# Covering Arrays:

# Generation and Post-optimization

by

Rushang Vinod Karia

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2015 by the
Graduate Supervisory Committee:

Charles Colbourn, Chair
Andrea Richa
Violet Syrotiuk

ARIZONA STATE UNIVERSITY

December 2015

ABSTRACT

Exhaustive testing is generally infeasible except in the smallest of systems. Research has shown that testing the interactions among fewer (up to 6) components is generally sufficient while retaining the capability to detect up to 99% of defects. This leads to a substantial decrease in the number of tests. Covering arrays are combinatorial objects that guarantee that every interaction is tested at least once.

In the absence of direct constructions, forming small covering arrays is generally an expensive computational task. Algorithms to generate covering arrays have been extensively studied yet no single algorithm provides the smallest solution. More recently research has been directed towards a new technique called post-optimization. These algorithms take an existing covering array and attempt to reduce its size.

This thesis presents a new idea for post-optimization by representing covering arrays as graphs. Some properties of these graphs are established and the results are contrasted with existing post-optimization algorithms. The idea is then generalized to close variants of covering arrays with surprising results which in some cases reduce the size by 30%. Applications of the method to generation and test prioritization are studied and some interesting results are reported.

ACKNOWLEDGMENTS

This thesis would have not been possible without the help of several people. Firstly, I would like to thank my advisor, Charles Colbourn for his constant advice and suggestions. Thank you to my committee members, Violet Syrotiuk and Andrea Richa for their time. Thanks to Charles Colbourn and Violet Syrotiuk for going through multiple revisions of this thesis and pointing out several typing errors that I missed. I would like to thank my parents and my sister for their love. You made me feel that I never left the country. Finally, I want to thank my companion, friend and shoulder to fall back on – Shrijal. This journey would have not been half as great without you.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# INTRODUCTION

The field of software testing continues to gain significance as the cost associated with faulty software continues to rise. Extensive research has been performed to reduce the cost of testing while taking into consideration the constraints imposed by the problem. Interaction testing involves testing specific subsets of components exhaustively. From a combinatorial perspective the problem is to find a covering array with the minimum number of rows. The number of rows is then considered the size of the test suite.

However, generation of covering arrays is by no means an easy task. Covering arrays have been studied extensively [19, 2, 15, 9, 16, 8, 3, 22, 24] and different techniques have been applied for their construction. Despite these efforts, the problem and its computational hardness in general remains open. To date, most of the work has concentrated on generating covering arrays from scratch. Techniques such as simulated annealing [24] appear to do well for smaller arrays, often beating greedy methods. However as the parameters become larger, the time required by the current state-of-art simulated annealing algorithms increases exponentially. Greedy methods [2, 15, 9] in general are faster but often yield low quality solutions. Many combinatorial design techniques [8] exploit the structure of covering arrays and other designs recursively to produce infinite families of covering arrays. However, the quality of these solutions depends on the quality of the ingredients and the construction.

More recently, [20] investigated post-optimization of covering arrays. Post-optimization attempts to reduce rows in an existing covering array by exploiting

certain redundancies in the covering array while maintaining coverage at every step of the process. The published results improve over several best-known results at the time and produced competitive results for several others using low quality solutions as the initial seed. Their research establishes that post-optimization is a viable technique since one may now start with a greedy solution which can be quickly obtained and then a post-optimizer can improve the solution.

## 1.1 Thesis Overview

This thesis explores a new idea by representing post-optimization of covering arrays as a vertex coloring problem. The idea is directly applicable to variants of covering arrays called quilting arrays and performs post-optimization successfully, in some cases reducing the number of rows by 30%.

Chapter 2 provides formal definitions of the terms used in this thesis. Chapter 3 surveys literature of existing post-optimization and generation techniques.

Chapter 4 introduces the new post-optimization method. The post-optimization method is explained and extended to work on quilting arrays and results are provided. Further, a bound on the clique number of the formed graphs is found for a specific family of covering arrays. Finally, the chapter concludes by analyzing factors under which post-optimization would be successful with limited effort.

Chapter 5 discusses the use of the proposed idea in applications other than post-optimization. Specifically, the generation of covering arrays, quilting arrays and providing test prioritization are discussed.

Chapter 6 concludes this thesis by listing the contributions made more formally and raises some interesting questions that could warrant future work.

# Chapter 2

# DEFINITIONS

This chapter formally describes the terms used throughout the rest of this thesis. Section 2.1 defines covering arrays. Section 2.2 defines quilting arrays - a generalization of covering arrays. Finally, Section 2.3 describes graphs.

## 2.1 Covering Arrays

Let $A = (a_{ij})$ be an $N \times k$ array with entries from an alphabet $\Sigma$ of size $v$. Let $N$ be the number of rows and $k$ be the number of columns of the array. We assume that the alphabet $\Sigma \equiv \{0, 1, 2, ..., v-1\}$, the rows are indexed by $\{0, 1, ..., N-1\}$ and the columns by $\{0, 1, ...., k-1\}$. Let $t$ be a positive integer such that $k \geq t$. Then a *t-way interaction* is a $t$-tuple $\{(c_i, v_i) : 0 \leq i < t\}$ where $c_i \in \{0, 1, ..., k-1\}$, $v_i \in \Sigma$ and $c_i \neq c_j$. Let $\Im$ be the set of all $\binom{k}{t} \times v^t$ $t$-way interactions. Array $A$ is a *covering array*, $\mathrm{CA}_\lambda(N; t, k, v)$, of strength $t$ and order $\lambda$ when every $t$-way interaction $\tau \in \Im$ appears *at least* $\lambda$ times in the covering array. Generally, $\lambda = 1$ and the notation is $\mathrm{CA}(N; t, k, v)$. For given values of $t, k$ and $v$ the minimum number of rows required for a covering array to exist is the *covering array number*; $\mathrm{CAN}(t, k, v)$.

Tables 1 and 2 are examples of a 2-covering array. The former is an optimal covering array since its size is equal to $\mathrm{CAN}(2, 4, 2) = 5$. The $\star$ symbols are referred to as *flexible positions* or *don't cares*. Such positions can take on any value without breaking coverage. The presence of flexible positions does not imply sub-optimality of a covering array.

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |

Table 1: CA(5;2,4,2)

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| $\star$ | $\star$ | 0 | 0 |
| $\star$ | $\star$ | 1 | 1 |

Table 2: CA(6;2,4,2)

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |

Table 3: PCA(4;2,4,2)

Table 3 represents a *partial* CA$(5; 2, 4, 2)$ since the 2-way interactions $\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\} \times \{(0,0)\}$[1] do not appear in the partial covering array. In fact, Table 3 is formed by deleting the last row from Table 1.

### 2.1.1 Known Bounds

Determining CAN$(t, k, v)$ appears to be a hard problem and in spite of extensive research only few bounds are known. Obviously CAN$(t, k, v) \geq v^t$. Bounds for trivial cases are easy, CAN$(1, k, v) = v$ and CAN$(t, k, 1) = 1$. When CAN$(t, k, v) = v^t$, every $t$-way interaction appears *exactly* once and the array is also known as an orthogonal array (OA).

Only for $t = v = 2$ is the covering array number determined [12, 13].

$$CAN(2, k, 2) = min \text{ n} : \binom{n-1}{\lfloor n/2 \rfloor - 1} \geq k \tag{2.1}$$

For arbitrary $v, t$ it is known that CAN grows logarithmically as a function of $k$ [10].

$$CAN(t, k, v) = \Theta(\log k) \tag{2.2}$$

[1] $(a,b) \times (c,d) = 2\text{–}way\ interaction\{(a,c),(b,d)\}$

Due to the difficulty in determining the covering array number, most construction methods focus on producing good upper bounds but provide no guarantees on the optimality of the solution.

## 2.2 Quilting Arrays

A relaxation of the coverage property of a covering array so that certain $t$-way interactions need not appear in the array is specified in [7]. These arrays are called quilting arrays. They define the *species* $\mathbb{S}$ of a $t$-way interaction to be the multiset $\{v_i : 0 \leq i \leq t-1\}$ and $v_i \in \Sigma$. A $family$ of a species is its orbit under the action of the symmetric group on $v$ symbols. Hence, a family is a partition of $t$ into at most $v$ parts.

Define $\mathbb{Q}_w^t$ to be a membership function:

$$\sum_{0 \leq i \leq v-1} \sigma_i = t \tag{2.3}$$

$$\sum_{\substack{0 \leq i,j \leq v-1 \\ i \neq j}} \sigma_i \sigma_j \geq w \tag{2.4}$$

where

$$\sigma_i = m(i) - \text{the multiplicity of } i \text{ in the family } \{v_i : 0 \leq i \leq t-1\}$$

$$w \geq 0.$$

Let $\mathscr{F}$ be the families satisfying $\mathbb{Q}_w^t$ and let $\mathcal{S}$ be the set of all species composed in $\mathscr{F}$. Lets $\Im$ be the set of all $\binom{k}{t} \times \mathcal{S}$ $t$-way interactions. Let $A = (a_{ij})$ be an $N \times k$ array. $A$ is a quilting array, $\mathbb{Q}_w^t$-QA$(N; t, k, v)$, of strength $t$ and weight $w$ if every interaction $\tau \in \Im$ appears *at least* once in $A$.

A further generalization is provided in [6]. Let $\mathbb{Q}^t_\Psi$ be a membership function where $\Psi$ is a set of weights $\psi \in \mathbb{N}$. Let $\mathscr{F}'$ be the set of all families satisfying *exactly* one $\mathbb{Q}^t_\psi$ with $\psi \in \Psi$ so that Equation 2.4 is met with strict equality. $A$ is a quilting array, $\mathbb{Q}^t_\Psi$-QA$(N; t, k, v)$, of strength $t$ and weights $\Psi$ if every interaction $\tau' \in \Im'$ appears at least once in $A$.

| | | | | | |
|---|---|---|---|---|---|
| 2 | 1 | 0 | 2 | 0 | 1 |
| 1 | 0 | 2 | 1 | 2 | 0 |
| 1 | 2 | 1 | 2 | 0 | 0 |
| 0 | 2 | 1 | 0 | 1 | 2 |
| 0 | 1 | 0 | 1 | 2 | 2 |
| 2 | 0 | 1 | 1 | 0 | 2 |
| 2 | 2 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 2 | 2 | 1 |
| 1 | 2 | 0 | 0 | 2 | 1 |
| 2 | 0 | 2 | 0 | 1 | 1 |
| 0 | 1 | 2 | 2 | 1 | 0 |
| 1 | 0 | 0 | 2 | 1 | 2 |
| 2 | 1 | 1 | 0 | 2 | 0 |
| 0 | 2 | 2 | 1 | 0 | 1 |
| 1 | 1 | 2 | 0 | 0 | 2 |

Table 4: $\mathbb{Q}^3_3$-QA$(15; 3, 6, 3)$

Table 4 gives an example of a quilting array. The membership function $\mathbb{Q}^3_3$ implies that only the family $\{0, 1, 2\}$ needs to be covered. Hence, the species that must be covered at least once in the quilting array are $012, 021, 102, 120, 201$ and $210$ while the other species may or may not appear.

**Definition 1.** A quilting array $\mathbb{Q}^t_0$-QA$(N; t, k, v)$ is a CA$(N; t, k, v)$.

### 2.2.1 Application of Quilting Arrays

Quilting arrays are used in recursive constructions for covering arrays. A construction using quilting arrays and hash families as the ingredients [7] yields less duplication

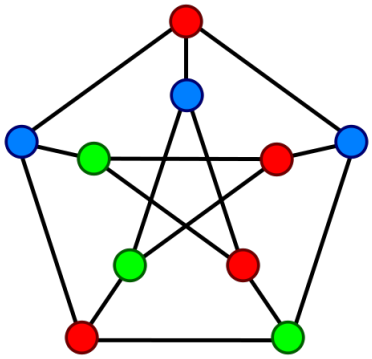


Figure 1: A Vertex Coloring of the Petersen Graph

than other constructions, ultimately leading to a solution with a fewer number of rows than the latter. It is important that the quilting array used be small since the number of rows of the resultant covering array depends on it.

## 2.3 Graphs

Let $\mathcal{G} = (V, E)$ be a simple graph where $V$ is the vertex set and $E$ is the edge set of the graph $E \subseteq [V]^2$. Let the *degree* of a vertex $v$, $d(v)$ be the total number of edges incident to it. Two vertices $u$ and $v$ are *adjacent* if edge $uv \in E$. Let the neighborhood of $v$, $N(v)$ be the set of all vertices adjacent to $v$. $\mathcal{G}$ is a *complete graph* $K^{|V|}$ if $\forall v \in V : N(v) = V \setminus v$. $\mathcal{G}$ contains a *clique* of size $r$ if $K^r \subseteq \mathcal{G}$. The clique number $\omega(\mathcal{G})$ is the largest number $\omega$ such that $K^\omega \subseteq \mathcal{G}$.

A proper vertex coloring of $\mathcal{G}$ is an assignment of colors $c : V \to C$ such that no two adjacent vertices are assigned the same color. A graph is $k$-colorable if there exists a proper vertex coloring using $k$ colors. The chromatic number of a graph $\chi(\mathcal{G})$ is the minimum number of colors required to color $\mathcal{G}$ properly. A graph is $k$-colorable if and only if $\chi(\mathcal{G}) \leq k$. Figure 1 shows a 3-coloring of the Petersen graph. The chromatic

number of the Petersen graph is 3 and hence it cannot be colored with fewer than 3 colors.

# Chapter 3

# RELATED WORK

This chapter surveys existing literature on the construction and post-optimization of covering arrays. Section 3.1 details methods for construction of covering arrays and quilting arrays. Section 3.2 reviews existing methods for post-optimization of covering arrays. Finally, Section 3.3 lists some well known graph coloring algorithms.

## 3.1 Construction Techniques for Covering Arrays

Due to the importance of covering arrays in fields such as software testing and cryptography, to name a few, a large body of work exists for solving the covering array number problem. Brute force, greedy [2, 15, 9], heuristic techniques such as simulated annealing [24] and tabu search [22], evolution based techniques such as genetic algorithms have been used for the construction of covering arrays. Due to their close relation to orthogonal arrays, techniques from number theory and design theory which primarily compose recursive and direct constructions [8] exist to construct covering arrays.

Brute force methods are feasible only for the smallest of instances since the work done is proportional to $O(k^{vN})$. Greedy methods typically focus on obtaining good solutions quickly. For example, the density algorithm [2], greedily picks the interactions by assigning weights to the factors. This is useful even if the solution is not optimal since the more important tests are identified and can be run first. Another greedy method, IPOG-F [9] differs from other methods in that it adds a column to a seed

covering array while introducing few new rows. Generally, greedy methods do not change a decision once made and hence often introduce duplication of coverage.

Heuristic techniques such as simulated annealing [24] and tabu search [22] successfully limit duplication but the time to convergence is often impractical except for smaller examples. *Roux–type* constructions [23, 8] rely on 'copy-pasting' covering arrays to obtain another covering array with a greater number of columns and perhaps rows. Direct constructions such as the starter vector method [16] do not rely on any computation to construct the covering array but instead rely on mathematical properties of the ingredients. Techniques that use a combination of combinatorial constructions and heuristic methods [3] to construct a covering array have also been researched. A more complete survey on all methods used to construct covering arrays can be found in [14]. A survey on combinatorial constructions can be found in [4]. A survey of commercial tools and the techniques they use to construct covering arrays can be found in [11]. However, despite the diversity of the techniques employed, no single method can always obtain the best results.

### 3.1.1 Construction Techniques for Quilting Arrays

Quilting arrays are a relatively new topic and as such there is not much literature surrounding them. Some recursive constructions appear in [7] but most are primarily formed by post-optimization of covering arrays. Since quilting arrays contain a subset of the interactions of a covering array with the same parameters, it is relatively easy to modify any existing covering array construction method to generate quilting arrays.

## 3.2 Post-optimization of Covering Arrays

Due to the difficulty in finding good recursive constructions and efficient heuristic algorithms to construct covering arrays, [20] proposed that one could ideally attempt to reduce the rows of an already constructed covering array. With respect to this problem, a good post-optimization method should

1. Take time inversely proportional to the redundancy for producing improvements.
2. Not get stuck in local optima.
3. Be applicable to related problems.
4. Preserve the characteristics of the input at every iteration.

Redundancy in this context means the characteristic that the post-optimizer uses to try to improve the array. An obvious choice is the total number of flexible symbols in the array but other measures could be used. A post-optimization method should be flexible to be extended to work with similar efficiency on related problems. For example the existing post-optimization method cannot work on a partial covering array and hence cannot be used as a part of a construction framework but it is widely applicable to other related problems such as hash families. Perhaps the most important criterion is that the method must preserve the input so that if the method were abruptly stopped the array would retain its input characteristics.

The existing post-optimization method (referred here as RandomPostOp) is now discussed in detail. RandomPostOp improves an existing covering array by exploiting flexible symbols in the covering array. A flexible set $F$ is a set consisting of entries $(r, c)$ where $r$ is the row index and $c$ is the column index. An element is added to $F$ if and only if $a_{rc} = \star$. As a convention the flexible symbols are marked in a bottom-up

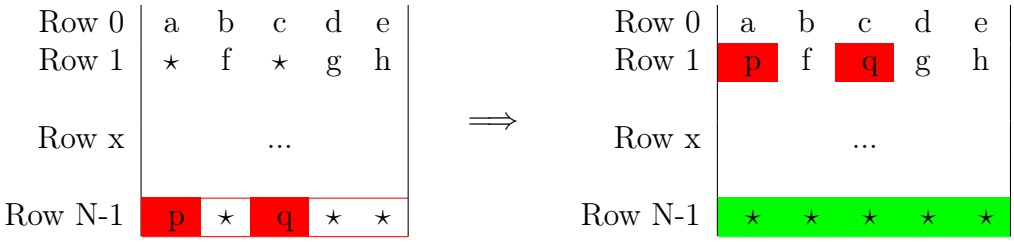



Table 5: An Illustration of RandomPostOpt

manner. This may not create the largest flexible set, but the problem of existence of a flexible set of size $l$ is known to be NP-complete [20].

For example, a flexible set for the covering array in Table 2 is $F = \{(4,0),(4,1),(5,0),(5,1)\}$. If all the entries of a row are present in the flexible set, then that row is not needed and can be deleted from the covering array while preserving coverage. Assigning any valid symbol to an entry in the flexible set could yield a new flexible set whilst also changing the covering array. This is the basic idea of RandomPostOpt. The method tries to change the flexible set so that all entries of a row appear in the set. The row is then deleted and the process is repeated. This places particular importance of selecting a candidate row to delete. RandomPostOp selects a row with the largest number of flexible symbols as the candidate row. For each of the valid entries of that row (entries not in the flexible set for that row) it tries to determine if there are other entries that are flexible. If so, then it assigns the symbol at the entry of the candidate row to the flexible entry or it may assign a random symbol (the latter empirically performs better). If there are several such entries then it chooses one at random. The consequence is that interactions covered by the candidate row might now be covered elsewhere, increasing the number of flexible symbols in the candidate row. The process is repeated until the candidate row is improved or until a specific number of iterations.

Table 5 illustrates an iteration of RandomPostOpt. The last row has 3 flexible symbols and is chosen as a candidate row to remove. The flexible set for the array is $F = \{(1,0),(1,2),(N-1,1),(N-1,3),...\}$. The entries of Row $N-1$ that are not in $F$ are $\{(N-1,0),(N-1,2)\}$ which cover a unique interaction. The method now replaces the symbols at those entries at other valid entries in $F$. This causes the entire candidate row to be in $F$ thus implying that it can be deleted. When RandomPostOp runs into local optima a new strategy is adopted. The flexible set is populated with valid symbols and the rows of the covering array are permuted. This leads to the formation of a new flexible set.

**Algorithm 1:** RandomPostOpt

```
input  : A covering array; CA(N; t, k, v)
output : A covering array; CA(N'; t, k, v) where N' ≤ N
begin
    while time limit not expired do
        if local optima then
            permute the covering array
        end
        else
            F ⇐ compute flexible set
            removeRows(F)
            r_c ⇐ select candidate row from F
            E_{r_c} ⇐ set of all entries of r_c
            F'' ⇐ E_{r_c} \ F(r_c)
            Replace symbols in F where entries in F'' have the same column
            randomly
        end
    end
end
```

RandomPostOpt, while performing well, suffers from two major drawbacks. The first is that it only uses flexible symbols to improve the array and does not alter symbols that are necessary. Many covering arrays constructed using simulated annealing or tabu

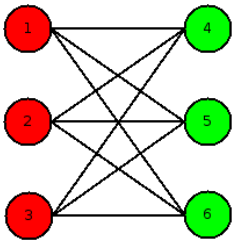
search work to limit the size of the flexible set making it very hard for RandomPostOpt to improve such arrays. Another drawback is that RandomPostOpt cannot be used as a part of a generative method since usually the flexible set is very small unless the covering array reaches a considerable percentage of coverage. Typically when a covering array gets too large it is sometimes feasible to reduce some rows possibly creating a partial array. It is desirable to remove many rows while reducing the coverage by a controllable percentage. Due to the element of randomness employed by RandomPostOpt this is not feasible without significant computation.

Algorithm 1 represents pseudo-code for a very basic version of RandomPostOpt. The method is easily extended to quilting arrays and $k$-restriction problems by changing the way the flexible set is computed.

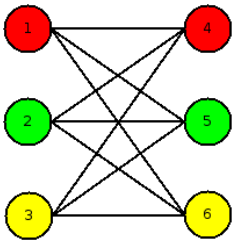
### 3.3 Graph Coloring Methods

Graph coloring is a hard computational problem. Except when $k = 1$ and $k = 2$ it is NP-complete to decide if a graph admits a $k$-coloring. Computing the chromatic number is NP-hard and unless P = NP approximating the chromatic number to within $n^{1-\epsilon}$ for $\epsilon > 0$ is not efficiently solvable [26]. As such, existing research focuses on greedy, heuristic and distributed methods to solve the problem. See [17] for a comprehensive survey of graph coloring algorithms.

Greedy algorithms generally color vertices in a certain order. For example *largest degree first coloring* [25] colors the vertices based on the non-increasing degree of vertices. Another popular greedy coloring algorithm, DSATUR [1], chooses the next vertex which has the maximum number of differently colored neighbors. The initial ordering of vertices is the same as that of the largest degree first coloring.




(a) An optimal coloring of $K_{3,3}$




(b) A sub-optimal coloring of $K_{3,3}$

Figure 2: Different Colorings of the Complete Bipartite Graph $K_{3,3}$

Greedy algorithms are generally fast but perform poorly since they do not assign a different color to an already colored vertex. Figure 2 illustrates the importance of the order of vertices on the coloring output by largest degree first coloring on the complete bipartite graph $K_{3,3}$ whose $\chi(K_{3,3}) = 2$. Assuming ties are broken randomly the method may produce different orders of vertices. Figure 2a is colored using the sequence *123456* and this yields an optimal coloring while Figure 2b is colored using the sequence *142536* and this requires more colors than the optimal. Other greedy techniques such as DSATUR try to dynamically order the vertices during coloring but in general have similar shortcomings. Local search techniques typically work using *iterative improvement* in that they choose a move minimizing a particular cost function. Typically larger cost moves are allowed when the method converges to a local optima. They generally produce better colorings but require a considerable amount of computation in terms of both time and space.

Chapter 4

# RECOLORING FOR COVERING AND QUILTING ARRAYS

This chapter introduces the Recoloring technique for post-optimization. Due to the modular nature of the idea different variations are examined and contrasted with some results improving the best known cases by 30%. Furthermore, given sufficient computation time Recoloring can produce the optimal solution. Section 4.4 studies the structure of the graphs formed and bounds the clique number for a certain family of graphs. Since it is difficult to obtain a deterministic bound on the improvement possible by post-optimization on any given array, we try to highlight some key factors for post-optimization to be successful.

## 4.1 Recoloring for Covering Arrays

To the best of our knowledge, the only work that represents covering arrays as graphs is [19, 18]. We follow a different approach here. Let $\tau$ be a $t$-way interaction and $\Re = \{0, 1, ..., N-1\}$ be a set of row indices. Row $r$ covers the $t$-way interaction $\tau \equiv \{(c_i, v_i) : 0 \leq i < t\}$ if $a_{rc_i} = v_i$ i.e. for every column $c_i$ of the interaction, the entry in the row indexed by $r$ is $v_i$. Let $\mathcal{R}(\tau)$ be the set of all rows covering $\tau$. Note that $\mathcal{R} \subseteq \Re$. A $t$-way interaction $\tau$ is *private* to $\mathcal{R}$ if for any $\mathcal{R}' = \Re \setminus \mathcal{R}$, $\mathcal{R}'(\tau) = \emptyset$.

Let $\wp(\mathcal{R})$ be a set of interactions that are private to $\mathcal{R}$. Form a graph $G$ with a vertex for each interaction $\tau \in \wp(\mathcal{R})$. Any two vertices $a \equiv \{(c_i, v_i) : 0 \leq i < t\}$ and $b \equiv \{(c'_i, v'_i) : 0 \leq i < t\}$ are connected by an edge if $c_i = c'_j$ but $v_i \neq v'_j$ for any $i, j$ satisfying $0 \leq i, j < t$. Thus an edge is placed between two interactions if and

only if they have at least one column in common and have different symbols in that column. It follows that such interactions cannot appear in the same row. Thus, the chromatic number $\chi(G)$ of $G$ is precisely the number of rows required to form $G$. If the computed chromatic number $\alpha(G) < |\mathcal{R}|$ then the coloring can be used to remove $|\mathcal{R}| - \alpha(\mathcal{G})$ rows of the covering array while covering all the interactions in $\wp(\mathcal{R})$.

**Algorithm 2:** RecolorPostOpt

```
   input  : A covering array CA(N; t, k, v)
   output : A covering array CA(N'; t, k, v) where N' ≤ N
 1 begin
 2     while time limit not expired do
 3         R ⇐ getRowSet()
 4         P ⇐ computePrivateInteractions(R)
 5         Form graph G
 6         α(G) ⇐ getProperVertexColoring(G)
 7         if α(G).size() ≤ R.size() then
 8             for every colored group g ∈ α(G) do
 9                 replaceRow ⇐ R[g]
10                 for every interaction τ ∈ g do
11                     CA[replaceRow][τ_{c_i}] ⇐ τ_{v_i}
12                 end
13             end
14             for i = α(G).size() to R.size() do
15                 deleteRow(R[i])
16             end
17         end
18     end
19 end
```

Algorithm 2 illustrates post-optimization using Recoloring. Lines 3 and 6 are generic modules which affect the performance of the method. Recoloring starts by choosing a set of rows for post-optimization. The private interactions of that set of rows are computed and a graph is formed. A coloring of the graph is then computed and each color class forms a new row. Thus, all the interactions are redistributed

among the rows. If the total number of color classes is less than the number of rows selected then some rows are not assigned any interaction and can be safely deleted from the covering array.

### 4.1.1 Finding the Private Interactions

Forming the graph involves $O(V^2t)$ comparisons between all vertices (interactions). As such, the time taken to form the graph can be restrictive when the graph grows beyond a certain size. One could maintain an adjacency matrix where every interaction would be linked to all interactions it could have edges with. This would reduce additional lookup but since an entry for every interaction needs to be maintained the space requirements are $\Omega(\binom{k}{t}v^t)$ which is impractical. This places particular importance on finding the private interactions quickly to keep the iteration time scalable.

We compute the private interactions using two approaches. The first approach adds all the $\binom{k}{t}$ $t$-way interactions covered by every row in $R$ to $\mathcal{P}(R)$. Note that $\mathcal{P}(R)$ is not a multiset. Thus interactions covered by two or more rows appear as one interaction in the set of private interactions. Flexible symbols are not considered to participate in any $t$-way interaction. Now for all remaining $N \setminus R$ rows, we check if any of the $\binom{k}{t}$ interactions are already in $\mathcal{P}(R)$ and remove those interactions from the set of private interactions. This approach requires $O(N\binom{k}{t})$ work for any $|R| > 0$. Pseudo-code for computing the private interactions appears in Algorithm 3.

Frequently the total work done can be reduced by making some observations. In any given column in a covering array, two rows may have the same symbol with probability $\frac{1}{v}$. We exploit this fact and use it to modify our definition of a private

**Algorithm 3:** computePrivateInteractions

```
input  : A covering array CA(N; t, k, v); A set of rows R ∈ N
output : Private interactions of the set of rows P(R)
begin
    P(R) ⇐ ∅
    forall the column tuples c ∈ (k choose t) do
        forall the rows r ∈ R do
            I ⇐ formInteraction(r, c)
            P(R) ⇐ P(R) ∪ I
        end
        forall the rows r' ∈ N \ R do
            I' ⇐ formInteraction(r', c)
            if P(R) contains I' then
                P(R) ⇐ P(R) − I'
                if P(R) = ∅ then
                    break
                end
            end
        end
    end
end
```

interaction. Let $\mathcal{N}(T)$ be the set of all rows covering the set of interactions $T$. It follows that $T$ is private to $\mathcal{N}$. Now, let $R$ be a set of rows such that $\mathcal{N} \subset R$. Recall that a set of interactions is private to $R$ if and only if $\mathcal{N}(T) \subseteq R$. Thus, $T$ is private to $R$. We tighten the definition here. A set of interactions $T$ is private to a set of rows $R$ if and only if $\mathcal{N}(T) = R$. Let $\Gamma(R)$ be the set of such interactions. This implies that only interactions covered by *all* rows in $R$ can appear as private interactions. However the new definition does not allow recoloring to work correctly. For example if two rows are selected then the private interaction set formed contains only interactions that are common to both rows. Any interaction private to a single row is not included causing these interactions to be lost. The old definition can be easily expressed in

terms of the new definition. Without loss of generality,

$$\mathcal{P}(R) = \bigcup_{\forall x \in 2^R} \Gamma(x) \tag{4.1}$$

where $2^R$ is the power set of $R$.

Thus $\mathcal{P}(a,b,c) = \Gamma(a) \cup \Gamma P(b) \cup \Gamma P(c) \cup \Gamma P(a,b) \cup ... \cup \Gamma(a,b,c)$. The advantage of finding the private interactions in this way is that significant computation is avoided. Whenever the private interactions of two or more rows need to be found we only consider the columns where all the rows share a common symbol. If all the rows have less than $t$ such columns then they cannot have any private interaction. Empirically this method outperforms the former method when $|R| \leq 4$. Computation of the private interactions using this technique is described in Algorithm 4.

The advantage of using Algorithm 4 is that for every column tuple $c$ at most one interaction is added to the private interaction set. This often helps the loop at line 10 to exit earlier. Since the power sets are being generated, this method is only practical for small values of $|R|$. Further, for $\mathcal{R} \in 2^R, |\mathcal{R}| > 1$ not all $\binom{k}{t}$ column tuples are checked. In fact, the number of common symbols between any two rows is strictly less than the total number of columns of the array unless all rows are the same which implies that except for the first row, all others rows may be deleted.

### 4.1.2 Generating the Row Set to Post-optimize

Both the choice of rows and the coloring algorithm are mutually dependent on each other. A good choice of rows can yield an easier to color graph while a good coloring can rearrange the rows so that further row choices are easier to determine. We describe heuristics for choosing the rows first.

**Algorithm 4:** powerSetPrivateInteractions

**input** : A covering array CA$(N; t, k, v)$; A set of rows $R \in N$
**output** : Private interactions of the set of rows $\mathcal{P}(R)$

```
1  begin
2      P(R) ⇐ ∅
3      forall the rowsets R ∈ 2^R \ ∅ do
4          P'(R) ⇐ ∅
5          k'[] ⇐ set of common entries of R
6          if k'.size() ≥ t then
7              forall the column tuples c ∈ (k' choose t) do
8                  I ⇐ formInteraction(R[0], c)
9                  P'(R) ⇐ P'(R) ∪ I
10                 forall the rows r' ∈ N \ R do
11                     I' ⇐ formInteraction(r', c)
12                     if P'(R) contains I' then
13                         P'(R) ⇐ P'(R) − I'
14                         if P'(R) = ∅ then
15                             break
16                         end
17                     end
18                 end
19             end
20         end
21         P(R) = P(R) ∪ P'(R)
22     end
23 end
```

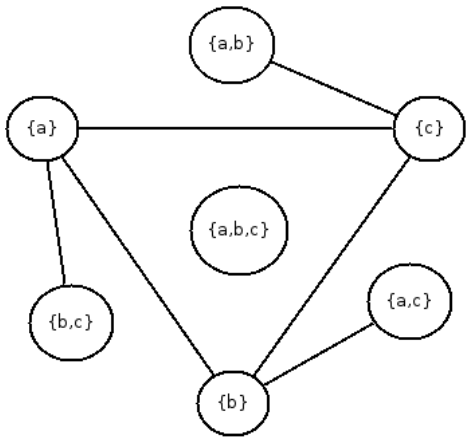



Figure 3: An Interaction Group Graph for Three Rows $R = \{a, b, c\}$

The best choice is to select rows in such a way that post-optimization is successful in the first iteration. However this is almost always not the case except when possibly the covering array is far from optimal. In such scenarios a good set of rows would be one which would either keep the size of the graph small and/or admit a coloring such that the size of the smallest color class is smaller than the number of private interactions of the smallest row. This effectively reduces the number of private interactions of that row. As such one could generate a new row set with the smallest row and hope to reduce its size even further until it is not assigned any interaction.

We now analyze conditions where the graph size is small. Unless specified, graph size in this context refers to the total number of edges of the graph. The goal is to keep the graph size as small as possible. We use the modified definition of a private interaction as defined in section 4.1.1. Let $R$ be a set of rows. A *unique* interaction is an interaction that is private to *exactly one* row in $R$. Let $\mu(r)$ be the set of all interactions unique to $r$. A *common* interaction is an interaction that is private to two or more rows. For any set of rows $R$, define an interaction group graph $G_I(V, E)$ where $V = 2^R \setminus \emptyset$. An edge exists between two vertices $u$ and $v$, $u \neq v$, if $u \not\subset v$

or $u \not\supset v$. This implies that any interaction of $\Gamma(u)$ may have an edge with any interaction in $\Gamma(v)$ if and only if the edge $uv$ exists in $G_I$. Note that when every vertex $u$ is substituted by $\Gamma(u)$ we recover the interaction graph for recoloring. Any vertex in the interaction graph does not have an edge with any other vertex if the corresponding interaction groups have no edge in the interaction group graph but the converse is not true. A simple example is two rows each having one private interaction with a common column and the same symbol in that column. These vertices have an edge in the interaction group graph but no edge in the interaction graph.

Based on the interaction group graph in Figure 3, it is desirable to find rows whose private interactions are in $\Gamma(a, b, c)$ since the vertices can have no edges with any of the other vertices of the graph. In general it is desirable to find sets of rows which have more common interactions than unique interactions. This is computationally expensive and as the number of rows increases the total number of common interactions is generally very small compared to the unique interactions.

We consider a greedy heuristic. Maintain a count of the unique interactions of each row. Choose the row with the fewest number of unique interactions and add it to the set. Form the interaction graph and color it. If recoloring is successful then the set of rows is cleared else select the next row which has the fewest number of private interactions and repeat the process. We clear the set if either an improvement is found or the number of vertices becomes large enough that the coloring method fails with a high probability. The number of vertices is empirically determined.

The greedy method suffers from a problem of symmetry. If the coloring used the same number of colors as the rows selected then the counts of the unique interactions remain more or less the same leading to the same row set being selected. We circumvent this problem by employing two different strategies; (i) We color the graph

using different vertex orderings until a sufficient unbalance is found or (ii) randomly selecting row sets instead of greedily. This often changes the configuration of the covering array. Our experiments show that this is often sufficient to escape local optima.

### 4.1.3 Choice of the Coloring Algorithm

The coloring algorithm plays an important part in determining the success of post-optimization. For any given choice of $R$ rows it is clear that $\chi(G) \leq |R|$ since we can form a graph and simply assign all private interactions of each row the same color. Therefore, a good coloring algorithm is one which either produces a coloring with fewer colors (if one exists) or uses at most $|R|$ colors and does so quickly. When the computed chromatic number $\alpha(G) = |R|$, the interactions can be redistributed in a way so as to be useful in other iterations. A good example is that if the color classes are as unbalanced as possible. All interactions $\Gamma(a), \Gamma(a,b), \Gamma(a,c)$ and $\Gamma(a,b,c)$ in figure 3 can be assigned a single color class. However $\Gamma(a,b), \Gamma(a,c)$ and $\Gamma(a,b,c)$ could be assigned a different color but doing so would balance the color classes. One would need to equip each interaction with a group tag in order for the coloring algorithm to utilize the information. This would require the power set method for computing the private interactions. We follow a simpler approach. If DSATUR fails to produce a balanced coloring we change the ordering of a few vertices and reinvoke DSATUR. This appears to perform reasonably well for smaller graphs. For larger graphs, DSATUR fails to produce a valid ($\alpha(G) \leq |R|$) coloring frequently.

| Row/Column | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 0 | 1 |
| 3 | 1 | 1 | 1 | 0 |
| 4 | $\star$ | $\star$ | 0 | 0 |
| 5 | $\star$ | $\star$ | 1 | 1 |

Table 6: CA(6;2,4,2)

#### 4.1.4 Example: Post-optimization on CA$(6; 2, 4, 2)$

We now walk through an iteration of Recoloring on a simple instance produced by the IPOG-F method (See Table 6). By equation 2.1 we know $\text{CAN}(2, 4, 2) = 5$.

Let $R = \{3, 4, 5\}$.

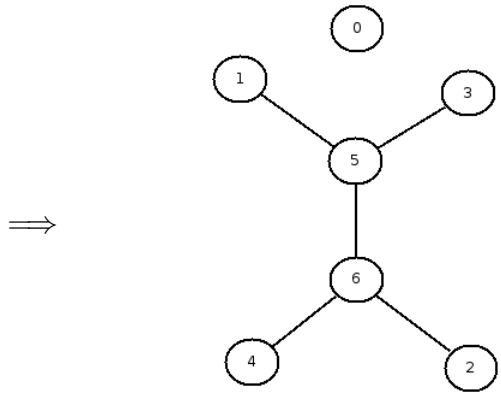
The private interactions of $R$ are computed and the graph is colored with 2 colors. Since the total number of color classes is less than the total number of rows selected we can delete 1 row. Figure 4d shows the coloring of the graph. The first color class is placed in the third row while the second color class forms the last row.

### 4.2 Recoloring for Quilting Arrays

Recoloring for quilting arrays is the same as for covering arrays except that one additional step is needed in the computation of the private interactions. Before an interaction is added to the private interaction set it must satisfy the membership function $\mathbb{Q}^t_\Psi$ of the quilting array. Some families that do not satisfy the membership function may be implicitly formed for some given column tuple. Thus this check is necessary for correctness. The rest of the procedure remains the same.

| Interaction | Group | Label |
|---|---|---|
| $\{(0,1),(1,1)\}$ | $\Gamma(3)$ | 0 |
| $\{(0,1),(2,1)\}$ | $\Gamma(3)$ | 1 |
| $\{(0,1),(3,0)\}$ | $\Gamma(3)$ | 2 |
| $\{(1,1),(2,1)\}$ | $\Gamma(3)$ | 3 |
| $\{(1,1),(3,0)\}$ | $\Gamma(3)$ | 4 |
| $\{(2,0),(3,0)\}$ | $\Gamma(4)$ | 5 |
| $\{(2,1),(3,1)\}$ | $\Gamma(5)$ | 6 |

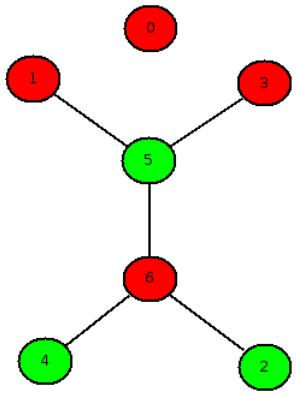
(a) Private Interactions




(b) The graph $G$ of the private interactions

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 |

(c) Post-optimized CA$(5;2,4,2)$




(d) A coloring of $G$

Figure 4: Recoloring of $R = \{3,4,5\}$ on CA$(6;2,4,2)$

### 4.3 Results of Post-optimization of Covering and Quilting Arrays Using Recoloring

We now analyze results produced by applying recoloring on covering arrays formed by IPOG-F. Recoloring is able to produce competitive results for several covering arrays and improved upon some previously best known bounds. Improvements in quilting arrays are more successful partially because the graphs of quilting arrays were comparatively easier to color. The program was run on an Intel i7-4790K processor with 8 cores each clocked at 4.4GHz and 16GB of RAM. Each of the cores was utilized by the program. The time limit for each instance varied with the size of the covering array but the maximum time allotted was 48 hours.

CA$(2, k, 3)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 5 | 11 | 13 | 11 | 11 |
| 6 | 12 | 15 | 12 | 12 |

CA$(2, k, 4)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 7 | 21 | 27 | 22 | 22 |

CA$(3, k, 2)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 5 | 10 | 11 | 10 | 10 |

CA$(3, k, 3)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 6 | 33 | 49 | 33 | 33 |
| 7 | 39 | 52 | 44 | 44 |
| 8 | 42 | 56 | 50 | 49 |
| 9 | 45 | 62 | 53 | 54 |
| 10 | 45 | 66 | 56 | 56 |

CA$(4, k, 2)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 6 | 21 | 26 | 24 | 21 |

CA$(4, k, 3)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 6 | 111 | 140 | 121 | 117 |
| 7 | 123 | 164 | 131 | 139 |
| 8 | 135 | 188 | 164 | 162 |
| 9 | 135 | 211 | 180 | 186 |

CA$(4, k, 4)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 7 | 412 | 530 | 464 | 447 |

CA$(4, k, 5)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 8 | 1212 | 1460 | 1247 | 1204 |

CA$(5, k, 2)$

| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
|---|---|---|---|---|
| 7 | 42 | 57 | 48 | 42 |

| CA$(5, k, 3)$ | | | | |
|---|---|---|---|---|
| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
| 7 | 351 | 467 | 394 | 387 |
| 8 | 405 | 557 | 475 | 471 |
| 9 | 405 | 652 | 560 | 561 |
| CA$(6, k, 3)$ | | | | |
| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
| 8 | 1134 | 1490 | 1263 | 1253 |
| CA$(6, k, 4)$ | | | | |
| $k$ | Best Known | NIST | RandomPostOpt | RecolorPostOpt |
| 8 | 7180 | 8579 | 7243 | 7221 |

Table 7: Post-optimization of Covering Arrays Using Recoloring

Table 7 lists results produced by performing post-optimization on the covering arrays produced by IPOG-F. The best known results mentioned here are those that are found in [5]. Covering arrays produced by IPOG-F can be found at [21]. Recoloring succeeded in improving the bound of CAN$(4, 8, 5) = 1212$ to CAN$(4, 8, 5) = 1204$. For the rest of the results, it either is competitive or better than the original post-optimization method.

As the size of the covering array increases in terms of the parameters, the coloring method experiences a higher failure rate directly reducing the efficiency of Recoloring. When $k > 10$ Recoloring has a very low success rate and seldom improves upon the randomized post-optimization method. Another important caveat is that while Recoloring does not tend to get stuck in local optima (given a good coloring method) the time taken per iteration of Recoloring is significantly larger due to additional time required to form the graph. As such, RandomPostOpt often runs and converges to a local optima very quickly and Recoloring catches up much later. Our experiments demonstrate that for CA$(4, 7, 4)$ the ratio of time taken by Recoloring compared to

randomized post-optimzation is as large as $5:1$. Once RandomPostOpt converges to a local optimum, Recoloring improves upon the results. In order to reduce the randomness we ran 100 seperate instances of RandomPostOpt to improve IPOG-F$(530; 4, 7, 4)$. RandomPostOpt converges to $467 \pm 5$. Recoloring on the other hand significantly improves the result to 447 before the time limit ran out. Another striking instance is IPOG-F$(8579; 6, 8, 4)$. Recoloring managed to improve the covering array to 7221 rows while RandomPostOpt converged at around 7243.

Recoloring in its current implementation does not seem to be effective for larger covering array instances. This is attributed to the coloring method which does extremely poorly even for a very small selection of rows in those cases. The heuristic methods that were tried improved the success rate but often took an excessive amount of time leading to a very limited number of iterations being executed. Nevertheless, it is surprising that Recoloring performs reasonably even with a poor coloring method.

The covering arrays that were used in the examples so far are small enough that the best known bounds known for them are quite difficult to improve even by heuristic methods. Quilting Arrays on the other hand have been produced by RandomPostOpt. Thus it appears worthwhile to compare RandomPostOpt and Recoloring on small Quilting Arrays. Recoloring improves upon several of the bounds sometimes producing an improvement as high as 30%. Even more surprising is that the bounds produced by RandomPostOpt used the best known covering arrays as the initial seeds while Recoloring used the IPOG-F examples which had a significantly larger number of rows. The quilting array improvements show the true strength of Recoloring to obtain a high quality solution even with a poor coloring method. Naturally, a better coloring method would yield even better or equal arrays. Table 8 lists some of the improvements found by Recoloring for quilting arrays where $k > t + 1$. Each entry represents the size of

the quilting array that was formed by post-optimizing the corresponding CA$(t, k, 3)$ generated by the IPOG-F method along with difference with the currently best known result as stated in [7].

| k | $\mathbb{Q}_2^3$ | $\mathbb{Q}_3^3$ | $\mathbb{Q}_3^4$ | $\mathbb{Q}_4^4$ | $\mathbb{Q}_5^4$ | $\mathbb{Q}_4^5$ | $\mathbb{Q}_6^5$ | $\mathbb{Q}_7^5$ | $\mathbb{Q}_8^5$ | $\mathbb{Q}_5^6$ | $\mathbb{Q}_8^6$ | $\mathbb{Q}_9^6$ | $\mathbb{Q}_{11}^6$ | $\mathbb{Q}_{12}^6$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 35 | 15 | 108 | 86 | 60 | - | - | - | - | - | - | - | - | - |
| | +2 | +0 | +0 | -22 | -24 | - | - | - | - | - | - | - | - | - |
| 7 | 42 | 21 | 110 | 107 | 83 | 348 | 339 | 286 | 172 | - | - | - | - | - |
| | +2 | +0 | -10 | -4 | -1 | +0 | -6 | -2 | -4 | - | - | - | - | - |
| 8 | 47 | 21 | 138 | 125 | 99 | 402 | 402 | 365 | 242 | 1149 | 1146 | 1203 | 860 | 374 |
| | +5 | -3 | +0 | -7 | -5 | +0 | +0 | +17 | -3 | +0 | +0 | +98 | -18 | -2 |
| 9 | 50 | 24 | 156 | 151 | 115 | 480 | 480 | 442 | 291 | 1428 | 1428 | 1421 | 1288 | 517 |
| | +5 | +0 | +0 | +3 | -2 | +0 | +0 | -20 | -11 | +0 | +0 | -1 | +193 | -9 |

Table 8: Post-optimization of $\mathbb{Q}_w^t$-QA$(N; t, k, 3)$ Using Recoloring

Perhaps the most important reason that Recoloring fails to improve some of the quilting arrays is the structure of the seed covering array used. A constant row of a covering array is a row where all columns have the same symbol. Naturally, a quilting array with weight $> 0$ does not need any such row and these can be deleted without forming a graph. The IPOG-F examples do not usually have constant rows while RandomPostOpt used covering arrays having at least one constant row in most of the cases. This leads to another important consideration of the structure of the initial covering array used in forming a quilting array.

### 4.4 Graph Analysis

In this section we provide some statistics about the graphs formed. We list the average number of vertices per selection of rows for different values of $t$ and $v$. We

analyze the success rate of the coloring algorithm in terms of the size of the graph and also prove that for $t = 2$ the clique number of the graph is at most $v^t$.

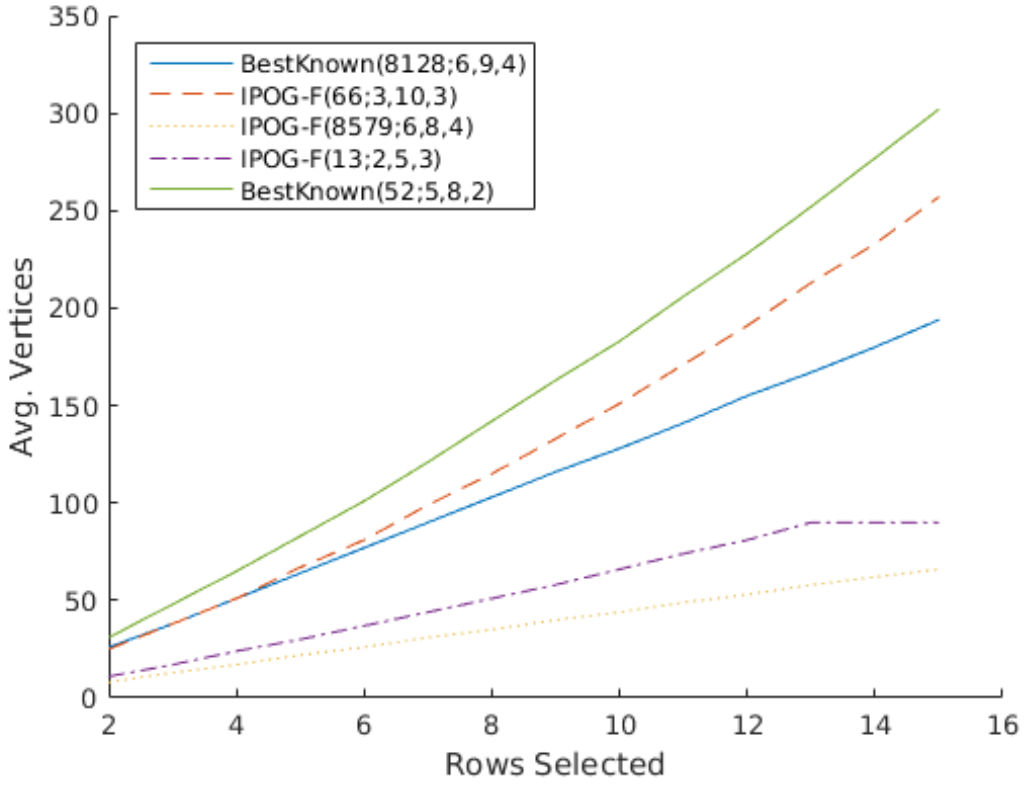



Figure 5: Average Number of Vertices in the Covering Array Graph

Figure 5 plots the average number of vertices of a graph formed by the selection of a specific number of rows. Even though there are much larger instances, $BestKnown(52; 5, 8, 2)$ has the largest graph size hinting that as an instance gets closer to best known the graph becomes larger and possibly more complex creating problems for the coloring method.

Figure 6 contrasts the performance of the coloring method based on the number of rows selected for different covering arrays. Success in this context means that the

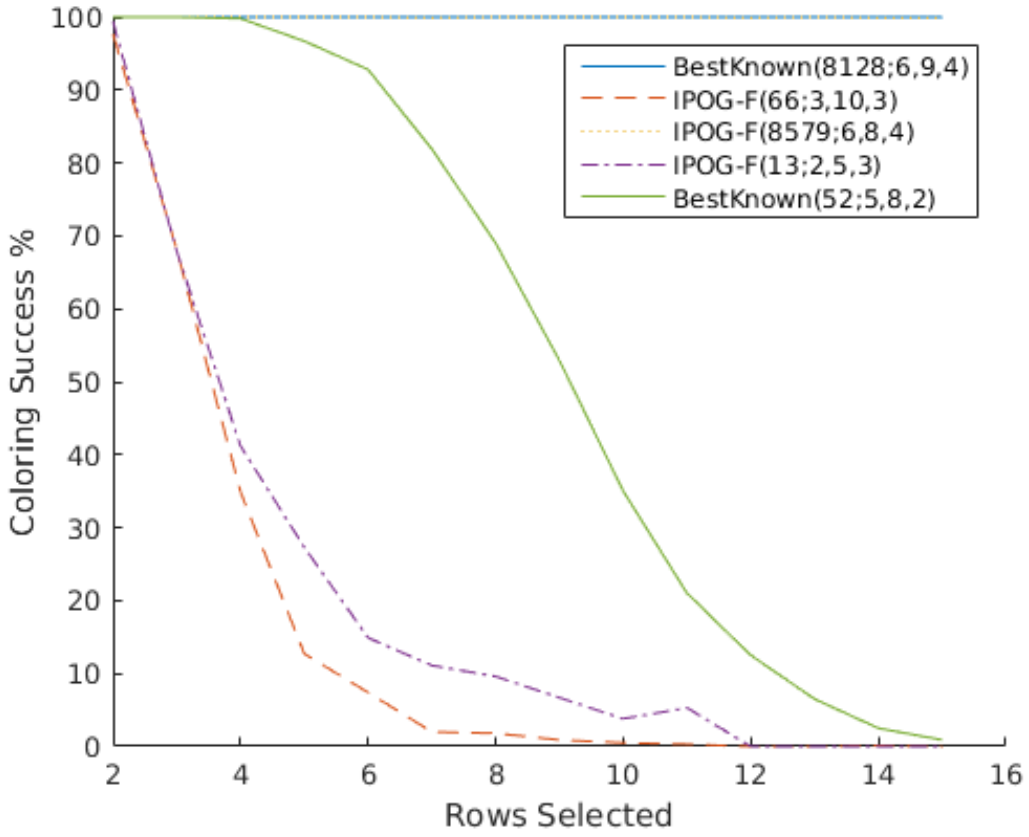



Figure 6: Performance of DSATUR Coloring

coloring method used no more colors than the total number of rows selected. An important observation is that a larger graph is not necessarily difficult to color. The average number of vertices for $BestKnown(52; 5, 8, 2)$ is greater than that for IPOG-F$(13; 2, 5, 3)$ or IPOG-F$(66; 3, 10, 3)$ for the same number of rows yet the coloring is worse for the latter instances. This is because the graph size for the latter cover a greater percentage of the total interactions leading to a much higher average edge density. For example, for four rows the graph sizes for $BestKnown(52; 5, 8, 2)$ and IPOG-F$(13; 2, 5, 3)$ are 65 and 24 respectively. However, they represent 3.6% and 30%

of the total number of interactions possible. Thus, the complexity of the latter is in practice much more than that of the former.

### 4.4.1 Bounding the Clique Number of the Graph

Knowing a bound on the clique number of the covering array graph serves two purposes. First, it is sufficient to say that post-optimization using rows as the vertices (two rows are connected by an edge if they have at least one pair of interactions having an edge in the interaction graph) is infeasible since the clique number in this case is arbitrarily large and thus is the chromatic number. Second, it implies that the covering array admits some structure which could be exploited.

We now bound the clique number of all families of covering arrays with strength 2. Let $\mathcal{I}$ be the set of all $\binom{k}{t} \times v^t$ $t$-way interactions of the covering array. Form a graph $\mathcal{G}$ using $I$. Call this graph the covering array graph $CAG$. Without loss of generality, we assume that the $t$-column tuples $\{c_0, c_1, ..., c_{t-1}\}$ are lexicographically ordered such that $c_i < c_j$ whenever $i < j$. The symbols are assumed to be represented as strings in the base $v$. Tables 9 and 10 illustrate the orderings of the columns and symbols respectively. Recall that an edge is placed between two interactions if and only if they have a column in common but distinct symbols at those columns. Let a column group refer to a specific $t$-column tuple. Each column forms $v^t$ interactions and there are $\binom{k}{t}$ such column groups. We assume that the parameters $t, k$ and $v$ are non-trivial.

**Lemma 1.** A covering array graph $CAG$ has a clique of size $v^t$ for any $t, k$ and $v$.

*Proof.* Let $C$ be a column group forming $v^t$ interactions. Each $v^t$ interaction has all $t$

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 0 | 1 | 3 |
| 0 | 1 | 4 |
| 0 | 2 | 3 |
| 0 | 2 | 4 |
| 0 | 3 | 4 |
| | ... | |
| 2 | 3 | 4 |

Table 9: Lexical Ordering for the Columns When $k = 5$ and $t = 2$

| | |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 0 | 2 |
| 1 | 0 |
| 1 | 1 |
| | ... |
| 2 | 2 |

Table 10: Lexical Ordering for the Columns When $v = 3$ and $t = 2$

columns in common but at least one symbol different. Thus, each interaction forms an edge with every interaction within the same column group $C$. □

**Lemma 2.** Any two column groups $C$ and $C'$ with $C \neq C'$, may have at most $t - 1$ columns in common.

**Theorem 1.** For $t = 2$, any sub-graph of the covering array graph for $\mathrm{CA}(N; 2, k, v)$ cannot have a clique of size greater than $v^t$.

*Proof.* We prove the result by using the interaction group graphs and bounding the size of maximal cliques. Note that a maximal clique is the largest clique that cannot be extended by the addition of any other vertex. We then prove that when the clique size is $v^t$ no other vertex can be added to it, i.e., it is maximal.

For the sake of convenience note that each of the $v^t$ symbol tuples has exactly $v$ occurrences of the same symbol in any given column. We group the symbols on any one column and there are $v$ such groups. Similarly, the column groups may be arranged in groups of size $k - 1$.

**Case 1.** A clique of size greater than $v^t$ does not exist between any two column groups having no columns in common.

A clique of size $v^t$ exists within any single column group. However, if two column groups have no columns in common then none of the vertices between the columns can have edges between them. The graph is then a disjoint graph with 2 cliques of size of $v^t$.

**Case 2.** A clique of size $v^t$ exists between any two column groups sharing exactly one column.

Select any column group $c$. Select all the interactions with the same symbol on the common column. There are $v$ such interactions which form a clique of size $v$ since they belong to the same column group $c$. Interactions in the other column groups need to have different symbols in-order to have edges with the interactions in the selected column group. There are $v^t - v$ such interactions. Select all $v^t - v$ interactions of the second column group. These form a clique of size $v^t - v$ since they belong to the same column group and also have edges with all the vertices in the select column group. Thus the size of the clique is $v^t$. A clique of size $v^t + 1$ or larger is not possible since if $v^t - v + c$ where $c > 0$ interactions are selected from the second column group then there is at least one interaction with the same symbol between both the column groups.

Similarly, selecting all $v^t$ vertices from the first column group and any one from the other cannot form a clique of size $v^t$ since the interaction selected has the same symbol with $v$ of the interactions in the first column group (thus this interaction forms a clique of size $v^t - v$).

**Case 3.** A clique of size $v^t$ exists between any $2 < l \leq v$ column groups sharing exactly one column between all column groups.

Case 2 is now generalized for $l$ column groups. The generalization is simple. Select $v$ interactions from each of the column groups. Fix a symbol for the first column group. There are $v$ such interactions. Now select a different symbol for the second column group. There are $v$ such interactions and these have an edge with the interactions in the first group. This process is repeated $v$ times. This leads to clique of size $v^t$. Now suppose $v+1$ vertices were selected from the second group. It then follows that selecting interactions from other groups cannot lead to clique of size $v^t$ since at least one of the interactions in the other groups could not have an edge with the additional vertex in the second group.

Similarly, when $v+1$ column groups are selected a clique of size greater than $v^t$ cannot be formed. Consider the case where a clique of size $v^t$ is formed with the first $v$ column groups, then any vertex from the last group cannot have $v$ edges with the remaining interactions. Any other way to select the vertices also yields a clique of size less than $v^t$.

**Case 4.** A maximal clique of size no more than $(v-1)^2+(v-1)+1$ exists between column groups with more than one column in between them.

This case deals with column groups where there are more than one column groups in common. An example is (0,1), (0,2) and (1,2) where not all column groups have the same column in common. In general, the largest possible selection of such column groups is when the groups are $(i, i+1), (i, i+2)$ and $(i+1, i+2)$. Any other selection reduces to case 2 since any other selection does not form a clique greater than or equal to three in the column graph. Without loss of generality, select $(i+1, i+2)$ as the first group to consider. Fix the first column to one symbol. $(v-1)$ symbols may be fixed with its common column so that they form a clique of size $v$. Do the same for the second column and the other group. We now have a graph with 2 disjoint

cliques of size $v$. Connect the graph by fixing the first column of the second group to a symbol and using the remain $(v-1)$ symbols for the other group. We claim this forms a clique of size $(v-1)^2+(v-1)+1$. There are $(v-1)$ vertices in the second group. We fix the other column of all these vertices to a symbol. The third group also has $(v-1)$ vertices. However, we fix $(v-1)$ symbols to the other column for each of the existing $(v-1)$ vertices thus creating $(v-1)^2$ vertices. It is easy to see that a clique of size $(v-1)^2+(v-1)+1$ is formed. Now take any other interaction from these three groups that do not belong to the graph. The addition of any interaction does not increase the size of the clique since there would be at least one vertex in the graph which would not have an edge with the new interaction.

All possible ways to form graphs can be handled by the cases above. Thus, no clique of size greater than $v^t$ exists in a covering array when $t=2$. □

## 4.5 Exact Coloring to Analyze the Success of Post-optimization

An interesting question to ask is when to post-optimize. It makes less sense to post-optimize an array which is very close to the best known or is in fact optimal. Certainly, indicators that would gauge the extent to which post-optimization would be successful are desirable. One possible indicator is the presence of *many* flexible positions in the covering array. Perhaps a more useful indicator is the minimum number of rows required to produce an improvement. Since the greedy coloring methods do not compute the true chromatic number we cannot be sure if a certain selection of $R$ rows admits an improvement or not. An alternative is to use exact vertex coloring in place of greedy coloring. This would be impractical from a post-optimization point of view but can be very useful from an analysis point of view.

**Algorithm 5:** Recolor with Exact Coloring

```
input  : A covering array CA(N; t, k, v)
output : A map X : x ⇒ {true, false}
begin
    X ⇐ ∅
    for numRows = 2 to N do
        improved ⇐ false
        R ⇐ all (N choose numRows) pairs of rows
        forall the pairs of rows r ∈ R do
            P(r) ⇐ computePrivateInteractions()
            Form graph G
            χ(G) ⇐ exactVertexColoring(G)
            if χ(G) < |r| then
                improved ⇐ true
                rearrangeAndDeleteRows()
            end
        end
        Add (numRows, improved) to X
    end
end
```

Algorithm 5 represents an exact coloring based implementation of Recoloring. A few differences from the original method are that we generate all row-sets possible except those of cardinality one and not according to any heuristic. Next, we use an exact coloring method to get the true chromatic number. Finally and perhaps more importantly, we only rearrange the rows when the chromatic number is *strictly* lesser than the number of rows selected. This is because we need not worry about local optima when the coloring method returns the chromatic number since if we colored all rows then the chromatic number would be the covering array number.

Since we do not reorder rows when there is no improvement, it is possible that some permutation $\pi_{r_1}\pi_{r_2}\pi_{r_3}...\pi_{r_n}$ exists that reordering the interactions may lead to an improvement. We provide a simple example in which we consider three rows $a, b$

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | - | - | - | ab | ab | ab | - | - | - | a | a | a |
| - | b | b | b | - | b | ab | ab | ab | - | - | - | - | - | - |
| - | - | - | - | - | c | c | c | c | - | - | - | c | c | c |

Table 11: An Example Where a 2-coloring without Reordering Exists Only When 3 Rows are Selected

and $c$ and their interactions as shown in Table 11. Without loss of generality, let $t = 3$ and further let the longest contiguous sequence of characters represent private interaction(s). Entries labeled $a$ belong to $\Gamma(a)$ and so forth. Selecting any pair of rows does not yield a 1-coloring, however if rows $a$ and $b$ were colored such that the interactions representing columns 5 through 8 in $b$ are now rearranged in $a$ then rows $b, c$ admit one coloring. Such an improvement depends on rearranging the interactions which the current implementation does not account for. However, when the method considers row sets of cardinality three it will report the improvement. Thus Algorithm 5 occasionally provides overestimates of the number of rows actually required. To circumvent this overestimation, the method would need to rearrange the interactions in such a way that later row sets may be able to produce an improvement. This is difficult to achieve and thus the simplistic method is much more convenient to use.

We now analyze some results on using exact coloring on some instances. Table 12 lists some analysis of small covering arrays. Performing exact coloring on large instances is not feasible due to the amount of time required iterating over all possible row sets. In general, one can observe that as the size of covering array approaches the best known instance the number of rows required to produce an improvement increases sometimes requiring as much as 40% of the rows of the covering array. This is significant since it implies that unless the coloring method can produce a successful coloring on larger graphs the extent to which Recoloring can post-optimize is limited.

| $CA_{IPOG-F}(13; 2, 5, 3)$ CAN = 11 | |
|---|---|
| Row Set size | Resulting Covering Array |
| 2 | 13 |
| 3 | 11 |
| 4 | 11 |
| $CA_{IPOG-F}(15; 2, 6, 3)$ CAN = 12 | |
| Row Set size | Resulting Covering Array |
| 2 | 15 |
| 3 | 13 |
| 4 | 13 |
| 5 | 12 |
| 6 | 12 |
| $CA_{IPOG-F}(52; 3, 7, 3)$ Best Known = 39 | |
| Row Set size | Resulting Covering Array |
| 2 | 47 |
| 3 | 43 |
| 4 | 42 |

Table 12: Results of Exact Coloring

The results in Table 12 depend on the initial structure of the covering array and may yield different results for the same covering array produced by different methods. Another observation is that when the covering array is far from optimal, many improvements can be found over the same size of the row set.

# Chapter 5

# ADDITIONAL APPLICATIONS OF RECOLORING

We now state additional applications of Recoloring. Section 5.1 discusses an application where Recoloring may be allowed to break coverage by a certain amount. Section 5.2 discusses the use of Recoloring as a method for constructing covering and quilting arrays.

## 5.1 Reducing Coverage by a Controlled Percentage

In industrial applications where time-to-market is a critical factor for success, the cost to exercise all tests in the covering array is prohibitive. Generally it is tolerable to lose some percentage of coverage provided that important tests are not discarded from the covering array. The approach to covering important tests is called test prioritization. Currently, greedy methods can produce such arrays but they often need to recompute the array from scratch if the set of important tests changes, or add redundant tests covering only the newly added tests. Recoloring can be adjusted to maximize the number of rows removed while retaining the minimum level of coverage and important tests. A naive implementation of Recoloring is specified in Algorithm 6. Certain heuristics could be used to improve the results even further.

Briefly, the algorithm post-optimizes using Recoloring procedure until it fails to produce an improvement for a certain number of iterations. It then tries to remove certain interactions (breaking coverage) such that the number of rows is reduced. It might be possible that the number of rows might not be reduced in spite of removing

**Algorithm 6:** recolorWithTestPrioritization

```
input  : A covering array CA(N; t, k, v)
         A set of required interactions Δ
         A minimum required coverage percent Θ
output : A covering array CA(N'; t, k, v) with coverage c ≥ Θ and N' ≤ N
begin
    localOptimaLimit ⇐ K
    failedCount ⇐ 0
    coverage ⇐ 100
    NUM_TRIALS ⇐ C
    while time limit not expired and coverage ≥ Θ do
        status ⇐ recolor()
        if status = REDUCED then
            failedCount ⇐ 0
        end
        else
            failedCount++
        end
        if failedCount = localOptimalLimit then
            trial ⇐ 0
            status ⇐ FAILED
            while status ≠ REDUCED and trial < NUM_TRIALS do
                Remove interactions τ ∉ Δ such that c ≥ Θ from the graph G
                coverage ⇐ newCoverage status ⇐ recolor()
                if status ≠ REDUCED then
                    Add back the removed interactions
                    coverage ⇐ OldCoverage
                end
            end
            if trials = NUM_TRIALS then
                Remove interactions τ ∉ Δ such that c ≥ Θ from the graph G
                coverage ⇐ newCoverage recolor()
            end
        end
    end
end
```

| Percent Coverage | Rows |
|---|---|
| 100.00 | 26 |
| 100.00 | 21 |
| 96.67 | 20 |
| 92.50 | 19 |
| 88.75 | 18 |

Table 13: PriortizeOpt on a CA$(26; 4, 6, 2)$ with Best= 21

| Percent Coverage | Rows |
|---|---|
| 100.00 | 140 |
| 99.00 | 90 |
| 98.00 | 88 |
| 96.00 | 71 |
| 94.00 | 62 |
| 92.00 | 56 |
| 90.00 | 52 |

Table 14: PriortizeOpt on a CA$(140; 5, 17, 2)$ with Best= 104

| Percent Coverage | Rows |
|---|---|
| 100.00 | 1460 |
| 100.00 | 1382 |
| 99.97 | 1381 |
| 99.93 | 1380 |
| 99.93 | 1379 |
| 99.93 | 1378 |
| 99.93 | 1377 |
| 99.01 | 1325 |
| 98.95 | 1324 |
| 98.93 | 1323 |
| 98.00 | 1288 |
| 97.98 | 1287 |
| 97.02 | 1259 |
| 96.99 | 1258 |
| 95.59 | 1212 |
| 94.47 | 1175 |

Table 15: PriortizeOpt on a CA$(1460; 4, 8, 5)$ with Best= 1212

interactions. The method still removes interactions in this case as long as the minimum coverage requirements are met.

Tables 13, 14, and 15 represent some results that were produced using Recoloring to provide test prioritization. Post-optimization was allowed to run for a specified amount of time after which recoloring was permitted to remove interactions. For a better view of the results we only consider reduction with respect to the best known covering array. For CA$(26; 4, 6, 2)$, prioritizedRecoloring deleted 10% of the rows losing 7.5% coverage in the process. While this may not appear worthwhile, a more striking example is CA$(140; 5, 17, 2)$ where prioritizedRecoloring removed 50% of the rows while losing only 10% coverage in the process. This highlights the power of recoloring to provide test prioritization. As the instances get more complex (as systems get larger), the percent of coverage by deleting a row is negligible and a significant saving

can be obtained. It is important to note that the above examples did not enforce any interaction to never be deleted but this could have just as easily be enforced.

## 5.2 Generation of Covering and Quilting Arrays Using Recoloring

Recoloring can also be used in the generation of covering arrays. To do so form the covering array graph $CAG$ (the graph formed using all interactions) and choose any sub-graph. Recolor this sub-graph and choose the $n$ *largest* color classes where $n$ is the number of rows to be added at each iteration. While one may not expect the generation technique to compete with the likes of simulated annealing, it does offer an interesting solution. The ability to generate $n$ rows at a time allows recoloring to generate a covering array relatively quickly. The current state-of-art methods focus on adding one row at-a-time causing scalability issues. Other maximum independent set problems such as vertex cover could be used to substitute the coloring algorithm. Naturally, an optimization in the selection of the sub-graph may yield even better results. Algorithm 7 describes a naive way of generating a covering or quilting array using recoloring. The algorithm randomly selects a sub-graph of the covering array graph and colors it. The largest color classes form the rows of the covering array and the process is repeated until the covering array graph is not empty.

Tables 16, 17, 18, 19 and 20 give arrays generated using recoloring. The quilting arrays now serve as the current bounds since no quilting arrays for $v > 3$ had yet been computed. The generation method was slightly modified so that post-optimization was used within the process. These results mainly demonstrate the capability of recoloring as a generation method. The quality of the solutions could be improved by increasing the computing power.

**Algorithm 7:** recolorGenerateCoveringArray

**input** : A set of parameters $t, k, v, w$
**output** : A covering or quilting array of the given parameters
**begin**
  $\mathcal{G} \Leftarrow$ generate $CAG$
  $p \Leftarrow$ probability to post-optimize
  $n \Leftarrow$ # of rows to be generated at each iteration
  **while** $CAG \neq \emptyset$ **do**
    $\mathcal{G}' \Leftarrow$ subgraph($\mathcal{G}$)
    $\alpha(\mathcal{G}') \Leftarrow$ getProperVertexColoring()
    Form new rows with the $n$ largest color classes
    Remove all vertices of the $n$ largest color classes from $CAG$
    Apply post-optimization with probability $p$
  **end**
**end**

| t | k | v | Best | IPOG-F | Recoloring |
|---|---|---|---|---|---|
| 2 | 600 | 6 | 115 | 164 | 183 |
| 4 | 9 | 5 | 1245 | 1638 | 1542 |
| 5 | 17 | 2 | 104 | 183 | 164 |
| 5 | 9 | 5 | 6634 | 8629 | 7868 |
| 6 | 9 | 5 | 35680 | 41210 | 39026 |

Table 16: Covering Arrays Generated Using Recoloring

| k | $\mathbb{Q}_3^4$ | $\mathbb{Q}_4^4$ | $\mathbb{Q}_5^4$ | $\mathbb{Q}_6^4$ | $\mathbb{Q}_4^5$ | $\mathbb{Q}_6^5$ | $\mathbb{Q}_7^5$ | $\mathbb{Q}_8^5$ | $\mathbb{Q}_9^5$ | $\mathbb{Q}_5^6$ | $\mathbb{Q}_8^6$ | $\mathbb{Q}_9^6$ | $\mathbb{Q}_{11}^6$ | $\mathbb{Q}_{12}^6$ | $\mathbb{Q}_{13}^6$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 412 | 392 | 360 | 128 | 1686 | 1628 | 1501 | 1076 | 614 | - | - | - | - | - | - |
| 8 | 504 | 481 | 436 | 163 | 2028 | 1998 | 1923 | 1510 | 851 | 7928 | 7831 | 7660 | 7010 | 4983 | 2146 |
| 9 | 504 | 504 | 487 | 203 | 2028 | 2027 | 2016 | 1895 | 1111 | 8124 | 8124 | 8124 | 7946 | 6517 | 3671 |
| 10 | - | - | - | - | - | - | - | - | - | 8172 | 8172 | 8172 | 8148 | 7525 | 5732 |
| 11 | - | - | - | - | - | - | 3152 | - | - | - | - | - | - | - | - |

Table 17: Quilting Arrays $\mathbb{Q}_w^t$-QA$(N; t, k, 4)$ Generated Using Recoloring

| k | $\mathbb{Q}_3^4$ | $\mathbb{Q}_4^4$ | $\mathbb{Q}_5^4$ | $\mathbb{Q}_6^4$ | $\mathbb{Q}_4^5$ | $\mathbb{Q}_6^5$ | $\mathbb{Q}_7^5$ | $\mathbb{Q}_8^5$ | $\mathbb{Q}_9^5$ | $\mathbb{Q}_{10}^5$ | $\mathbb{Q}_5^6$ | $\mathbb{Q}_8^6$ | $\mathbb{Q}_9^6$ | $\mathbb{Q}_{11}^6$ | $\mathbb{Q}_{12}^6$ | $\mathbb{Q}_{13}^6$ | $\mathbb{Q}_{14}^6$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 909 | 909 | 899 | 366 | 5568 | 5466 | 5244 | 4247 | 3015 | - | - | - | - | - | - | - | - |
| 8 | 1207 | 1200 | 1157 | 505 | 6202 | 6171 | 6007 | 5304 | 4055 | 1119 | 24999 | 24989 | 24951 | 24680 | 22573 | 18609 | 7010 |
| 9 | 1240 | 1240 | 1240 | - | 6617 | 6614 | 6590 | 6271 | 4952 | 1567 | 35629 | 35627 | 35610 | 35336 | 32902 | 28614 | 18030 |
| 10 | - | - | - | - | - | - | - | - | - | - | 43337 | 43336 | 43334 | 43164 | 41464 | 38218 | 28892 |
| 13 | - | - | - | 1859 | - | - | - | - | - | - | - | - | - | - | - | - | - |

Table 18: Quilting Arrays $\mathbb{Q}_w^t$-QA$(N; t, k, 5)$ Generated Using Recoloring

| k | $\mathbb{Q}_3^4$ | $\mathbb{Q}_4^4$ | $\mathbb{Q}_5^4$ | $\mathbb{Q}_6^4$ | $\mathbb{Q}_4^5$ | $\mathbb{Q}_6^5$ | $\mathbb{Q}_7^5$ | $\mathbb{Q}_8^5$ | $\mathbb{Q}_9^5$ | $\mathbb{Q}_{10}^5$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 1838 | 1838 | 1832 | 828 | 17028 | 16977 | 16968 | 13754 | 11482 | - |
| 8 | 1974 | 1969 | 1964 | 1241 | 20095 | 20232 | 19615 | 17779 | 15374 | 9468 |
| 9 | 2768 | 2764 | 2754 | 1641 | 21679 | 21607 | 21556 | 20556 | 18506 | 14858 |
| 10 | - | 3658 | - | - | - | - | - | - | - | - |

Table 19: Quilting Arrays $\mathbb{Q}_w^t$-QA$(N; t, k, 6)$ Generated Using Recoloring

| k | $\mathbb{Q}_5^6$ | $\mathbb{Q}_8^6$ | $\mathbb{Q}_9^6$ | $\mathbb{Q}_{11}^6$ | $\mathbb{Q}_{12}^6$ | $\mathbb{Q}_{13}^6$ | $\mathbb{Q}_{14}^6$ | $\mathbb{Q}_{15}^6$ |
|---|---|---|---|---|---|---|---|---|
| 8 | 70777 | 70767 | 70726 | 70378 | 67518 | 62586 | 38331 | 29289 |
| 9 | 129966 | 129936 | 129627 | 128676 | 124375 | 120920 | 121788 | 113040 |
| 10 | 131743 | 131742 | 131730 | 131571 | 129955 | 126999 | - | - |

Table 20: Quilting Arrays $\mathbb{Q}_w^t$-QA$(N; 6, k, 6)$ Generated Using Recoloring

# Chapter 6

# CONCLUSION

## 6.1 Contribution

The primary contribution of this thesis is the introduction of a new idea to effectively post-optimize covering and quilting arrays. Recoloring has a smaller likelihood of converging to a local optimum than its counterparts since recoloring does not require flexible symbols in post-optimization. In fact, the chromatic number of the covering array graph is the covering array number. Thus given enough computational time, Recoloring can find an optimal covering array. The method is competitive and often outperforms the existing algorithms for smaller instances.

Several new bounds for quilting arrays were computed. Recursive constructions exist which use quilting arrays to produce covering arrays. A reduction of a few rows in the seed quilting array can lead to a large decrease in the size of the resultant array. These new bounds can certainly improve bounds on existing results.

Another surprising application of Recoloring towards test prioritization was demonstrated wherein Recoloring could reduce the cost by more than half while still retaining a significant proportion of coverage. Finally, a scalable and fast generation technique which can generate more than one row at a time was described.

An important theoretical aspect of any post-optimization algorithm is the ability to anticipate the likelihood of success. This reduces unnecessary work performed by a post-optimizer. The analysis for post-optimization metrics performed in this thesis

can provide valuable pointers towards the choice of the coloring method and the rows to select.

Finally, by bounding the clique number, one can justify that using the interaction graph is more suitable for post-optimization than another graph with a possibly arbitrarily large clique number.

## 6.2 Future Work

Based on observations, it is likely that the clique number of the covering array for general $t, k$ and $v$ is $v^t$. This thesis proved the case for $t = 2$. It would be interesting to generalize the proof. This could lead to interesting insight about the structure of the covering array and perhaps help in finding embedded designs in covering arrays.

Exact coloring showed that the scalability of the method heavily depends on the coloring algorithm. The results produced by a poor coloring method are competitive enough to warrant the use of heuristic coloring methods.

Recoloring could be extended to other problems similar to covering arrays. One close relative is *mixed covering arrays* which generalize the set of symbols such that each column gets its own set. In general recoloring could be adapted to several $k$-restriction problems.

The application of recoloring indicates a close relation between covering arrays and the maximum independent set problem. Perhaps another approach could be explored and more details about the covering array graph could be unveiled.

# REFERENCES


[1] D. Brelaz. "New methods to color the vertices of a graph". In: *A.C.M* 7 (1979), pp. 494–498.

[2] R. C. Bryce and C. J. Colbourn. "The density algorithm for pairwise interaction testing". In: *Software Testing, Verification and Reliability* 17.3 (2007), pp. 159–182.

[3] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. "Augmenting simulated annealing to build interaction test suites". In: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. Nov. 2003, pp. 394–405.

[4] C. J. Colbourn. "Combinatorial aspects of covering arrays". In: *Le Matematiche* 59.1,2 (2006), pp. 125–172.

[5] C. J. Colbourn. "Covering array tables". In: (2005). [Online; Accessed: 1st October 2015]. URL: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html.

[6] C. J. Colbourn. "Covering arrays, augmentation, and quilting arrays". In: *Discrete Mathematics, Algorithms and Applications* 06.03 (2014), p. 1450034.

[7] C. J. Colbourn and J. Zhou. "Improving two recursive constructions for covering arrays". In: *Journal of Statistical Theory and Practice* 6.1 (2012), pp. 30–47.

[8] C. J. Colbourn et al. "Roux-type constructions for covering arrays of strengths three and four". In: *Designs, Codes and Cryptography* 41.1 (2006), pp. 33–57. ISSN: 0925-1022.

[9] M. A. Forbes et al. "Refining the in-parameter-order strategy for constructing covering arrays". In: *Journal of Research of the National Institute of Standards and Technology* 113.5 (Sept. 2008).

[10] A. P. Godbole, D. E. Skipper, and R. A. Sunley. "t-Covering arrays: Upper bounds and poisson approximations". In: *Combinatorics, Probability and Computing* 5 (June 1996), pp. 105–117.

[11] M. Grindal, J. Offutt, and S. F. Andler. "Combination testing strategies: A survey". In: *Software Testing, Verification and Reliability* 15.3 (2005), pp. 167–199.

[12] G. O. H. Katona. "Two applications (for search theory and truth functions) of sperner type theorems". In: *Periodica Mathematica Hungarica* 3.1-2 (1973), pp. 19–26.

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

[14] V. V. Kuliamin and A. A. Petukhov. "A survey of methods for constructing covering arrays". In: *Programming and Computer Software* 37.3 (2011), pp. 121–146.

[15] Y. Lei et al. "IPOG: A general strategy for t-way software testing". In: *In Proceedings of the International Conference on engineering of Computer-Based Systems (eCBS*. 2007, pp. 549–556.

[16] J. R. Lobb et al. "Cover starters for covering arrays of strength two". In: *Discrete Mathematics* 312.5 (2012), pp. 943–956.

[17] E. Malaguti and P. Toth. "A survey on vertex coloring problems". In: *International Transactions in Operational Research* 17.1 (2010), pp. 1–34.

[18] E. Maltais. "Covering arrays avoiding forbidden edges and edge clique covers". MA thesis. University of Ottawa, 2009.

[19] K. Meagher. "Covering arrays on graphs: Qualitative independence Graphs and extremal set partition theory". PhD thesis. University of Ottawa, 2005.

[20] P. Nayeri. "Post-Optimization: Necessity analysis for combinatorial arrays". PhD thesis. Arizona State University, 2011.

[21] NIST. "Covering arrays generated by IPOG-F". In: (2008). [Online]. URL: http://math.nist.gov/coveringarrays/ipof/ipof-results.html.

[22] K. J. Nurmela. "Upper bounds for covering arrays by tabu search". In: *Discrete Applied Mathematics* 138.1–2 (2004). Optimal Discrete Structures and Algorithms, pp. 143–152.

[23] G. Roux. "k-propriétés dans des tableaux de n colonnes; cas particulier de la k-surjectivité et de la k-permutivité". PhD thesis. University of Paris, 1987.

[24] J. Torres-Jimenez and E. Rodriguez-Tello. "New bounds for binary covering arrays using simulated annealing". In: *Information Sciences* 185.1 (2012), pp. 137–152.

[25] D. J. A. Welsh and M. B. Powell. "An upper bound for the chromatic number of a graph and its application to timetabling problems". In: *The Computer Journal* 10.1 (1967), pp. 85–86.

[26] D. Zuckerman. "Linear degree extractors and the inapproximability of max clique and chromatic number". In: *Theory of Computing* 3.6 (2007), pp. 103–128.