements could be made to speed up the algorithms. For example, the *evaluate* member function can be improved, simply by including a variable in the BKG_Array objects themselves indicating their current level of fitness. Rather than calling a function to evaluate an object, the score variable could be simply increased or decreased automatically while performing the local update. This would give the function a complexity of $O(1)$, reducing the complexities of the SA and TS algorithms by roughly a factor of k.

Also, in the TS algorithm, the searches through a tabu list could have been more efficient. Rather than making L comparisons, a binary search method could have been implemented, together with appropriate insertions to and deletions from the tabu list at any time to perform the searches in $O(logL)$ operations. Considering the usually small value of L, this would not have a large effect on the complexity of the TS algorithm.

Algorithm	Current Algorithmic Complexity	With Future Enhancements
SA	$Mk^2 + C(k^2g^2 + kb)$	$Mk + C(kb)$
TS	$Mkbg(k^2 + L) + C(k^2g^2 + kb)$	$Mkbg(k + logL) + C(kb)$
GA	$MSk^2(b + g^2)$	no change

Table 2.1: The result of enhancements on the complexity of our metaheuristic searches.

Finally, a major concern in all three algorithms, but particularly the genetic algorithm, is the amount of copies that need to be made as the object in question becomes more fit throughout the search. Considering that these copies are made only when the current object is more fit than the best on record, the number C of such copies made in one step of the binary search could be reduced by creating a a more fit initial object prior to the search, by deterministic or heuristic means. During the search, the three algorithms could be instructed to only make copies of objects which are more fit than this initial object. If the fitness of this initial object is N, then N becomes an upper bound on C, thereby reducing each algorithm's runtime. A further enhancement would be to only retain the object's *blocks* array without all of the other variables and arrays in the class.

We present in Table 2.1 a summary of the complexity improvements due to the implementation of the above algorithmic enhancements.

Chapter 3

Quantitative Analysis

In using the three metaheuristics described in the last chapter, it is desirable to know not only which algorithm is the most effective in general, but also to discover which set of input parameters optimizes the effectiveness of each algorithm.

There are many different ways of measuring effectiveness. For example, given positive integers k and g, as well as the type of array desired, we are naturally interested in the algorithm that can determine either the best corresponding value of b or the best average value of b over a set of trials. However, this effectiveness can be refined by examining how many moves or the amount of time needed by each algorithm to generate the array in question. With data detailing these run parameters for each algorithm, one can hope to assess which algorithm would be most useful, based on personal weights associated with each search criterion.

3.1 The Parameters of the SA Algorithm

Structurally, the simulated annealing algorithm is the simplest of the three metaheuristics employed. As determined in Chapter 2, each step in the binary search using simulated annealing requires $O(M(kb + k^2g^2))$ lines of code to run completely. The three input parameters for the annealing algorithm are the maximum number of moves allowed, M, the initial temperature, t_0, and the temperature decrement factor, δ_t. In order to determine the optimal choices for t_0 and δ_t, t_0 was initially fixed while

δ_t was varied, and then the reverse experiment was performed. For each pair of values (t_0, δ_t), the algorithm was run five times and best arrays found were noted. This was repeated for two different covering arrays, $CA(6, 4 : 1)$, and $CA(7, 5 : 1)$, as well as the packing array $PA(7, 5 : 1)$. These structures were known to have optimal sizes of 19 rows, 29 rows, and 15 rows, respectively [24]. The results of these trials are presented in Tables A.1 - A.4 in Appendix A. When looking for the two covering arrays, the upper bounds on the number of rows were set to 25 rows for $CA(6, 4 : 1)$ and 45 rows for $CA(7, 5 : 1)$. The lower bound on the number of rows for the $PA(7, 5 : 1)$ was taken to be 7.

An interesting piece of information gleaned from any one of the four tables is that a cooling function which reduces the temperature too quickly renders the algorithm useless. For values of $\delta_t \leq 0.5$, the annealing algorithm was often unable to locate a covering array for any of the five runs, resulting in an entry of "–" in the table. However, it would also seem that in some cases, as the value of δ_t approaches 1, the algorithm also has difficulty finding good arrays. In the cases $t_0 = 1$ and $t_0 = 1,000$, the greater values of δ_t yield comparatively poor results. As a result, it was concluded that a good choice for δ_t would be a value in between 0.9 and 0.9925. Also, an augmentation in the number of moves, thereby allowing the algorithm to search longer, seemed to increase the effectiveness of the annealing algorithm. Of the 103 trials for which results were obtained for two values of M, an increase in the number of moves improved the average number of rows of the constructed covering arrays in 59 cases and improved on the size of the best covering array constructed in 37 cases. Furthermore, an increase in the value of M did not affect the average number of rows in 17 additional cases and had no effect on the size of the best covering array constructed in another 46 cases.

While an increase in the given quantity M increases the chance of finding a better covering or packing array, it also underlines a major weakness of the SA algorithm. It is known that $ca(7, 5 : 1) = 29$ and after the 560 trials performed in order to compile the four tables were examined, it was noted that only one trial produced a $CA(7, 5 : 1)$ with less than 32 rows. This includes all of the cases where M was already as large as 10^6 moves. After this many moves, the quantity $t = t_0 \cdot (\delta_t)^M$ is so small, regardless

of the values of t_0 and δ_t, that a move to a less fit array is almost never accepted. As a result, if the algorithm has not found an optimal array by this point, it will take a great number of iterations to generate a random number small enough to move the search to a different part of the search space. The only way to attempt to counter this situation is to choose a value of δ_t which is extremely close to one, thus ensuring that t does not become too small for a large number of moves. Unfortunately, this renders the initial portion of the run ineffective, as it will take a much longer time to reach the focal point point in the search. It is important that this point is eventually reached, as the algorithm will have a lot of difficulty finding any local optima while it is easy to accept a move to a less fit array. This suggests that the SA algorithm may not be very well suited for finding optimal arrays which are sparsely located throughout a particular search space.

Some information about the choice of t_0 can also be obtained by comparing the information between tables. As t_0 increases, the range of values of δ_t with which the algorithm performed the best seems to slide closer to 0.75. This makes sense intuitively, as it is desirable to eventually make it difficult to accept a less fit array, and for higher t_0 and fixed M, this is accomplished by reducing the value of δ_t. Also, while the algorithm seemed to perform better for smaller values of t_0 (by a sheer count of when the algorithm generated the best array found across all trials), this is likely due to the fact that M was fixed across all trials. A value of $t_0 = 1$ is likely as effective as a value of $t_0 = 1000$, so long as M is modified appropriately in order to ensure that the algorithm eventually reaches the focal point.

With these findings in mind, it was decided that when searching for optimal covering and packing arrays, we would give input parameters $\delta_t = 0.925$ and appropriate values of t_0 and M that would guarantee that the focal point is reached.

3.2 The Parameters of the TS Algorithm

As opposed to the SA algorithm, which has a pair of input parameters, the tabu search algorithm has a single input parameter, the tabu lifetime L. As with the testing performed with the annealing algorithm, the three structures $CA(6, 4 : 1)$,

$CA(7,5:1)$, and $PA(7,5:1)$ were chosen as the objects for which to search. A maximum number of search moves, M, was fixed and a range of values of L were tested. The bounds on the number of rows, b, were the same as those set for the SA algorithm tests. The results of these tests are presented in Table A.5.

Intuitively, augmenting the number of moves an algorithm can make prior to stopping should increase the effectiveness of the algorithm in finding optimal arrays. As with the simulated annealing algorithm, this is indeed the case with the tabu search algorithm. This is not entirely obvious from examining the columns corresponding to the searches for a $CA(6,4:1)$, as the optimal 19-row array was fairly simple to find for all values of L, even for the lesser value of M. The columns of Table A.5 which correspond to the searches for a $CA(7,5:1)$, however, show evidence of this claim, as no trial for $M = 1000$ yielded an optimal, 29-row array, while more than half of the trials for $M = 5000$ found optimal arrays.

With regards to choosing an optimal value of L, it seems that the selection of extreme values renders the algorithm less effective in searching for arrays. For example, L needs to be suitably large to reduce the chance of cycling. Referring to the results of the trials pertaining to the $CA(7,5:1)$ searches, the smaller values of L were fairly ineffective at locating small arrays. Also, if the value of L gets too large, it is possible that too many moves could be designated as forbidden, relegating the bulk of the search to areas of the search space not containing global optima. This was evident in the search for a $PA(7,5:1)$, which should result in the optimal discovery of a 15-row, 7-column array with entries from Z_5: as L gets large, the search is unable to find an array with more than 12 rows.

The choice of $L = 5$ yielded the best array found in every case, while very few trials for $L \geq 150$ yielded optimal arrays. While this leaves a fairly wide range of values from which L can be selected, many of the values chosen in between these bounds yielded very good results. It should be noted, though, that an upper bound of 150 is purely circumstantial. As the sought arrays get larger, they contain more possible entries. Consequently, the number of moves that must be deemed forbidden in order to make the tabu list restrictive must also increase.

For most searches, the value of L varied between 10 and 100, depending on the

size of the desired array.

3.3 The Parameters of the Genetic Algorithm

Unlike the SA and TS algorithms, which require only a small set of input parameters to commence the search, the genetic algorithm requires input parameters which changed the very structure of the algorithm itself. Not only is there an input parameter for the size of the population in the algorithm, but there is also a second parameter dictating the mating method. Furthermore, two different genetic algorithms were compared to one another, one of which employed a tournament selection method when choosing mates, and the other which employed a population cull which forced the population to converge to a local optimum more quickly.

Recall from Chapter 2.5 that the algorithm employing the tournament selection heuristic first randomly partitions the population into groups of four and then deletes from each group of four the two weakest genes. The two remaining genes mate and their offspring take the place of the deleted genes. The algorithm equipped with the quick convergence heuristic randomly partitions the population into two ordered groups. The i^{th} arrays from each group are mated with each other, for $1 \leq i \leq S/2$ and the S most fit arrays from the original population and their offspring are selected to be the new population.

3.3.1 Tournament Selection and Quick Convergence

The first decision that needed to be made was with regards to which algorithm should be used, the quick convergence algorithm or the tournament selection algorithm. These algorithms were used to search for six structures - a $CA(7,4:1)$, a $CA(8,5:1)$, a $CA(5,6:1)$, a $PA(5,4:1)$, a $PA(6,5:1)$, and a $PA(5,6:1)$ - in order to determine which one performed the best, on the average. The results are presented in Tables A.10 through A.12.

Before making reference to the results of the tests, it should be noted that a small error in the code prevented the program from ever looking for a packing array with

g^2 rows. After each step in the binary search, the code naturally rounds down when determining the next value of b to explore. As a result, when it is searching for packing arrays, the program will only check for arrays with sizes up to and including $g^2 - 1$.

The most peculiar finding is that within each table, the tournament selection algorithm found better covering arrays, while the quick convergence method found better packing arrays. The only trials in which this was not the case is when the two algorithms tied, or in the cases of $CA(8, 5 : 1)$ and $CA(5, 6 : 1)$ in Table A.11 and in the case of $PA(5, 4 : 1)$ in Table A.12. In some cases, one algorithm beat the other soundly, as in the case of $PA(6, 5 : 1)$ throughout all three tables, and in the case of $PA(5, 4 : 1)$ in Table A.10.

One other notable feature of the three tables is that in almost every case, the tournament selection method is the faster of the two methods. This was expected, as a particularly time consuming segment of code was a part of the quick convergence algorithm at the time of testing. Since the testing occurred, the code has been improved upon and although the new quick convergence algorithm runs faster than the old one, no testing was done to see how the run times compared to the tournament selection algorithm.

Without regard to runtime, the method of tournament selection appears to find better covering arrays, while the quick convergence algorithm seems to be more effective in searching for packing arrays. These findings might be related to the size of the structures sought. Perhaps the packing array for which we searched were not as sparsely located in their search space as were the covering arrays. If this were so, then it would be easier for a steepest-ascent-type algorithm to produce an object of optimal fitness.

3.3.2 Point, Row and Column Crossover

The next step was to choose which crossover method should be employed to mate the population members. Recall that the point crossover method randomly selects a single pivot entry (i, j) for each pair of arrays to be mated. If the pair of mates contain entries A_{mn} and B_{mn} and the pair of offspring contain entries C_{mn} and D_{mn},

then $C_{mn} = A_{mn}$ ($D_{mn} = B_{mn}$) for $m < i$ or $m = i$ and $n < j$ and $C_{mn} = B_{mn}$ ($D_{mn} = A_{mn}$) for $m > i$ or $m = i$ and $n > j$. The row and column crossover methods work in much the same way, except that the pivot entry selected for recombination must have $i = 0$ for the row method and $j = 0$ for the column method. At first glance, it might seem that no one of these methods would hold a clear advantage over either of the others: all three methods are simply different ways of mixing arrays, all running with the same algorithmic complexity. However, after performing some tests, some trends became apparent.

Referring once again to Tables A.10 through A.12, the tournament selection algorithm equipped with the column crossover method found smaller covering arrays than nearly any other tournament selection algorithm. The only two exceptions were the algorithm employing the point crossover method, which found a smaller $CA(5, 6 : 1)$, and the algorithm using a row crossover method, which found smaller $CA(7, 4 : 1)$ arrays on the average. Unfortunately, the column crossover method took longer than the others to run in every case except for when it was searching for a $CA(8, 5 : 1)$. Therefore, ignoring runtime, when using the tournament selection genetic algorithm to find covering arrays, the column crossover method should be used to find better arrays. In particular, as the alphabet size g increases, the difference between the runtime of the algorithm mating by column crossover and the runtime of the other two becomes quite small (less than five minutes).

One more set of tests was performed to attempt to discern which crossover method should be employed when looking for a covering array with the tournament selection method. The algorithm was instructed to search for a $CA(13, 11 : 1)$ with 198 rows within a fixed period of time. This array does exist, but we intentionally supplied various amounts of time within which it would be unlikely that the TS or the GA algorithm would find the optimal array. For each specified amount of time, the algorithm was given ten chances to cover as many ordered pairs as it could. The results are summarized in Table A.13. The results show that the algorithm which employed a row crossover method took less time on the average to complete a move, while the algorithm using the point crossover produced the lowest average score in most cases. While it is important to complete each move as quickly as possible, it is essential that

each move be able to cover as many uncovered ordered pairs as possible. Therefore, when searching for covering arrays of larger dimension, the point crossover method of mating should be used in order to cover a greater portion of the ordered pairs in less time.

The results obtained for the quick convergence algorithm are not as easy to interpret. While searching for a $PA(7, 5 : 1)$, the algorithms using the row and column crossover methods excelled at finding large packing arrays, bettering the third algorithm by an average of nearly one row per trial. However, for the other two sets of trials, it was the algorithm which mated via the point crossover method which performed better on the average. Furthermore, the only algorithm which took noticeably longer to locate its packing arrays was the one mating by column crossover. This suggests that the best algorithm is either the one using the point crossover method or the one using the row crossover method.

3.3.3 Population Size and Number of Generations

The final step in optimizing the genetic algorithm was to select an appropriate initial population size, together with a suitable number of generations over which the population could evolve. As with the testing for the SA and TS algorithms, the genetic algorithm was tested while looking for the same three covering and packing arrays. In order to get some additional data, two trials were added: one for the $CA(7, 5 : 1)$ and one for the $PA(7, 5 : 1)$, both with the number of generations, M, set to 500. The results of the search for covering arrays are presented in Table A.6, while the packing array search results are located in Table A.7. In both cases, the genetic algorithm employed tournament selection to select mates and then combined them using the point crossover method. One set of heuristics was chosen in order to reduce the number of separate runs to be performed. Of all the combinations of genetic algorithm heuristics, previous tests suggested that the algorithms employing the point crossover method were the most effective in general.

Within the set of searches for either of the covering arrays, an increase in the number of generations clearly increased the ability of the algorithm to locate arrays with

less rows. In the search for packing arrays, the effect of such an increase was a little more subtle: while the average number of rows in the best structure found increased in nearly every case, the maximum number of rows present in the best packing arrays located remained almost identical after raising the number of generations allowed. In most cases, as suggested by the complexity of the algorithm, an increase in the number of generations proportionally inflated the amount of time necessary to complete the set of trials for a fixed value of S. For example, taking $S = 100$ and the sought array to be $CA(7, 5 : 1)$, the algorithm iterated through 500 generations in roughly 20 minutes, 1000 generations in 31 minutes and 5000 generations in 3 hours.

In general, as with the increase in the number of generations, augmenting the size S of the initial population has a positive effect on the capability of the genetic algorithm to find better covering and packing arrays. Aside from the $M = 500$ trials in Table A.7, the bulk of the best optimal row counts occur in the lower half of the tables, when $S \geq 100$. Unfortunately, the runtime is sharply increased when considering greater values of S. Taking the sought array again to be $CA(7, 5 : 1)$, but by increasing the population size by a factor of five ($S = 500$), the algorithm iterated through 500 generations in roughly 85 minutes, 1000 generations in 180 minutes and 5000 generations in 23 hours. While the complexity of the algorithm suggests that the runtime should increase with a factor proportional to S, the factor is not this high for smaller values of M, but much worse for larger M.

One other question of interest deals with a claim of Ashlock's [1], that it is more effective, with regards to run time and usage of memory, to perform r runs with a population size S in parallel, rather than a single run with a population size $r \cdot S$. Clearly, the use of memory is greatly reduced as it is primarily used to store a set of S covering or packing arrays. As for effectiveness, it was shown in the previous paragraph that a larger S tends to produce better average and optimal results. In Table A.8, a list of times required for the genetic algorithm to complete a binary search while searching for a $CA(7, 5 : 1)$ for three different generation sizes is presented. For $M = 500$, there is no case where the total time required to run five trials with population size S is less than the average time required to run one trial with population size $5S$. For $M = 1000$, this only occurs for $S \geq 60$. Finally, Ashlock's conjecture

holds true for all values of S when $M = 5000$. We conjecture that this may be due to the fact that it takes many generations for a smaller population to explore the amount of search space covered initially by a larger population. The key to searching through the space effectively is not through mating, which essentially homogenizes the population, but through the mutation which constantly introduces new arrays to the population. In a few initial steps, the randomly generated arrays in a large population may differ in many entries, ensuring a wide initial coverage of the search space. As both large and small populations homogenize over time, the rates of addition of new members to these populations likely approach one another, allowing the smaller population to catch up in terms of area coverage.

After he had observed an immense standard deviation in the amount of moves required by a genetic algorithm to find an particular object, Ashlock concluded that it would be more effective to run the program many times with a smaller population size. As shown in Table A.15, our findings were similar. In fact, in two consecutive runs contributing to the values resulting from the search for a $CA(5, 6 : 1)$, the genetic algorithm found the object in 1740 moves in the first trial, and then in the next trial, was unable to locate the object until it had completed $4,514,907$ moves.

This idea of Ashlock's also leads us to examine the effectiveness of the algorithm for many values of M and S, where the quantity $M \cdot S$ is constant. In each such set of trials, the total number of genes created through evolution is constant. In effect, this test provides a measure of whether it is more productive to use a large population size and evolve it over a short period of time, or to use a small population size and evolve the population over a larger number of generations. The results for the trials where the sought array is $CA(7, 5 : 1)$ are presented in Table A.9. Clearly, using a smaller population size and a larger number of generations yields better arrays in less time. Therefore, this shall be taken into account when using the genetic algorithm to search for covering and packing arrays.

3.4 Comparing the Three Algorithms

After having identified good ranges for the input parameters for each algorithm, three tests were run in which all three competed to find the best arrays possible in the shortest amount of time. As explained in Section 3.3.2, the first test forced each of the metaheuristics to search for a $CA(13, 11 : 1)$ with 198 rows within a fixed period of time. The amounts of time supplied were such that it was presumed that some of the algorithms might not be able to find the optimal array within the time alotted. Each algorithm attempted to find the desired structure ten times for each amount of time specified. The results are summarized in Table A.13. It should be noted that the input parameters were set to $t_0 = 10$, $\delta_t = 0.999$, $L = 40$, and $S = 40$. Also, the tournament selection form of the genetic algorithm was employed. In each trial, the number of possible moves was unlimited.

The results clearly show that the genetic algorithm, in all its forms, is by far the weakest metaheuristic. Even in the later trials, where the algorithms were allowed to run for three hours, the genetic algorithm could not cover the unordered pairs as effectively as the other two algorithms. In fact, the SA algorithm located a $CA(13, 11 : 1)$ with 198 rows many times, the earliest occurrence being for a time limit of half an hour. Due to the complexity of the tabu search algorithm's inner loop, this metaheuristic is at a great disadvantage when running for shorter amounts of time. However, the tabu search algorithm has a very powerful neighbourhood move which allows it to cover many ordered pairs effectively in relatively few moves. By the time the TS algorithm was searching for two hours, it was leaving less than ten pairs uncovered across all possible pairs of columns.

A further comparison between the SA and TS algorithms can be made by comparing the number of basic operations each requires to locate a covering array. If the SA algorithm found a $CA(13, 11 : 1)$ with 198 rows in 2.1×10^7 moves (less than 30 minutes) and the SA algorithm performs $O(M(k^2g^2 + kb))$ operations per search, then the SA algorithm required approximately 4.8×10^{11} operations to locate the array. For the TS algorithm to match this operational efficiency, it would have to locate an array in $\frac{SAoperations}{k^3gb}$ moves. This quantity works out to be approximately 1.0×10^5

moves. By the data in this table, it appears that the tabu search algorithm might only need another hour or two to locate the array. In Table A.13, the relationship between a given amount of time and the number of moves an algorithm made in this time seems to suggest that the TS algorithm would require almost 35 hours to perform this many moves. This suggests that the TS algorithm performs operations much more efficiently than the SA algorithm.

A similar test was conducted for a $CA(9, 7 : 1)$ with 62 rows to determine if the results would be similar when looking for a smaller structure. These results are presented in Table A.14. Prior to commencing these tests, it was unknown whether a $CA(9, 7 : 1)$ with 62 rows existed as the current upper bound on the number of rows is 63. While the genetic algorithm again performed much more poorly than the other two algorithms, it was interesting to note that by the time one hour had elapsed, the genetic algorithm performed almost as well as the annealing algorithm. In fact, giving the SA algorithm more time to search didn't seem to help its performance at all. After five minutes had elapsed, the TS algorithm was already outperforming the SA algorithm and it continued to improve for each time increment. We believe that if a $CA(9, 7 : 1)$ with 62 rows were to exist, there wouldn't be many different ways to construct it and that, throughout the entire search space, very few such arrays exist. In fact, it would appear that very few arrays in the search space have a fitness of 20 or less. The results of Table A.14 strengthen our belief that while the SA algorithm can quickly locate objects that are dense in the search space, it is not effective at all in locating objects that are more difficult to find.

One other telling test was performed to compare the three metaheuristics. Just as it is of interest to determine which algorithms are effective within a fixed time period, it is equally interesting to determine which algorithms find a particular array in the shortest amount of time. Two arrays were considered: a $PA(7, 6 : 1)$ with 19 rows and a $CA(5, 6 : 1)$ with 42 rows. Through experience, it was observed that these arrays were not particularly easy to find and may provide a challenge for the three metaheuristics. The input parameters were set to $t_0 = 1$, $\delta_t = 0.925$, $L = 10$, and $S = 60$ for the packing array search, while S was augmented to 80 for the covering array search. Again, the tournament selection form of the genetic algorithm was

employed. Each algorithm was given ten opportunities to locate the specified array. In each trial, the amount of time available to the algorithm was unlimited.

Referring to Table A.15 for the results of this test, it became apparent once again that the genetic algorithm performs far worse than the other two algorithms. When searching for the packing array, the genetic algorithm required roughly two hours to find the array on 19 rows while on the average, the SA and TS algorithms accomplished this feat in less than twelve seconds. While the SA and TS algorithms took roughly the same amount of time to locate the packing array, the annealing algorithm required 300 times as many moves to succeed. These performance ratios were almost identical when searching for a $CA(5, 6 : 1)$ with 42 rows.

It will be shown in the next chapter that the simulated annealing algorithm found the best covering array results, while the tabu search bettered the most packing array bounds. This was largely due to the fact that the only remaining improvable covering array bounds were for values of k and g large enough to slow the tabu search down by a great amount. While the annealing algorithm could proceed through an entire binary search in a matter of hours, the tabu search would take this long to look for a single array.

Many more tests and trials were conducted for the genetic algorithm than for the SA and TS algorithms, even though the former seemed immediately to be the weakest of the three. While it may have been more productive to attempt to modify one of the stronger algorithms, it was out of a sense of disbelief that the genetic algorithm could perform so poorly relative to the other algorithms that so many attempts were made to modify the structure of the genetic algorithm. Unfortunately, none of the modifications performed on the GA algorithm produced changes in effectiveness drastic enough to suggest that it was as well suited for finding good packing and covering arrays as either of the other two algorithms.

Chapter 4

Bounds Improved by Randomized Search

4.1 A Brief History of the Covering Array Problem

The most basic of results regarding the existence of optimal covering arrays stems from the realization that orthogonal arrays are in fact covering arrays where every ordered pair occurs precisely once in every possible pair of columns. Clearly, if there exists an $OA(k, g)$, then we also have a $CA(k, g : 1)$ with g^2 rows. As explained in Chapter 1, the existence of a set of l mutually orthogonal idempotent Latin squares on g symbols implies the existence of an $OA(l + 2, g)$ with g disjoint rows and hence the existence of a $CA(l + 2, g : g)$ with g^2 rows.

Many approaches have been taken when attempting to produce better bounds on the size of a packing or covering array. Rényi first posed the problem of trying to find maximal qualitatively t-independent g-partitions of a b-set. Independently, Katona [12] and Kleitman and Spencer [13] completely solved the problem for $t = g = 2$. Gargano, Körner and Vaccaro [9] then determined the first asymptotic bounds for the problem. Their work shows that

$$\lim_{n\to\infty} \frac{ca(k,g:1)}{\log_2 k} = \frac{g}{2}.$$

This result says nothing, however, about the relationship between k, g and $ca(k, g : 1)$ for smaller values of k.

A variety of methods have been used to improve on this bound for smaller parameter values. Poljak and Rödl [21] developed the first set of upper bound improvements that were useful for smaller k and almost a decade later, Poljak and Tuza [22] published the first bound improvements which took the number of disjoint rows in the covering array into account. Sloane [23] later published a series of results for $g = 3$ and small values of k, due to Applegate, Cook, Östergård, and Sloane.

Stevens, Moura and Mendelsohn [27] have developed lower bounds on the sizes of covering arrays, while Stevens and Mendelsohn [26] have developed upper bounds on the size of packing arrays. In [24], an extremely useful construction is presented. This **Blocksize recursive** construction produces good upper and lower bounds on the size of covering arrays, based on the bounds for arrays with smaller k. The following is a direct result of the construction.

Theorem 4.1. *[24] Let n, $m \leq g$. Then,*

$$ca(k, g : n) \leq \min_{2 \leq i \leq \lceil \frac{k}{2} \rceil} (ca(i, g : n) + ca(\tfrac{k}{i}, g : m) - m).$$

Clearly, this theorem becomes more powerful for higher m, which denotes the number of disjoint rows in the base array. It is therefore desirable to consider using covering arrays with as many disjoint rows as possible when attempting this construction to build arrays with larger k.

Stevens also uses a generalization of Wilson's Theorem, a construction involving group divisible designs and a method of completing ITDs to generate other upper bounds on the size of covering arrays [24]. The least effective of the three is the method involving the filling of holes in an ITD, but this method is solely responsible for providing the best known bounds for $ca(4, 6 : n)$. This method will also be adapted in Chapter 5 to produce some optimal covering arrays with alphabet sizes independent of the columns in the array. Stevens, Moura and Mendelsohn [27] also provide lower bounds for covering arrays by examining the intersection properties of set systems and by presenting some set-packing arguments.

Sloane's initial work dealt with strength 3 covering arrays. These structures have also been recently researched, culminating in the most recent publication of Chateauneuf and Kreher's [2]. Chateauneuf and Kreher construct objects called starter arrays from one-factorizations of K_{2v} and use these to obtain strength 3 covering arrays.

Stevens and Mendelsohn [26] have used recursive methods to provide bounds on the size of packing arrays. They have considered packing arrays as error correcting codes and generalized the Plotkin bound defined in Chapter 1 to tighten bounds on the size of packing arrays. They also use the maximum sizes of packing designs to improve the upper bounds on packing arrays containing a large set of disjoint rows.

Aside from these theoretical methods, metaheuristics have also been used previously in an attempt to achieve better bounds on the size of covering arrays. In [24], Stevens designs a simulated annealing algorithm similar to our own, which was used to find covering arrays with small parameter values. Stevens's algorithm differed from the one implemented here in its main loop. Instead of decreasing the temperature after every attempted move, Stevens's algorithm performed a number of moves, N, at a fixed temperature and kept the best array within the chain of N moves. The number N was input by the user. The temperature was decreased in the same fashion as ours and the kept array was used as the starting point of the next chain of N moves. The algorithm was successful for small values of k and g, finding many new upper bounds at the time his thesis was published. Seven of these bounds are still the best known and are labelled as such in Table B.2.

In [19], Nurmela devised a tabu search algorithm to find similarly small covering arrays. The results of Nurmela's tabu searches were very successful, lowering many bounds in the range $3 \leq g \leq 10$ and $k \leq 15$. Nurmela's tabu search uses a more sophisticated heuristic than our algorithm. At each step, two columns in the array and an ordered pair not covered in those columns are selected at random. A list of rows in the array is generated, such that every row in the list requires a single element change to cover the pair in the selected columns. The list of these changes required to cover the pair forms the set of moves at any step. The cost of each move is calculated and the move that is most beneficial to the fitness of the array is selected. If there is

a tie, one is selected at random. Also, Nurmela rules that a move is forbidden if the element to be switched is an element that has been switched in any way over the last L moves, where L is the supplied tabu lifetime. Nurmela claims that the computations required to complete each move are quick, but offers no data with regards to the degree of computation speed. Nurmela's more sophisticated tabu search heuristic does reduce the size of each array's neighbourhood, suggesting that his tabu search is likely able to complete each move in less time.

Researchers at Bellcore labs have also devised an algorithm referred to as the **Automatic Efficient Test Generator**, or **AETG** [3, 4, 6]. As opposed to our metaheuristics, which manipulate the entries in an array of fixed size, the AETG algorithm starts with just one row of the array and then builds the array one row at a time as follows. The algorithm refers to a list of possible rows that could be added to the array at each step and greedily selects the row which covers the most as yet uncovered pairs. Nurmela [19] claims that while the AETG algorithm is fast, it does not always generate optimal arrays. Furthermore, as presented in [2], the methods of Chateauneuf and Kreher performed better than those of the AETG system in every trial when searching for strength 3 covering arrays. As the AETG algorithm has not yet been fully described in publications and very little specific data is known about its speed in finding covering arrays, it is impossible to conjecture whether its runtime compares to that of our algorithms. The only bound attributed to the AETG algorithm is that of $ca(10, 20 : 1) \leq 180$, to which reference is made in each of the papers cited above.

4.2 Covering Arrays

As mentioned before, there is a lot of active research into lowering the best known upper bounds on the size of covering arrays. Consequently, very few better bounds were found by us. Those bounds that we improved upon were for arrays with parameters k and g not considered by Stevens and Nurmela.

The bounds improved by our metaheuristic searches are presented in Table 4.1. Of these sixteen improvements, the first was found by both the SA algorithm and

Value Improved	New Bound	Previous Bound
$ca(16, 6 : 1)$	65 rows	69 rows
$ca(16, 7 : 1)$	88 rows	91 rows
$ca(16, 8 : 1)$	113 rows	120 rows
$ca(17, 8 : 1)$	116 rows	120 rows
$ca(18, 8 : 1)$	118 rows	120 rows
$ca(16, 9 : 1)$	145 rows	153 rows
$ca(17, 9 : 1)$	148 rows	153 rows
$ca(18, 9 : 1)$	151 rows	153 rows
$ca(16, 10 : 1)$	177 rows	180 rows
$ca(13, 11 : 1)$	198 rows	231 rows
$ca(14, 12 : 1)$	205 rows	231 rows
$ca(15, 12 : 1)$	210 rows	231 rows
$ca(16, 12 : 1)$	216 rows	231 rows
$ca(17, 12 : 1)$	221 rows	231 rows
$ca(18, 12 : 1)$	225 rows	231 rows
$ca(15, 13 : 1)$	253 rows	255 rows

Table 4.1: New covering array bounds discovered by our metaheuristic searches.

the TS algorithm in 422568 moves, 638 seconds and 2129 moves, 83286 seconds, respectively. The other fifteen results were only found by the simulated annealing algorithm. Clearly, this does not mean that the TS algorithm can not find these arrays, but that, given the parameter sets we supplied, it was unable to locate them in a reasonable amount of time.

Other methods were used to improve the most recent table of bounds we could locate, found in [2]. We applied Stevens's Blocksize recursive algorithm to obtain better bounds for $26 \leq k \leq 30$, $g = 5$, and for $17 \leq k \leq 30$, $g = 6$. Also, the hole of an $ITD(5, 18, 4)$ was filled to get $ca(k, g : 1) = 324$ for $g = 18$, $k = 4, 5$. These methods are described explicitly in [24].

For a complete table of the best known upper and lower bounds on covering arrays for $g \leq 20$ and $k \leq 30$, please refer to Table B.2 and Table B.3.

4.3 Packing Arrays

Recently, packing arrays have been much less studied than covering arrays. Considering that the possible sizes of packing arrays are bounded both above and below, as opposed to covering arrays which can be as large as required, we saw a good opportunity to use the metaheuristic search techniques to tighten whatever bounds were loose.

The most recent table detailing the best known bounds on the size of packing arrays is found in [24]. This table contains 60 loose lower bounds. After performing our metaheuristic search, 53 of these lower bounds were improved. Moreover, 13 of these improvements led to tight bounds.

Of the 53 new best bounds, 31 were found by the SA algorithm, 37 were found by the TS algorithm and 26 were found by the GA algorithm. Furthermore, fourteen of these new bounds were located solely by tabu search, seven new bounds were located solely by simulated annealing and another seven were located solely by the genetic algorithm. In five cases, the genetic algorithm was the only algorithm of the three unable to find the new bound within the allotted number of trials. This situation also occurred with the annealing and tabu algorithms, once apiece. One interesting point to note is that in the cases where both the genetic algorithm and the simulated annealing algorithm found a new bound, the genetic algorithm required 40 percent as many generations as the SA algorithm required moves to locate the solution. Considering the difference in time between a single run through the annealing loop and the processing time for an entire generation in the genetic algorithm, this demonstrates how much more time it took the GA algorithm to find the same solution.

For a table of the best known upper and lower bounds on packing arrays for $g \leq 7$ and $k \leq 29$, please refer to Table B.4.

Chapter 5

Heterogeneous Alphabet Sizes

One of the primary motivations for the study of covering arrays is to attempt to develop efficient schemes for testing software and network stability. As stated in Chapter 1, one can represent each of the k nodes in a network or k variables in a computer program by a column in a covering array, where the nodes or variables can each be in one of g states. Then, each row of the array represents a trial in a testing scheme where the entry with coordinates (i, j) corresponds to placing node or variable i in state j. The ordered pair covering property of the array then guarantees that after all of the tests have been performed, each pair of nodes will have taken on every possible ordered pair of states. Unfortunately, the vast majority of these real-world systems do not contain objects which all take on the same fixed number of states. This fact suggests that most of what is already known about covering arrays may be too simplified to be widely applicable.

In order to construct covering arrays which are more useful for creating real-world tests, one must consider cases where column i has alphabet size g_i, where it is possible that $g_i \neq g_j$ for two columns i and j in the array. We extend our definition of a strength 2 covering array to include these cases.

Definition 5.1. *A strength 2 covering array of type $\prod_{i=1}^{k} g_i$, denoted symbolically as $CA(\prod_{i=1}^{k} g_i : n)$, is a $b \times k$ array A. The entries in column i are filled from a g_i-ary alphabet, which is taken to be Z_{g_i}, by convention. Given any two columns i and j and any ordered pair (u, v) from $Z_{g_i} \times Z_{g_j}$, there exists at least one row r such that entry*

$a_{r,i} = u$ *and entry* $a_{r,j} = v$. *Moreover, there is a set of* $n \leq min\{g_i\}$ *disjoint rows. The smallest number of rows possible in a* $CA(\prod_{i=1}^{k} g_i : n)$ *is denoted by* $ca(\prod_{i=1}^{k} g_i : n)$.

By this definition, a covering array $CA(k, g : n)$, as defined in Chapter 1, is now denoted by $CA(g^k : n)$. Furthermore, the Blocksize Recursive construction of Stevens can be modified as follows.

Theorem 5.1. *For given* $g_1 \leq g_2 \leq \ldots \leq g_k$ *and any* ρ, *we have*
$ca(g_1^\rho g_2^\rho \cdots g_k^\rho : n) \leq (ca(g_1 g_2 \cdots g_k : n) + ca(g_k^\rho : m) - m)$,
where m *is the number of disjoint rows in the appended array.*

The question of interest remains to try to find the smallest possible value for $ca(\prod_{i=1}^{k} g_i : n)$. For $k \leq 3$, the problem was solved entirely in [18] through the use of the following theorems.

Theorem 5.2. *[18]*

$ca(g_1 g_2 : n) = g_1 g_2$

Theorem 5.3. *[18]*

$ca(g_1 g_2 g_3 : n) = g_2 g_3$,
where it is assumed that $g_1 \leq g_2 \leq g_3$. *The only exception is for* $g_1 = g_2 = g_3 = n = 2$, *for which* $ca(2^3 : 2) = 5$.

In order to completely solve the problem for $k = 4$, two Lemmata are required. Lemma 5.1 considers the effect of increasing the size of some g_i on the value of $ca(\prod_{i=1}^{k} g_i : n)$, while Lemma 5.2 describes the result of decreasing the value of some g_i.

Lemma 5.1. *[18]*

Assuming that $g_1 \leq g_2 \leq \ldots \leq g_k$, *we have*
$ca(g_1 g_2 \ldots (g_i + e) \ldots g_k : min\{n + e, g_1\}) \leq ca(g_1 g_2 \ldots g_i \ldots g_k : n) + e \cdot g_{k'}$,
where $k' = k - 1$ *if* $i = k$ *and* $k' = k$ *otherwise.*

Proof. Given a $CA(g_1 g_2 \ldots g_k : n)$, add to it $e \cdot g_{k'}$ new rows to be filled in by the appropriate alphabet members. The only pairs that remain to be covered in order to

transform the old array into a $CA(g_1 g_2 \ldots (g_i + e) \ldots g_k : n)$ are those pairs with a coordinate in column i belonging to the alphabet $Z_{g_i+e} \setminus Z_{g_i}$. Fill the empty entries in column i with each element of this alphabet appearing $g_{k'}$ times apiece. As $g_{k'}$ is the largest possible alphabet size, regardless of the value of i, the set of $e \cdot g_{k'}$ rows is clearly sufficient to cover the remaining ordered pairs. As the symbols in the empty columns added can be placed independently, the set of disjoint blocks can be increased by no more than e rows.

□

Lemma 5.2. *[18]*

Assuming that $g_1 \leq g_2 \leq \ldots \leq g_k$, *we have* $ca(g_1 g_2 \ldots (g_i - e) \ldots g_k : min\{n, g_i - e\}) \leq ca(g_1 g_2 \ldots g_i \ldots g_k : n)$.

Proof. Given a $CA(g_1 g_2 \ldots g_k : n)$, relabel all of the entries in column i so that any entry from $Z_{g_i} \setminus Z_{g_i-e}$ is arbitrarily mapped to some element of Z_{g_i-e}. As the original structure was a covering array, so is the new structure as there are no new ordered pairs to cover. Also, given that the original covering array contained n disjoint rows, this quantity is only necessarily reduced in the case where the alphabet size $g_i - e$ becomes less than n.

□

With these two results in hand, we show the following.

Theorem 5.4. $ca(g_1 g_2 g_3 g_4 : n) = g_3 g_4$, *where it is assumed that* $n \leq g_1 \leq g_2 \leq g_3 \leq g_4$. *The only exceptions are as follows:*

- $ca(2^4 : 1) = 5$;
- $ca(2^4 : 2) = 6$;
- $ca(2^3 3^1 : 2) = 7$;
- $ca(3^4 : 2) = 10$;
- $ca(3^4 : 3) = 11$;

- $ca(3^3 4^1 : 3) = 13$;
- $ca(6^4 : n) = 37$ *for* $n \leq 5$; *and*
- $37 \leq ca(6^4 : 6) \leq 38$. *This is the only open case.*

Proof. For $g_3 \neq 2, 3$ or 6, then by results in [5, 28] there exists a set of two mutually orthogonal idempotent Latin squares of side g_3. Therefore, for all such g_3, there exists a $CA(g_3^4 : g_3)$ with g_3^2 rows. By Lemma 5.1 and Lemma 5.2, we have $ca(g_1 g_2 g_3 g_4 : n) = g_3 g_4$.

For $g_3 = 2$, if at least one of g_1 and g_2 are equal to 1, then the case in question is covered by either Theorem 5.3 or Theorem 5.2. THerefore, we may assume that the only cases left are the $CA(2^3 g_4^1 : 1)$ and $CA(2^3 g_4^1 : 2)$, where $g_4 \geq 2$. It has been shown that $ca(2^4 : 1) = 5$ [12] and $ca(2^4 : 2) = 6$ [24]. As for the smallest possible

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

Figure 5.1: A $CA(2^3 3^1 : 1)$ with six rows.

$CA(2^3 3^1 : 1)$, it must contain at least 6 rows to cover all ordered pairs from $Z_2 \times Z_3$. A $CA(2^3 3^1 : 1)$ with exactly 6 rows is presented in Figure 5.1.

Now consider the partially filled $CA(2^3 3^1 : 2)$ in Figure 5.2. To cover all ordered

0	0	0	0
x	x	x	0
1	1	1	1
y	y	y	1
			2
			2

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

Figure 5.2: An incomplete $CA(2^3 3^1 : 2)$ with six rows; a $CA(2^3 3^1 : 2)$ with seven rows.

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

Figure 5.3: A $CA(2^3 4^1 : 2)$ with eight rows.

pairs between column i and column 4, for $i < 4$, in only six rows, all entries labelled x must be 1 and all those labelled y must be 0. It is easy to see that this does not leave enough clearance in the final two rows to cover pairs $(0,1)$ and $(1,0)$ within the first three columns. A $CA(2^3 3^1 : 2)$ with exactly 7 rows is presented in Figure 5.2. Finally, a $CA(2^3 4^1 : 2)$, must contain at least 8 rows to cover all ordered pairs from $Z_2 \times Z_4$. A $CA(2^3 4^1 : 2)$ with precisely 8 rows is presented in Figure 5.3. Lemma 5.1 now shows that $ca(2^3 g_4^1 : n) = 2g_4$ for all $g_4 \geq 4$.

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

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

Figure 5.4: A $CA(2^1 3^3 : 2)$ with nine rows; a $CA(3^3 4^1 : 2)$ with twelve rows.

Consider the case $g_3 = 3$. An $OA(4,3)$ with nine rows exists, therefore by Lemma 5.1 and Lemma 5.2, we have $ca(g_1 g_2 3 g_4 : 1) = 3g_4$, for $g_1 \leq g_2 \leq 3 \leq g_4$. A $CA(2^1 3^3 : 2)$ must contain at least 9 rows to cover all ordered pairs from $Z_3 \times Z_3$. Figure 5.4 displays an optimal $CA(2^1 3^3 : 2)$ with precisely nine rows.

Combining the first array in Figure 5.4 with Lemma 5.1 and Lemma 5.2, we have $ca(2g_2 3g_4 : 2) = 3g_4$ for all $g_4 \geq 3$ and $2 \leq g_2 \leq 3$. It is known that $ca(3^4 : 2) = 10$ [24]. $CA(3^3 4^1 : 2)$ must contain at least 12 rows to cover all ordered pairs from $Z_3 \times Z_4$. The rightmost array in Figure 5.4 displays an optimal $CA(3^3 4^1 : 2)$ with exactly twelve rows. Applying Lemma 5.1 to the second array of Figure 5.4 yields $ca(3^3 g_4^1 : 2) = 3g_4$ for all $g_4 \geq 4$. In all cases where $n = g_3 = 3$, we must have $g_1 = g_2 = 3$ as well. It is known that $ca(3^4 : 3) = 11$. Applying Lemma 5.1 to the optimal $CA(3^4 : 2)$ with ten rows, we get $ca(3^3 4^1 : 3) \leq 13$. We need only show that a $CA(3^3 4^1 : 3)$ with twelve rows can not exist to complete this case.

0	0	0	0
x	x	x	0
x	x	x	0
1	1	1	1
y	y	y	1
y	y	y	1
2	2	2	2
z	z	z	2
z	z	z	2
		0	3
		1	3
		2	3

0	0	0	0
1	1	2	0
2	2	1	0
1	1	1	1
0	2	2	1
2	0	0	1
2	2	2	2
1	0	1	2
0	1	0	2
u	u	0	3
v		1	3
	v	2	3

Figure 5.5: Attempting to construct a $CA(3^3 4^1 : 3)$ with twelve rows.

Start with the leftmost incomplete $CA(3^3 4^1 : 3)$ shown in Figure 5.5. In order to cover all ordered pairs between column i and column 4, for $i < 4$, in exactly twelve rows, all entries labelled x must be 1 or 2, all those labelled y must be 0 or 2 and all those labelled z must be 0 or 1. Also, without loss of generality, it has been assumed that the final three entries of column 3 are $0, 1$ and 2, in that order. To cover the most ordered pairs possible within the first three columns, none of the rows of values x, y, or z should contain a string of three consecutive 0s, 1s or 2s, as those pairs are already covered. Without loss of generality, let the two rows of values x be $1, 1, 2$ and $2, 2, 1$, respectively, let the two rows of values y be $0, 2, 2$ and $2, 0, 0$, respectively and let the two rows of values z be $1, 0, 1$ and $0, 1, 0$, respectively. Without making this

assumption, some pair of columns 1, 2 and 3 would be left with at least four pairs to be covered in the last three rows of the array. The array created is the rightmost array depicted in Figure 5.5.

Each of these three newly filled pairs of rows covers four new ordered pairs and it is easy to see that no other arrangement of symbols can do better. Our choices for entries labelled x, y and z have left only six pairs uncovered within the first three columns. Clearly the final three entries in each of the first two columns must be 0, 1 and 2, in some order. Setting the two entries labelled u to be $1, 2$, respectively, covers three of these pairs, leaving only $(2, 1)$ in columns 1 and 2, $(0, 2)$ in columns 2 and 3, and $(0, 1)$ in columns 1 and 3 uncovered. Clearly, these can not be covered in the remaining two rows. In fact, the best that can be done is achieved by setting the two values v to be 0 and then completing the rest arbitrarily. This leaves one ordered pair uncovered. Therefore, one needs at least thirteen rows to construct such an array, giving us $ca(3^3 4^1 : 3) = 13$. Lemma 5.1, combined with the fifteen row $CA(3^3 5^1 : 3)$ in Figure 5.6 finally gives us that $ca(3^3 g_4 : 3) = 3g_4$ for all $g_4 \geq 5$.

0	0	0	0
1	1	2	0
2	2	1	0
1	1	1	1
0	2	2	1
2	0	0	1
2	2	2	2
0	1	0	2
1	0	1	2
0	0	2	3
1	2	0	3
2	1	1	3
0	0	1	4
1	1	1	4
2	2	2	4

Figure 5.6: A $CA(3^3 5^1 : 3)$ with fifteen rows.

Finally, we examine the case $g_3 = 6$. An $ITD(6, 4 : 2)$ with 32 rows exists as shown on the left side of Figure 5.7. Recall that this structure covers all ordered pairs

0	0	4	0
0	1	5	1
0	2	2	4
0	3	3	5
0	4	0	2
0	5	1	3
1	0	1	5
1	1	0	4
1	2	5	0
1	3	4	1
1	4	2	3
1	5	3	2
2	0	5	3
2	1	4	2
2	2	0	5
2	3	1	4
2	4	3	0
2	5	2	1
3	0	3	4
3	1	2	5
3	2	4	3
3	3	5	2
3	4	1	1
3	5	0	0
4	0	0	1
4	1	3	3
4	2	1	2
4	3	2	0
5	0	2	2
5	1	1	0
5	2	3	1
5	3	0	3

4	4	5	4
4	5	4	4
5	5	5	4
5	4	4	4

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

Figure 5.7: Subarrays required to construct a $CA(5^1 6^3 : 5)$ with 36 rows and a $CA(6^3 7^1 : 6)$ with 42 rows.

from $Z_6 \times Z_6$ in every pair of columns, except that no ordered pair from $\{4,5\} \times \{4,5\}$ occurs in any pair of columns. Collapse the set of symbols in the last column into the set of symbols of Z_5, making sure that the symbol 5 in row 15 is replaced by a 1 and that the symbol 5 in row 20 is replaced by a 3. Then, by affixing the second array in Figure 5.7 to the bottom of the collapsed ITD, we get a $CA(5^1 6^3 : 5)$ with 36 rows, where the disjoint rows are rows $1, 12, 15$ and 20 of the ITD and row 1 of the affixed array. Furthermore, if we instead affix the rightmost array in Figure 5.7 to the bottom of the ITD without collapsing the symbol sets, we get a $CA(6^3 7^1 : 6)$ with 42 rows, where the disjoint rows are rows $1, 8, 18$ and 27 of the ITD and rows 4 and 8 of the affixed array. It is currently unknown whether $ca(6^4 : 6)$ is 37 or 38. Both bounds are derived in [24]. Combining the $CA(5^1 6^3 : 5)$ of Figure 5.7 with Lemma 5.2 yields $ca(g_1 g_2 6^2 : n) = 36$ for all $n \leq g_1 < 6$, $g_2 \leq 6$ and $g_1 \leq g_2$. Applying Lemma 5.1 and Lemma 5.2 to the $CA(6^3 7^1 : 6)$ of Figure 5.7 gives $ca(g_1 g_2 6 g_4 : n) = 6 g_4$ for all $n \leq g_1 \leq g_2 \leq 6$ and $g_4 \geq 7$.

□

Most covering arrays with higher values of k can also be constructed in a similar fashion. Once an orthogonal array is constructed, the alphabet expansion and reduction lemmata can be used to obtain a lot of the remaining cases. Unfortunately, as k grows, there exists less $OA(k, g)$ as it becomes much more difficult to find smaller values of g for which there exist $k - 2$ mutually orthogonal Latin Squares of side g. Furthermore, for cases with higher n, we would like to start with either an optimal covering array or a set of $k - 2$ mutually orthogonal idempotent Latin Squares of side g to construct a covering array with the appropriate number of disjoint rows. As with the ordinary Latin squares, these large sets become scarce for larger k and we must rely more on results known about ordinary covering arrays.

Theorem 5.5. *For $g_4 \neq 4, 6, 10$, $ca(g_1 g_2 g_3 g_4 g_5 : n) = g_4 g_5$, where it is assumed that $n \leq g_1 \leq g_2 \leq g_3 \leq g_4 \leq g_5$. For $g_4 = 4$, $ca(g_1 g_2 g_3 4 g_5 : 1) = g_4 g_5$, where the g_i remained ordered in the same way as above. The only exceptions are as follows:*

- $ca(2^5 : 1) = ca(2^5 : 2) = 6$;

- $ca(2^4 3^1 : 2) = 7$;
- $ca(2^1 3^4 : 1) = 10$;
- $ca(3^5 : 1) = 11$;
- $ca(3^5 : 2) = ca(3^5 : 3) = 12$;
- $ca(3^4 4^1 : 2) = ca(3^4 4^1 : 3) = 13$;
- $10 \leq ca(2^1 3^4 : 2) \leq 11$;

This last one is the only open cases for $g_4 \leq 3$.

Proof. For $g_4 \neq 2, 3, 6, 10$, there exists a set of 3 mutually orthogonal Latin squares [5], from which an $OA(5, g_4)$ with g_4^2 rows can be constructed. Lemma 5.1 and Lemma 5.2 can be used to obtain the result for $n = 1$ and $g_4 \neq 2, 3, 6, 10$. Moreover, for $g_4 \neq 2, 3, 4, 6, 10$, there exists a set of three mutually orthogonal idempotent Latin squares [5], from which an $OA(5, g_4)$ with g_4^2 rows, g_4 of which are disjoint, can be constructed. The same lemmata are used to get the result for all $n \leq g_1$ and $g_4 \neq 2, 3, 4, 6, 10$.

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

Figure 5.8: A $CA(2^5 : 2)$ and a $CA(2^4 3^1 : 1)$ with six rows apiece.

For $g_4 = 2$, Katona shows in [12] that $ca(2^5 : 1) = 6$. In order to not reduce this situation to a case equivalent to one with lesser k, we may assume that $g_1 = g_2 = g_3 = 2$. Lemma 5.1 can be applied to the $CA(2^4 3^1 : 1)$ with six rows in Figure 5.8 to get $ca(g_1 g_2 g_3 2 g_5 : 1) = 2g_5$ for $g_1 \leq g_2 \leq g_3 \leq 2$ and $g_5 \geq 3$. A $CA(2^4 3^1 : 2)$ must have at least six rows, in order to cover all pairs from $Z_2 \times Z_3$. However, it was shown in the

previous proof that one requires at least seven rows to construct a $CA(2^3 3^1 : 2)$, which is a subarray of the object we are looking to construct. Therefore, $ca(2^4 3^1 : 2) \geq 7$. A $CA(2^4 3^1 : 2)$ and a $CA(2^4 4^1 : 2)$ with seven and eight rows, respectively, are presented in Figure 5.9. Finally, Lemma 5.1 can be applied to the $CA(2^4 4^1 : 2)$ to show $ca(g_1 g_2 g_3 2 g_5 : 2) = 2g_5$ for all remaining cases of g_1, g_2, g_3, and g_5.

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

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

Figure 5.9: A $CA(2^4 3^1 : 2)$ with seven rows and a $CA(2^4 4^1 : 2)$ with eight rows.

For $g_4 = 3$, [23] reports that $ca(3^5 : 1) = 11$. A $CA(2^2 3^3 : 2)$ with nine rows is presented in Figure 5.10. Together with Lemma 5.2, we have $ca(2^2 g_3 3^2 : n) = 9$ for $n \leq 2$ and $g_3 = 2, 3$ and $9 \leq ca(2^1 3^4 : 1) \leq 11$. It is also known that $ca(3^5 : 2) = 12$ and $ca(3^5 : 3) = 12$.

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

x	0	0	0	0
x	0	1	1	1
x	0	2	2	2
y	1	0	2	1
y	1	1	0	2
y	1	2	1	0
z	2	0	1	2
z	2	1	2	0
z	2	2	0	1

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

Figure 5.10: A $CA(2^2 3^3 : 2)$ with nine rows; an incomplete $CA(2^1 3^4 : 1)$; a $CA(2^1 3^4 : 1)$ with ten rows.

Applying Lemma 5.2 to the value $ca(3^5 : 2)$, we get $9 \leq ca(2^1 3^4 : 2) \leq 12$. Consider the incomplete $CA(2^1 3^4 : 1)$ in Figure 5.10. The only way to construct the $CA(3^4 : 1)$ subarray is to use the MOLS construction mentioned in Chapter 1[5]. Symbols 0 and 1 must each occur in each set of cells x, y and z. Choose a symbol, say 0, to be placed first, once in each set of three rows. Initially, 24 pairs are yet to be covered between the first column and each other column. Due to the structure of the MOLS which were used to build the ternary subarray, each symbol must occur exactly once in each column in each set of rows prefixed by x, y or z. Also, in each row prefixed by y or z, the symbols in columns $3, 4$ and 5 must all be different by orthogonality. This guarantees that placing a 0 once in each set of three commonly prefixed rows results in one of two cases: either one pair is covered three times in a single pair of columns or all but three pairs get covered, with each uncovered pair straddling a different pair of columns $(1, i)$, for $i = 3, 4, 5$. The first case covers 10 pairs, but requires a 0 to be placed in two extra cells in the first column, thereby forcing the usage of five rows in the array. The second case covers 9 pairs. Consider placing a 0 in the first and fourth cells of the first column. In order to avoid the first case, the third 0 must be placed in the second or third cell labeled z. Both cases leave some pair $(0, j)$ uncovered, for $j = 1$ or 2. Clearly, as each symbol only occurs once per row in each of columns $3, 4$ and 5, it must be impossible to cover the remaining pairs from $\{0\} \times Z_3$ in a single row. As a result, five rows are needed to cover all pairs of $\{0\} \times Z_3$. No assumption was made with regards to the symbol to be placed first and therefore, it must be impossible to cover all pairs from $Z_2 \times Z_3$ in less than ten rows. A $CA(2^1 3^4 : 1)$ with ten rows is presented in Figure 5.10. This result also shows that $10 \leq ca(2^1 3^4 : 2) \leq 11$ and that $10 \leq ca(2^1 3^4 : 3) \leq 12$. We do not believe that a $CA(2^1 3^4 : 2)$ with less than 11 rows can be constructed. In fact, the best that we have achieved is a $CA(2^1 3^4 : 2)$ with 10 rows having 3 pairs uncovered, all within the same pair of columns.

A $CA(3^4 4^1 : 3)$ must have at least twelve rows, in order to cover all pairs from $Z_3 \times Z_4$. It was shown in the last proof that thirteen rows are needed to construct a $CA(3^3 4^1 : 3)$. This is a subarray of the object we are looking to construct, so we have $ca(3^4 4^1 : 3) \geq 13$. A $CA(3^4 4^1 : 3)$ with thirteen rows is presented in Figure 5.11. The

cells labelled x may be filled arbitrarily from Z_3.

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

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

0	0	0	0	0
1	1	2	2	0
2	2	1	1	0
0	2	0	2	1
1	1	1	1	1
2	0	2	0	1
0	1	1	0	2
1	0	0	1	2
2	2	2	2	2
0	0	2	1	3
1	2	1	0	3
2	1	0	2	3
0	0	1	2	4
1	1	2	0	4
2	2	0	1	4

Figure 5.11: A $CA(3^4 4^1 : 1)$ with twelve rows; a $CA(3^4 4^1 : 3)$ with thirteen rows; a $CA(3^4 5^1 : 3)$ with fifteen rows.

Also shown in Figure 5.11 are a $CA(3^4 4^1 : 1)$ with twelve rows, as well as a $CA(3^4 5^1 : 3)$ with fifteen rows. These two arrays, together with Lemma 5.1 show the result for all arrays with $g_4 = 3$, completing the proof.

□

Chapter 6

Conclusion

6.1 Summary of Results

We implemented three metaheuristic algorithms to search for better bounds on covering and packing arrays. The simulated annealing, tabu search, and genetic algorithms were implemented, taking into account comments from literature and experimental data in choosing the algorithms' input parameters. All three were able to find new bounds on the size of packing arrays with the TS algorithm finding the majority of the new bounds. The tabu search algorithm also improved on the bound for $CA(16, 6)$, while the simulated annealing algorithm found new bounds on that and four other covering arrays.

After comparing the three algorithms, we found out that the genetic algorithm was ineffective at finding quality arrays, relative to the performance of the SA and TS algorithms. Not only did the genetic algorithm take more time to execute moves, but it seemed to require more moves to find a good covering array. The SA algorithm, capable of executing many moves in a very short time, was very useful for finding covering arrays of various sizes. However, when the size of an array's neighbourhood was smaller, the TS algorithm was often able to find much better arrays, as shown in Tables A.5 and A.14. In particular, the results of Table A.14 suggest that while the SA algorithm operates very quickly, it does not converge to optimal regions in the search space as well as the TS algorithm. In this case, by a smaller neighbourhood

we mean arrays with $k \leq 9$ and $g \leq 7$. These findings were purely experimental.

One way to search for covering arrays while combining the strengths of these two algorithms would be to consider some values of M, k and g and to run the SA algorithm from a very large upper bound for b, say $b > \frac{g \log_2 k}{2}$. While processing through the binary search, either the algorithm would find covering arrays at each step or it would output the best score it was able to find before performing M moves. Values of b for which the SA algorithm left only a small number of pairs uncovered could be attempted again, using the more powerful TS algorithm. This process would be particularly useful as the SA algorithm could provide a quick glimpse at those values of b for which a $CA(k, g : n)$ with b rows might exist, at which point the much slower TS algorithm could used to perform a more thorough search.

We believe that while our algorithms may not have produced as many new bounds as expected, metaheuristic search algorithms are useful for finding better bounds on covering and packing arrays where the best known constructions do not produce very tight bounds. Metaheuristic searches have been performed before by Stevens and Nurmela and their algorithms were very successful for smaller parameter sets. Unfortunately, the amount of memory required to store information needed for more complex heuristics may become too great for larger parameter sets, possibly causing their algorithms to run very slowly in these cases. Our algorithms used very simple heuristics and were able to improve on bounds for structures as large as an 11×198 array with entries from a 13-ary alphabet.

After many trials with large values of M, the tabu search algorithm was unable to improve the known bounds on certain small covering and packing arrays, suggesting that the current bounds might be tight. The covering array bounds that could not be improved upon after this sort of extensive testing were the following: $ca(8, 3 : 1) \leq 13$, $ca(5, 4 : 3) \leq 17$, $ca(5, 4 : 4) \leq 18$ and $ca(9, 7 : 1) \leq 63$. In particular, if this last bound were tight, it would refute a conjecture of Stevens's, that for a fixed g and n, $ca(k + 1, g : n) - ca(k, g : n) \leq g - 1$. For packing arrays, some of the seemingly unbeatable bounds were $pa(5, 4 : 2) \geq 12$, $pa(5, 4 : 3) \geq 9$, $pa(8, 5 : 3) \geq 9$, $pa(9, 6 : 4) \geq 12$, and $pa(10, 6 : 4) \geq 9$.

As well as using search algorithms to construct covering and packing arrays, we

considered the problem of determining the minimum number of rows for a covering array with potentially different alphabet sizes for each column. Aside from one case, the problem for $k \leq 4$ was completely solved. This was accomplished by applying Lemma 5.2 and Lemma 5.1 to known orthogonal and covering arrays and constructing the rest of the examples by hand. We also presented some partial results for the case $k = 5$, not handling the solutions to the cases where the second largest alphabet size was 4, 6 or 10.

6.2 Future Endeavours

The amount of time required to proceed through a full binary search for a $CA(k, g : n)$ for larger values of k and g limited the amount of parameter sets that could be tested over the course of researching and writing this thesis. It is possible that our algorithms are able to produce better bounds for many as yet untested parameter sets. We will continue to search for new bounds until the date of publication.

The major barrier to finding more results using metaheuristics search algorithms is that of time. Many of the structures created in the code could have been eliminated to reduce runtime, but they were kept as it was never decided whether or not they would be required. Within the *BKG_Array* structures, only one of the score arrays was ever used to evaluate the objects. Nonetheless, we would like to explore ways of using the other score arrays to modify the algorithm heuristics. For example, rather that having the TS algorithm randomly select an array at each step from a set of equally fit neighbouring arrays, the information contained in the structure which counts the number times each member of the alphabet occurs in any column could be used to preserve some sort of balance property. While including these score arrays in the *BKG_Array* class does not affect the complexity of any of the algorithms, there is a time cost associated with updating them, which increases with k, g and b.

The one member function which is the most expensive is *fill_pair_arrays*, which fills the pertinent score arrays after the generation of a *BKG_Array*. This function executes $O(k^2(g^2 + b))$ operations when called. In the annealing and tabu algorithms, this is only performed once and outside of the main loop, thereby having

no influence on the complexity of these algorithms. Unfortunately, in the genetic algorithm, depending on the mating method used, $O(S)$ offspring arrays are produced and this in turn raises the complexity of the algorithm to $O(SMk^2(g^2+b))$. We could not devise an alternative to tabulating the pairs occurring by using the count array *ord_pair_chk_array*. If there were a local way to count the occurrence of pairs, then the complexity of the genetic algorithm would be reduced to $O(SMbk^2)$.

After the three randomized search algorithms were selected, we wanted to test the effectiveness of variations on these algorithms. We were able to implement a few variations on the genetic algorithm, for different population culling methods and for different crossover strategies. Unfortunately, none of these implementations were significantly more effective than any other, to the point where it could compete with the SA and TS algorithms. Some alternative implementations of the annealing and tabu search algorithms were mentioned in Chapter 2, but none were attempted. It would be interesting to see if any of these alternate versions of the more successful algorithms are any more effective than their counterparts implemented over the course of this thesis.

6.3 Two Generalizations of Covering Arrays

The primary application of covering arrays discussed in Chapter 1.3 was that of software or network testing. Unfortunately, as mentioned in Chapter 5, most real life systems of this sort do not conform to the basic definition of a covering array. In order to make covering arrays fully applicable, one must consider the case of generalized covering arrays with non-uniform alphabet sets for each column. There is much work left to be done on this problem. As k becomes larger, we have fewer values of g for which there exists a set of $k-2$ MOLS and therefore, more special cases that must be constructed by some alternative means. While most of the examples displayed in the Figures of Chapter 5 were constructed by hand, this becomes a difficult task for even $k = 5$ and a second largest alphabet size g_i of six. Furthermore, the constructions developed for basic covering arrays may not work as well when generalized for heterogeneous alphabet covering arrays. We propose that for these special

cases, an alternative to developing constructions would be to use metaheuristic search techniques. In particular, the code we have written to search for covering arrays with uniform alphabet size may be modified to construct these objects as well. While the algorithms themselves would sustain only minor changes, the method of evaluating and storing the new arrays would be very different and it would take no small amount of thought and time to design these structures in a clever way.

A second variation of a covering array not discussed in this thesis is that of the structured network array. Consider the case of a network which requires testing, but it is known in advance that certain pairs of nodes do not interact. An optimal test suite for such a network would be a sort of covering array where the pair covering property needs only occur between pairs of columns corresponding to nodes which are known to interact with one another. Meagher [25] has already made appropriate modifications to our code to handle this problem. Representing the network as a graph with edges connecting vertices corresponding to interacting nodes, it can be shown that the size of the covering array is related to $ca(k, g : 1)$, where k is the colorability of the graph. The problem has already been solved for many classes of graphs, but much is left to be done in terms of solving this problem completely.

Appendix A

Tables of Parameter Test Results

In order to determine optimal ranges for the input parameters of each of the main metaheuristics employed throughout this thesis, it was necessary to conduct several tests. These tests provide much insight on the effect of the variation of these parameters with respect to the algorithms' runtimes and effectiveness. Further tests were also required in order to compare the algorithms to one another.

The tables in this appendix contain the data noted as a result of these extensive tests.

	$CA(6,4:1)$				$CA(7,5:1)$				$PA(7,5:1)$	
	$M=10^4$		$M=10^5$		$M=10^5$		$M=10^6$		$M=10^4$	
δ_t	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best
0.01	–	–	–	–	42.4	42	43.3	43	8.0	10
0.1	24.0	24	24.0	24	42.0	41	42.8	42	8.0	9
0.25	–	–	–	–	42.4	42	42.4	41	7.6	8
0.5	23.3	23	24.0	24	41.8	40	41.4	40	8.6	9
0.75	20.8	20	19.6	19	33.8	33	34.2	33	11.2	12
0.9	20.8	20	19.8	19	33.6	33	33.6	33	11.2	12
0.925	20.8	20	20.8	20	33.4	33	34.0	33	11.0	11
0.95	20.8	20	20.0	19	34.0	33	33.2	32	11.4	12
0.975	20.8	20	20.2	20	33.8	33	33.0	33	11.4	12
0.99	20.2	19	20.2	20	34.4	34	33.0	32	11.6	13
0.9925	20.0	19	20.4	20	34.0	33	33.4	32	11.4	12
0.995	20.4	19	20.0	19	34.0	34	33.8	33	11.0	11
0.9975	20.8	20	20.0	19	33.4	32	34.0	32	11.2	12
0.999	21.0	21	20.4	20	34.2	33	33.4	33	11.2	12

Table A.1: Arrays found by the SA algorithm for $t_0 = 1$.

	$CA(6,4:1)$				$CA(7,5:1)$				$PA(7,5:1)$	
	$M=10^4$		$M=10^5$		$M=10^5$		$M=10^6$		$M=10^4$	
δ_t	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best
0.01	23.8	23	–	–	41.8	40	43.3	42	7.8	9
0.1	–	–	23.5	23	42.6	41	41.6	40	7.0	9
0.25	24.0	24	24.0	24	42.7	42	42.2	40	8.6	9
0.5	23.7	23	23.7	23	41.8	41	42.2	42	8.4	9
0.75	20.6	19	20.8	20	33.4	33	33.8	33	11.2	12
0.9	20.4	19	20.4	20	33.2	33	33.6	33	11.6	12
0.925	20.6	20	20.6	20	33.6	33	33.4	33	11.0	12
0.95	20.6	19	20.4	20	33.6	33	33.2	32	11.8	13
0.975	20.6	20	20.4	20	34.0	33	33.4	33	10.8	11
0.99	20.4	19	20.6	20	33.6	32	34.0	32	11.0	11
0.9925	20.6	20	20.2	19	33.8	33	33.8	32	11.4	13
0.995	20.6	19	20.0	19	33.6	33	33.0	33	11.2	12
0.9975	20.6	20	20.0	20	33.8	33	32.6	32	11.4	12
0.999	20.6	20	20.2	19	33.6	32	33.2	33	10.8	11

Table A.2: Arrays found by the SA algorithm for $t_0 = 10$.

	$CA(6,4:1)$				$CA(7,5:1)$				$PA(7,5:1)$	
	$M = 10^4$		$M = 10^5$		$M = 10^5$		$M = 10^6$		$M = 10^4$	
δ_t	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best
0.01	–	–	24.0	24	42.0	42	42.3	40	8.0	9
0.1	24.0	24	–	–	42.0	41	42.4	42	8.6	10
0.25	24.0	24	23.5	23	42.0	40	43.0	42	8.4	9
0.5	–	–	23.5	23	42.0	41	42.6	42	8.2	9
0.75	20.4	19	20.4	20	33.4	33	33.0	32	11.2	12
0.9	20.6	20	20.8	20	33.8	33	33.4	32	11.6	12
0.925	20.6	20	20.4	19	33.2	33	33.6	33	11.0	12
0.95	20.6	20	20.4	19	34.0	33	34.2	33	11.4	12
0.975	20.4	19	20.4	19	34.2	33	33.2	33	11.4	12
0.99	20.2	19	19.8	19	34.6	34	33.4	33	11.0	11
0.9925	20.6	20	20.0	19	33.4	33	33.4	33	11.4	12
0.995	20.6	20	21.0	20	34.0	33	33.2	32	11.0	11
0.9975	20.6	20	20.4	20	34.8	34	33.0	32	11.2	12
0.999	21.4	21	20.6	20	33.6	32	33.6	33	10.8	11

Table A.3: Arrays found by the SA algorithm for $t_0 = 100$.

	$CA(6,4:1)$				$CA(7,5:1)$				$PA(7,5:1)$	
	$M = 10^4$		$M = 10^5$		$M = 10^5$		$M = 10^6$		$M = 10^4$	
δ_t	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best
0.01	24.0	24	24.0	24	41.6	40	42.0	40	8.8	11
0.1	24.0	24	23.7	23	42.2	41	42.8	42	8.0	10
0.25	–	–	24.0	24	43.3	42	42.2	42	8.6	9
0.5	–	–	–	–	43.0	42	42.4	42	9.0	10
0.75	20.8	20	20.8	20	34.0	33	33.0	33	11.6	12
0.9	20.8	20	19.8	19	33.6	33	33.2	33	11.8	13
0.925	20.8	19	20.2	20	33.6	33	33.4	33	11.0	11
0.95	20.4	20	20.0	19	34.0	33	33.2	33	11.6	12
0.975	20.8	20	20.0	20	33.8	33	33.4	33	11.6	12
0.99	20.6	20	20.6	20	34.4	33	33.2	33	11.0	11
0.9925	20.8	20	19.8	19	33.6	33	33.8	32	11.2	12
0.995	21.2	21	20.4	20	34.0	33	33.8	33	11.2	12
0.9975	21.0	21	20.3	20	33.0	30	33.2	33	11.4	12
0.999	22.8	22	20.4	20	33.4	33	33.8	33	10.6	11

Table A.4: Arrays found by the SA algorithm for $t_0 = 1000$.

	$CA(6,4:1)$				$CA(7,5:1)$				$PA(7,5:1)$	
	$M = 500$		$M = 1000$		$M = 1000$		$M = 5000$		$M = 1000$	
L	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best
1	20.0	19	20.4	20	33.2	33	32.8	32	13.2	14
3	20.2	19	20.6	20	33.0	32	31.4	30	13.4	14
5	20.2	19	20.2	19	32.6	30	30.2	29	13.0	14
7	20.0	20	19.8	19	33.2	32	30.6	30	13.6	14
10	20.2	19	20.2	20	32.6	32	29.6	29	13.2	14
30	20.2	19	20.4	19	31.8	30	30.0	29	12.4	13
50	20.6	20	20.2	19	33.0	32	29.6	29	12.6	13
70	20.4	20	19.8	19	32.4	32	30.0	29	12.6	14
90	19.8	19	19.6	19	31.6	30	30.2	29	12.2	13
110	20.0	19	20.2	19	32.0	30	30.6	29	12.0	12
130	20.6	19	19.8	19	32.8	32	30.2	29	12.0	12
150	19.8	19	20.0	19	33.2	33	32.2	32	12.2	13
200	20.2	19	20.6	20	33.6	33	32.4	32	12.0	12
500	20.8	20	21.0	21	34.2	33	33.8	33	11.4	12

Table A.5: Arrays found by the tabu search algorithm with lifetime L.

	$CA(6,4:1)$				$CA(7,5:1)$					
	$M = 1000$		$M = 5000$		$M = 500$		$M = 1000$		$M = 5000$	
S	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best
12	22.6	22	21.0	20	40.0	39	38.0	37	36.2	35
20	22.2	21	20.4	19	39.0	37	37.4	37	35.6	35
40	21.8	21	20.6	20	38.2	38	36.2	36	35.2	35
60	21.6	21	20.6	20	37.6	37	37.0	37	35.6	35
80	20.6	19	20.6	20	37.4	37	36.6	36	34.8	34
100	21.4	21	20.2	20	37.6	37	36.4	36	34.6	33
140	21.0	21	20.2	19	37.2	37	36.6	36	35.0	34
200	21.0	21	20.2	20	37.0	36	35.4	35	34.6	34
300	21.2	21	20.2	19	37.0	37	35.4	35	34.2	34
400	19.8	19	20.6	20	37.0	36	35.0	35	35.0	35
500	20.8	19	19.6	19	36.6	36	35.8	35	34.6	34

Table A.6: Covering arrays found by the genetic algorithm, using tournament selection, the point crossover method and population size S.

	$PA(7,5:1)$			
	$M = 500$		$M = 1000$	
S	Avg.	Best	Avg.	Best
12	10.2	11	10.8	11
20	10.4	11	11.0	11
40	10.8	11	11.0	12
60	11.4	12	11.0	11
80	10.8	11	11.0	11
100	11.2	12	11.4	12
140	11.0	11	11.0	11
200	11.2	12	11.8	13
300	11.0	11	11.4	12
400	11.2	12	11.4	12
500	11.4	12	11.6	12

Table A.7: Packing arrays found by the genetic algorithm, using the point crossover method and population size S.

	$M = 500$		$M = 1000$		$M = 5000$	
S	Avg.	Total	Avg.	Total	Avg.	Total
12	34.4	172	62.0	310	212.0	1060
20	57.2	286	81.2	406	368.2	1841
40	120.8	604	159.6	798	712.0	3560
60	134.4	672	201.6	1008	1153.4	5767
80	175.2	876	285.4	1427	1860.2	9301
100	233.0	1165	377.2	1886	2085.4	10427
140	308.0	1540	469.0	2345	3050.4	15252
200	424.8	2124	787.6	3938	3644.4	18222
300	594.0	2970	1303.0	6515	8051.4	40257
400	840.0	4200	1679.8	8399	10821.8	54109
500	1022.0	5110	2129.4	10647	16494.8	82474

Table A.8: Time (in seconds) required for the GA algorithm to locate a $CA(7,5:1)$ of optimal size, for various values of population size S.

$S \cdot M = 100,000$				
S	M	Avg. Rows	Least Found	Time
20	5000	35.6	35	1841
100	1000	36.4	36	1886
200	500	37.0	36	2124
$S \cdot M = 200,000$				
S	M	Avg. Rows	Least Found	Time
40	5000	35.2	35	3560
200	1000	35.4	35	3938
400	500	37.0	36	4200

Table A.9: Measuring the effectiveness of the GA algorithm in constructing a $CA(7, 5 : 1)$ for different values of S and M, where the quantity $S \cdot M$ is constant.

	Quick Conv. Method			Tournament Method		
Array	Avg. Rows	Least Found	Time	Avg. Rows	Least Found	Time
$CA(7, 4 : 1)$	22.2	22	19653	22.2	22	14661
$CA(8, 5 : 1)$	37.3	36	46854	36.6	36	39288
$CA(5, 6 : 1)$	43.0	42	26783	43.0	41	21531
$PA(5, 4 : 1)$	14.3	15	440	12.5	15	442
$PA(6, 5 : 1)$	14.6	16	2391	14.0	15	2383
$PA(5, 6 : 1)$	26.3	28	2250	25.3	27	2015

Table A.10: Arrays found by two genetic algorithms, using the point crossover method.

	Quick Conv. Method			Tournament Method		
Array	Avg. Rows	Least Found	Time	Avg. Rows	Least Found	Time
$CA(7, 4 : 1)$	22.4	22	20113	22.1	22	17762
$CA(8, 5 : 1)$	36.9	36	48312	37.0	36	37397
$CA(5, 6 : 1)$	42.8	42	29688	43.2	42	21774
$PA(5, 4 : 1)$	13.9	15	416	13.4	15	581
$PA(6, 5 : 1)$	15.2	19	2572	14.7	16	2149
$PA(5, 6 : 1)$	26.1	27	2167	25.4	27	1899

Table A.11: Arrays found by two genetic algorithms, using the row crossover method.

Array	Quick Conv. Method			Tournament Method		
	Avg. Rows	Least Found	Time	Avg. Rows	Least Found	Time
$CA(7,4:1)$	22.4	22	16988	22.2	22	20999
$CA(8,5:1)$	36.9	36	43933	36.6	35	36540
$CA(5,6:1)$	42.8	42	24763	42.8	42	22097
$PA(5,4:1)$	13.5	15	593	13.8	15	431
$PA(6,5:1)$	15.2	19	2508	13.8	15	2127
$PA(5,6:1)$	25.7	27	2536	25.1	27	2050

Table A.12: Arrays found by two genetic algorithms, using the column crossover method.

Time	Algorithm	Score	Moves	Algorithm	Score	Moves
60	ANNEAL	21.4	8.5×10^5	GENETIC(PT)	1459.0	77.4
	TABU	1476.4	51.0	GENETIC(ROW)	1468.2	77.6
				GENETIC(COL)	1448.4	77.6
300	ANNEAL	1.0	4.4×10^6	GENETIC(PT)	406.2	908.0
	TABU	244.6	244.6	GENETIC(ROW)	407.4	913.2
				GENETIC(COL)	407.8	916.6
1800	ANNEAL	2.6	2.7×10^7	GENETIC(PT)	312.4	2489.4
	TABU	37.4	1453.0	GENETIC(ROW)	316.4	2494.8
				GENETIC(COL)	317.6	2494.8
3600	ANNEAL	1.6	5.5×10^7	GENETIC(PT)	202.8	4995.0
	TABU	17.4	2902.8	GENETIC(ROW)	212.6	5003.6
				GENETIC(COL)	207.6	4996.8
7200	ANNEAL	0.4	1.1×10^8	GENETIC(PT)	146.8	10000.8
	TABU	7.2	5764.0	GENETIC(ROW)	144.2	10010.2
				GENETIC(COL)	145.6	9994.2
10800	ANNEAL	2.6	1.6×10^8	GENETIC(PT)	121.2	15012.0
	TABU	5.6	8641.6	GENETIC(ROW)	122.4	15032.8
				GENETIC(COL)	124.0	15009.4

Table A.13: Best score found for a CA(13,11:1) with 198 rows in a fixed amount of time, given in seconds.

Time	Algorithm	Score	Moves	Algorithm	Score	Moves
60	ANNEAL	56.3	1.1×10^6	GENETIC(PT)	157.8	460.2
	TABU	62.7	553.7	GENETIC(ROW)	154.5	464.5
				GENETIC(COL)	156.2	456.7
300	ANNEAL	54.0	5.5×10^6	GENETIC(PT)	102.8	2346.0
	TABU	44.0	2768.7	GENETIC(ROW)	100.5	2352.7
				GENETIC(COL)	98.8	2353.5
1800	ANNEAL	57.8	3.4×10^7	GENETIC(PT)	73.3	14159.7
	TABU	31.5	16659.8	GENETIC(ROW)	79.3	14160.7
				GENETIC(COL)	75.5	14139.7
3600	ANNEAL	57.7	6.9×10^7	GENETIC(PT)	74.3	28575.0
	TABU	28.3	33391.0	GENETIC(ROW)	72.5	28590.0
				GENETIC(COL)	69.7	28543.8

Table A.14: Best score found for a CA(9,7:1) with 62 rows in a fixed amount of time, given in seconds.

$PA(7, 6:1)$, 19 rows			
Algorithm	Time	Moves	σ
ANNEAL	8.0	133903.4	169766.8
TABU	11.4	352.7	258.2
GENETIC(PT)	7112.9	54745.4	89514.8
GENETIC(ROW)	8519.6	103194.4	103676.2
GENETIC(COL)	5826.6	69807.1	85042.0
$CA(5, 6:1)$, 42 rows			
ANNEAL	5.1	108952.7	94729.3
TABU	17.2	538.7	355.9
GENETIC(PT)	11092.4	218285.8	288874.4
GENETIC(ROW)	19511.4	385690.5	642029.5
GENETIC(COL)	33228.6	657088.6	1354448.7

Table A.15: Number of moves and amount of time needed, averaged over five trial runs, for the metaheuristics to locate a $PA(7, 6:1)$ with 19 rows and a $CA(5, 6:1)$ with 42 rows.

Appendix B

Tables of Bounds

In this Appendix, we present tables containing the best known upper and lower bounds on the size of covering and packing arrays. Table B.1 displays the key for the results presented in Tables B.2 and B.3. In Table B.4, all bounds without superscript indices were obtained from tables in [24]. Those given a superscript of x were improved by constructions in [26]. Those given a superscript of y were improved by metaheuristic search.

Index	Reference
a	orthogonal array exists
b	no pair of MOLS(2) exists
c	Stevens [24]
d	Applegate [23]
e	Chateauneuf and Kreher [2]
f	AETG: Dalal et. al [3, 4, 6]
g	Our metaheuristic searches
h	Katona [12]
i	Nurmela [19]
j	symbol collapsing

Table B.1: Key for Table B.2 and Table B.3.

$k\backslash g$	2	3	4	5	6	7	8	9	10	11
3	4^a	9	16	25	36^a	49	64	81	100	121
4	5^b	9^a	16	25	37^c	49	64	81	100^a	121
5	6	11^d	16^a	25	$37,39^i$	49	64	81	100,102	121
6	6	12	19^c	25^a	$37,41^i$	49	64	81	$100,102^c$	121
7	6	12	$19,21^c$	29^c	37,42	49	64	81	100,120	121
8	6	12,13	$19,23^c$	$29,33^i$	$39^c,42^i$	49^a	64	81	100,120	121
9	6	$12,13^c$	19,24	$29,35^i$	$39,48^e$	$52^c,63$	64^a	81	100,120	121
10	6^h	$12,14^i$	$19,24^i$	$29,37^i$	$39,52^i$	$52,63^c$	67^c, 80	81^a	100,120	121
11	7	12,15	$19,25^i$	$29,38^i$	$39,55^i$	$52,73^i$	67, 80^c	84^c,120	100,120	121
12	7	12,15	$19,26^i$	$29,40^i$	$39,57^i$	$52,76^i$	67, 99^i	84,120	103^c,120	121^a
13	7	12,15	19,27	$29,41^i$	$39,58^i$	$52,79^i$	$67,102^i$	$84,120^j$	$103,120^c$	124^c,**198**^{g}
14	7	12,15	$19,27^i$	$29,42^i$	$39,60^i$	$52,81^i$	$67,104^i$	$84,131^i$	$103,162^i$	124,**205**^{g}
15	7^h	12,15	19,28	$29,43^i$	$39,61^i$	$52,83^i$	$67,107^i$	$84,135^i$	$103,166^i$	124,**210**^{g}
16	8	12,15	19,28	29,45	39,**65**^{g}	52,**88**^{g}	67,**113**^{g}	84,**145**^{g}	103,**177**^{g}	124,**216**^{g}
17	8	12,15	19,28	29,45	39,71	52,91	67,**116**^{g}	84,**148**^{g}	103,180	124,**221**^{g}
18	8	12,15	19,28	29,45	39,71	52,91	67,**118**^{g}	84,**151**^{g}	103,180	124,**225**^{g}
19	8	12,15	19,28	29,45	39,71	52,91	67,120	84,153	103,180	124,231
20	8	$12,15^i$	$19,28^c$	29,45	39,71	52,91	67,120	84,153	$103,180^j$	124,231
21	8	12,17	19,31	29,45	39,73	52,91	67,120	84,153	103,202	124,231
22	8	12,17	19,31	29,45	39,73	52,91	67,120	84,153	103,202	124,231
23	8	12,17	19,31	29,45	39,73	52,91	67,120	84,153	103,202	124,231
24	8	$12,17^i$	19,31	29,45	39,73	52,91	67,120	84,153	103,202	124,231
25	8	12,18	$19,31^c$	29,45	39,74	52,91	67,120	84,153	103,202	124,231
26	8	12,18	19,32	29,45	39,74	52,91	67,120	84,153	103,202	124,231
27	8	12,18	19,32	29,45	39,74	52,91	67,120	84,153	103,202	124,231
28	8	12,18	19,32	29,45	39,74	52,91	67,120	84,153	103,202	124,231
29	8	12,18	19,32	29,45	39,74	52,91	67,120	84,153	103,202	124,231
30	8^h	$12,18^i$	$19,32^c$	$29,45^c$	$39,74^c$	$52,91^c$	$67,120^c$	$84,153^c$	103,202	124,231

Table B.2: Best known lower and upper bounds on $ca(k, g : 1)$ for $g \leq 11$.

$k\backslash g$	12	13	14	15	16	17	18	19	20
3	144	169	196[a]	225	256	289	324	361	400
4	144	169	196,225	225	256	289	324	361	400
5	144	169	196,225	225	256	289	324[c]	361	400[e]
6	144	169	196,225[j]	225[a]	256	289	324,360	361	400,525[e]
7	144[a]	169	196,255	225,255	256	289	324,360	361	400,529
8	144,169	169	196,255	225,255	256	289	324,360	361	400,529
9	144,169	169	196,255	225,255	256	289	324,360	361	400,529
10	144,169	169	196,255	225,255	256	289	324,360	361	400,529
11	144,169	169	196,255	225,255	256	289	324,360	361	400,529
12	144,169[j]	169	196,255	225,255	256	289	324,360	361	400,529
13	144,169	169	196,255	225,255	256	289	324,360	361	400,529
14	147[c],169	169[a]	196,255	225,255	256	289	324,360	361	400,529
15	147,**253**[g]	172[c],255	196,255	225,255	256	289	324,360	361	400,529
16	147,255	172,255	199[c],255	225,255	256	289	324,360	361	400,529
17	147,255	172,255	199,255	228[c],255	256[a]	289	324,360	361	400,529
18	147,255[j]	172,255	199,255[j]	228,255[c]	259[c],288	289[a]	324,360	361	400,529
19	147,276	172,325	199,360	228,288[j]	259,288[c]	292[c],360	324,360	361	400,529
20	147,276	172,325	199,360	228,360	259,360	292,360	327[c],360	361[a]	400,529
21	147,276[e]	172,325	199,360[j]	228,360[j]	259,360[j]	292,360[j]	327,360	364,529	400,529
22	147,288	172,325	199,437	228,450	259,496	292,529	327,529	364,529	403[c],529
23	147,288	172,325	199,437	228,450	259,496	292,529	327,529	364,529	403,529
24	147,288	172,325	199,437	228,450	259,496	292,52[j]9	327,529[j]	364,529[j]	403,529[j]
25	147,288	172,325	199,437	228,450	259,496	292,561	327,600[j]	364,600[j]	403,600[j]
26	147,288	172,325	199,437	228,450	259,496	292,561	327,625[j]	364,625[j]	403,625[j]
27	147,288	172,325	199,437	228,450	259,496	292,561	327,666	364,703	403,729
28	147,288	172,325	199,437	228,450	259,496	292,561	327,666	364,703	403,729[j]
29	147,288	172,325	199,437	228,450	259,496	292,561	327,666	364,703	403,841
30	147,288[e]	172,325[e]	199,437[e]	228,450[e]	259,496[e]	292,561[e]	327,666[e]	364,703[e]	403,841[j]

Table B.3: Best known upper and lower bounds on $ca(k, g : 1)$ for $12 \geq g \leq 20$.

k	4	5	6	7
$pa(k,3:1)$	9	6	4	3
$pa(k,3:2)$	6	4	3	3
$pa(k,3:3)$	3	3	3	3

k	4	5	6	7	8	9	10	11
$pa(k,4:1)$	16	16	9	8	5	5	5	4
$pa(k,4:2)$	16	12^{ν},14	8	6	5	5	4	4
$pa(k,4:3)$	16	9^{ν},13	$7^{\neq\nu}$	5	4	4	4	4
$pa(k,4:4)$	16	4	4	4	4	4	4	4

k	4	5	6	7	8	9	10	11	12	13	14	15	16
$pa(k,5:1)$	25	25	25	15	10	10	7	6	6	6	6	6	5
$pa(k,5:2)$	25	25	20^{ν},23	13^{ν}	10	8	7	6	6	6	6	5	5
$pa(k,5:3)$	25	25	16^{ν},22	12^{ν}	9^{ν},10	7	6	6	6	5	5	5	5
$pa(k,5:4)$	25	25	14^{ν},$19^{\neq}$	11	6	6	5	5	5	5	5	5	5
$pa(k,5:5)$	25	25	5	5	5	5	5	5	5	5	5	5	5

k	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$pa(k,6:1)$	34	31^{ν},34	31^{ν},34	20^{ν},34	17^{ν},19	14^{ν}	12	12	9	8	8	7	7	7	7	7	7	7	6
$pa(k,6:2)$	34	31^{ν},34	31^{ν},34	20^{ν},34	17^{ν},19	14^{ν}	12	10	8	8	7	7	7	7	7	7	7	6	6
$pa(k,6:3)$	34	31^{ν},34	28^{ν},34	20^{ν},33	16^{ν},19	13^{ν}	12	9	8	7	7	7	7	7	7	6	6	6	6
$pa(k,6:4)$	34	31^{ν},34	28^{ν},34	19^{ν},32	15^{ν},18	12^{ν},13	9,12	8	7	7	7	7	6	6	6	6	6	6	6
$pa(k,6:5)$	34	31^{ν},34	26^{ν},34	17^{ν},$26^{\neq}$	$13^{\neq\nu}$	$8^{\neq}$	7	7	6	6	6	6	6	6	6	6	6	6	6
$pa(k,6:6)$	33^{ν},34	31^{ν},34	23^{ν},34	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

k	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
$pa(k,7:1)$	49	49	28	21	15^{ν}	14	14	10	10	9	9	9	8	8	8	8	8	8	8	8	8	8	7
$pa(k,7:2)$	49	28,47	21,26	17^{ν},19	15^{ν}	14	12	10	9	9	9	8	8	8	8	8	8	8	8	8	8	7	7
$pa(k,7:3)$	49	28,46	21,25	16^{ν},18	15^{ν}	13^{ν},14	11	10	9	9	8	8	8	8	8	8	8	8	8	7	7	7	7
$pa(k,7:4)$	49	28,45	19^{ν},24	16^{ν},17	14^{ν},15	13^{ν}	10	9	9	8	8	8	8	8	8	8	7	7	7	7	7	7	7
$pa(k,7:5)$	49	21^{ν},44	17^{ν},23	15^{ν},16	12^{ν},14	9	9	8	8	8	8	8	7	7	7	7	7	7	7	7	7	7	7
$pa(k,7:6)$	49	19^{ν},$34^{\neq}$	15^{ν},$18^{\neq}$	$11^{\neq\nu}$	8	8	8	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
$pa(k,7:7)$	49	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Table B.4: Best known upper and lower bounds for packing arrays with $3 \leq g \leq 7$ and $4 \leq k \leq 29$.

Bibliography

[1] D. Ashlock. Finding designs with genetic algorithms. In W.D. Wallis, editor, *Mathematics and its Applications: Computational and Constructive Design Theory*, volume 368, chapter 4, pages 49–65. Kluwer Academic Publishers, Dordrecht, 1996.

[2] M. Chateauneuf and D.L. Kreher. On the state of strength three covering arrays. submitted to J. Comb. Des.

[3] 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. Soft. Eng.*, 23:437–444, 1997.

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

[5] C.J. Colbourn and J.H. Dinitz, editors. *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.

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

[7] P. Danziger and B. Stevens. Class-uniformly resolvable designs. *J. Comb. Des.*, 9:79–99, 2001.

[8] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley and Sons, 1998.

[9] L. Gargano, J. Körner, and U. Vaccaro. Qualitative independence and Sperner problems for directed graphs. *J. Comb. Theory, Ser. A*, 61:173–192, 1992.

[10] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1998.

[11] R.L. Haupt and S.E. Haupt. *Practical Genetic Algorithms.* John Wiley and Sons, 1998.

[12] G.O.H. Katona. Two applications of Sperner type theorems. *Per. Math. Hung.*, 3:19–26, 1973.

[13] D.J. Kleitman and J. Spencer. Families of k-independent sets. *Discr. Math.*, 6:255–262, 1973.

[14] J. Körner and M. Lucertini. Compressing inconsistent data. *IEEE Trans. Inform. Theory*, 40(3):706–715, May 1994.

[15] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search.* CRC Press, 1999.

[16] E. Lamken, R. Rees, and S. Vanstone. Class-uniformly resolvable pairwise balanced designs with block sizes two and three. *Discr. Math.*, pages 197–209, 1991.

[17] L. Moura. Personal communication.

[18] L. Moura, J. Stardom, B. Stevens, and A. Williams. Covering arrays with heterogeneous factor levels. in preparation.

[19] K. Nurmela. Upper bounds for covering arrays by tabu search. unpublished.

[20] R.H.J.M. Otten and L.P.P.P. Van Ginneken. *The Annealing Algorithm.* Kluwer Academic Publishers, 1989.

[21] S. Poljak and V. Rödl. Orthogonal partitions and coverings of graphs. *Czech. Math. Journal*, 30, 1980.

[22] S. Poljak and Z. Tuza. On the maximum number of qualitatively independent partitions. *J. Comb. Theory*, 51:111–116, 1989.

[23] N.J.A. Sloane. Covering arrays and intersecting codes. *J. Comb. Des.*, 1:51–63, 1993.

[24] B. Stevens. *Transversal Coverings and Packings.* PhD thesis, University of Toronto, 1998.

[25] B. Stevens, K. Meagher, and J. Stardom. Structured network arrays. in preparation.

[26] B. Stevens and E. Mendelsohn. Packing arrays and packing designs. submitted to Des. Codes and Crypt.

[27] B. Stevens, L. Moura, and E. Mendelsohn. Lower bounds for transversal covers. *Des., Codes and Crypt.*, 15(3):279–299, 1998.

[28] W.D. Wallis. *Combinatorial Designs.* Marcel Dekker, Inc., 1988.

[29] A.W. Williams. Determination of test configurations for pairwise interaction coverage. Thirteenth International Conference on the Testing of Communications Systems, pp 57–74, 2000.