A NEW ALGORITHM FOR OBTAINING MIXED-LEVEL ORTHOGONAL AND NEARLY-ORTHOGONAL ARRAYS

by

Ryan Lekivetz

B.Sc., University of Regina, 2004

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Statistics and Actuarial Science

SIMON FRASER UNIVERSITY
Fall 2006

APPROVAL

Name: Ryan Lekivetz

Degree: Master of Science

Title of Project: A New Algorithm for Obtaining Mixed-Level Orthogonal and Nearly-Orthogonal Arrays

Examining Committee: Dr. Richard Lockhart
Chair

Dr. Derek Bingham
Senior Supervisor
Simon Fraser University

Dr. Randy Sitter
Simon Fraser University

Dr. Boxin Tang
External Examiner
Simon Fraser University

Date Approved: December 11, 2006

Abstract

Orthogonal arrays are frequently used in industrial experiments for quality and productivity improvement. Due to run-size constraints and level combinations, an orthogonal array may not exist, in which case a nearly-orthogonal array can be used. Orthogonal and nearly-orthogonal arrays can be difficult to find. This project will introduce a new algorithm for the construction of orthogonal arrays and nearly-orthogonal arrays with desirable statistical properties, and compare the new algorithm to a pre-existing algorithm.

Acknowledgments

There are far too many people I have met during my two years at SFU that should be thanked. In fear of forgetting to mention someone, I will just send out a big thank you to everyone. Chances are, if you are reading this now, you are one of the people I would like to thank (so thank you).

Big thanks to the graduate students I have met during my time at SFU. The fun times certainly made these two years fly by. And I certainly know that without your help I would not have gotten through all of my classes.

My sincerest gratitude to Dr. Derek Bingham and Dr. Randy Sitter. Without their knowledge and help, this project would not exist. Because of them and the great professors in the department, I've been inspired to continue my studies.

I can not forget to thank my family for their constant support and encouragement.

Last but certainly not least, my eternal gratitude to Cindy Feng for all her help and encouragement. I don't know how I would have survived this without her.

Contents

List of Tables

Chapter 1

Introduction

The concept of orthogonal arrays dates back to the 1940's to Rao (1947). Orthogonal arrays are frequently used in industrial experiments for quality and productivity improvement. When an experimenter believes a number of factors may impact a process, orthogonal arrays can be used to investigate which factors are active before further studies are done. Running all possible combinations of the levels for the factors may not be practical for a variety of reasons. When the model of interest is a normal linear regression model, orthogonal arrays give designs that allow an experimenter to consider a relatively large number of factors in relatively few trials while maintaining desirable statistical properties. For situations when orthogonal arrays do not exist, we consider the concept of nearly-orthogonal arrays. There are a variety of ways to measure the "goodness" of orthogonal arrays and nearly-orthogonal arrays, and also a number of ways to actually find them.

Orthogonal arrays, and nearly-orthogonal arrays, often have desirable statistical properties, but are not always easy to find. For some orthogonal arrays, construction can be done through existing theory, but in situations where theory does not apply or it is too time-consuming for an experimenter to find appropriate theory, an algorithm for constructing orthogonal arrays is needed. A number of different algorithms have been proposed for constructing orthogonal and nearly-orthgonal arrays; some of these include a Federov exchange algorithm from Miller and Nguyen (1994), an interchange algorithm from Nguyen (1996), a threshold accepting technique from Ma et al. (2000), an algorithm for a mixed level orthogonal array with many 2-level factors from DeCock and Stufken (2000), columnwise-pairwise algorithms from Li and Wu (1997), and a state-of-the-art algorithm from Xu (2002).

The strength of an orthogonal array is related to the estimability of interaction terms

in the normal linear regression model. The higher the strength of an orthogonal array, the more interaction terms can be estimated independently of each other. In most cases, orthogonal arrays of interest are of strength 2, as for strength 3 or higher, construction of such an array may not be very easy.

This project addresses the problem of trying to find an efficient algorithm for constructing orthogonal arrays and nearly-orthogonal arrays in a timely manner. In addition, we look to algorithms which can be adjusted to find orthogonal arrays with higher strength. Comparison will be made between Xu's algorithm (2002) and a new algorithm which will be introduced in this project.

In this project, Chapter 2 will introduce the concept of orthogonal and nearly-orthogonal arrays and some of their uses, and look at some of the criteria used to measure near-orthogonality. Chapter 3 will discuss two algorithms, one of which is new, that take a sequential approach to find orthogonal and nearly-orthogonal arrays. That is, the algorithms find designs by adding one column at a time. Also discussed in Chapter 3 is the extension of the algorithms to try and find orthogonal arrays of higher strength. Chapter 4 will compare the algorithms in the construction of some orthogonal arrays and nearly-orthogonal arrays with small runs. For orthogonal arrays, the algorithms will be compared in terms of efficiency for finding orthogonal arrays and speed. The construction of nearly-orthogonal arrays is compared in terms of a statistically justified criterion.

Chapter 2

Orthogonal and Nearly-Orthogonal Arrays

Experimenters are often concerned about how changes of certain factors impact a process, and want to investigate the effects of these factors simultaneously. When the aim of the experiment is to estimate the effects of these factors (mean effects, interactions,...), orthogonal arrays can be used. We will first examine areas in which orthogonal arrays are useful, followed by a formal definition.

As we will see, orthogonal arrays are desirable for their properties in estimating main effects and interactions in the normal regression model. In some situations, orthogonal arrays do not exist. In such cases, a nearly-orthogonal array is often a good alternative. In section 2.3, we introduce nearly-orthogonal arrays and discuss what it means to be "nearly" orthogonal.

2.1 Factorial Designs

2.1.1 Full Factorial Experiments

For many scientific settings, investigators are interested in studying a number of factors (variables to be studied) simultaneously. Often the goal of the experiment is to study the impact that factors have on a response variable of interest. At the beginning of an experiment, there may be a large number of factors which can impact the response. Before continuing study on these factors, it is useful to "screen" out the inert variables and identify

the important factors. If the set of (potentially) important factors can be reduced, more time can be spent studying the effect of the active factors.

Setting the factors at a fixed number of values, called levels, a factorial design is often used as a plan to run the experiment. When a set of levels has been determined for each factor, one way to examine how factors impact the response is to study all possible level combinations of all the factors. The hope is to discover how the factors impact the response individually, and how they may interact with each other.

For k factors, with s_1, $s_2, \cdots, s_k$ levels respectively, there are $s_1 \times s_2 \times \cdots \times s_k$ different combinations for the k factors. To run an experiment which involves each possible combination, the experiment requires $N = s_1 \times s_2 \times \cdots \times s_k$ runs.

For a general factorial design, we consider the standard normal regression model for a design $\mathbf{d}$,

$$Y = X_0\alpha_0 + X_1\alpha_1 + \cdots + X_m\alpha_m + \epsilon, \tag{2.1}$$

where Y is the vector of observations, α_j the vector of j-factor interactions, X_j the matrix of coefficients for α_j (column i corresponds to the coefficient for the ith effect) , and ϵ the vector of independent random errors which are distributed as $N(0, \sigma^2)$. When using a full factorial design, the main effects and j-factor interactions can be estimated independently of each other.

Example 2.1 Consider an experiment with three factors, each having two levels. We refer to these factors as A, B, and C. For each factor, if we consider one level to be "low", and the other "high", it is convenient to consider these levels as being -1 and $+1$ respectively. The full factorial design can be represented

$$\mathbf{X_1} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & +1 \\ -1 & +1 & -1 \\ -1 & +1 & +1 \\ +1 & -1 & -1 \\ +1 & -1 & +1 \\ +1 & +1 & -1 \\ +1 & +1 & +1 \end{pmatrix}. \tag{2.2}$$

Each row in the design represents one of the experimental runs, which would be randomized when implementing the experiment. The number in column 1 refers

to the level for factor 1, likewise for columns/factors 2 and 3. We refer to $\mathbf{X_1}$ as the design matrix.

The choice of using -1 and 1 as levels gives the added convenience of being able to estimate the main effects of the three factors given the response vector $\mathbf{y}$ with values corresponding to observations for the level settings of the factors in each row. The vector of main effects, $\hat{\mathbf{c}}_1$, can be calculated as

$$\hat{\mathbf{c}}_1 = \frac{1}{2^{n-1}}\mathbf{X}_1'\mathbf{y}.$$

In designing an experiment as a full factorial, not only can main effects be estimated independently, but also the interaction effects. The main effects and all interactions can be represented as

$$\mathbf{X} = \begin{pmatrix} -1 & -1 & -1 & +1 & +1 & +1 & -1 \\ -1 & -1 & +1 & +1 & -1 & -1 & +1 \\ -1 & +1 & -1 & -1 & +1 & -1 & +1 \\ -1 & +1 & +1 & -1 & -1 & +1 & -1 \\ +1 & -1 & -1 & -1 & -1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & -1 \\ +1 & +1 & -1 & +1 & -1 & -1 & -1 \\ +1 & +1 & +1 & +1 & +1 & +1 & +1 \end{pmatrix}. \tag{2.3}$$

Notice that the first three columns are still the same as the design matrix, $\mathbf{X_1}$, representing the main effects, A,B, and C, while the other columns represent the interaction effects, AB, AC, BC, and ABC respectively, and can be obtained by multiplying the columns across each row (another added convenience of the ± 1 coding). We refer to $\mathbf{X}$ as the model matrix. For a response vector $\mathbf{y}$ with values corresponding to observations for the level settings of the factors in each row, the vector of estimated main effects and interactions, $\hat{\mathbf{c}}$, can be calculated as

$$\hat{\mathbf{c}} = \frac{1}{2^{n-1}}\mathbf{X}'\mathbf{y}.$$

It turns out that the covariance matrix for $\hat{\mathbf{c}}$ under model (2.1) is diagonal, so the effects can be estimated independently of each other.

2.1.2 Fractional Factorial Experiments

While two-level full factorial designs are desirable as they can estimate all main effects and all linear interactions, running all possible combinations of the factors may not be feasible for a variety of reasons. These can range from economic limitations, ethical concerns for certain combinations of factors, to combinations that do not make practical sense or are not possible to be run together. In such situations, a fractional factorial design is frequently used.

Recall that running all possible combinations of factors allows for estimation of all main effects and linear interactions. It is still possible to use a fraction of the runs in the full factorial and still estimate many factorial effects. However, in not running all possible combinations of factors, the estimates of some factorial effects cannot be fully distinguished from each other. We refer to this inability to distinguish between effects as *aliasing*. A desirable design will attempt to ensure that those effects of most interest will not be aliased with each other.

If information is not available as to what effects may be of interest before creating a design, there is the need for a set of working assumptions to rank the importance of factorial effects. Three fundamental principles for factorial effects which are used to choose fractional factorial designs (eg. see Wu & Hamada (2000)) are:

1. **Hierarchical Ordering Principle**: (i) Lower order effects are more likely to be important than higher order effects, and (ii) effects of the same order are equally likely to be important.

2. **Effect Sparsity Principle**: The number of relatively important effects in a factorial experiment is small.

3. **Effect Heredity Principle**: In order for an interaction to be significant, at least one of its parent factors should be significant.

Using these principles, regular fractional factorial designs can be constructed and ranked. Under the principles, the main effects are the most important factorial effects to be estimated, followed by 2-factor interactions, 3-factor interactions, etc...

> **Example 2.2** An experimenter has seven 2-level variables of interest, $(A, \cdots, G)$, which are to be studied simultaneously. To run a full factorial would take 2^7

runs, but the experimenter wishes to perform the experiment in 2^3 runs. In each of the runs, care must be taken in how to select the level for each factor. For instance, if two factors are at the same level for each run, it is impossible to distinguish the main effects for these factors. In order to run the experiment, the experimenter can use the matrix $\mathbf{X}$ defined in (2.3), where the interaction columns of $\mathbf{X}$ will be the factor levels for which to run the additional variables. This assignment of variables can be shown through the following *generators*

$$\begin{aligned} D &= AB \\ E &= AC \\ F &= BC \\ G &= ABC. \end{aligned}$$

The set of all columns equal to the identity column I of all $1's$ is referred to as the *defining contrast subgroup*:

$$\begin{aligned} I &= ABD = ACE = BCF = ABCG \\ &= BCDE = ACDF = CDG = ABEF = BGE = AFG \\ &= DEF = ADEG = BDFG = CEFG = ABCDEFG. \end{aligned}$$

The defining contrast subgroup also gives rise to the *alias pattern*, which is the grouping of all factorial effects which are aliased. For example, for the main effect of A, the aliased factorial effects are

$$\begin{aligned} A &= BD = CE = ABCF = BCG \\ &= ABCDE = CDF = ACDG = BEF = ABGE = FG \\ &= ADEF = DEG = ABDFG = ACEFG = BCDEFG. \end{aligned}$$

The effects can be estimated using the same method for full factorials, keeping in mind that now some effects cannot be distinguished from each other. Using this fractional factorial design, the experimenter can only estimate 7 factorial effects. In fact, under the fundamental principles, the only effects to be estimated here would be the main effects for factors $A, \cdots, G$, under the assumption that all other effects are negligible.

In the previous example, the main effects for each of the factors could be estimated independently of each other. Keeping in mind the fundamental principles, if our run size is sufficiently large enough, we would like to be able to estimate lower order interactions which are independent from other lower order factorial effects (interactions involving fewer factors). This consideration leads to an important concept for fractional factorial designs: *resolution.*

A design is said to have resolution R (usually denoted by Roman numerals) if no p-factor effect is aliased with another effect having less than $R - p$ factors (see for instance, Montgomery (1997)). For example, in a resolution **III** design, no main effects are aliased with other main effects, but there is aliasing between main effects and two-factor interactions, and possibly two-factor interactions with each other. In a resolution **IV** design, main effects are not aliased with other main effects or two-factor interactions, but may be aliased with three-factor interactions, and two-factor interactions may be aliased with other two-factor interactions. The resolution of a fractional factorial is, in general, the length of the smallest word in the defining contrast subgroup. A high resolution is desirable, since as the resolution increases it allows more of the lower order interactions to be estimated independently of other lower order factorial effects, just as the main effects can be estimated independently.

As mentioned, we would like to find a design with resolution as high as possible. However, there may be many designs having the same resolution. Further comparison of designs can be done through the *minimum aberration criteria.* The minimum aberration criteria sequentially minimizes the elements of the word length pattern (if there are A_i words of length i in the defining contrast subgroup, the word length pattern is the vector $W = (A_3, A_4, \cdots)$) of the defining contrast subgroup. The purpose of the minimum aberration criteria is to keep the aliasing of lower-order interactions and main effects as minimal as possible.

If an experimenter can assume certain higher-order interactions are negligible, then information on main effects and lower-order interactions can be obtained by using a fraction of the runs of a full factorial design by using a fractional factorial design. Fractional factorial designs are particularly useful in screening experiments, to identify those factors that have large effects. Regular fractional factorial designs (those with a defining relation) can be represented by an orthogonal array, which we will introduce in the next section.

Not all fractional factorial designs have a defining contrast subgroup. We refer to these designs as nonregular designs. While in regular designs, any two factorial effects are either estimated independently or fully aliased, nonregular designs do not have this property.

Methods for ranking nonregular designs have been examined by Deng and Tang (1999), Tang and Deng (1999), and Xu and Wu (2001). Under these rankings, the best designs are generally orthogonal arrays. That is, orthogonal arrays are desiribable for both regular and nonregular fractional factorial designs.

2.2 Orthogonal Arrays

We now give a formal definition of orthogonal arrays.

Definition 2.1 *(orthogonal array) Let S be a set of s symbols denoted by $0, 1, \ldots, s-1$. An* orthogonal array A *with s symbols, strength t and index λ is an $N \times k$ array with entries from S such that every $N \times t$ subarray of A contains each t-tuple from S in exactly λ rows.*

An $N \times k$ orthogonal array with s levels, strength t and index λ will be denoted by $OA(N, k, s, t)$. For reasons that will be made clear, the parameter N will be referred to as run size, k as the number of factors, and s as the number of levels. These terms will be used interchangeably throughout. The parameter λ need not be mentioned in this notation, as it can be determined by the property

$$\lambda = N/s^t.$$

Example 2.3 Consider the following 8×7 array:

$$\begin{array}{ccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

This is an $OA(8, 7, 2, 2)$ since entries from the array are either 0 or 1, and given any two columns, the pairs (0, 0), (0, 1), (1, 0), (1, 1) occur the same number of times (twice), implying strength 2. Note that this array is not strength 3, as looking at columns 1, 2, and 4, the triplet (0, 0, 1) occurs twice, but (1, 0, 1) does not occur at all.

Some useful properties of orthogonal arrays:

1. An orthogonal array of strength t is also of strength t^*, where $1 \leq t^* < t$. For this reason, we will consider the strength of an orthogonal array as the largest strength attributed to that array.

2. A permutation of runs or factors of an orthogonal array creates an orthogonal array with the same parameters.

3. A permutation of the levels for any factor creates an orthogonal array with the same parameters.

While our definition of orthogonal arrays stated that the set S has symbols $0, \cdots, s-1$, we can replace these symbols with distinct symbols of our choice. For example, if S has symbols 0 and 1, we can replace these symbols by -1 and +1 and the resulting array is still an orthogonal array. From this, (2.2) is an $OA(8,3,2,3)$, and (2.3) is an $OA(8,7,2,2)$. In fact, by replacing the symbol 1 by -1 and 0 by +1 in example 2.1, the resulting orthogonal array is the matrix (2.3). In further examples, for a factor with s levels we will use the symbols $0, \cdots, s-1$.

For full factorial designs discussed in section 2.1.1, in the situation that we have m factors with s levels, the full factorial can be represented by an $OA(s^m, m, s, m)$.

For fractional factorial designs from section 2.1.2, a fractional factorial design can be represented by an orthogonal array. If a design has resolution R, it contains full factorial designs on any subset of $R-1$ columns, which by definition makes the orthogonal array strength $R-1$.

2.2.1 Mixed Orthogonal Arrays

The orthogonal arrays discussed in section 2.2 are such that each factor has the same number of levels. It may not be desirable or possible to use such an orthogonal array in some situations (ie. where a machine has one component which can be set to 2 levels while another which can be set to 3). The concept of orthogonal arrays can be extended to situations where factors have different numbers of levels.

Definition 2.2 *(mixed orthogonal array) Let S_i be a set of s_i levels denoted by $0, 1, \ldots, s_i - 1$ for $1 \leq i \leq v$ for some positive integer v ($s_i \geq 2$). We define a* mixed orthogonal array

$OA(N, s_1^{k_1} s_2^{k_2} \cdots s_v^{k_v}, t)$ to be an array of size $N \times k$ such that $k = k_1 + k_2 + \cdots + k_v$ and the first k_1 columns have symbols from S_1, the next k_2 columns have symbols from S_2, and so on, such that given any $N \times t$ subarray, each possible t-tuple appears in the same number of rows.

This definition does not require $s_1, s_2, \ldots, s_v$ to be distinct, but for simplicity we generally combine factors with the same number of levels. For example, instead of using the notation $2^2 2^1$, we would use 2^3.

We refer to t as the strength of the mixed orthogonal array. The previous comments for the orthogonal arrays in section 2.2 hold for mixed orthogonal arrays as well. Note that for mixed orthogonal arrays, we no longer consider the concept of *index*, as the number of times a t-tuple can occur may depend on the columns considered, as can be seen in the next example.

Example 2.4 A mixed orthogonal array $OA(12, 3^1 2^4, 2)$

$$
\begin{array}{ccccc}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 \\
2 & 0 & 0 & 1 & 1 \\
2 & 0 & 1 & 0 & 0 \\
2 & 1 & 0 & 1 & 0 \\
2 & 1 & 1 & 0 & 1
\end{array}
$$

The orthogonal arrays from section 2.1 can be written in the notation of mixed orthogonal arrays. For instance, example 2.3 would be an $OA(8, 2^7, 2)$. Such orthogonal arrays can be called *symmetrical orthogonal arrays*. From this point forward, we use the notation for mixed orthogonal arrays. Just as we can use symmetrical orthogonal arrays for full factorial designs and fractional factorial designs, we can do the same for mixed orthogonal arrays, and estimation of factorial effects can be done in the same manner. A full factorial design will be of the form $OA(N, s_1 s_2 \cdots s_k, k)$ where $N = \prod_{i=1}^{k} s_i$.

2.3 Nearly-Orthogonal Arrays

For a given run size, an orthogonal array with factors/levels specified may not exist. Likewise, for a given set of factors and levels, the run size required for an orthogonal array may not be feasible for reasons as discussed in section 2.1.2. Just because an orthogonal array does not exist should not mean that the experiment cannot be performed. An ideal compromise is to create a design that is in some way as close to an orthogonal array as possible. Wang and Wu (1992) considered the concept of these so-called *nearly-orthogonal arrays.*

Definition 2.3 *(nearly-orthogonal array (Xu(2002)))* *Let S_i be a set of s_i levels denoted by $0, 1, \ldots, s_i - 1$ for $1 \leq i \leq v$ for some positive integer v. We define a* nearly-orthogonal array $NOA(N, s_1^{k_1} s_2^{k_2} \cdots s_v^{k_v})$ *to be an array of size $N \times k$ such that $k = k_1 + k_2 + \cdots + k_v$ and the first k_1 columns have symbols from S_1, the next k_2 columns have symbols from S_2, and so on, such that the array is optimal according to some criterion.*

Non-orthogonality can cause problems for data analysis. For non-orthogonal arrays, the order for which effects enter into the model is important, and interactions can be partially aliased (neither fully aliased nor orthogonal) with main effects. Hamada and Wu (1992) and Chipman, Hamada, and Wu (1997) presented some data-analysis strategies for partially aliased effects.

One of the major issues in looking at nearly-orthogonal arrays is deciding what it means to be "nearly" orthogonal. A criterion should be attempting to measure a notion of departure from orthogonality and the ability to compare different designs for their near-orthogonality.

We will now look at some of the approaches that have been taken for measuring orthogonality, and compare some of the similarities in section 2.3.1. We begin with criteria based on the model matrix.

If we consider the main effects model, which drops the interaction terms from (2.1), then

$$E(Y) = \beta_0 + \sum_{i=1}^{m} x_i \beta_i$$

where β_0 is the grand mean, β_i is the ith effect, x_i is the corresponding coefficient, and y's have errors iid $N(0, \sigma^2)$. For an array with N runs, the model can be rewritten in matrix form

$$E(Y) = X\beta, \qquad Cov(Y) = \sigma^2 \mathbf{I}_N \tag{2.4}$$

where Y is the vector of N runs, $\beta = (\beta_0, \beta_1, \cdots, \beta_m)'$, $X = (\mathbf{1}, \mathbf{x_1}, \cdots, \mathbf{x_m})$, $\mathbf{x_i}$ being the vector of x_i values for the N runs of the array (the level for factor i at each of the runs) and $m = \sum_i (s_i - 1)$.

Let $\tilde{X} = [\mathbf{x_1}/||\mathbf{x_1}||, \cdots, \mathbf{x_m}/||\mathbf{x_m}||]$. Wang and Wu (1992) proposed the D criterion

$$D = |\tilde{X}'\tilde{X}|^{1/m}$$

to measure the overall efficiency of a nearly orthogonal array, where due to the standardization of $\tilde{X}$, $D = 1$ iff the $\mathbf{x}_i$'s are orthogonal to each other.

In trying to estimate the effects $\beta_1, \ldots, \beta_m$, then the variance of the least squares estimator of β_i is minimized when $\mathbf{x_i}$, the vector of x_i values for the N runs, is orthogonal to the other columns of X. For any design, $|\tilde{X}'\tilde{X}| \leq 1$, and $|\tilde{X}'\tilde{X}| = 1$ iff the original design $\mathbf{d}$ is an orthogonal array. Then the D criterion measures the efficiency of estimating $\beta_1, \ldots, \beta_m$ in (2.4).

If we let $\tilde{X}'\tilde{X} = [r_{ij}]$, then the A_2 criterion is defined as

$$A_2 = \sum_{i<j} r_{ij}^2.$$

The A_2 criterion measures the overall aliasing between all pairs of columns. An A_2-*optimal* design minimizes A_2, which is useful in that $A_2 = 0$ iff $\mathbf{d}$ is an orthogonal array of strength 2.

Another way to view A_2 is to consider the ANOVA model for a design $\mathbf{d}$ as defined by (2.1). Xu and Wu (2001) defined $A_j(\mathbf{d})$ as a measure of the aliasing between the j-factor interactions and the general mean. For $X_j = [x_{ik}^{(j)}]$, let

$$A_j(\mathbf{d}) = \frac{1}{N^2} \sum_k \left| \sum_{i=1}^{N} x_{ik}^{(j)} \right|^2 .$$

The generalized minimum aberration criterion is to sequentially minimize the terms in $(A_1(\mathbf{d}), A_2(\mathbf{d}), A_3(\mathbf{d}), \ldots)$. The generalized minimum aberration criterion is equivalent to other measures of minimum aberration: the minimum aberration criterion (Fries and Hunter (1980)) discussed in Section 2.1.2 for regular designs, Tang and Deng's (1999) minimum G_2-aberration criterion for two-level nonregular designs, and the minimum generalized aberration criterion (Ma and Fang (2001)) for multi-level nonregular designs.

Instead of working with the model matrix, it may be desirable to work instead with the design matrix only. For design $\mathbf{d} = [x_{ik}]_{N \times n}$, let $n_{kl}(a, b)$ be the number of rows

with column k at level $a \in \{0, \cdots, s_k\}$ and column l at level $b \in \{0, \cdots, s_l\}$. Denote $n_{kl}(a, \cdot) = \sum_{b=0}^{s_l-1} n_{kl}(a, b)$ and $n_{kl}(\cdot, b) = \sum_{a=0}^{s_k-1} n_{kl}(a, b)$, where $n_{kl}(a, \cdot)$ can be thought of as the number of times $a \in \{0, \cdots, s_k - 1\}$ appears in column k, while $n_{kl}(\cdot, b)$ is the number of times $b \in \{0, \cdots, s_l - 1\}$ appears in column l. Ye and Sudjianto (2003) used

$$\chi^2_{kl}(\mathbf{d}) = \sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} \frac{[n_{kl}(a, b) - n_{kl}(a, \cdot)n_{kl}(\cdot, b)/N]^2}{n_{kl}(a, \cdot)n_{kl}(\cdot, b)}, \tag{2.5}$$

relating this to the sum of squared correlations of pairs of orthonormal two-level contrasts for columns k and l. If $\chi^2_{kl}(\mathbf{d}) = 0$ then the two columns have orthogonal main effects.

Cramer (1946) defined the related measure

$$V_{kl}(\mathbf{d}) = \left[\frac{\chi^2_{kl}(\mathbf{d})/N}{min(s_k, s_l) - 1}\right]^2,$$

which takes on values between 0 and 1, and is equal to 0 if columns k and l have orthogonal main effects and is equal to 1 if the columns are completely aliased. Ye and Sudjianto (2003) proposed

$$E[V^2(\mathbf{d})] = \frac{1}{n(n-1)/2} \sum [V_{kl}(\mathbf{d})]^2$$

as a measure of nearly orthogonal main effects.

If the columns are balanced, then all elements in a column appear the same number of times, so $n_{kl}(a, \cdot) = N/s_k$ and $n_{kl}(\cdot, b) = N/s_l$. This simplifies $\chi^2_{kl}(\mathbf{d})$ to

$$\chi^2_{kl}(\mathbf{d}) = \sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} \frac{[n_{kl}(a, b) - N/(s_k s_l)]^2}{N/(s_k s_l)}. \tag{2.6}$$

If $n_{kl}(a, b) = N/(s_k s_l)$ for all $k < l$, then $\mathbf{d}$ is an orthogonal array. The simplified criterion (2.6) is used by Yamada and Lin (1999), Yamada et al. (1999) and Yamada and Matsui (2002). They proposed

$$ave(\chi^2(\mathbf{d})) = \sum_{1 \le k < l \le m} \chi^2_{kl}(\mathbf{d})/[m(m-1)/2]$$

as a measure of the average dependency of all columns.

Given a design $\mathbf{d}$, Ma et al. (2000) proposed the following criterion to measure orthogonality between columns k and l of $\mathbf{d}$:

$$f_\phi(k, l) = \sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} \phi\left(|n_{kl}(a, b) - N/(s_k s_l)|\right),$$

where $\phi(\cdot)$ is a monotonic increasing function on $[0, \infty)$ with $\phi(0) = 0$. As a measure of non-orthogonality of a design $\mathbf{d}$, for $\theta(\cdot)$ a monotonic increasing function on $[0, \infty)$ with $\theta(0) = 0$, the function

$$D_{\phi,\theta}(\mathbf{d}) = \sum_{1 \le k < l \le n} \theta(f_\phi(k, l))$$

is used as a measure of orthogonality. While more general than the χ^2 criterion, the $D_{\phi,\theta}$ criterion is still ultimately a measure looking at the balance of factor combinations between two columns.

Xu (2002) introduced the J_2 criterion for measuring orthogonality. For an $N \times n$ matrix $\mathbf{d} = [x_{ik}]$ where column k has s_k levels, define

$$\delta_{i,j}(\mathbf{d}) = \sum_{k=1}^{n} w_k \delta(x_{ik}, x_{jk}), \qquad 1 \le i, j \le N, \tag{2.7}$$

where $\delta(a, b) = 1$ if $a = b$, 0 otherwise, and w_k is the weight of the column. For rows i and j, $\delta_{i,j}(\mathbf{d})$ is a measure of the similarity between these rows. If $w_k = 1$ for all k, $\delta_{i,j}(\mathbf{d})$ is the number of columns in which rows i and j coincide. Xu (2002) defined

$$J_2(\mathbf{d}) = \sum_{1 \le i < j \le N} [\delta_{i,j}(\mathbf{d})]^2 \tag{2.8}$$

as a measure of orthogonality in a design. Xu (2003) and Xu and Lau (2006) used J_2 in power moments for supersaturated designs. Whereas many criterion have a value of 0 for an orthogonal array, Xu (2002) established a lower bound on J_2:

$$\begin{aligned} J_2(\mathbf{d}) &\ge L(n) \\ &= 2^{-1} \left[\left(\sum_{k=1}^{n} N s_k^{-1} w_k \right)^2 + \left(\sum_{k=1}^{n} (s_k - 1)(N s_k^{-1} w_k)^2 \right) \right. \\ &\quad \left. - N \left(\sum_{k=1}^{n} w_k \right)^2 \right], \end{aligned} \tag{2.9}$$

where equality holds iff $\mathbf{d}$ is an orthogonal array.

2.3.1 Connection Between Criteria

In the preceding section, a number of different measures for near-orthogonality were introduced. Many of these criteria, while appearing to be dissimilar, are actually measuring

near-orthogonality in the same way.

Xu (2002) showed that for a balanced design $NOA(N, s_1 \cdots s_m)$,

$$J_2(\mathbf{d}) = N^2 A_2(\mathbf{d}) + 2^{-1} N \left[Nm(m-1) + N \sum s_k - \left(\sum s_k \right)^2 \right]. \tag{2.10}$$

The second term of the right-hand side is constant, so for the balanced design, J_2 and A_2 are equivalent.

Also equivalent in the balanced case is $ave(\chi^2)$ and A_2 through the equality

$$ave(\chi^2) = NA_2/[m(m-1)/2].$$

The significance of these equivalences is that these criterion are all measuring the balance of the $n_{kl}(a,b)'s$ over all a, b for each pair k, l. This is made even more clear by the following lemma from Xu (2003):

Lemma 2.1 *For integers $m, n \geq 0$, define $h(m,n) = \lfloor m/n \rfloor^2 n + (2\lfloor m/n \rfloor + 1)(m - \lfloor m/n \rfloor n)$. Let $x_1, \cdots, x_n$ be nonnegative integers such that $\sum x_i = m$. Then $\sum x_i^2 \geq h(m,n)$ with equality iff all x_i equal $\lfloor m/n \rfloor$ or $\lfloor m/n \rfloor + 1$.*

From this lemma, to find optimal designs with the A_2 criterion (and those it is equivalent to), it is sufficient to find a design such that $n_{k,l}(a,b)'s$ are within one for all a, b, given any pair of columns k, l. In fact, if such a design exists, Ma et. al's (2000) $D_{\phi,\theta}(\mathbf{d})$, in the case where $\phi(x) = x^2$, $\theta(x) = 1$ (which was used by Ma et al. (2000)) will be minimized as well. Also, $\phi(x) = x^2$ is related to the χ^2_{kl} criterion from equation (2.6).

In an orthogonal array all level combinations in columns k and l appear equally often. When considering a nearly-orthogonal array, making $n_{k,l}(a,b)$'s as balanced as possible would intuitively seem a method of making the array nearly-orthogonal. The previous discussion on criterion shows that this line of reasoning also has a statistical justification, and that most approaches to considering near-orthogonality are handled using this approach.

2.4 Construction of Orthogonal and Nearly-Orthogonal Arrays

In some settings, there are a substantial number of designs available which can be obtained through existing theory or from tables of designs. However, in some situations, finding

optimal designs can be difficult. Some orthogonal arrays require a number of different mathematical techniques to find all orthogonal arrays (an essential resource for these techniques is Hedayat, Sloane, and Stufken (1999)). If an optimal or even a good design is not available, it is likely too time-consuming to examine all potential theoretical methods to find a desirable array. We would like a computer algorithm that can find orthogonal arrays or nearly-orthogonal arrays based on some criterion in a fast and efficient manner. Chapter 3 will examine some algorithms which can be used in the construction of orthogonal and nearly-orthogonal arrays.

Chapter 3

Two Algorithms for the Construction of Orthogonal and Nearly-Orthogonal Arrays

While the mathematical theory exists for the construction of many orthogonal arrays, it may not always be enough. If an experimenter does not have the mathematical theory which applies for a given situation (if it even exists) there needs to be a method to construct the best design possible. Ultimately, the experimenter wants a design that has some nice statistical properties for the run size with the desired factors/levels. In such situations, it is ideal to have an algorithm to find an optimal design, or at least one that is near-optimal. In this chapter, an algorithm for finding "good" designs is presented, and a new one is proposed.

3.1 Xu's Forward Procedure Algorithm using the J_2 criterion

Many attempts have been made at finding efficient algorithms for the construction of orthogonal arrays. Xu (2002) discusses some of the different algorithms, and introduces his own, which will be discussed in this section. In that article, the author shows his algorithm to be superior in terms of both speed and efficiency compared to existing approaches.

The algorithm sequentially adds columns to an existing design, attempting to find a new column orthogonal to the columns already in the design. Furthermore, the algorithm uses

swapping of symbols in the new column and makes switches based on one which gives the greatest reduction in the J_2 criterion.

We begin by examining the operations which will be used in Xu's algorithm involving column addition and symbol switching.

For an $N \times n$ matrix $\mathbf{d} = [x_{ik}]$ where factor(column) k has s_k levels, define $\delta_{i,j}(\mathbf{d})$ as (2.7) and $J_2(\mathbf{d})$ as (2.8) as a measure of orthogonality in a design.

Consider adding column $\mathbf{c}$ to $\mathbf{d}$, denoting the resulting $N \times (n+1)$ matrix as $\mathbf{d}_+$. The columns $1, \ldots, n$ remain unchanged, so the only difference between $\delta_{i,j}(\mathbf{d}_+)$ and $\delta_{i,j}(\mathbf{d})$ is the consideration of the symbols in $\mathbf{d}_+$ from the new column $\mathbf{c}$ in rows i and j. For weight w_k which is pre-assigned to column $\mathbf{c}$ with s_k levels,

$$\delta_{i,j}(\mathbf{d}_+) = \delta_{i,j}(\mathbf{d}) + w_k \delta_{i,j}(\mathbf{c}), \tag{3.1}$$

for $1 \leq i, j \leq N$. To update J_2,

$$\begin{aligned} J_2(\mathbf{d}_+) &= \sum_{i<j} [\delta_{i,j}(\mathbf{d}_+)]^2 \\ &= \sum_{i<j} [\delta_{i,j}(\mathbf{d}) + w_k \delta_{i,j}(\mathbf{c})]^2 \\ &= \sum_{i<j} [\delta_{i,j}(\mathbf{d})]^2 + 2w_k \sum_{i<j} [\delta_{i,j}(\mathbf{d}) \delta_{i,j}(\mathbf{c})] + w_k^2 \sum_{i<j} [\delta_{i,j}(\mathbf{c})]^2 \\ &= J_2(\mathbf{d}) + 2w_k \sum_{i<j} [\delta_{i,j}(\mathbf{d}) \delta_{i,j}(\mathbf{c})] + w_k^2 \sum_{i<j} [\delta_{i,j}(\mathbf{c})], \end{aligned} \tag{3.2}$$

where the last equality comes from the fact that $\delta_{i,j}(\mathbf{c})$ can only take on values of 0 or 1. This also allows for fast computation of $J_2(\mathbf{d}_+)$

Now consider switching distinct symbols in rows a and b of the newly added column $\mathbf{c}$. Then for $j \neq a, b$, $\delta_{a,j}(\mathbf{c}) = \delta_{j,a}(\mathbf{c})$ and $\delta_{b,j}(\mathbf{c}) = \delta_{j,b}(\mathbf{c})$ are switched. The switch does not effect $\delta_{a,b}(\mathbf{c})$, as the values are distinct, so this value is 0. To examine the difference in $J_2(\mathbf{d}_+)$, first consider fixing a certain row $j \neq a, b$. The effect of the switch only effects $\delta_{i,j}(\mathbf{d}_+)$ for $i = a, b$, all other rows are unchanged. In the old calculation for J_2, the terms which have been changed in the calculation are

$$\delta_{a,j}(\mathbf{d}) \delta_{a,j}(\mathbf{c}) + \delta_{b,j}(\mathbf{d}) \delta_{b,j}(\mathbf{c}).$$

After the switch, these terms will now be

$$\delta_{a,j}(\mathbf{d}) \delta_{b,j}(\mathbf{c}) + \delta_{b,j}(\mathbf{d}) \delta_{a,j}(\mathbf{c}).$$

Looking at the difference the swap has made, the results are:

$$\begin{aligned} & (\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})) \times \delta_{a,j}(\mathbf{d}) + (\delta_{b,j}(\mathbf{c}) - \delta_{a,j}(\mathbf{c})) \times \delta_{b,j}(\mathbf{d}) \\ = \quad & (\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})) \times (\delta_{a,j}(\mathbf{d}) - \delta_{b,j}(\mathbf{d})). \end{aligned}$$

For the overall effect on $J_2(\mathbf{d}_+)$, the symbol swap will reduce $J_2(\mathbf{d}_+)$ by $\Delta(a,b)$ such that

$$\Delta(a,b) = 2w_k \sum_{j \neq a,b} [\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})][\delta_{a,j}(\mathbf{d}) - \delta_{b,j}(\mathbf{d})]. \tag{3.3}$$

3.1.1 The Algorithm

As discussed, Xu's algorithm sequentially adds columns to an existing design. A random, balanced column is added, and the algorithm searches all possible switches of elements in the new column, performing the best switch if it exists. This process continues until a lower bound is reached, or no improvement is made. If the lower bound is not reached, another attempt will be made at finding an orthogonal column, up to a prespecified number of times.

The algorithm proceeds as follows:

1. For $k = 1, \cdots, n$, compute the lower bound $L(k)$ by equation (2.9).

2. Specify an initial design $\mathbf{d}$ with columns $(0, \ldots, 0, 1, \ldots, 1, \ldots, s_1 - 1, \ldots, s_1 - 1)'$ and $(0, \cdots, s_2 - 1, 0, \cdots, s_2 - 1, \cdots, 0, \cdots, s_2 - 1)'$, and compute $\delta_{i,j}(\mathbf{d})$ and $J_2(\mathbf{d})$ by definition.

3. For $k = 3, \cdots, n$, do the following:

   i. Generate a random balanced s_k-level column $\mathbf{c}$. Compute $J_2(\mathbf{d}_+)$ by equation (3.2). If $J_2(\mathbf{d}_+) = L(k)$, go to step (iv).

   ii. For all pairs of rows a and b with distinct symbols, compute $\Delta(a,b)$ according to equation (3.3). Choose a pair of rows with the largest $\Delta(a,b)$ and exchange the symbols in rows a and b of column $\mathbf{c}$. Reduce $J_2(\mathbf{d}_+)$ by $\Delta(a,b)$. If $J_2(\mathbf{d}_+) = L(k)$, go to (iv); otherwise repeat (ii) until no further improvement is made.

   iii. Repeat (i) and (ii) T times and choose column $\mathbf{c}$ that produces the smallest $J_2(\mathbf{d}_+)$.

   iv. Add column $\mathbf{c}$ as the kth column of $\mathbf{d}$, let $J_2(\mathbf{d}) = J_2(\mathbf{d}_+)$, and update $\delta_{i,j}(\mathbf{d})$ by equation (3.1).

4. Return the final $N \times n$ design $\mathbf{d}$.

The quantity T in the algorithm represents the number of times we try to add an additional column. That is, when the algorithm attempts to add a column and does not meet the lower bound when switches are exhausted, then we try again with another random column. We refer to T as the number of *restarts*, that is, the number of times we have to start a column from the beginning. The number of restarts play an important role in the efficiency of the algorithm. The effect of T will be studied in Chapter 4.

Xu's algorithm has the advantage of being columnwise, which allows for balance in each column at each iteration. In addition, if the current design at any time is not orthogonal, it has been chosen as "near" orthogonal as possible according to the J_2 criteria. While the algorithm is designed in a way to keep speed in mind, the number of rows has a major impact on the speed. This is due to having to look at all possible distinct pairs in a column for the design, as well as having to provide updates for a number of rows after a switch.

3.1.2 Xu's Algorithm using the χ^2 criterion

Recall from section 2.3.1 that in the situation where columns are balanced, the J_2 and χ^2 criterion are equivalent. In fact, Xu's algorithm can be used with the χ^2 criterion instead of J_2. In order to use the χ^2 criterion, we will establish operations for column addition and symbol switching used in the algorithm for the χ^2 criteria as opposed to J_2.

For an added column, we need to count the $n_{kl}(i,j)$'s for all previous columns $k = 1, \ldots, l-1$, from which we can get χ^2_{kl} and calculate χ^2 by definition. For symbol switching, suppose that a column $\mathbf{c}$ has been added to the current design, such that it is the lth column. Call this design $\mathbf{d_l}$. Let $c_{\alpha l}$ and $c_{\beta l}$ be the elements in rows α and β of $\mathbf{c}$ that we wish to switch. Denote $x_{\alpha i}$ and $x_{\beta i}$ as the elements in column i for rows α and β. Making the switch decreases $n_{kl}(x_{\alpha k}, c_{\alpha l})$ and $n_{kl}(x_{\beta k}, c_{\beta l})$ by 1, and increases $n_{kl}(x_{\alpha k}, c_{\beta l})$ and $n_{kl}(x_{\beta k}, c_{\alpha l})$ by 1, for all $k = 1, \ldots, l-1$. If the design after the switch is denoted as $\mathbf{d}_\mathbf{l}^+$, for a particular column k, if $x_{\alpha k} = x_{\beta k}$, then $\chi^2_{kl}(\mathbf{d}_\mathbf{l}^+)$ is unchanged. If the elements in column k are different,

the impact on $\chi^2_{kl}(\mathbf{d}_l)$ is

$$\begin{aligned}
\chi^2_{kl}(\mathbf{d}_l^+) &= \sum_{i=0}^{s_k-1}\sum_{j=0}^{s_l-1}[n_{kl}^{(\mathbf{d}_l^+)}(i,j) - N/(s_k s_l)]^2 \\
&= \sum_{(i,j)\notin\{x_{\alpha k},x_{\beta k}\}\times\{c_{\alpha l},c_{\beta l}\}} [n_{kl}^{(\mathbf{d}_l)}(i,j) - N/(s_k s_l)]^2 \\
&\quad +[n_{kl}^{(\mathbf{d}_l)}(x_{\alpha k},c_{\alpha l}) - 1 - N/(s_k s_l)]^2 + [n_{kl}^{(\mathbf{d}_l)}(x_{\beta k},c_{\beta l}) - 1 - N/(s_k s_l)]^2 \\
&\quad +[n_{kl}^{(\mathbf{d}_l)}(x_{\beta k},c_{\alpha l}) + 1 - N/(s_k s_l)]^2 + [n_{kl}^{(\mathbf{d}_l)}(x_{\alpha k},c_{\beta l}) + 1 - N/(s_k s_l)]^2 \\
&= \chi^2_{kl}(\mathbf{d}_l) + 2[n_{kl}^{(\mathbf{d}_l)}(x_{\alpha k},c_{\beta l}) + n_{kl}^{(\mathbf{d}_l)}(x_{\beta k},c_{\alpha l}) \\
&\quad -n_{kl}^{(\mathbf{d}_l)}(x_{\alpha k},c_{\alpha l}) - n_{kl}^{(\mathbf{d}_l)}(x_{\beta k},c_{\beta l})] + 4. \qquad (3.4)
\end{aligned}$$

In general, for a symbol switch in rows α and β, we can define

$$\chi^2_{kl}(\mathbf{d}_l^+) = \chi^2_{kl}(\mathbf{d}_l) + \Delta_{kl}(\alpha,\beta)$$

where $\Delta_{kl}(\alpha,\beta)$ is computed as the remaining terms of (3.4) if $x_{\alpha k}$ and $x_{\beta k}$ are different, and 0 otherwise. Then a swap of the elements in column l of rows α and β will reduce $\chi^2(\mathbf{d}_l)$ by $\Delta_{\chi^2}(\alpha,\beta)$, where

$$\Delta_{\chi^2}(\alpha,\beta) = \sum_{k=1}^{l-1}\Delta_{kl}(\alpha,\beta). \qquad (3.5)$$

We now adapt Xu's algorithm to the χ^2 criterion as follows:

- replace all occurrences of J_2 with χ^2.
- replace $\Delta(a,b)$ by $\Delta_{\chi^2}(a,b)$ given by (3.5).
- replace the lower bounds L(n) by 0.
- in step 3(a), the updated χ^2 is calculated by determining the $n_{kl}(i,j)$'s for $k = 1,\ldots,l-1$.

3.1.3 Comments on the Algorithm

With the equivalence between the J_2 and χ^2 criterion for a balanced column, for any random balanced column added to a current design, using either criteria will ultimately lead to the same design. This is due to the best symbol switch for one criterion necessarily being the best for the other criterion. However, the manner in which the algorithm makes calculations

for the criteria is markedly different.

The J_2 criterion is driven by the $\delta_{i,j}$'s, the similarities between rows. When a column is added, the update is based on checking the similarity in the added column. When searching for switches to be made and upon making a switch, changes are based upon all rows affected. One of the nice properties of the J_2 criterion is that calculations for the current column are not influenced by how many columns already exist in the design. While it may be harder to find a suitable column with many pre-existing columns, the actual calculations are based on the $\delta_{i,j}$'s, which is not impacted if the present column is the third or the twentieth.

On the other hand, the χ^2 criterion is driven by the $n_{kl}(i,j)'s$, the number of times each pair of symbols between columns k and l occur. To compute χ^2 with a new column, the update involves calculating these counts for the new column and each of the previous columns. This situation also occurs with the switching of elements, in that the calculations involve having to look at the influence the switch has on the current column with the previous columns. In contrast to the J_2 criterion, as the number of columns in the design grows, the χ^2 criterion takes longer to calculate.

With modern computation, one criterion may be able to use computational features of a programming language to gain an edge over the other in terms of speed. However, we can still get a sense of how these criterion can impact the speed of the algorithm. The χ^2 criterion will run faster when there are less columns, as it is quicker to go over the columns rather than the number of rows in looking for switches. However, as the number of columns increases, the χ^2 criterion should take longer to calculate, and may lose the advantage to the J_2 criterion.

3.2 A New Algorithm Using a Sequential Approach

This section will introduce a new algoithmm using $\chi^2(\mathbf{d}) = \sum_{k<l} \chi^2_{kl}(\mathbf{d})$ as defined by (2.6) as a measure of near-orthogonality. Dropping the denominator, and instead using the simpler

$$\chi^2_{kl}(\mathbf{d}) = \sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} [n_{kl}(a,b) - N/(s_k s_l)]^2 \tag{3.6}$$

allows for a fast update, and can be used to drive the algorithm.

The algorithm follows Xu's in the idea of adding columns sequentially. However, instead

of adding a random column and attempting to make switches to improve the orthogonality of the design, we will instead take a sequential approach for the rows to add the new column. That is, elements will be added to a new column one row at a time, based upon which element seems best at that time.

Before presenting the algorithm, we will introduce some new notation and examine some of the quantities used in the algorithm for adding one symbol at a time in the new column. For an $N \times (l-1)$ design $\mathbf{d}$ in which we are looking to add column l, let

$$d_{lb^*}^{(h)} = [x_1^{(h)}|x_2^{(h)}|\cdots|x_{lb^*}^{(h)}], \tag{3.7}$$

the first h rows, where $x_{lb^*}^{(h)} = (x_{1l}, \cdots, x_{(h-1)l}, b^*)'$.

Denote $\chi_l^{2(hb^*)}$ as the criterion evaluated with h rows using symbol b^* in the last row (row h) for column l (ie. using $d_{lb^*}^{(h)}$ from equation (3.7)). To evaluate the criteria using symbol $b^* \in \{0, \ldots, s_l - 1\}$, the update can be done easily using $\chi^{2(h-1)}$, the χ^2 criteria using the first $h-1$ rows. Looking at column k and the new column, there is now one more instance of (x_{hk}, b^*), so

$$\begin{aligned}
\chi_{kl}^{2(hb^*)} &= \sum_{a=0, a\neq x_{hk}}^{s_k-1} \sum_{b=0, b\neq b^*}^{s_l-1} [n_{kl}^{(h)}(a,b) - N/(s_k s_l)]^2 \\
&\quad + [n_{kl}^{(h-1)}(x_{hk}, b^*) + 1 - N/(s_k s_l)]^2 \\
&= \chi_{kl}^{2(h-1)} + 2[n_{kl}^{(h-1)}(x_{hk}, b^*) - N/(s_k s_l)] + 1,
\end{aligned} \tag{3.8}$$

where $n_{kl}^{(h-1)}(a, b^*)$ is the number of occurrences of (a, b^*) in the previous $h-1$ rows.

When adding a symbol for the first row, any of the elements $0, \cdots, s_l - 1$ can be chosen. Choosing b^* from these elements and looking to the previous columns (the symbols in the previous columns are fixed), the pair $n_{kl}(a, b^*)$ will have a value of 1, since there is one occurence. All other combinations for elements from columns k and l is 0 for $k = 1, \ldots, l-1$. Then

$$\begin{aligned}
\chi_{kl}^{2(1)} &= \sum_{i=0}^{s_k-1} \sum_{j=0}^{s_l-1} [n_{kl}(i,j) - N/(s_k s_l)]^2 \\
&= [1 - N/(s_k s_l)]^2 + \sum_{(i,j)\neq(a,b^*)} [0 - N/(s_k s_l)]^2 \\
&= [1 - 2N/(s_k s_l) + [N/(s_k s_l)]^2] + (s_k s_l - 1)[N/(s_k s_l)]^2 \\
&= s_k s_l [N/(s_k s_l)]^2 - 2N/(s_k s_l) + 1.
\end{aligned}$$

Regardless of the choice for the first row, the criterion is the same, so it need only be calculated once at the onset of the algorithm.

To look at the χ^2 criterion after h rows, we simply need to add the criterion over the previous columns. That is,

$$\chi_l^{2(h)} = \sum_{k=1}^{l-1} \chi_{kl}^{2(hb^*)}.$$

If $\chi_l^{2(N)} = 0$, then column l is orthogonal to the previous l-1 columns.

3.2.1 The New Algorithm

Just as in Xu's algorithm, we still use a set number of restarts for attempting to find an orthogonal column if one cannot be found. Also, after a column has been found, and is not orthogonal, Xu's algorithm can be used if desired to try and find an orthogonal column. However, use of Xu's algorithm combined with the new one will add considerable time in computation.

The algorithm proceeds as follows:

1. Specify an initial design $\mathbf{d}$ with columns $(0, \cdots, 0, 1, \cdots, 1, \cdots, s_1 - 1, \cdots, s_1 - 1)$ and $(0, \cdots, s_2 - 1, 0, \cdots, s_2 - 1, \cdots, 0, \cdots, s_2 - 1)$. Let $l = 3$.
2. Randomize the rows of $\mathbf{d}$.
3. Let $x_l = 0$ and $h = 1$.
4. Let $d_{lb^*}^{(h)}$ be (3.7), the first h rows, where $x_{lb^*}^{(h)} = (x_{1l}, \cdots, x_{(h-1)l}, b^*)'$.
5. For $b^* = 0, \cdots, s_l - 1$, calculate $\chi_l^{2(hb^*)} = \sum_{k=1}^{l-1} \chi^{2(hb^*)}$ is calculated using (3.8) for k and l from $d_{lb^*}^{(h)}$. Use the best b^* such that $n_{kl}(a, b^*) \leq N/(s_k s_l)$ for $k = 1, \cdots, l-1$. If no such choice exists, take the best b^* with $n_{kl}(a, b^*) > N/(s_k s_l)$. In the case of equally good choices, take the largest or randomly choose between them.
6. Repeat Steps 4-5 for $h = 1, \cdots, N$.
7. If $\chi^2(\mathbf{d}) = 0$, add column $\mathbf{c}$ to the design, go to 9.
8. repeat 5-7 T times. Choose the column $\mathbf{c}$ which minimizes $\chi^2(\mathbf{d}_+)$.
9. Repeat Steps 2-8 for $l = 3, \cdots, n$.
10. Return the final $N \times n$ design $\mathbf{d}$.

3.2.2 Algorithm Using the J_2 Criterion

The new algorithm can be used with the J_2 criterion instead of the χ^2 criterion. To use the new algorithm, we now establish those quantities which will be needed.

Using J_2, we store the δ's from the design with $l-1$ columns, $\mathbf{d}_{l-1}$. For an added element,

$$\begin{aligned}\delta_{\alpha h}(\mathbf{d}_{lb^*}^{(h)}) &= \sum_{k=1}^{l} w_k \delta(x_{\alpha k}, x_{hk}) \\ &= \sum_{k=1}^{l-1} w_k \delta(x_{\alpha k}, x_{hk}) + w_l \delta(x_{\alpha l}, b^*) \\ &= \delta_{\alpha h}(\mathbf{d}_{l-1}) + w_l \delta(x_{\alpha l}, b^*).\end{aligned}$$

Then calculating the J_2 criterion at the current row is a matter of adding in the δ's for the previous $h-1$ rows, so

$$\begin{aligned}J_2(\mathbf{d}_l^{(hb^*)}) &= J_2(\mathbf{d}_l^{(h-1)}) + \sum_{\alpha=1}^{h-1} [\delta_{\alpha h}(\mathbf{d}_{l-1}) + w_l \delta(x_{\alpha l}, b^*)]^2 \\ &= J_2(\mathbf{d}_l^{(h-1)}) + \sum_{\alpha=1}^{h-1} [\delta_{\alpha h}(\mathbf{d}_{l-1})]^2 \\ &\quad + 2w_l \sum_{\alpha=1}^{h-1} \delta(x_{\alpha l}, b^*) \delta_{\alpha h}(\mathbf{d}_{l-1}) + w_l^2 \sum_{\alpha=1}^{h-1} \delta(x_{\alpha l}, b^*). \qquad (3.9)\end{aligned}$$

In addition, from equation (4) in Xu (2002),

$$\begin{aligned}J_2(\mathbf{d}_l^{(h-1)}) &= J_2(\mathbf{d}_{l-1}^{(h-1)}) + 2w_l \sum_{1 \le \alpha < \beta \le h-1} \delta(x_{\alpha l}, x_{\beta l}) \delta_{\alpha\beta}(\mathbf{d}_{l-1}^{(h-1)}) \\ &\quad + w_l^2 \sum_{1 \le \alpha < \beta \le h-1} \delta(x_{\alpha l}, x_{\beta l}). \qquad (3.10)\end{aligned}$$

Combining equations (3.9) and (3.10),

$$\begin{aligned}J_2(\mathbf{d}_l^{(hb^*)}) = &= J_2(\mathbf{d}_{l-1}^{(h-1)}) + 2w_l \sum_{1 \le \alpha < \beta \le h-1} \delta(x_{\alpha l}, x_{\beta l}) \delta_{\alpha\beta}(\mathbf{d}_{l-1}^{(h-1)}) \\ &\quad + w_l^2 \sum_{1 \le \alpha < \beta \le h-1} \delta(x_{\alpha l}, x_{\beta l}) + \sum_{\alpha=1}^{h-1} [\delta_{\alpha h}(\mathbf{d}_{l-1})]^2 \\ &\quad + 2w_l \sum_{\alpha=1}^{h-1} \delta(x_{\alpha l}, b^*) \delta_{\alpha h}(\mathbf{d}_{l-1}) + w_l^2 \sum_{\alpha=1}^{h-1} \delta(x_{\alpha l}, b^*).\end{aligned}$$

This decomposition implies a fast update, as only the last two terms of this expression depend on the element b^*. All the other terms can be stored from the previous rows and do not need to be recalculated. In addition, the $\delta(x_{\alpha l}, x_{\beta l})$ take on values of 0 or 1, so the last two terms can be calculated quickly.

The algorithm proceeds the same way as for χ^2, simply by using J_2 instead of χ^2 in step 5. However, in finding the best choice using J_2, if we wish to ensure $n_{kl}(a,b) \leq N/(s_k s_l)$ (which we need for an orthogonal array), the n_{kl}'s are additional information to be stored, as they are not used in the calculation of J_2. In addition, the equivalence of the criteria is assuming a balanced column. The impact of this will be discussed in the next subsection. As the algorithm is based on a subset of rows in the design, the criteria may not be equivalent at a certain row, and the best choice may be different.

3.2.3 Comments

From equation (2.10), the columns of the array must be balanced (each element appears the same number of times in the column) for the equivalence of J_2 and χ^2 . This balance is more important than it might initially seem, as can be demonstrated by the following example.

Example 3.1 Consider the designs $\mathbf{D_1}$ and $\mathbf{D_2}$:

$$\mathbf{D_1} = \begin{pmatrix} +1 & +1 \\ +1 & -1 \\ -1 & +1 \\ -1 & -1 \\ +1 & +1 \\ -1 & -1 \end{pmatrix}, \mathbf{D_2} = \begin{pmatrix} +1 & +1 \\ +1 & -1 \\ -1 & +1 \\ -1 & -1 \\ +1 & +1 \\ +1 & -1 \end{pmatrix}.$$

The design $\mathbf{D_1}$ is balanced, but in design $\mathbf{D_2}$ the first column is not balanced. Calculating χ^2 and J_2 for both designs gives $\chi^2(\mathbf{D_1}) = \chi^2(\mathbf{D_2}) = 2/3$, whereas $J_2(\mathbf{D_1}) = 16$ while $J_2(\mathbf{D_2}) = 17$. Using the χ^2 criterion as setup in algorithm (which assumes balance) shows no difference between the two designs, while under the J_2 criterion, $\mathbf{D_1}$ is preferred. In fact, if we calculate χ^2 by definition (2.5), $\chi^2(\mathbf{D_1}) = 2/3$, while $\chi^2(\mathbf{D_2}) = 0$, as the cross-product of the two columns is 0.

The preceding example illustrates that some care must be taken when considering the equivalence of the different criteria. In particular, for columns k and l, if N/s_k is integer (and/or N/s_l) where it is possible to have a balanced column, if $N/s_k s_l$ is not integer, balance may need to be forced in constructing the column row by row.

In light of this, it is useful to introduce the notion of *weak strength* (Xu (2003)). A design is said to be of weak strength t, denoted by t^-, if all level combinations appear as equally often as possible. In other words, the difference in the count of possible level combinations for any given t columns does not exceed one. If a design is of strength t, all possible level combinations occur the same number of times, so it has weak strength t^-. On the other hand, a design being weak strength t^- does not imply weak strength $(t-1)^-$, as can be seen in example 3.1.

In the case of orthogonal arrays, from the properties of strength, strength t ensures not only all subsets of t columns are balanced, but also all smaller subset of $t^* = 1, 2, \cdots, t-1$. As noted above, this does not hold true for nearly-orthogonal arrays. Simply trying to make the $n'_{ij}s$ as close to each other as possible for a given i and j is not enough to ensure column balance throughout the design. The reason for this can be more clearly seen by the decomposition of J_2 given by:

$$2J_2(\mathbf{d}) = \sum_{k=1}^{n}\left[\sum_{a=0}^{s_k-1} n_{kk}(a,a)^2\right] + \sum_{1\le k\ne l\le n}\left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1} n_{kl}(a,b)^2\right] - Nn^2.$$

The first part of this decomposition of J_2 corresponds to the balance within each single column and is minimized if the array is of weak strength 1^-, while the second part corresponds to column pairs, and is minimized if the array is of strength 2^-. The χ^2 criterion is only based on weak strength 2^-.

Since χ^2 and J_2 are based on the occurrences of the n_{kl}'s, there are instances where we can construct an optimal design using a pre-existing orthogonal array. For instance, if we have an $OA(N, s_1 \cdots s_m, 2)$, then removing any row will still have all the n_{kl}'s within 1 (since they all occur the same number of times in columns k and l by the definition of orthogonal array), so by Lemma 2.1 the design is an A_2 optimal $NOA(N-1, s_1 \cdots s_m)$.

We can also add rows to an existing orthogonal array to make it A_2 optimal. If we have an existing $OA(N, s_1 \cdots s_m, 2)$, adding the row $(0, \ldots, 0)$ will still result in an A_2 optimal $NOA(N+1, s_1 \cdots s_m)$ by Lemma 2.1. Furthermore, adding the row $(1, \ldots, 1)$ is also an A_2 optimal $NOA(N+2, s_1 \cdots s_m)$. In fact, we can continue this process by adding up to

$min(s_i)$ rows in this manner.

3.3 Extension to Higher Strength

Even if an orthogonal array of strength 2 can be found, if a strength 3 or higher orthogonal array can be obtained, it would be preferred. This section will look at extending the J_2 and χ^2 criteria and algorithms to higher strength.

3.3.1 The J_3 Criterion

The J_2 criterion can be extended to J_3 and later to arbitrary J_k for strength 3 and higher.

For an $N \times n$ matrix $d = [x_{ij}]$, with weight w_k for column k having s_k levels, define $\delta_{i,j}(\mathbf{d})$ as (2.7). Define

$$J_3(\mathbf{d}) = \sum_{1 \le i < j \le N} [\delta_{i,j}(d)]^3, \tag{3.11}$$

which is similar to J_2, but the power is replaced by 3. By defining J_3 as (3.11), J_3 can be used as a measure of closeness to strength 3 as can be seen by the following lemma.

Lemma 3.1 *For an $N \times n$ matrix $d = [x_{ij}]$, with weight w_k for column k having s_k levels,*

$$\begin{aligned} 2J_3(d) &\ge L_3(n) \\ &= \sum_{k=1}^{n} w_k^3 N^2/s_k + 3\sum_{k \ne l} w_k^2 w_l [N^2/(s_k s_l)] \\ &\quad + \sum_{k \ne l \ne m} w_k w_l w_m [N^2/(s_k s_l s_m)] - N\left(\sum_{k=1}^{n} w_k\right)^3, \end{aligned}$$

where equality holds iff $\mathbf{d}$ *is an orthogonal array of strength 3.*

Proof of Lemma 3.1. Define $n_{klm}(a,b,c) = |i : x_{ik} = a, x_{il} = b, x_{im} = c|$, the number of times the levels a, b, and c appear in columns k, l, and m, respectively. Note that

$$\sum_{i=1}^{N}\sum_{j=1}^{N} \delta(x_{ik}, x_{jk})\delta(x_{il}, x_{jl})\delta(x_{im}, x_{jm}) = \sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1}\sum_{c=0}^{s_m-1} n_{klm}(a,b,c)^2.$$

This holds since the left hand side is over all pairs of rows, and takes a value of 1 for each of the $n_{klm}(a,b,c)$'s and the other $n_{klm}(a,b,c)$ rows it occurs with.

Note that the right hand side of the previous formula can be partitioned into three

sums based on the number of distinct columns are being looked at: one, two, or three. In addition, the order in which the columns are considered makes no difference on the count (ie. $n_{klm}(a,b,c) = n_{mkl}(c,a,b)$).

To establish the inequality,

$$
\begin{aligned}
2J_3(d) &= 2\sum_{i<j}[\delta_{i,j}(d)]^3 \\
&= 2\sum_{i<j}\left[\sum_{k=1}^{n} w_k\delta(x_{ik},x_{jk})\right]^3 \\
&= \sum_{i=1}^{N}\sum_{j=1}^{N}\left[\sum_{k=1}^{n} w_k\delta(x_{ik},x_{jk})\right]^3 - N\left(\sum_{k=1}^{n} w_k\right)^3 \\
&= \sum_{i=1}^{N}\sum_{j=1}^{N}\left[\sum_{k=1}^{n} w_k\delta(x_{ik},x_{jk})\right]\times\left[\sum_{l=1}^{n} w_l\delta(x_{il},x_{jl})\right] \\
&\quad \times\left[\sum_{m=1}^{n} w_m\delta(x_{im},x_{jm})\right] - N\left(\sum_{k=1}^{n} w_k\right)^3 \\
&= \sum_{i=1}^{N}\sum_{j=1}^{N}\left[\sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} w_k\delta(x_{ik},x_{jk})w_l\delta(x_{il},x_{jl})w_m\delta(x_{im},x_{jm})\right] \\
&\quad -N\left(\sum_{k=1}^{n} w_k\right)^3 \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} w_k w_l w_m\left[\sum_{i=1}^{N}\sum_{j=1}^{N}\delta(x_{ik},x_{jk})\delta(x_{il},x_{jl})\delta(x_{im},x_{jm})\right] \\
&\quad -N\left(\sum_{k=1}^{n} w_k\right)^3 \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} w_k w_l w_m\left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1}\sum_{c=0}^{s_m-1} n_{klm}(a,b,c)^2\right] - N\left(\sum_{k=1}^{n} w_k\right)^3 \\
&= \sum_{k=1}^{n} w_k^3\left[\sum_{a=0}^{s_k-1} n_{kkk}(a,a,a)^2\right] + 3\sum_{k\neq l} w_k^2 w_l\left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1} n_{kkl}(a,a,b)^2\right] \\
&\quad +\sum_{k\neq l\neq m} w_k w_l w_m\left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_k-1}\sum_{c=0}^{s_k-1} n_{klm}(a,b,c)^2\right] - N\left(\sum_{k=1}^{n} w_k\right)^3
\end{aligned}
$$

$$\geq \sum_{k=1}^{n} w_k^3 N^2/s_k + 3\sum_{k\neq l} w_k^2 w_l [N^2/(s_k s_l)] + \sum_{k\neq l\neq m} w_k w_l w_m [N^2/(s_k s_l s_m)] - N\left(\sum_{k=1}^{n} w_k\right)^3,$$

where equality holds when the design is an orthogonal array of strength 3.

For the inequality, the Cauchy-Schwartz inequality is used. The Cauchy-Schwartz inequality states that for $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n \in \Re$, we have:

$$(x_1^2 + x_2^2 + \cdots + x_n^2)(y_1^2 + y_2^2 + \cdots + y_n^2) \geq (x_1 y_1 + x_2 y_2 + \cdots + x_n y_n)^2,$$

where equality holds iff $x_1/y_1 = x_2/y_2 = \cdots = x_n/y_n$.

For an $N \times n$ design matrix $\mathbf{d}$, note that for fixed columns k, l and m with s_k, s_l, and s_m levels respectively, we have

$$\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1}\sum_{c=0}^{s_m-1} n_{klm}(a,b,c) = N,$$

$$\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1} n_{kkl}(a,a,b) = N,$$

$$\sum_{a=0}^{s_k-1} n_{kkk}(a,a,a) = N.$$

The number of summations reduces when the subscripts reference multiple columns which are actually the same column. By the definition of $n_{klm}(a,b,c)$, the columns which are referenced that are the same can only be counted if they are also at the same level (it is impossible for a row to have one factor at two different levels).

Looking at distinct columns k, l, and m, there are $s_k s_l s_m$ possible combinations for the levels of these factors. By Cauchy-Schwartz

$$\left(\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1}\sum_{c=0}^{s_m-1} n_{klm}(a,b,c)^2\right)(1^2 + 1^2 + \cdots + 1^2)/s_k s_l s_m$$

$$\geq \left(\sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} \sum_{c=0}^{s_m-1} n_{klm}(a,b,c) \right)^2 / s_k s_l s_m$$

$$= N^2 / s_k s_l s_m,$$

where equality holds iff all $n_{klm}(a,b,c)$ are the same for all possible values of a, b, and c. This implies that for these columns, every triplet appears the same number of times. If this holds for every k, l, and m, the design is an orthogonal array of strength 3.

Similarly, if we consider having two distinct columns, k and m, with $s_k s_l$ different combinations, by Cauchy-Schwartz,

$$\left(\sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} n_{kkl}(a,a,b)^2 \right) (1^2 + 1^2 + \cdots + 1^2)/s_k s_l$$

$$\geq \left(\sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} n_{kkl}(a,a,b) \right)^2 / s_k s_l$$

$$= N^2 / s_k s_l,$$

where equality holds iff all $n_{kkl}(a,a,b)$ are the same for all possible values of a and b. This implies that for columns k and l every pair appears the same number of times.

Finally, for a single distinct column,

$$\left(\sum_{a=0}^{s_k-1} n_{kkk}(a,a,a)^2 \right) (1^2 + 1^2 + \cdots + 1^2)/s_k$$

$$\geq \left(\sum_{a=0}^{s_k-1} n_{kkk}(a,a,a) \right)^2 / s_k$$

$$= N^2 / s_k,$$

where equality holds iff all $n_{kkk}(a,a,a)$ are the same for all possible values of a. This implies that column k is balanced, with each level occurring the same number of times.

As mentioned, if for every k, l, and m, each triplet occurs the same number of times, the design is an orthogonal array of strength 3. If it is an orthogonal array of strength 3, then for any pair of columns, all possible pairs of levels occur the same number of times, and the

design is balanced. Then the equality for $2J_3(\mathbf{d})$ holds iff the design is an orthogonal array of strength 3.

Example 3.2 Consider the following design matrix

$$\mathbf{d} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

For this matrix, the first 3 columns form an $OA(8, 2^3, 3)$. Checking the inequality for these first 3 columns, $2J_3(\mathbf{d}_{(\mathbf{1-3})}) = L_3(3) = 216$. The entire matrix is not an orthogonal array of strength 3, and checking the inequality, $2J_3(\mathbf{d}) = 432$ while $L_3(4) = 384$.

With the extension of J_2, the hope is to extend Xu's algorithm to the criterion for higher strength. In order to do so, we need to extend the calculations for column addition and symbol swapping. We start with column addition.

Let column $\mathbf{c}$ with weight w_c and s_c levels be added to the original design $\mathbf{d}$, resulting in $\mathbf{d}_+$. The new design is such that

$$\delta_{i,j}(\mathbf{d}_+) = \delta_{i,j}(\mathbf{d}) + w_k \delta_{i,j}(\mathbf{c}).$$

Updating J_3,

$$\begin{aligned} J_3(\mathbf{d}_+) &= \sum_{i<j}[\delta_{i,j}(\mathbf{d}_+)]^3 \\ &= \sum_{i<j}[\delta_{i,j}(\mathbf{d}) + w_k\delta_{i,j}(\mathbf{c})]^3 \\ &= \sum_{i<j}[\delta_{i,j}(\mathbf{d})]^3 + 3w_k\sum_{i<j}[\delta_{i,j}(\mathbf{d})^2\delta_{i,j}(\mathbf{c})] \\ &\quad +3w_k^2\sum_{i<j}[\delta_{i,j}(\mathbf{d})\delta_{i,j}(\mathbf{c})^2] + w_k^3\sum_{i<j}[\delta_{i,j}(\mathbf{c})]^3 \\ &= J_3(\mathbf{d}) + 3w_k\sum_{i<j}[\delta_{i,j}(\mathbf{d})^2\delta_{i,j}(\mathbf{c})] \\ &\quad +3w_k^2\sum_{i<j}[\delta_{i,j}(\mathbf{d})\delta_{i,j}(\mathbf{c})] + w_k^3\sum_{i<j}[\delta_{i,j}(\mathbf{c})], \end{aligned}$$

where the simplification of the formula is due to $\delta_{i,j}(\mathbf{c})$ only taking on values of 0 or 1. In addition, when weights are all assigned a value of 1, if $\delta_{i,j}(\mathbf{d})^2$ is stored, then this updating involves no multiplication.

Another aspect of Xu's algorithm involves switching symbols. Consider switching distinct symbols in rows a and b of the newly added column. Then for $j \neq a, b$, the values $\delta_{a,j}(\mathbf{c}) = \delta_{j,a}(\mathbf{c})$ and $\delta_{b,j}(\mathbf{c}) = \delta_{j,b}(\mathbf{c})$ are switched. The switch does not effect $\delta_{a,b}(\mathbf{c})$, as the values are distinct, so this value is 0.

Looking at the difference in $J_3(d_+)$, first consider fixing a certain row $j \neq a, b$. In regards to row j, the change on J_3 of the switch are based on rows a and b, all other rows have been unchanged. In the old calculation for J_3, the terms of interest in calculation are

$$\delta_{a,j}(\mathbf{d})^2\delta_{a,j}(\mathbf{c}) + \delta_{b,j}(\mathbf{d})^2\delta_{b,j}(\mathbf{c})$$

and

$$\delta_{a,j}(\mathbf{d})\delta_{a,j}(\mathbf{c}) + \delta_{b,j}(\mathbf{d})\delta_{b,j}(\mathbf{c}).$$

In the calculation of J_3 after the switch, these terms will now be

$$\delta_{a,j}(\mathbf{d})^2\delta_{b,j}(\mathbf{c}) + \delta_{b,j}(\mathbf{d})^2\delta_{a,j}(\mathbf{c})$$

and

$$\delta_{a,j}(\mathbf{d})\delta_{b,j}(\mathbf{c}) + \delta_{b,j}(\mathbf{d})\delta_{a,j}(\mathbf{c}).$$

Looking at the difference the swap has made,

$$\begin{aligned} & (\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})) \times \delta_{a,j}(\mathbf{d})^2 + (\delta_{b,j}(\mathbf{c}) - \delta_{a,j}(\mathbf{c})) \times \delta_{b,j}(\mathbf{d})^2 \\ = & \quad (\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})) \times (\delta_{a,j}(\mathbf{d})^2 - \delta_{b,j}(\mathbf{d})^2) \end{aligned}$$

and

$$\begin{aligned} & (\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})) \times \delta_{a,j}(\mathbf{d}) + (\delta_{b,j}(\mathbf{c}) - \delta_{a,j}(\mathbf{c})) \times \delta_{b,j}(\mathbf{d}) \\ = & \quad (\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})) \times (\delta_{a,j}(\mathbf{d}) - \delta_{b,j}(\mathbf{d})). \end{aligned}$$

For the overall effect on $J_3(\mathbf{d}_+)$, the symbol swapping will reduce $J_3(\mathbf{d}_+)$ by $\Delta(a,b)$, where

$$\begin{aligned} \Delta(a,b) &= 3w_k \sum_{j \neq a,b} [\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})][\delta_{a,j}(\mathbf{d})^2 - \delta_{b,j}(\mathbf{d})^2] \\ &\quad +3w_k^2 \sum_{j \neq a,b} [\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})][\delta_{a,j}(\mathbf{d}) - \delta_{b,j}(\mathbf{d})] \\ &= 3w_k \sum_{j \neq a,b} [\delta_{a,j}(\mathbf{c}) - \delta_{b,j}(\mathbf{c})] \times [\delta_{a,j}(\mathbf{d})^2 - \delta_{b,j}(\mathbf{d})^2 + \delta_{a,j}(\mathbf{d}) - \delta_{b,j}(\mathbf{d})]. \end{aligned} \tag{3.12}$$

We note that $\Delta(a,b)$ requires no multiplication, as $\delta_{a,j}(\mathbf{c})$ and $\delta_{b,j}(\mathbf{c})$ are either 0 or 1 and $\delta_{a,j}(\mathbf{d})^2$ and $\delta_{b,j}(\mathbf{d})^2$ can be stored for faster calculation.

3.3.2 An Algorithm

Following the same idea as Xu's algorithm, a new algorithm is proposed. The new algorithm adds columns to an existing design to find an OA of strength 3, or an NOA_3 (nearly-orthogonal array optimized using the J_3 criteria). The algorithm will repeat T times if no lower bound is achieved. Depending on the nature of the problem, this T can be adjusted depending on the orthogonality of the existing design. The functions used will be swap, involving the interchange of symbols, and exchange, involving changing a candidate column with a new one.

The algorithm:

1. Compute the lower bound $L_3(k)$ for $k = 2, \cdots, n$.

2. Setup the first 2 columns, specified as $(0, \cdots, 0, 1, \cdots, 1, \cdots, s_1 - 1, \cdots, s_1 - 1)$ for column 1, and $(0, \cdots, 0, 1, \cdots, 1, \cdots, s_2 - 1, \cdots, s_2 - 1)$ for column 2. Compute $\delta_{i,j}(\mathbf{d})$ and $J_3(\mathbf{d})$.

3. For $k = 3, \cdots, n$:

   i. Generate a random balanced s_k level column $\mathbf{c}$. Compute $J_3(\mathbf{d}_+)$ and test against $L_3(k)$, goto iv) if equality holds

   ii. For all rows with different symbols, calculate $\Delta(a, b)$ according to (3.12). Choose the a and b for which $\Delta(a, b)$ has the greatest value. Swap these 2 symbols, and reduce $J_3(d_+)$ by $\Delta(a, b)$. If $J_3(d_+) = L_3(k)$, goto (iv). Otherwise, repeat step (ii) until no improvement can be made.

   iii. Repeat i) and ii) T times, choosing column $\mathbf{c}$ which results in the smallest value for $J_3(d_+)$.

   iv. Add column $\mathbf{c}$ to the design, and update values of $J_3(\mathbf{d})$, $\delta_{i,j}(\mathbf{d})$ and $\delta_{i,j}(\mathbf{d})^2$ with $J_3(\mathbf{d}_+)$, $\delta_{i,j}(\mathbf{d}_+)$ and $\delta_{i,j}(\mathbf{d}_+)^2$, respectively.

4. Return the $N \times n$ design $\mathbf{d}$.

If the algorithm is to result in an OA of strength 3, then in step 3.ii), the equality must hold. The choice for T makes a bigger impact on the speed of this algorithm when using J_3. In finding an orthogonal column that maintains strength 3, the potential columns the algorithm can find is generally much smaller than that of strength 2. A large value for T will enable more time to find an orthogonal column, or at least a nearly-orthogonal column, but will also take much time. A small value will take less computation, but may result in a poor choice for the added column.

3.3.3 Comments on J_3

A useful expansion of J_3 is

$$\begin{aligned}
J_3(d) &= \sum_{i<j}[\delta_{i,j}(d)]^3 \\
&= \sum_{i<j}\left[\sum_{k=1}^{n} w_k\delta(x_{ik},x_{jk})\right]^3 \\
&= \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\left[\sum_{k=1}^{n} w_k\delta(x_{ik},x_{jk})\right]^3 \\
&= \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\left[\sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} w_k\delta(x_{ik},x_{jk})w_l\delta(x_{il},x_{jl})w_m\delta(x_{im},x_{jm})\right] \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} w_kw_lw_m\left[\sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\delta(x_{ik},x_{jk})\delta(x_{il},x_{jl})\delta(x_{im},x_{jm})\right] \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} w_kw_lw_m\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1}\sum_{c=0}^{s_m-1}\binom{n_{klm}(a,b,c)}{2},
\end{aligned}$$

as we look at the occurrence of (a,b,c) over all possible pairs of rows.

Using this derivation of J_3, if an NOA of $N \times n$ is desired (that is, we know an OA does not exist), if we start with an $N^* \times n$ design that is an orthogonal array of strength 3, where $N^* \leq N$, then any added row will have the same effect on J_3 in regards to the $N^* \times n$ design. Hence, to find a design with reasonable J_3, we can look at minimizing the $(N - N^*) \times n$ matrix for the remaining rows according to the J_3 criterion. This holds as well for the J_2 criterion. By trying to find an optimal design over less rows, the algorithm(s) can run faster and hopefully more efficiently.

3.3.4 Extension to Higher Strength

J_3 can be extended to look for higher strength. We consider the case where all weights have a value of 1, but the incorporation of weights is natural. In general, for strength S, define

$$J_S(d) = \sum_{1\leq i<j\leq N}[\delta_{i,j}(d)]^S$$

The main reason we can make this generalization is that the definition is still over pairs of rows. In particular, consider

$$n_{k_1 \cdots k_S}(a_1, \cdots, a_S) = |i : x_{ik_1} = a_1, \cdots, x_{ik_S} = a_S|,$$

the number of rows that the combination of levels $a_1, a_2, \cdots, a_S$ appears in columns $k_1, \cdots, k_S$ respectively. Then

$$\sum_{i=1}^{N}\sum_{j=1}^{N} \delta(x_{ik_1}, x_{jk_1}) \cdots \delta(x_{ik_S}, x_{jk_S}) = \sum_{a_1=0}^{s_{k_1}-1} \cdots \sum_{a_S=0}^{s_{k_S}-1} n_{k_1 \cdots k_S}(a_1, \cdots, a_S)^2,$$

since the left hand side is over all pairs of rows, and each of the $n_{k_1 \cdots k_S}(a_1, \cdots, a_S)$'s will be counted with the $n_{k_1 \cdots k_S}(a_1, \cdots, a_S)$ rows it occurs with.

The Cauchy-Schwartz inequality will still be applied in a similar manner as was done for J_3, where

$$\sum_{k_1=1}^{n} \cdots \sum_{k_S=1}^{n} \left[\sum_{a_1=0}^{s_{k_1}-1} \cdots \sum_{a_S=0}^{s_{k_S}-1} n_{k_1 \cdots k_S}(a_1, \cdots, a_S)^2 \right] - N \left(\sum_{k=1}^{n} 1 \right)^S$$

can be partitioned into subsets such that the number of distinct columns being considered are $1, \cdots, S$. The added calculation to column addition and symbol switching should be proportional to the number of these partitions, just as we saw in the case of J_3. Likewise, Xu's algorithm can still be used for the desired J_i.

3.3.5 The χ^2 Criterion for Higher Strength

Instead of using the χ^2 criterion to measure the similarity of two columns at a time, a modification can be made to consider three or more columns at a time using the new algorithm. We first look at extension to three columns and those quantities which will be needed for the algorithm. Our notation remains the same, with the exception that quantities involving two columns now involve three columns. This can be easily extended to more than three columns for higher strength.

Let $n_{klm}(a, b, c)$ be the number of rows in which symbols a, b, and c appear in column k, l, and m respectively. Define

$$\chi^2_{klm} = \sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} \sum_{c=0}^{s_m-1} \frac{[n_{klm}(a, b, c) - N/(s_k s_l s_m)]^2}{N/(s_k s_l s_m)},$$

which has a value of 0 iff all $n_{klm}(a,b,c) = N/(s_k s_l s_m)$. Then we can use $\chi^2(\mathbf{d}) = \sum_{k<l<m} \chi^2_{klm}(\mathbf{d})$ as a measure of near-strength 3, which takes on a value of 0 if $\chi^2_{klm} = 0$ for all k, l, and m, which implies a strength 3 orthogonal array. In order to simplify calculations, we use the simpler

$$\chi^2_{klm} = \sum_{a=0}^{s_k-1} \sum_{b=0}^{s_l-1} \sum_{c=0}^{s_m-1} [n_{klm}(a,b,c) - N/(s_k s_l s_m)]^2.$$

A fast update over three columns is done in the same manner as was done over two columns. If we are trying to add column m,

$$\begin{aligned} \chi^{2(hc^*)}_{klm} &= \sum_{a=0,a\neq x_{hk}}^{s_k-1} \sum_{b=0,b\neq x_{hl}}^{s_l-1} \sum_{c=0,c\neq c^*}^{s_m-1} [n^{(h)}_{klm}(a,b,c) - N/(s_k s_k s_m)]^2 \\ &\quad + [n^{(h-1)}_{klm}(x_{hk}, x_{hl}, c^*) + 1 - N/(s_k s_l s_m)]^2 \\ &= \chi^{2(h-1)}_{klm} + 2[n^{(h-1)}_{klm}(x_{hk}, x_{hl}, c^*) - N/(s_k s_l s_m)] + 1. \end{aligned} \tag{3.13}$$

When considering the first row, any of the symbols $0, 1, \ldots, s_m - 1$ can be used, and

$$\chi^{2(1)}_{klm} = s_k s_l s_m [N/(s_k s_l s_m)]^2 - 2N/(s_a s_b s_l) + 1.$$

To measure the near-strength 3 orthogonality of the new column compared to all previous columns, we use

$$\chi^{2(hc^*)}_{m} = \sum_{1\leq k<l\leq m-1} \chi^{2(hc^*)}_{klm}. \tag{3.14}$$

For a strength S orthogonal array, we simply need to use the χ^2 criterion over S columns. For the columns $k_1, k_2, \ldots, k_S$, with $s_1, s_2, \ldots, s_S$ symbols respectively, let $n_{k_1 k_2 \cdots k_S}(a_1, a_2, \ldots, a_S)$ be the number of rows in which the combination of $(a_1, a_2, \ldots, a_S)$ appears in columns $k_1, k_2, \ldots, k_S$. Then

$$\chi^2_{k_1 k_2 \cdots k_S} = \sum_{a_1=0}^{s_1-1} \sum_{a_2=0}^{s_2-1} \cdots \sum_{a_S=0}^{s_S-1} [n_{k_1 k_2 \cdots k_S}(a_1, a_2, \ldots, a_S) - N/(s_1 s_2 \cdots s_S)]^2$$

can be used, and quantities can be established as was done for strength 3.

3.3.6 The Algorithm

The algorithm proceeds in the same way as for 2 columns. We will demonstrate how the algorithm proceeds for three columns, but can easily be extended to higher strength:

1. Specify an initial design $\mathbf{d}$ with columns $(0, \cdots, 0, 1, \cdots, 1, \cdots, s_1 - 1, \cdots, s_1 - 1)$ and $(0, \cdots, s_2 - 1, 0, \cdots, s_2 - 1, \cdots, 0, \cdots, s_2 - 1)$. Let $m = 3$.

2. Randomize the rows of $\mathbf{d}$.

3. Let $x_m = 0$ and $h = 1$.

4. Let $d_{mc^*}^{(h)} = [x_1^{(h)}|x_2^{(h)}|\cdots|x_{mc^*}^{(h)}]$, the first h rows, where $x_{mc^*}^{(h)} = (x_{1l}, \cdots, x_{(h-1)m}, c^*)'$.

5. For $c^* = 0, \cdots, s_m - 1$, calculate $\chi_m^{2(hc^*)} = \sum_{k=1}^{m-1} \chi^{2(hc^*)}$ is (3.6) calculated using columns k, l and m from $d_{mc^*}^{(h)}$. Use the best c^* such that $n_{klm}(a, b, c^*) \leq N/(s_k s_l s_m)$ for $1 \leq k < l \leq m - 1$. If no such choice exists, take the best c^* with $n_{klm}(a, b, c^*) > N/(s_k s_l s_m)$. In the case of equally good choices, take the largest or randomly choose between them.

6. Repeat Steps 4-5 for $h = 1, \cdots, N$.

7. If $\chi^2(\mathbf{d}) = 0$, go to 8.

8. repeat 5-7. T times. Choose the column $\mathbf{c}$ which minimizes $\chi^2(\mathbf{d}_+)$.

9. Repeat Steps 2-8 for $m = 3, \cdots, n$.

10. Return the final $N \times n$ design $\mathbf{d}$.

3.3.7 Comments

As in the case with the χ^2 criterion for strength 2, some care must be taken for the balance of a design. For strength 3, if the n_{klm}'s are all equal, then the n_{kl}'s must be all equal as well. However, if equality cannot be achieved, minimizing χ^2 may not necessarily result in a design which has nice properties in terms of the n_{kl}'s.

Example 3.3 Consider the designs $\mathbf{D_1}$ and $\mathbf{D_2}$

$$\mathbf{D_1} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & +1 \\ -1 & +1 & -1 \\ -1 & +1 & +1 \\ +1 & -1 & -1 \\ +1 & -1 & +1 \\ +1 & +1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & +1 \\ -1 & -1 & -1 \\ +1 & +1 & -1 \\ +1 & +1 & +1 \end{pmatrix}, \mathbf{D_2} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & +1 \\ -1 & +1 & -1 \\ -1 & +1 & +1 \\ +1 & -1 & -1 \\ +1 & -1 & +1 \\ +1 & +1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & -1 \\ -1 & +1 & +1 \\ +1 & -1 & +1 \\ +1 & +1 & -1 \end{pmatrix}.$$

Calculating χ^2 and J_2 for both designs gives $\chi^2(\mathbf{D_1}) = \chi^2(\mathbf{D_2}) = 2$, whereas $J_3(\mathbf{D_1}) = 342$ while $J_3(\mathbf{D_2}) = 330$. In this example, the J_3 criterion ranks D_2 as a better design. The columns are balanced, and we have a design of weak strength 3^- for both designs, but D_2 is weak strength 2^-, while D_1 is not.

Example 3.3 enforces one of the issues that can arise using the new algorithm for finding nearly-orthogonal arrays. In order to use the χ^2 criterion when we can not assume that all n_{klm}'s can be balanced, we should also check for balance or near balance among the n_{kl}'s and balance within each column. This check for balance is already considered in the J-criterion. If we are looking for a near-strength 3 (or higher) orthogonal array, the J-criterion will try to keep balance of lower levels of near-strength as well. If instead one uses the χ^2 criterion, extra time must be spent to ensure balance on smaller dimensions.

3.4 Designs with a Larger Number of Runs

There are some limitations with the algorithms presented as the run size increases. For Xu's algorithm, having to check all possible switches can be too time-consuming due to having to consider too many switches. In the new algorithm, too many runs means that there is much more unknown at a certain row, and makes it harder to find a good column.

3.4.1 A modification on Xu's Algorithm

The idea behind our modification of Xu's algorithm is that if an orthogonal array exists, each element of the added column must occur the same number of times with each element of the previous columns. We can choose the new column so that it is at least orthogonal to the first column. In addition, if we restrict switches such that we will not lose the orthogonality to the first column, the number of switches to check is greatly reduced.

The algorithm proceeds as follows:

1. For $k = 1, \cdots, n$, compute the lower bound $L(k)$ by equation (2.1).

2. Specify an initial design $\mathbf{d}$ with columns $(0, \ldots, 0, 1, \ldots, 1, \ldots, s_1 - 1, \ldots, s_1 - 1)' = (x_{(0^*)1}, x_{(1^*)1}, \ldots, x_{(s_1-1^*)1})$, where $x_{(i^*)1}$ denotes the rows in column 1 having element i^*, and $(0, \cdots, s_2 - 1, 0, \cdots, s_2 - 1, \cdots, 0, \cdots, s_2 - 1)'$, and compute $\delta_{i,j}(\mathbf{d})$ and $J_2(\mathbf{d})$ by definition. If $J_2(\mathbf{d}) = L(2)$, then $n_0 = 2$ and $T = T_1$; otherwise, $n_0 = 0$ and $T = T_2$.

3. For $k = 3, \cdots, n$, do the following:

   (a) Generate a random balanced s_k-level column $\mathbf{c}$ as follows: create s_1 random balanced columns of size $N^* = N/s_1$, call these $\mathbf{c}_i$, for $i = 1, \ldots, s_1$. Match these $\mathbf{c}_i$ to $x_{(i^*)1}$. The column $\mathbf{c}$ is orthogonal to column 1. Compute $J_2(\mathbf{d}_+)$. If $J_2(\mathbf{d}_+) = L(k)$, go to (d).

   (b) For all pairs of rows a and b in $x_{(i^*)1}$ with distinct symbols in $\mathbf{c}$ for $i = 1, \ldots, s_1$, compute $\Delta(a, b)$. This forces orthogonality with column 1. Choose a pair of rows with the largest $\Delta(a, b)$ and exchange the symbols in rows a and b of column $\mathbf{c}$. Reduce $J_2(\mathbf{d}_+)$ by $\Delta(a, b)$. If $J_2(\mathbf{d}_+) = L(k)$, go to (d); otherwise repeat (b) until no further improvement is made.

   (c) Repeat (a) and (b) T times and choose column $\mathbf{c}$ that produces the smallest $J_2(\mathbf{d}_+)$.

   (d) Add column $\mathbf{c}$ as the kth column of $\mathbf{d}$, let $J_2(\mathbf{d}) = J_2(\mathbf{d}_+)$, and update $\delta_{i,j}(\mathbf{d})$.

4. Return the final $N \times n$ design $\mathbf{d}$.

The main advantage to this algorithm is the reduction of the number of switches to consider. While this may seem to be too restrictive on where switches can be made, it should be kept in mind that in an optimal design, orthogonality would be kept with this

first column for an orthogonal array. That is, if we were to switch distinct elements from $\mathbf{c}$ in $x_{(i^*)1}$ and $x_{(j^*)1}$, losing orthogonality with column 1, switches must be made that would eventually make the column orthogonal once again. In addition, column 1 can be randomly chosen among any of the existing columns, and more restarts can be used as less time needs to be spent considering all possible switches.

Even if we are dealing with nearly-orthogonal arrays, we can likewise ensure that the added column has weak strength 2 in regards to the first column. In particular, if the first column has a large number of levels, this seems reasonable if using natural weights.

3.4.2 Discussion of Weights

The weights for the J_2 criterion are chosen based on the purpose of the design. If we want an orthogonal array, the lower bound can be reached, so we can use weights of 1 for all columns to simplify calculations. Recalling the relationship between J_2 and A_2 when using natural weights (weights of a column equal to the number of levels), for nearly-orthogonal arrays, using natural weights will try to create an array optimal for the A_2 criterion. For factors which are deemed to be of higher importance, a larger weight can be assigned to them. If a factor has a higher weight, it is more likely that other factors will be chosen to be orthogonal to the factor with higher weight.

The use of weights is directly related to the χ^2 criteria. Using weights of 1 for J_2 is equivalent to using the simplified equation (3.6) for the χ^2 criteria. The use of natural weights is equivalent to using equation (2.6), which will be used for nearly-orthogonal arrays to optimize A_2. Setting the weights to 1 to speed up the algorithm may not be advisable with nearly-orthogonal arrays, as can be seen by example 3.4.

Example 3.4 Let $\mathbf{D_1}$ and $\mathbf{D_2}$ be $NOA(12, 6^1 3^3)$, with

$$\mathbf{D_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 \\ 3 & 0 & 1 & 0 \\ 3 & 1 & 0 & 1 \\ 4 & 0 & 1 & 1 \\ 4 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 5 & 1 & 1 & 1 \\ 6 & 0 & 0 & 1 \\ 6 & 1 & 1 & 0 \end{pmatrix}, \mathbf{D_2} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 3 & 0 & 1 & 0 \\ 3 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 \\ 4 & 1 & 1 & 0 \\ 5 & 0 & 0 & 0 \\ 5 & 1 & 1 & 1 \\ 6 & 0 & 0 & 1 \\ 6 & 1 & 1 & 0 \end{pmatrix}.$$

Using the unweighted criterion, we get $J_2(\mathbf{D_1}) = 172$ and $J_2(\mathbf{D_2}) = 172$, while the weighted criterion gives $J_2(\mathbf{D_1}) = 912$ and $J_2(\mathbf{D_2}) = 880$. For $\mathbf{D_1}$, the non-orthogonal columns are 1 and 4, and in $\mathbf{D_2}$ columns 3 and 4. The unweighted J_2 views these as the same, as both have four $n_{kl}(a, b)$'s that are either one above or one below $N/(s_k s_l)$. The weighted J_2 considers this more serious when column 1 is involved as it has more levels, so in this example $\mathbf{D_2}$ is preferred. From a statistical standpoint, this makes sense, as in $\mathbf{D_1}$, some level combinations for columns 1 and 4 do not appear, whereas in $\mathbf{D_2}$ every level combination for all pairs of columns occurs at least once, including the non-orthogonal columns 3 and 4.

3.5 Advancements

This chapter introduced a new algorithm for finding orthogonal and nearly-orthogonal arrays. It adds new columns one row at a time by making the "best" choice for each row. We also used the χ^2 criterion with Xu's algorithm.

The J_2 criteria was extended to higher strength, and although results are mentioned by Xu (2003) in regards to minimum moment aberration, more detail was paid attention to here from an algorithmic standpoint. As well, both algorithms were adapted to be used for

higher strength. Discussion was given for trying to find a way to speed up the algorithm when dealing with a larger run size.

Chapter 4

Performance and Comparison

In this chapter we will compare the performance of Xu's algorithm and the new algorithm in terms of the construction of orthogonal and nearly-orthogonal arrays according to some measure of optimality. Orthogonal arrays and nearly-orthogonal arrays will be discussed in separate sections, as the approaches we use to compare the algorithms are different.

4.1 Construction of Orthogonal Arrays

For small run sizes, we often know if an orthogonal array exists. For settings where we know one exists, we can compare how each algorithm performs in finding an orthogonal array. To do this, a meaningful basis of comparison must be used. For practical purposes, the main aspects we want to study in an algorithm are speed and the ability to find orthogonal arrays. The algorithm should be successful at finding orthogonal arrays, but it should also be fast. If an algorithm can find orthogonal arrays easily, but takes a long time to do so, it may be impractical, particularly for large run sizes. For either algorithm, if we set the number of restarts very high, we expect the algorithms to find orthogonal arrays more often, but may end up sacrificing speed in doing so. On the other hand, if an algorithm can be run very quickly, but rarely finds an orthogonal array, the practicality is also questionable. For both algorithms, if we use a small number of restarts, we anticipate that the algorithms will be very fast, but may not find an orthogonal array very often.

What we really desire is a balance between speed and success. To find a reasonable balance, we use the average time to find an orthogonal array as a criterion to compare algorithms. That is, for an algorithm being started from its initial point for a certain

number of times,

$$Time_{OA} = OA_{found}/Time_{total}. \tag{4.1}$$

where $Time_{OA}$ is the average time to find an orthogonal array, OA_{found} is the total number of tiames an orthogonal arrays was found, and $Time_{total}$ is the total time to run the algorithm from its starting point for a specified number of times.

As mentioned, the speed and efficiency of the algorithms are determined by the number of restarts - the number of times the algorithm tries to add a new column until finally giving up. Too many restarts can mean a lot of time spent attempting to find an orthogonal column. If an orthogonal (to the existing design) column is difficult for the algorithm to find or does not exist, the algorithm is using time trying to reach a lower bound that it can not achieve. On the other hand, too few restarts may cause an algorithm to give up prematurely and have to start again from the initial design of just two columns. In this situation, the algorithm may ultimately take more time to find an orthogonal array, as it may take many runs of the algorithm before an orthogonal array is actually found.

Xu (2002) suggests 100 restarts as a suitable choice for his algorithm. While not based directly on the expected time to an orthogonal array, the suggestion of 100 restarts is based upon considering the success rate and time. The author shows his algorithm to be supereior to a Federov exchange algorithm from Miller and Nguyen (1994), and an interchange algorithm from Nguyen (1996). Using $Time_{OA}$ and comparing the two algorithms, the number of restarts to be tried for these algorithms are 50, 100, 200, 300, 500, and 1000. Varying the number of restarts in this way allows for better exploration of how the algorithms perform. In order to find a "good" number of restarts to use for an algorithm, we can use $Time_{OA}$. We hope to determine how many restarts should be used in a general setting for decent results (according to $Time_{OA}$).

Another issue for examining the algorithms is the number of times each algorithm should be repeated from the initial point in trying to find an orthogonal array, referred to as *tries*. If the number of tries is too low, the results may not accurately reflect how the algorithm does on average as the expected time to an orthogonal array is based on the number of orthogonal arrays found. This is particularly troublesome in situations where finding an orthogonal array is a very rare occurrence. After a set number of attempts, it is possible that an orthogonal array is found only once or twice, perhaps even not at all. When the number of tries is large, if one set of tries results in one orthogonal array while the other finds two orthogonal arrays, the difference in average time to an orthogonal array will be

markedly different. However, if an algorithm can find an orthogonal array often and we use too many tries, then too much time is spent without much gain in information on the algorithm.

When we study an algorithm, we want to ensure that we have an accurate representation of how we can expect the algorithm to perform on average. A reasonable question to ask is "how many times should the algorithm find an orthogonal array before we stop?" To do this, we want to estimate p, the probability of any given try resulting in an orthogonal array. If we are comfortable with our estimate of p, we do not need to worry that one set of tries happened to get "lucky." The more tries that use to estimate p, the better we anticipate our estimate will be, simply as the number of orthogonal arrays found divided by the total number of tries. However, at some point we need to decide that we have used enough tries to be satisfied with our estimate of p, otherwise we will just keep running the algorithm. To estimate p, we will use the geometric distribution.

The geometric distribution counts the number of trials until a success is observed. The random variable X denotes this number of trials. The geometric distribution has the following properties:

$$\begin{aligned} E(X) &= \frac{1}{p} \\ Var(X) &= \frac{1-p}{p^2} \end{aligned}$$

where p is the probability of success. As tries are independent of each other, if we consider finding an orthogonal array as a success and a non-orthogonal array as a failure, to find n orthogonal arrays, we can consider n iid random variables from a geometric distribution, stopping when the nth orthogonal array is found. With $X_1, X_2, \ldots, X_n$ iid from a geometric distribution with parameter p, a 95% confidence interval for $E(X)$ (ie. $1/p$) is

$$\bar{X} \pm 1.96\frac{\sqrt{1-p}}{p\sqrt{n}}.$$

To get a 95% confidence interval for p, we can use

$$\hat{p} \pm 1.96\frac{\hat{p}\sqrt{1-\hat{p}}}{\sqrt{n}},$$

where $\hat{p}$ is the number of orthogonal arrays found divided by the total number of tries. We want to use the geometric distribution to remove some of the chance that one algorithm

may have happened to have a "lucky" set of tries and ensures that the results reflect how well the algorithms do on average.

The issue is now how accurate we want our estimate of p to be. For orthogonal arrays in which the algorithms have a reasonable success rate, say greater than 0.10, an interval half-width of 0.01 is considered. This amount is still small enough to be comfortable with the results, and we do not want to have the half-width too small, or else there may be too many successes to be found from a time standpoint.

From the 95% confidence interval for p, the number of geometric random variables to sample for a specified half-width w is

$$n = \frac{1.96^2 p^2 (1-p)}{w^2}.$$

For the half-width of 0.01, n achieves its maximum when $p = 2/3$, giving $n = 5692$. Looking at this situation as a worst case scenario, in running the code until it has found an orthogonal array 6000 times, the confidence interval for p will have a half-width of at most 0.01.

For those orthogonal arrays in which the success rate for the algorithms is very low, 6000 successes may take substantial time to be achieved, so 6000 is not feasible. In addition, for small success rates, an interval half-width of 0.01 is likely not that desirable. In this situation, an interval half-width proportional to p is preferred. Using $w = p/10$, the number of successes needed is

$$n = 100 * 1.96^2 (1-p).$$

In this situation, as p decreases n increases, so using $n = 400$ allows for intervals to have have a half-width of $p/10$ or less. While this number of successes may seem small in comparison to the 6000 discussed for a larger success rate, for rare events there is more information contained in each observation of the geometric random variable. For these rare events, we can expect to see a number of failures before a success. For a higher rate of success, we expect to see fewer failures.

To make compasison of the algorithms fair, we try to speed up calculations as much as possible. The new algorithm will be used with the simplified χ^2 given as equation (3.6). Xu's algorithm will be used with weights of one which is suggested when finding orthogonal arrays.

4.1.1 Results

Both algorithms are suited for designs with a smaller number of runs, so it is orthogonal arrays of this nature which we will study. We use the orthogonal arrays studied in Xu (2002). Considering these orthogonal arrays, we have some prior information on what we can anticipate on the performance of Xu's algorithm (Xu (2002)). These arrays are typical of the mixed-level designs used for industrial experiments with small runs, and are also diverse in the number of runs, factors, and levels.

For each orthogonal array, each setting of restarts to be tested was used for Xu's algorithm with the J_2, Xu's algorithm with χ^2, and the new algorithm with χ^2. The number of orthogonal arrays found, the number of tries, and the total time spent were recorded in each case. Determination of the number of tries resulting in an orthogonal array (400 or 6000) was based on previous results from Xu (2002), and where information was not available, using the amount of time/tries until one orthogonal array was found. In a few situations, the time to find one orthogonal array was so extreme that the test was stopped after one orthogonal array for time considerations.

For the number of restarts that were tested, the expected time to find an orthogonal array is presented for Xu's algorithm with the J_2 criterion (Table 4.1), Xu's algorithm with the χ^2 criterion (Table 4.2), and the new algorithm using the χ^2 criterion (Table 4.3). The new algorithm using the J_2 criterion was not tested, as in testing it, it performs very poorly. The reason for this is an additional condition in using the χ^2 criterion that checks whether $n_{kl}(a,b)$ exceeds its expected value which fits in naturally with the χ^2 criterion, but is not a natural extension with the J_2 criterion. Removing this condition with the χ^2 criterion also results in poor performance. While we could use n_{kl} to force balance, if we need to store these in addition to the components for J_2, it is just as well to use χ^2. For all settings, more detailed tables giving tries resulting in an orthogonaly array, total number of tries, total time spent, estimated $\hat{p}$, $95\%CI$ for p, and estimated expected time to an orthogonal array are presented in tables 4.6-4.23.

Moving on to the results, Xu (2002) recommended using 100 column restarts, considering both the efficiency of finding orthogonal arrays and speed. Most of the consideration was placed on when the proportion of tries resulting in an orthogonal array appeared to be constant. However, in considering expected time to an orthogonal array, 50 restarts seems to be enough in most situations for Xu's algorithm with either criterion. Using more

restarts, we expect to find an orthogonal array more often, but the expected time to find an orthogonal array increases since extra time is spent in situations where the algorithm can not find an orthogonal array.

For the new algorithm, there does not seem to be a clearcut choice for the number of restarts to use. A try in the new algorithm is generally very quick in comparison to Xu's, but does not find an orthogonal array as often. For smaller run sizes (< 27), 300-500 restarts is a reasonable choice for expected time to an orthogonal array. For the higher run sizes studied, 1000 restarts is a better choice. In contrast to Xu's algorithm, where just 50 restarts is often enough for finding orthogonal arrays and can be used in most situations, too few restarts for the new algorithm can cause a very low success rate and high expected time to an orthogonal array.

Table 4.4 compares the best expected time to an orthogonal array for each algorithm among the different orthogonal arrays tested and lists the best time among those and which algorithm achieved that time. It is apparent from Table 4.4 that there is no universal winner in terms of expected time to an orthogonal array. However, we notice a trend in which the new algorithm performs better when the number of columns is small, and performs worse when the number of columns is close to the number of rows. This makes sense because the new algorithm has added calculations compared to Xu's as the number of columns grows.

Looking at Table 4.4 for Xu's algorithm, comparing the J_2 and χ^2 criteria, we see that in many cases the χ^2 criterion is an improvement over the J_2 criterion. This usually occurs when the number of columns is small relative to the run size. To get a better idea of why this occurs, we can look at the manner in which the criteria are computed. If the current design has $m-1$ columns, and we are trying to add the mth column, the χ^2 criterion uses $2*(m-1)$ of the n_{ij}'s for evaluating/making a symbol switch between rows i and j. For the J_2 criterion, the calculation is based on $2*N/s_m$ of the $\delta_{i,j}$'s. If the number of columns is small relative to the number of rows, the χ^2 criterion will perform better, with the J_2 criterion preferred as the number of columns grows. To get the greatest efficiency from Xu's algorithm, it may be worthwhile to use the χ^2 criterion for some initial set columns, and then switch to J_2 for the remaining as it is not influenced by the number of columns.

While the new algorithm may not outperform Xu's in all situations, it provides an effective means for constructing orthogonal arrays with a small number of rows and a modest number of columns relative to the number of rows. For studying the χ^2 criterion with Xu's algorithm, we found that there may be improvements in the situations where there is a

moderate number of columns relative to the number of rows.

4.1.2 Discussion

Based on the observation that the new algorithm seems to perform much better when the number of factors is small relative to the run size, it may be worthwhile to try and use the expansive replacement method. For the *expansive replacement* method, let A be an orthogonal array of strength 2 with a factor l having s_l levels, and B be an orthogonal array of strength 2 having s_l runs. By making a one-to-one correspondence between each level of factor l of A and the s_l runs of B, replacing each occurrence of the levels in A with the corresponding run in B, the resultant is an orthogonal array of strength 2 with at least as many factors as A. This can be useful with the observations about a smaller number of factors in comparison to the run size, as we may be able to find a design faster by using expansive replacement afterward. An example of this in the designs studied here are the $OA(27, 9^1 3^9)$ and $OA(27, 3^{13})$. The new algorithm can find an $OA(27, 9^1 3^9)$ very quickly in comparison to the $OA(27, 3^{13})$, but using expansive replacement, we can get an $OA(27, 3^{13})$ using an $OA(27, 9^1 3^9)$ with the 9-level factor replaced with an $OA(9, 3^4)$.

4.2 Nearly-Orthogonal Arrays

For studying nearly-orthogonal arrays, we want to find a design optimal according to the A_2 criterion due to the statistical justification described in Chapter 2. In comparing the algorithms for orthogonal arrays, the comparison was done on the efficiency for finding an optimal design - an orthogonal array. For nearly-orthogonal arrays, we generally do not know if a design is optimal. As such, our approach to comparing the algorithms must be modified.

Xu (2002) compared his algorithm to other existing algorithms (an interchange algorithm from Nguyen (1996), a threshold accepting technique from Ma et al (2000), and combinatorial construction methods from Wang and Wu (1992)) in regards to the A_2 criterion for nearly-orthogonal arrays with small run sizes. Xu's proved to be a clear winner. We take the same approach, comparing the new algorithm to Xu's in terms of the A_2 criterion. We use the mixed-level nearly-orthogonal arrays tested by Xu (2002) as a basis of comparison.

Even though we are now dealing with nearly-orthogonal arrays, the algorithms still proceed in the same way they did for the orthogonal case. For the χ^2 criterion, there are situations where the operations will not be integer, but this does not change the methodology of the algorithm. Since we do not know if we have an optimal design, the geometric approach can not be used to select the number of tries. Instead we choose the number of tries to be 10,000 based on past experience. From the observations about the number of restarts for orthogonal arrays, the number of restarts used for the new algorithm will be 300, 500, and 1000.

The J_2 equivalence to A_2 in equation (2.8) comes from setting the weight of each column to be the number of levels for that column. Due to this, instead of using the faster equation (3.6), we now use (2.6) to drive the algorithm. In the orthogonal array case, $\chi^2_{kl} = 0$ for all k,l, so the denominator was not necessary. In the nearly-orthogonal case, the denominator serves the same purpose as natural weights in Xu's algorithm, which we use for the statistical purpose because of the relationship to A_2. This distinction can be illustrated by example 3.4.

4.2.1 Results

For each nearly-orthgonal array, 10,000 tries were used for 300, 500, and 1000 restarts with the new algorithm. The algorithm returned the design with lowest A_2 among all the tries. This A_2 value is recorded along with the best nearly-orthogonal designs reported by Xu (2002). Since we are looking for designs having lower A_2 values, the slower equation (2.6) is used to drive the algorithm due to its relationship to A_2.

The results from each of the nearly-orthogonal arrays are shown in Table 4.5. The new algorithm results in comparable A_2 designs for every nearly-orthogonal array. The exceptions are the $NOA(24, 2^1 3^{11})$, in which case the new algorithm did not achieve the A_2 of 2.01 that Xu (2002) reported. However, for the $NOA(24, 3^1 4^7)$ and $NOA(12, 2^7 3^2)$, the new algorithm provides designs with better A_2 values, and thus new designs were found.

In some situations, we know if a nearly-orthogonal array is A_2-optimal. From Lemma 2.1, if for any pair of rows i and j, the $n_{ij}(a, b)$'s are within one, the design is A_2 optimal. While it is not always possible to find such a design, if one does exist, lower bounds can be adjusted for all $n_{ij}(a, b)$'s being within 1, and we could proceed in the same way as when dealing with orthogonal arrays. Since we do not know beforehand if such an optimal design exists, we must see if the results are different than those reported by Xu (2002).

When we can not find an optimal design like that described above, it is not always clear how the design performs in terms of A_2. From our results, the best design for both algorithms tends to have the same A_2 value. In fact, for most of the nearly-orthogonal arrays, these designs are found within the first few tries, and increasing the number of restarts/tries does not seem to give any improvement.

Some nearly-orthogonal arrays cause problems for the algorithms in trying to find a best nearly-orthogonal array. For instance, using the new algorithm, the best $NOA(24, 2^1 3^{11})$ is not as good as that found by Xu (2002). As well, all three different restarts result in different A_2 values, and the 500 restarts actually finds a better design than 1000 restarts. If we use 2000 restarts (not listed in table), the best design has an A_2 value of 1.91, which is better than the result from Xu (2002). Since we are trying to make A_2 as small as possible, this suggests that there may be instances in which it is worthwhile to use a greater number of restarts, even though it may increase the runtime.

When dealing with a mixed-level array, the order in which we add columns into the design can have an impact on the design which the algorithm returns as having the best A_2. For instance, using the first column as the 6-level column in the $NOA(12, 6^1 2^5)$, the best design has an A_2 value of 0.444. If we instead consider the 6-level column as the last column added, the best design has an A_2 value of 1.000. A possible explanation is related to the natural weights. When the 6-level column is added first, it is likely that added columns will be forced into orthogonality with the 6-level column because more weight is assigned to this column. When used as the last column, the algorithm must try to make the 6-level column orthogonal with the other five columns which appears more difficult.

The previous discussion on the $NOA(12, 6^1 2^5)$ highlights one of the major concerns in using a columnwise algorithm to find nearly orthogonal arrays: the best nearly-orthogonal array for some k columns may not help in creating the best nearly-orthogonal array for $k+1$ columns. For the $NOA(12, 6^1 2^5)$, if we consider adding the 6-level column last, we may be trying to add this column to an $OA(12, 2^5)$, which would be an optimal design for the first five columns. However, the resulting best NOA for six columns will not be as good as if considering the six-level column first.

Xu (2002) advised to arrange the columns in decreasing order of levels, due to the number of possible balanced columns. We make the same suggestion due to the use of natural weights, to try and force orthogonality with the higher level columns. Since the sequential nature may not work well with a particular ordering, it may be worthwhile to randomize

the order of the factors, so that if one ordering is better, it should get used by the algorithm at some point. For the $NOA(12, 6^1 2^5)$, if the 6-level column is used as column 1, 2, or 3, a design can be found with A_2 of 0.444.

4.2.2 Discussion

Using the χ^2 criterion for nearly-orthogonal arrays, it is possible that $N/(s_k s_l)$, and hence χ^2, is not integer. However, in not using integer calculations, the speed is greatly increased. In an attempt to speed up calculation, if we examine the χ^2 criterion again,

$$\begin{aligned}
\chi^2_{kl}(\mathbf{d}) &= \sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1}\frac{[n_{kl}(a,b) - N/(s_k s_l)]^2}{N/(s_k s_l)} \\
&= (s_k s_l)/N \sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1}\left[n_{kl}(a,b)^2 - 2N n_{kl}(a,b)/(s_k s_l) + N^2/(s_k s_l)^2\right] \\
&= (s_k s_l)/N \left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1} n_{kl}(a,b)^2\right] - (s_k s_l)/N\left[2N^2/(s_k s_l) - N^2/(s_k s_l)\right] \\
&= (s_k s_l)/N \left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1} n_{kl}(a,b)^2\right] - N.
\end{aligned}$$

Looking at χ^2 in this way, the only control we have on the A_2 criterion is to minimize $(s_k s_l)/N\left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1} n_{kl}(a,b)^2\right]$. When dealing with nearly-orthogonal arrays, it is quicker to deal with the simpler

$$\chi^{2*}_{kl} = (s_k s_l)\left[\sum_{a=0}^{s_k-1}\sum_{b=0}^{s_l-1} n_{kl}(a,b)^2\right]$$

which is also beneficial in that it is integer. When using the original χ^2_{kl} criterion, a $n_{kl}(a,b)$ exceeding the expected number $N/(s_k s_l)$ by one or two may not be very desirable, especially in the case of orthogonal arrays. However, the impact on χ^2 of exceeding the expected number is not necessarily very large, and a choice that does so may be deemed the best choice. The χ^{2*}_{kl} criterion will treat exceeding the expected number as being more serious, hopefully causing more balance among the columns.

Table 4.1: Expected time (in secs) to OA for Xu's algorithm - J_2 criteria.

OA	50	100	200	300	500	1000
$OA(9, 3^4)$	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004
$OA(20, 2^{19})$	0.01149	0.01661	0.02809	0.03155	0.04614	0.08553
$OA(16, 2^{15})$	0.00116	0.00115	0.00115	0.00116	0.00116	0.00116
$OA(12, 2^{11})$	0.00033	0.00037	0.00046	0.00054	0.00078	0.00121
$OA(16, 8^1 2^8)$	0.00086	0.00086	0.00086	0.00086	0.00086	0.00086
$OA(16, 4^5)$	0.03428	0.06703	0.13163	0.19256	0.32710	0.65567
$OA(18, 3^7 2^1)$	0.00583	0.00728	0.01049	0.01336	0.01989	0.03358
$OA(18, 6^1 3^6)$	0.04579	0.08438	0.16620	0.24315	0.40683	0.78767
$OA(20, 5^1 2^8)$	0.02689	0.03583	0.06080	0.08595	0.13052	0.25249
$OA(24, 2^{23})$	0.09150	0.13929	0.21202	0.27655	0.40633	0.68883
$OA(24, 4^1 2^{20})$	0.05414	0.08309	0.12126	0.16527	0.23114	0.41733
$OA(24, 3^1 2^{16})$	0.73488	1.14190	2.04770	2.68075	4.33018	8.22000
$OA(24, 12^1 2^{12})$	0.01193	0.01199	0.01233	0.01261	0.01311	0.01423
$OA(24, 4^1 3^1 2^{13})$	0.48003	0.78833	1.36018	1.94388	2.66440	5.19288
$OA(24, 6^1 4^1 2^{11})$	0.22801	0.40483	0.68750	0.94183	1.45883	2.68383
$OA(25, 5^6)$	0.33263	0.65250	1.27650	1.847	3.10167	6.01467
$OA(27, 9^1 3^9)$	0.05280	0.05086	0.05068	0.05074	0.05074	0.05074
$OA(27, 3^{13})$	41.72750	47.55500	76.78000	101.08750	156.34000	337.31750
$OA(28, 2^{27})$	6.52250	9.94250	18.55000	25.91250	37.98750	70.76500
$OA(32, 16^1 2^{16})$	0.18622	0.13603	0.12982	0.12935	0.12957	0.12957
$OA(32, 8^1 4^2 2^{18})$	0.23916	0.36100	0.54467	0.68917	0.90900	0.90900
$OA(40, 20^1 2^{20})$	53.3675	4.80238	1.86400	1.62917	1.75650	1.75650

Table 4.2: Expected time (in secs) to OA for Xu's algorithm - χ^2 criteria.

OA	50	100	200	300	500	1000
$OA(9, 3^4)$	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004
$OA(20, 2^{19})$	0.01335	0.01942	0.02391	0.03694	0.05448	0.10160
$OA(16, 2^{15})$	0.00162	0.00162	0.00162	0.00162	0.00162	0.00162
$OA(12, 2^{11})$	0.00045	0.00047	0.00053	0.00057	0.00071	0.00098
$OA(16, 8^1 2^8)$	0.00080	0.00079	0.00079	0.00079	0.00079	0.00079
$OA(16, 4^5)$	0.02012	0.03879	0.07666	0.11208	0.19046	0.38133
$OA(18, 3^7 2^1)$	0.00419	0.00511	0.00711	0.00888	0.01303	0.02152
$OA(18, 6^1 3^6)$	0.02925	0.05427	0.10672	0.15582	0.26043	0.50617
$OA(20, 5^1 2^8)$	0.02064	0.02786	0.04687	0.06634	0.10126	0.19501
$OA(24, 2^{23})$	0.10361	0.16673	0.26203	0.34567	0.51900	0.89367
$OA(24, 4^1 2^{20})$	0.05605	0.09100	0.13695	0.18436	0.27164	0.49717
$OA(24, 3^1 2^{16})$	0.72308	1.18685	2.19533	2.90705	4.73555	9.13750
$OA(24, 12^1 2^{12})$	0.01045	0.01052	0.01065	0.01081	0.01106	0.01162
$OA(24, 4^1 3^1 2^{13})$	0.42285	0.73217	1.30083	1.88060	2.62585	5.18485
$OA(24, 6^1 4^1 2^{11})$	0.16844	0.31107	0.54033	0.74700	1.16950	2.16600
$OA(25, 5^6)$	0.12614	0.24076	0.48900	0.70783	1.18900	2.30350
$OA(27, 9^1 3^9)$	0.02815	0.02730	0.02730	0.02726	0.02726	0.02726
$OA(27, 3^{13})$	23.34500	27.91500	46.85000	62.85250	99.24750	217.97750
$OA(28, 2^{27})$	5.93750	9.91250	19.12500	35.65500	42.08750	80.05250
$OA(32, 16^1 2^{16})$	0.14521	0.11180	0.10755	0.10725	0.10739	0.10739
$OA(32, 8^1 4^2 2^{18})$	0.15462	0.26077	0.41950	0.54550	0.75033	1.27150
$OA(40, 20^1 2^{20})$	27.575	2.90823	1.27483	1.15383	1.25817	1.62800

Table 4.3: Expected time (in secs) to OA for new algorithm - χ^2 criteria.

OA	50	100	200	300	500	1000
$OA(9, 3^4)$	0.00002	0.00002	0.00002	0.00002	0.00002	0.00002
$OA(20, 2^{19})$	0.07220	0.02051	0.01773	0.01998	0.02451	0.03443
$OA(16, 2^{15})$	0.00232	0.00219	0.00219	0.00217	0.00217	0.00222
$OA(12, 2^{11})$	0.00043	0.00042	0.00044	0.00044	0.00046	0.00051
$OA(16, 8^1 2^8)$	0.00037	0.00037	0.00037	0.00037	0.00037	0.00037
$OA(16, 4^5)$	0.00210	0.00399	0.00816	0.01171	0.01918	0.03925
$OA(18, 3^7 2^1)$	0.00408	0.00336	0.00340	0.00366	0.00436	0.00598
$OA(18, 6^1 3^6)$	0.01359	0.02604	0.04648	0.07052	0.11091	0.21725
$OA(20, 5^1 2^8)$	0.05308	0.02822	0.02751	0.02391	0.02872	0.04863
$OA(24, 2^{23})$	-	16.38250	0.47333	0.20637	0.19994	0.25612
$OA(24, 4^1 2^{20})$	150.80000	1.44953	0.12949	0.08880	0.09116	0.13025
$OA(24, 3^1 2^{16})$	6216	398.76000	17.06250	5.20098	2.13895	1.98565
$OA(24, 12^1 2^{12})$	0.00378	0.00330	0.00325	0.00327	0.00333	0.00343
$OA(24, 4^1 3^1 2^{13})$	1902.97	125.39000	2.87870	1.36700	0.88230	1.09703
$OA(24, 6^1 4^1 2^{11})$	2.80718	0.35610	0.15058	0.14990	0.18453	0.31941
$OA(25, 5^6)$	0.01844	0.02788	0.05401	0.07787	0.12475	0.25365
$OA(27, 9^1 3^9)$	0.05666	0.05574	0.02701	0.02318	0.02161	0.02336
$OA(27, 3^{13})$	719115	-	-	629.8275	471.75000	129.03
$OA(28, 2^{27})$	-	-	-	26150	920.660	56.31
$OA(32, 16^1 2^{16})$	0.27375	0.05029	0.02787	0.02504	0.02437	0.02403
$OA(32, 8^1 4^2 2^{18})$	-	449.28000	268.06000	13.57750	1.52683	0.49383
$OA(40, 20^1 2^{20})$	-	2942	124.17500	20.32250	0.40800	0.26325

Table 4.4: Best Expected time (in secs) to OA for each algorithm.

OA	New	Xu-χ^2	Xu-J_2	Best	Algorithm
$OA(9, 3^4)$	0.00002	0.00004	0.00004	0.00002	New
$OA(20, 2^{19})$	0.01773	0.01335	0.01149	0.01149	Xu-J_2
$OA(16, 2^{15})$	0.00217	0.00162	0.00115	0.00115	Xu-J_2
$OA(12, 2^{11})$	0.00042	0.00045	0.00033	0.00033	Xu-J_2
$OA(16, 8^1 2^8)$	0.00037	0.00079	0.00086	0.00037	New
$OA(16, 4^5)$	0.00210	0.02012	0.03428	0.00210	New
$OA(18, 3^7 2^1)$	0.00336	0.00419	0.00583	0.00336	New
$OA(18, 6^1 3^6)$	0.01359	0.02925	0.04579	0.01359	New
$OA(20, 5^1 2^8)$	0.02391	0.02064	0.02689	0.02064	Xu-χ^2
$OA(24, 2^{23})$	0.19994	0.10361	0.09150	0.09150	Xu-J_2
$OA(24, 4^1 2^{20})$	0.08880	0.05605	0.05414	0.05414	Xu-J_2
$OA(24, 3^1 2^{16})$	1.98565	0.72308	0.73488	0.72308	Xu-χ^2
$OA(24, 12^1 2^{12})$	0.00325	0.01045	0.01193	0.00325	New
$OA(24, 4^1 3^1 2^{13})$	0.88230	0.42285	0.48003	0.42285	Xu-χ^2
$OA(24, 6^1 4^1 2^{11})$	0.14990	0.16844	0.22801	0.14990	New
$OA(25, 5^6)$	0.01844	0.12614	0.33263	0.01844	New
$OA(27, 9^1 3^9)$	0.02161	0.02726	0.05068	0.02161	New
$OA(27, 3^{13})$	129.03000	23.34500	41.72750	23.34500	Xu-χ^2
$OA(28, 2^{27})$	56.31000	5.93750	6.52250	5.93750	Xu-χ^2
$OA(32, 16^1 2^{16})$	0.02403	0.10725	0.12935	0.02403	New
$OA(32, 8^1 4^2 2^{18})$	0.49383	0.15462	0.23916	0.15462	Xu-χ^2
$OA(40, 20^1 2^{20})$	0.26325	1.15383	1.62917	0.26325	New

Table 4.5: Comparison of the new algorithm to Xu's in terms of A_2.

NOA	Xu	300	500	1000
$NOA(6, 3^1 2^3)$	0.333	0.333	0.333	0.333
$NOA(10, 5^1 2^5)$	0.4	0.4	0.4	0.4
$NOA(12, 4^1 3^4)$	0.75	0.75	0.75	0.75
$NOA(12, 2^3 3^4)$	0.75	0.75	0.75	0.75
$NOA(12, 6^1 2^5)$	0.444	0.444	0.444	0.444
$NOA(12, 6^1 2^6)$	0.667	0.667	0.667	0.667
$NOA(12, 3^1 2^9)$	0.778	0.833	0.833	0.778
$NOA(12, 2^1 3^5)$	1.25	1.25	1.25	1.25
$NOA(12, 2^7 3^2)$	0.861	0.792	0.792	0.792
$NOA(12, 2^5 3^3)$	0.875	0.764	0.764	0.764
$NOA(15, 5^1 3^5)$	0.8	0.8	0.8	0.8
$NOA(18, 2^1 3^8)$	0.5	0.5	0.5	0.5
$NOA(18, 3^7 2^3)$	0.333	0.333	0.333	0.333
$NOA(18, 9^1 2^8)$	0.346	0.346	0.346	0.346
$NOA(20, 5^1 2^{15})$	0.76	0.76	0.76	0.76
$NOA(24, 8^1 3^8)$	0.875	0.875	0.875	0.875
$NOA(24, 3^1 2^{21})$	0.722	0.833	0.819	0.722
$NOA(24, 6^1 2^{15})$	0.111	0.111	0.111	0.111
$NOA(24, 6^1 2^{18})$	0.667	0.667	0.667	0.667
$NOA(24, 2^1 3^{11})$	2.01	2.208	2.083	2.115
$NOA(24, 3^1 4^7)$	2.56	2.58	2.53	2.472

Table 4.6: Results for Xu's algorithm - J_2 criterion, 50 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.24	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9464	68.96	0.63398	0.62428	0.64369	0.01149
$OA(16, 2^{15})$	6000	6000	6.94	1.00000	1.00000	1.00000	0.00116
$OA(12, 2^{11})$	6000	6354	1.97	0.94429	0.93865	0.94993	0.00033
$OA(16, 8^1 2^8)$	6000	6000	5.17	1.00000	1.00000	1.00000	0.00086
$OA(16, 4^5)$	6000	35987	205.7	0.16673	0.16288	0.17058	0.03428
$OA(18, 3^7 2^1)$	6000	7321	34.95	0.81956	0.81075	0.82837	0.00583
$OA(18, 6^1 3^6)$	6000	37386	274.74	0.16049	0.15677	0.16421	0.04579
$OA(20, 5^1 2^8)$	6000	21035	161.34	0.28524	0.27914	0.29134	0.02689
$OA(24, 2^{23})$	6000	20297	548.98	0.29561	0.28933	0.30189	0.09150
$OA(24, 4^1 2^{20})$	6000	13641	324.81	0.43985	0.43152	0.44818	0.05414
$OA(24, 3^1 2^{16})$	400	11498	293.95	0.03479	0.03144	0.03814	0.73488
$OA(24, 12^1 2^{12})$	6000	6100	71.55	0.98361	0.98042	0.98679	0.01193
$OA(24, 4^1 3^1 2^{13})$	400	7863	192.01	0.05087	0.04601	0.05573	0.48003
$OA(24, 6^1 4^1 2^{11})$	6000	66054	1368.04	0.09083	0.08864	0.09303	0.22801
$OA(25, 5^6)$	6000	53268	1995.75	0.11264	0.10995	0.11532	0.33263
$OA(27, 9^1 3^9)$	6000	7222	316.77	0.83079	0.82215	0.83944	0.05280
$OA(27, 3^{13})$	400	354156	16691	0.00113	0.00102	0.00124	41.72750
$OA(28, 2^{27})$	400	61539	2609	0.00650	0.00587	0.00713	6.52250
$OA(32, 16^1 2^{16})$	6000	12686	1117.31	0.47296	0.46427	0.48165	0.18622
$OA(32, 8^1 4^2 2^{18})$	6000	17032	1434.93	0.35228	0.34510	0.35945	0.23916
$OA(40, 20^1 2^{20})$	400	132250	21347	0.00305	0.00273	0.00332	53.3675

Table 4.7: Results for Xu's algorithm - J_2 criterion, 100 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.24	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9581	99.63	0.62624	0.61655	0.63593	0.01661
$OA(16, 2^{15})$	6000	6000	6.9	1.00000	1.00000	1.00000	0.00115
$OA(12, 2^{11})$	6000	6322	2.22	0.94907	0.94365	0.95449	0.00037
$OA(16, 8^1 2^8)$	6000	6000	5.14	1.00000	1.00000	1.00000	0.00086
$OA(16, 4^5)$	6000	35924	402.15	0.16702	0.16316	0.17088	0.06703
$OA(18, 3^7 2^1)$	6000	7240	43.68	0.82873	0.82005	0.83741	0.00728
$OA(18, 6^1 3^6)$	6000	36929	506.28	0.16247	0.15871	0.16624	0.08438
$OA(20, 5^1 2^8)$	6000	17612	214.96	0.34068	0.33368	0.34768	0.03583
$OA(24, 2^{23})$	6000	19660	835.74	0.30519	0.29875	0.31163	0.13929
$OA(24, 4^1 2^{20})$	6000	13720	498.55	0.43732	0.42902	0.44562	0.08309
$OA(24, 3^1 2^{16})$	400	10132	456.76	0.03948	0.03569	0.04327	1.14190
$OA(24, 12^1 2^{12})$	6000	6065	71.91	0.98928	0.98669	0.99187	0.01199
$OA(24, 4^1 3^1 2^{13})$	6000	108982	4730	0.05505	0.05370	0.05641	0.78833
$OA(24, 6^1 4^1 2^{11})$	6000	66851	2429	0.08975	0.08759	0.09192	0.40483
$OA(25, 5^6)$	6000	52403	3915	0.11450	0.11177	0.11722	0.65250
$OA(27, 9^1 3^9)$	6000	6153	305.15	0.97513	0.97124	0.97902	0.05086
$OA(27, 3^{13})$	400	196711	19022	0.00203	0.00183	0.00223	47.55500
$OA(28, 2^{27})$	400	45571	3977	0.00878	0.00792	0.00963	9.94250
$OA(32, 16^1 2^{16})$	6000	6728	816.19	0.89180	0.88437	0.89922	0.13603
$OA(32, 8^1 4^2 2^{18})$	6000	14704	2166	0.40805	0.40011	0.41600	0.36100
$OA(40, 20^1 2^{20})$	400	5710	1920.95	0.07005	0.06343	0.07667	4.80238

Table 4.8: Results for Xu's algorithm - J_2 criterion, 200 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.24	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9496	168.53	0.63184	0.62214	0.64155	0.02809
$OA(16, 2^{15})$	6000	6000	6.9	1.00000	1.00000	1.00000	0.00115
$OA(12, 2^{11})$	6000	6305	2.75	0.95163	0.94633	0.95692	0.00046
$OA(16, 8^1 2^8)$	6000	6000	5.14	1.00000	1.00000	1.00000	0.00086
$OA(16, 4^5)$	6000	35874	789.76	0.16725	0.16339	0.17111	0.13163
$OA(18, 3^7 2^1)$	6000	7271	62.92	0.82520	0.81647	0.83393	0.01049
$OA(18, 6^1 3^6)$	6000	37663	997.18	0.15931	0.15561	0.16300	0.16620
$OA(20, 5^1 2^8)$	6000	17466	364.78	0.34352	0.33648	0.35057	0.06080
$OA(24, 2^{23})$	6000	20103	1272.1	0.29846	0.29214	0.30479	0.21202
$OA(24, 4^1 2^{20})$	6000	13619	727.58	0.44056	0.43222	0.44890	0.12126
$OA(24, 3^1 2^{16})$	400	10603	819.08	0.03773	0.03410	0.04135	2.04770
$OA(24, 12^1 2^{12})$	6000	6072	73.96	0.98814	0.98542	0.99086	0.01233
$OA(24, 4^1 3^1 2^{13})$	400	7357	544.07	0.05437	0.04919	0.05955	1.36018
$OA(24, 6^1 4^1 2^{11})$	6000	65669	4125	0.09137	0.08916	0.09357	0.68750
$OA(25, 5^6)$	6000	53731	7659	0.11167	0.10900	0.11433	1.27650
$OA(27, 9^1 3^9)$	6000	6003	304.07	0.99950	0.99893	1.00007	0.05068
$OA(27, 3^{13})$	400	159970	30712	0.00250	0.00226	0.00275	76.78000
$OA(28, 2^{27})$	400	42839	7420	0.00934	0.00843	0.01025	18.55000
$OA(32, 16^1 2^{16})$	6000	6036	778.89	0.99404	0.99209	0.99598	0.12982
$OA(32, 8^1 4^2 2^{18})$	6000	14678	3268	0.40878	0.40082	0.41673	0.54467
$OA(40, 20^1 2^{20})$	6000	16621	11184	0.36099	0.35369	0.36829	1.86400

Table 4.9: Results for Xu's algorithm - J_2 criterion, 300 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.24	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9501	189.28	0.63151	0.62181	0.64121	0.03155
$OA(16, 2^{15})$	6000	6000	6.98	1.00000	1.00000	1.00000	0.00116
$OA(12, 2^{11})$	6000	6289	3.25	0.95405	0.94887	0.95922	0.00054
$OA(16, 8^1 2^8)$	6000	6000	5.14	1.00000	1.00000	1.00000	0.00086
$OA(16, 4^5)$	6000	35248	1155.34	0.17022	0.16630	0.17415	0.19256
$OA(18, 3^7 2^1)$	6000	7217	80.14	0.83137	0.82273	0.84001	0.01336
$OA(18, 6^1 3^6)$	6000	37211	1458.92	0.16124	0.15751	0.16498	0.24315
$OA(20, 5^1 2^8)$	6000	17583	515.71	0.34124	0.33423	0.34825	0.08595
$OA(24, 2^{23})$	6000	20216	1659.3	0.29679	0.29050	0.30309	0.27655
$OA(24, 4^1 2^{20})$	6000	13843	991.61	0.43343	0.42518	0.44169	0.16527
$OA(24, 3^1 2^{16})$	400	10235	1072.3	0.03908	0.03533	0.04284	2.68075
$OA(24, 12^1 2^{12})$	6000	6072	75.63	0.98814	0.98542	0.99086	0.01261
$OA(24, 4^1 3^1 2^{13})$	400	7729	777.55	0.05175	0.04681	0.05669	1.94388
$OA(24, 6^1 4^1 2^{11})$	6000	64738	5651	0.09268	0.09045	0.09492	0.94183
$OA(25, 5^6)$	6000	52427	11082	0.11444	0.11172	0.11717	1.847
$OA(27, 9^1 3^9)$	6000	6000	304.44	1.00000	1.00000	1.00000	0.05074
$OA(27, 3^{13})$	400	144062	40435	0.00278	0.00250	0.00305	101.0875
$OA(28, 2^{27})$	400	43360	10365	0.00923	0.00833	0.01012	25.91250
$OA(32, 16^1 2^{16})$	6000	6003	776.11	0.99950	0.99893	1.00007	0.12935
$OA(32, 8^1 4^2 2^{18})$	6000	14809	4135	0.40516	0.39725	0.41307	0.68917
$OA(40, 20^1 2^{20})$	6000	10987	9775	0.54610	0.53679	0.55541	1.62917

Table 4.10: Results for Xu's algorithm - J_2 criterion, 500 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.24	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9526	276.81	0.62986	0.62016	0.63955	0.04614
$OA(16, 2^{15})$	6000	6000	6.94	1.00000	1.00000	1.00000	0.00116
$OA(12, 2^{11})$	6000	6328	4.65	0.94817	0.94270	0.95363	0.00078
$OA(16, 8^1 2^8)$	6000	6000	5.14	1.00000	1.00000	1.00000	0.00086
$OA(16, 4^5)$	6000	35886	1962.58	0.16720	0.16334	0.17106	0.32710
$OA(18, 3^7 2^1)$	6000	7270	119.35	0.82531	0.81658	0.83404	0.01989
$OA(18, 6^1 3^6)$	6000	37620	2441	0.15949	0.15579	0.16319	0.40683
$OA(20, 5^1 2^8)$	6000	17172	783.12	0.34941	0.34227	0.35654	0.13052
$OA(24, 2^{23})$	6000	20531	2438	0.29224	0.28602	0.29846	0.40633
$OA(24, 4^1 2^{20})$	6000	13803	1386.83	0.43469	0.42642	0.44296	0.23114
$OA(24, 3^1 2^{16})$	400	10936	1732.07	0.03658	0.03306	0.04009	4.33018
$OA(24, 12^1 2^{12})$	6000	6071	78.63	0.98831	0.98560	0.99101	0.01311
$OA(24, 4^1 3^1 2^{13})$	400	7053	1065.76	0.05671	0.05132	0.06211	2.66440
$OA(24, 6^1 4^1 2^{11})$	6000	65020	8753	0.09228	0.09005	0.09450	1.45883
$OA(25, 5^6)$	6000	53061	18610	0.11308	0.11038	0.11577	3.10167
$OA(27, 9^1 3^9)$	6000	6000	304.44	1.00000	1.00000	1.00000	0.05074
$OA(27, 3^{13})$	400	139853	62536	0.00286	0.00258	0.00314	156.34000
$OA(28, 2^{27})$	400	42548	15195	0.00940	0.00848	0.01032	37.98750
$OA(32, 16^1 2^{16})$	6000	6000	777.42	1.00000	1.00000	1.00000	0.12957
$OA(32, 8^1 4^2 2^{18})$	6000	14591	5454	0.41121	0.40323	0.41920	0.90900
$OA(40, 20^1 2^{20})$	6000	9250	10539	0.64865	0.63892	0.65838	1.75650

Table 4.11: Results for Xu's algorithm - J_2 criterion, 1000 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.24	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9649	513.18	0.62183	0.61215	0.63150	0.08553
$OA(16, 2^{15})$	6000	6000	6.98	1.00000	1.00000	1.00000	0.00116
$OA(12, 2^{11})$	6000	6307	7.28	0.95132	0.94601	0.95663	0.00121
$OA(16, 8^1 2^8)$	6000	6000	5.14	1.00000	1.00000	1.00000	0.00086
$OA(16, 4^5)$	6000	36125	3934	0.16609	0.16225	0.16993	0.65567
$OA(18, 3^7 2^1)$	6000	7194	201.5	0.83403	0.82543	0.84263	0.03358
$OA(18, 6^1 3^6)$	6000	36872	4726	0.16273	0.15896	0.16649	0.78767
$OA(20, 5^1 2^8)$	6000	17322	1514.93	0.34638	0.33929	0.35347	0.25249
$OA(24, 2^{23})$	6000	19945	4133	0.30083	0.29446	0.30719	0.68883
$OA(24, 4^1 2^{20})$	6000	13925	2504	0.43088	0.42265	0.43910	0.41733
$OA(24, 3^1 2^{16})$	400	11461	3288	0.03490	0.03154	0.03826	8.22000
$OA(24, 12^1 2^{12})$	6000	6065	85.36	0.98928	0.98669	0.99187	0.01423
$OA(24, 4^1 3^1 2^{13})$	400	7483	2077.15	0.05345	0.04836	0.05855	5.19288
$OA(24, 6^1 4^1 2^{11})$	6000	63704	16103	0.09419	0.09192	0.09645	2.68383
$OA(25, 5^6)$	6000	51715	36088	0.11602	0.11326	0.11878	6.01467
$OA(27, 9^1 3^9)$	6000	6000	304.44	1.00000	1.00000	1.00000	0.05074
$OA(27, 3^{13})$	400	159955	134927	0.00250	0.00226	0.00275	337.31750
$OA(28, 2^{27})$	400	46731	28306	0.00856	0.00772	0.00939	70.76500
$OA(32, 16^1 2^{16})$	6000	6000	777.42	1.00000	1.00000	1.00000	0.12957
$OA(32, 8^1 4^2 2^{18})$	6000	14591	5454	0.41121	0.40323	0.41920	0.90900
$OA(40, 20^1 2^{20})$	6000	9250	10539	0.64865	0.63892	0.65838	1.75650

Table 4.12: Results for Xu's algorithm - χ^2 criterion, 50 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.22	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9464	80.07	0.63398	0.62428	0.64369	0.01335
$OA(16, 2^{15})$	6000	6000	9.69	1.00000	1.00000	1.00000	0.00162
$OA(12, 2^{11})$	6000	6354	2.67	0.94429	0.93865	0.94993	0.00045
$OA(16, 8^1 2^8)$	6000	6000	4.78	1.00000	1.00000	1.00000	0.00080
$OA(16, 4^5)$	6000	35987	120.69	0.16673	0.16288	0.17058	0.02012
$OA(18, 3^7 2^1)$	6000	7321	25.15	0.81956	0.81075	0.82837	0.00419
$OA(18, 6^1 3^6)$	6000	37386	175.51	0.16049	0.15677	0.16421	0.02925
$OA(20, 5^1 2^8)$	6000	21035	123.84	0.28524	0.27914	0.29134	0.02064
$OA(24, 2^{23})$	6000	20297	621.63	0.29561	0.28933	0.30189	0.10361
$OA(24, 4^1 2^{20})$	6000	13641	336.29	0.43985	0.43152	0.44818	0.05605
$OA(24, 3^1 2^{16})$	400	11498	289.23	0.03479	0.03144	0.03814	0.72308
$OA(24, 12^1 2^{12})$	6000	6100	62.69	0.98361	0.98042	0.98679	0.01045
$OA(24, 4^1 3^1 2^{13})$	400	7863	169.14	0.05087	0.04601	0.05573	0.42285
$OA(24, 6^1 4^1 2^{11})$	6000	66054	1010.61	0.09083	0.08864	0.09303	0.16844
$OA(25, 5^6)$	6000	53268	756.86	0.11264	0.10995	0.11532	0.12614
$OA(27, 9^1 3^9)$	6000	7222	168.9	0.83079	0.82215	0.83944	0.02815
$OA(27, 3^{13})$	400	354156	9338	0.00113	0.00102	0.00124	23.34500
$OA(28, 2^{27})$	400	61539	2375	0.00650	0.00587	0.00713	5.93750
$OA(32, 16^1 2^{16})$	6000	12686	871.24	0.47296	0.46427	0.48165	0.14521
$OA(32, 8^1 4^2 2^{18})$	6000	17032	927.73	0.35228	0.34510	0.35945	0.15462
$OA(40, 20^1 2^{20})$	400	132250	11030	0.00302	0.00273	0.00332	27.575

Table 4.13: Results for Xu's algorithm - χ^2 criterion, 100 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.22	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9581	116.5	0.62624	0.61655	0.63593	0.01942
$OA(16, 2^{15})$	6000	6000	9.71	1.00000	1.00000	1.00000	0.00162
$OA(12, 2^{11})$	6000	6322	2.82	0.94907	0.94365	0.95449	0.00047
$OA(16, 8^1 2^8)$	6000	6000	4.74	1.00000	1.00000	1.00000	0.00079
$OA(16, 4^5)$	6000	35924	232.71	0.16702	0.16316	0.17088	0.03879
$OA(18, 3^7 2^1)$	6000	7240	30.63	0.82873	0.82005	0.83741	0.00511
$OA(18, 6^1 3^6)$	6000	36929	325.59	0.16247	0.15871	0.16624	0.05427
$OA(20, 5^1 2^8)$	6000	17612	167.16	0.34068	0.33368	0.34768	0.02786
$OA(24, 2^{23})$	6000	19660	1000.36	0.30519	0.29875	0.31163	0.16673
$OA(24, 4^1 2^{20})$	6000	13720	545.97	0.43732	0.42902	0.44562	0.09100
$OA(24, 3^1 2^{16})$	400	10132	474.74	0.03948	0.03569	0.04327	1.18685
$OA(24, 12^1 2^{12})$	6000	6065	63.14	0.98928	0.98669	0.99187	0.01052
$OA(24, 4^1 3^1 2^{13})$	6000	108982	4393	0.05505	0.05370	0.05641	0.73217
$OA(24, 6^1 4^1 2^{11})$	6000	66851	1866.41	0.08975	0.08759	0.09192	0.31107
$OA(25, 5^6)$	6000	52403	1444.55	0.11450	0.11177	0.11722	0.24076
$OA(27, 9^1 3^9)$	6000	6153	163.81	0.97513	0.97124	0.97902	0.02730
$OA(27, 3^{13})$	400	196711	11166	0.00203	0.00183	0.00223	27.91500
$OA(28, 2^{27})$	400	45571	3965	0.00878	0.00792	0.00963	9.91250
$OA(32, 16^1 2^{16})$	6000	6728	670.8	0.89180	0.88437	0.89922	0.11180
$OA(32, 8^1 4^2 2^{18})$	6000	14704	1564.64	0.40805	0.40011	0.41600	0.26077
$OA(40, 20^1 2^{20})$	400	5710	1163.29	0.07005	0.06343	0.07667	2.90823

Table 4.14: Results for Xu's algorithm - χ^2 criterion, 200 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.22	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9496	143.45	0.63184	0.62214	0.64155	0.02391
$OA(16, 2^{15})$	6000	6000	9.7	1.00000	1.00000	1.00000	0.00162
$OA(12, 2^{11})$	6000	6305	3.19	0.95163	0.94633	0.95692	0.00053
$OA(16, 8^1 2^8)$	6000	6000	4.75	1.00000	1.00000	1.00000	0.00079
$OA(16, 4^5)$	6000	35874	459.96	0.16725	0.16339	0.17111	0.07666
$OA(18, 3^7 2^1)$	6000	7271	42.67	0.82520	0.81647	0.83393	0.00711
$OA(18, 6^1 3^6)$	6000	37663	640.29	0.15931	0.15561	0.16300	0.10672
$OA(20, 5^1 2^8)$	6000	17466	281.2	0.34352	0.33648	0.35057	0.04687
$OA(24, 2^{23})$	6000	20103	1572.15	0.29846	0.29214	0.30479	0.26203
$OA(24, 4^1 2^{20})$	6000	13619	821.7	0.44056	0.43222	0.44890	0.13695
$OA(24, 3^1 2^{16})$	400	10603	878.13	0.03773	0.03410	0.04135	2.19533
$OA(24, 12^1 2^{12})$	6000	6072	63.88	0.98814	0.98542	0.99086	0.01065
$OA(24, 4^1 3^1 2^{13})$	400	7357	520.33	0.05437	0.04919	0.05955	1.30083
$OA(24, 6^1 4^1 2^{11})$	6000	65669	3242	0.09137	0.08916	0.09357	0.54033
$OA(25, 5^6)$	6000	53731	2934	0.11167	0.10900	0.11433	0.48900
$OA(27, 9^1 3^9)$	6000	6003	163.78	0.99950	0.99893	1.00007	0.02730
$OA(27, 3^{13})$	400	159970	18740	0.00250	0.00226	0.00275	46.85000
$OA(28, 2^{27})$	400	42839	7650	0.00934	0.00843	0.01025	19.12500
$OA(32, 16^1 2^{16})$	6000	6036	645.27	0.99404	0.99209	0.99598	0.10755
$OA(32, 8^1 4^2 2^{18})$	6000	14678	2517	0.40878	0.40082	0.41673	0.41950
$OA(40, 20^1 2^{20})$	6000	16621	7649	0.36099	0.35369	0.36829	1.27483

Table 4.15: Results for Xu's algorithm - χ^2 criterion, 300 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.22	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9501	221.63	0.63151	0.62181	0.64121	0.03694
$OA(16, 2^{15})$	6000	6000	9.71	1.00000	1.00000	1.00000	0.00162
$OA(12, 2^{11})$	6000	6289	3.42	0.95405	0.94887	0.95922	0.00057
$OA(16, 8^1 2^8)$	6000	6000	4.75	1.00000	1.00000	1.00000	0.00079
$OA(16, 4^5)$	6000	35248	672.47	0.17022	0.16630	0.17415	0.11208
$OA(18, 3^7 2^1)$	6000	7217	53.3	0.83137	0.82273	0.84001	0.00888
$OA(18, 6^1 3^6)$	6000	37211	934.89	0.16124	0.15751	0.16498	0.15582
$OA(20, 5^1 2^8)$	6000	17583	398.03	0.34124	0.33423	0.34825	0.06634
$OA(24, 2^{23})$	6000	20216	2073.99	0.29679	0.29050	0.30309	0.34567
$OA(24, 4^1 2^{20})$	6000	13843	1106.17	0.43343	0.42518	0.44169	0.18436
$OA(24, 3^1 2^{16})$	400	10235	1162.82	0.03908	0.03533	0.04284	2.90705
$OA(24, 12^1 2^{12})$	6000	6072	64.87	0.98814	0.98542	0.99086	0.01081
$OA(24, 4^1 3^1 2^{13})$	400	7729	752.24	0.05175	0.04681	0.05669	1.88060
$OA(24, 6^1 4^1 2^{11})$	6000	64738	4482	0.09268	0.09045	0.09492	0.74700
$OA(25, 5^6)$	6000	52427	4247	0.11444	0.11172	0.11717	0.70783
$OA(27, 9^1 3^9)$	6000	6000	163.54	1.00000	1.00000	1.00000	0.02726
$OA(27, 3^{13})$	400	144062	25141	0.00278	0.00250	0.00305	62.85250
$OA(28, 2^{27})$	400	43360	14262	0.00923	0.00833	0.01012	35.65500
$OA(32, 16^1 2^{16})$	6000	6003	643.47	0.99950	0.99893	1.00007	0.10725
$OA(32, 8^1 4^2 2^{18})$	6000	14809	3273	0.40516	0.39725	0.41307	0.54550
$OA(40, 20^1 2^{20})$	6000	10987	6923	0.54610	0.53679	0.55541	1.15383

Table 4.16: Results for Xu's algorithm - χ^2 criterion, 500 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.22	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9526	326.9	0.62986	0.62016	0.63955	0.05448
$OA(16, 2^{15})$	6000	6000	9.69	1.00000	1.00000	1.00000	0.00162
$OA(12, 2^{11})$	6000	6328	4.27	0.94817	0.94270	0.95363	0.00071
$OA(16, 8^1 2^8)$	6000	6000	4.75	1.00000	1.00000	1.00000	0.00079
$OA(16, 4^5)$	6000	35886	1142.78	0.16720	0.16334	0.17106	0.19046
$OA(18, 3^7 2^1)$	6000	7270	78.16	0.82531	0.81658	0.83404	0.01303
$OA(18, 6^1 3^6)$	6000	37620	1562.57	0.15949	0.15579	0.16319	0.26043
$OA(20, 5^1 2^8)$	6000	17172	607.57	0.34941	0.34227	0.35654	0.10126
$OA(24, 2^{23})$	6000	20531	3114	0.29224	0.28602	0.29846	0.51900
$OA(24, 4^1 2^{20})$	6000	13803	1629.86	0.43469	0.42642	0.44296	0.27164
$OA(24, 3^1 2^{16})$	400	10936	1894.22	0.03658	0.03306	0.04009	4.73555
$OA(24, 12^1 2^{12})$	6000	6071	66.38	0.98831	0.98560	0.99101	0.01106
$OA(24, 4^1 3^1 2^{13})$	400	7053	1050.34	0.05671	0.05132	0.06211	2.62585
$OA(24, 6^1 4^1 2^{11})$	6000	65020	7017	0.09228	0.09005	0.09450	1.16950
$OA(25, 5^6)$	6000	53061	7134	0.11308	0.11038	0.11577	1.18900
$OA(27, 9^1 3^9)$	6000	6000	163.54	1.00000	1.00000	1.00000	0.02726
$OA(27, 3^{13})$	400	139853	39699	0.00286	0.00258	0.00314	99.24750
$OA(28, 2^{27})$	400	42548	16835	0.00940	0.00848	0.01032	42.08750
$OA(32, 16^1 2^{16})$	6000	6000	644.33	1.00000	1.00000	1.00000	0.10739
$OA(32, 8^1 4^2 2^{18})$	6000	14591	4502	0.41121	0.40323	0.41920	0.75033
$OA(40, 20^1 2^{20})$	6000	9250	7549	0.64865	0.63892	0.65838	1.25817

Table 4.17: Results for Xu's algorithm - χ^2 criterion, 1000 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.22	1.00000	1.00000	1.00000	0.00004
$OA(20, 2^{19})$	6000	9649	609.57	0.62183	0.61215	0.63150	0.10160
$OA(16, 2^{15})$	6000	6000	9.69	1.00000	1.00000	1.00000	0.00162
$OA(12, 2^{11})$	6000	6307	5.85	0.95132	0.94601	0.95663	0.00098
$OA(16, 8^1 2^8)$	6000	6000	4.75	1.00000	1.00000	1.00000	0.00079
$OA(16, 4^5)$	6000	36125	2288	0.16609	0.16225	0.16993	0.38133
$OA(18, 3^7 2^1)$	6000	7194	129.09	0.83403	0.82543	0.84263	0.02152
$OA(18, 6^1 3^6)$	6000	36872	3037	0.16273	0.15896	0.16649	0.50617
$OA(20, 5^1 2^8)$	6000	17322	1170.03	0.34638	0.33929	0.35347	0.19501
$OA(24, 2^{23})$	6000	19945	5362	0.30083	0.29446	0.30719	0.89367
$OA(24, 4^1 2^{20})$	6000	13925	2983	0.43088	0.42265	0.43910	0.49717
$OA(24, 3^1 2^{16})$	400	11461	3655	0.03490	0.03154	0.03826	9.13750
$OA(24, 12^1 2^{12})$	6000	6065	69.71	0.98928	0.98669	0.99187	0.01162
$OA(24, 4^1 3^1 2^{13})$	400	7483	2073.94	0.05345	0.04836	0.05855	5.18485
$OA(24, 6^1 4^1 2^{11})$	6000	63704	12996	0.09419	0.09192	0.09645	2.16600
$OA(25, 5^6)$	6000	51715	13821	0.11602	0.11326	0.11878	2.30350
$OA(27, 9^1 3^9)$	6000	6000	163.54	1.00000	1.00000	1.00000	0.02726
$OA(27, 3^{13})$	400	159955	87191	0.00250	0.00226	0.00275	217.97750
$OA(28, 2^{27})$	400	46731	32021	0.00856	0.00772	0.00939	80.05250
$OA(32, 16^1 2^{16})$	6000	6000	644.33	1.00000	1.00000	1.00000	0.10739
$OA(32, 8^1 4^2 2^{18})$	6000	14591	7629	0.41121	0.40323	0.41920	1.27150
$OA(40, 20^1 2^{20})$	6000	9250	9768	0.64865	0.63892	0.65838	1.62800

Table 4.18: Results for the new algorithm - χ^2 criterion, 50 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.12	1.00000	1.00000	1.00000	0.00002
$OA(20, 2^{19})$	6000	146050	433.22	0.04108	0.04006	0.04210	0.07220
$OA(16, 2^{15})$	6000	6659	13.92	0.90104	0.89386	0.90821	0.00232
$OA(12, 2^{11})$	6000	6111	2.56	0.98184	0.97849	0.98518	0.00043
$OA(16, 8^1 2^8)$	6000	6005	2.23	0.99917	0.99844	0.99990	0.00037
$OA(16, 4^5)$	6000	20737	12.62	0.28934	0.28317	0.29551	0.00210
$OA(18, 3^7 2^1)$	6000	20012	24.47	0.29982	0.29347	0.30617	0.00408
$OA(18, 6^1 3^6)$	6000	81629	81.53	0.07350	0.07171	0.07529	0.01359
$OA(20, 5^1 2^8)$	6000	218539	318.45	0.02746	0.02677	0.02814	0.05308
$OA(24, 2^{23})$	6000	5660089	14698	0.00106	0.00103	0.00109	2.44967
$OA(24, 4^1 2^{20})$	400	24571115	60320	0.00002	0.00001	0.00002	150.80000
$OA(24, 3^1 2^{16})$	1	2579056	6216	0.00000	0.00000	0.00000	6216
$OA(24, 12^1 2^{12})$	6000	7792	22.68	0.77002	0.76068	0.77936	0.00378
$OA(24, 4^1 3^1 2^{13})$	1	895291	1902.97	0.00000	0.00000	0.00000	1902.97
$OA(24, 6^1 4^1 2^{11})$	400	565875	1122.87	0.00071	0.00064	0.00078	2.80718
$OA(25, 5^6)$	6000	75156	110.64	0.07983	0.07789	0.08177	0.01844
$OA(27, 9^1 3^9)$	6000	83649	339.95	0.07173	0.06998	0.07348	0.05666
$OA(27, 3^{13})$	1	415130993	719115	0.00000	-0.00000	0.00000	719115.00000
$OA(28, 2^{27})$	-	-	-	-	-	-	-
$OA(32, 16^1 2^{16})$	400	21103	109.5	0.01895	0.01711	0.02079	0.27375
$OA(32, 8^1 4^2 2^{18})$	-	-	-	-	-	-	-
$OA(40, 20^1 2^{20})$	-	-	-	-	-	-	-

Table 4.19: Results for the new algorithm - χ^2 criterion, 100 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.12	1.00000	1.00000	1.00000	0.00002
$OA(20, 2^{19})$	6000	16437	123.04	0.36503	0.35767	0.37239	0.02051
$OA(16, 2^{15})$	6000	6027	13.14	0.99552	0.99383	0.99721	0.00219
$OA(12, 2^{11})$	6000	63546087	2.54	0.00009	0.00009	0.00010	0.00042
$OA(16, 8^1 2^8)$	6000	6000	2.24	1.00000	1.00000	1.00000	0.00037
$OA(16, 4^5)$	6000	20703	23.93	0.28981	0.28363	0.29599	0.00399
$OA(18, 3^7 2^1)$	6000	10046	20.14	0.59725	0.58766	0.60684	0.00336
$OA(18, 6^1 3^6)$	6000	83826	156.23	0.07158	0.06983	0.07332	0.02604
$OA(20, 5^1 2^8)$	6000	61576	169.31	0.09744	0.09510	0.09978	0.02822
$OA(24, 2^{23})$	6000	16923285	98295	0.00035	0.00035	0.00036	16.38250
$OA(24, 4^1 2^{20})$	400	72278	579.81	0.00553	0.00499	0.00608	1.44953
$OA(24, 3^1 2^{16})$	100	7776675	39876	0.00001	0.00001	0.00002	398.76000
$OA(24, 12^1 2^{12})$	6000	6193	19.78	0.96884	0.96451	0.97316	0.00330
$OA(24, 4^1 3^1 2^{13})$	100	2775808	12539	0.00004	0.00003	0.00004	125.39000
$OA(24, 6^1 4^1 2^{11})$	6000	510443	2136.57	0.01175	0.01146	0.01205	0.35610
$OA(25, 5^6)$	6000	61087	167.26	0.09822	0.09586	0.10058	0.02788
$OA(27, 9^1 3^9)$	6000	83649	334.45	0.07173	0.06998	0.07348	0.05574
$OA(27, 3^{13})$	-	-	-	-	-	-	-
$OA(28, 2^{27})$	-	-	-	-	-	-	-
$OA(32, 16^1 2^{16})$	6000	22476	301.73	0.26695	0.26117	0.27273	0.05029
$OA(32, 8^1 4^2 2^{18})$	100	8851063	44928	0.00001	0.00001	0.00001	449.28000
$OA(40, 20^1 2^{20})$	1	298659	2942	0.00000	0.00000	0.00001	2942

Table 4.20: Results for the new algorithm - χ^2 criterion, 200 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.12	1.00000	1.00000	1.00000	0.00002
$OA(20, 2^{19})$	6000	9542	106.4	0.62880	0.61911	0.63849	0.01773
$OA(16, 2^{15})$	6000	6000	13.11	1.00000	1.00000	1.00000	0.00219
$OA(12, 2^{11})$	6000	6093	2.61	0.98474	0.98166	0.98781	0.00044
$OA(16, 8^1 2^8)$	6000	6000	2.22	1.00000	1.00000	1.00000	0.00037
$OA(16, 4^5)$	6000	21097	48.96	0.28440	0.27831	0.29049	0.00816
$OA(18, 3^7 2^1)$	6000	7652	20.38	0.78411	0.77489	0.79333	0.00340
$OA(18, 6^1 3^6)$	6000	82924	278.85	0.07236	0.07059	0.07412	0.04648
$OA(20, 5^1 2^8)$	6000	28327	165.06	0.21181	0.20705	0.21657	0.02751
$OA(24, 2^{23})$	6000	191423	2840	0.03134	0.03056	0.03212	0.47333
$OA(24, 4^1 2^{20})$	6000	43210	776.92	0.13886	0.13560	0.14212	0.12949
$OA(24, 3^1 2^{16})$	400	616769	6825	0.00065	0.00059	0.00071	17.06250
$OA(24, 12^1 2^{12})$	6000	6052	19.5	0.99141	0.98908	0.99373	0.00325
$OA(24, 4^1 3^1 2^{13})$	400	116168	1151.48	0.00344	0.00311	0.00378	2.87870
$OA(24, 6^1 4^1 2^{11})$	400	7165	60.23	0.05583	0.05051	0.06114	0.15058
$OA(25, 5^6)$	6000	61364	324.03	0.09777	0.09542	0.10012	0.05401
$OA(27, 9^1 3^9)$	6000	18499	162.05	0.32434	0.31760	0.33109	0.02701
$OA(27, 3^{13})$	-	-	-	-	-	-	-
$OA(28, 2^{27})$	-	-	-	-	-	-	-
$OA(32, 16^1 2^{16})$	6000	7700	167.23	0.77922	0.76996	0.78849	0.02787
$OA(32, 8^1 4^2 2^{18})$	100	2225068	26806	0.00004	0.00004	0.00005	268.06000
$OA(40, 20^1 2^{20})$	400	2182071	49670	0.00018	0.00017	0.00020	124.17500

Table 4.21: Results for the new algorithm - χ^2 criterion, 300 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.12	1.00000	1.00000	1.00000	0.00002
$OA(20, 2^{19})$	6000	9266	119.9	0.64753	0.63780	0.65726	0.01998
$OA(16, 2^{15})$	6000	6000	13.03	1.00000	1.00000	1.00000	0.00217
$OA(12, 2^{11})$	6000	6092	2.65	0.98490	0.98184	0.98796	0.00044
$OA(16, 8^1 2^8)$	6000	6000	2.21	1.00000	1.00000	1.00000	0.00037
$OA(16, 4^5)$	6000	20885	70.26	0.28729	0.28115	0.29342	0.01171
$OA(18, 3^7 2^1)$	6000	7231	21.97	0.82976	0.82110	0.83842	0.00366
$OA(18, 6^1 3^6)$	6000	85210	423.1	0.07041	0.06870	0.07213	0.07052
$OA(20, 5^1 2^8)$	6000	22219	143.47	0.27004	0.26420	0.27588	0.02391
$OA(24, 2^{23})$	6000	48056	1238.21	0.12485	0.12190	0.12781	0.20637
$OA(24, 4^1 2^{20})$	6000	19433	532.81	0.30875	0.30226	0.31525	0.08880
$OA(24, 3^1 2^{16})$	400	120476	2080.39	0.00332	0.00300	0.00364	5.20098
$OA(24, 12^1 2^{12})$	6000	6054	19.59	0.99108	0.98871	0.99345	0.00327
$OA(24, 4^1 3^1 2^{13})$	400	35130	546.8	0.01139	0.01028	0.01250	1.36700
$OA(24, 6^1 4^1 2^{11})$	6000	71571	899.42	0.08383	0.08180	0.08586	0.14990
$OA(25, 5^6)$	6000	59967	467.2	0.10006	0.09765	0.10246	0.07787
$OA(27, 9^1 3^9)$	6000	11358	139.09	0.52826	0.51908	0.53744	0.02318
$OA(27, 3^{13})$	400	22434606	251931	0.00002	0.00002	0.00002	629.82750
$OA(28, 2^{27})$	1	1351132	26150	0.00000	0.00000	0.00000	26150
$OA(32, 16^1 2^{16})$	6000	6394	150.26	0.93838	0.93249	0.94427	0.02504
$OA(32, 8^1 4^2 2^{18})$	400	245064	5431	0.00163	0.00147	0.00179	13.57750
$OA(40, 20^1 2^{20})$	400	196937	8129	0.00203	0.00183	0.00223	20.32250

Table 4.22: Results for the new algorithm - χ^2 criterion, 500 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.12	1.00000	1.00000	1.00000	0.00002
$OA(20, 2^{19})$	6000	9138	147.07	0.65660	0.64686	0.66633	0.02451
$OA(16, 2^{15})$	6000	6000	13.03	1.00000	1.00000	1.00000	0.00217
$OA(12, 2^{11})$	6000	6085	2.76	0.98603	0.98308	0.98898	0.00046
$OA(16, 8^1 2^8)$	6000	6000	2.21	1.00000	1.00000	1.00000	0.00037
$OA(16, 4^5)$	6000	20793	115.08	0.28856	0.28240	0.29472	0.01918
$OA(18, 3^7 2^1)$	6000	7176	26.14	0.83612	0.82756	0.84469	0.00436
$OA(18, 6^1 3^6)$	6000	82655	665.45	0.07259	0.07082	0.07436	0.11091
$OA(20, 5^1 2^8)$	6000	18642	172.32	0.32185	0.31515	0.32856	0.02872
$OA(24, 2^{23})$	6000	23534	1199.61	0.25495	0.24938	0.26052	0.19994
$OA(24, 4^1 2^{20})$	6000	13947	546.95	0.43020	0.42198	0.43842	0.09116
$OA(24, 3^1 2^{16})$	400	28671	855.58	0.01395	0.01259	0.01531	2.13895
$OA(24, 12^1 2^{12})$	6000	6051	19.97	0.99157	0.98927	0.99388	0.00333
$OA(24, 4^1 3^1 2^{13})$	400	13216	352.92	0.03027	0.02735	0.03319	0.88230
$OA(24, 6^1 4^1 2^{11})$	6000	59283	1107.2	0.10121	0.09878	0.10364	0.18453
$OA(25, 5^6)$	6000	58632	748.51	0.10233	0.09988	0.10479	0.12475
$OA(27, 9^1 3^9)$	6000	7915	129.66	0.75805	0.74862	0.76749	0.02161
$OA(27, 3^{13})$	400	10213648	188700	0.00004	0.00004	0.00004	471.75000
$OA(28, 2^{27})$	400	11141117	368264	0.00004	0.00003	0.00004	920.66000
$OA(32, 16^1 2^{16})$	6000	6049	146.21	0.99190	0.98964	0.99416	0.02437
$OA(32, 8^1 4^2 2^{18})$	6000	172627	9161	0.03476	0.03389	0.03562	1.52683
$OA(40, 20^1 2^{20})$	6000	28491	2448	0.21059	0.20586	0.21533	0.40800

Table 4.23: Results for the new algorithm - χ^2 criterion, 1000 restarts.

OA	Found	Tries	Time (secs)	$\hat{p}$	p_L	p_U	$\hat{E}_{Time_{OA}}(s)$
$OA(9, 3^4)$	6000	6000	0.12	1.00000	1.00000	1.00000	0.00002
$OA(20, 2^{19})$	6000	9190	206.6	0.65288	0.64315	0.66262	0.03443
$OA(16, 2^{15})$	6000	6000	13.31	1.00000	1.00000	1.00000	0.00222
$OA(12, 2^{11})$	6000	6085	3.08	0.98603	0.98308	0.98898	0.00051
$OA(16, 8^1 2^8)$	6000	6000	2.22	1.00000	1.00000	1.00000	0.00037
$OA(16, 4^5)$	6000	20845	235.5	0.28784	0.28169	0.29399	0.03925
$OA(18, 3^7 2^1)$	6000	7172	35.86	0.83659	0.82803	0.84514	0.00598
$OA(18, 6^1 3^6)$	6000	81687	1303.49	0.07345	0.07166	0.07524	0.21725
$OA(20, 5^1 2^8)$	6000	18315	291.76	0.32760	0.32080	0.33440	0.04863
$OA(24, 2^{23})$	6000	19998	1536.7	0.30003	0.29368	0.30638	0.25612
$OA(24, 4^1 2^{20})$	6000	13115	781.47	0.45749	0.44897	0.46602	0.13025
$OA(24, 3^1 2^{16})$	400	13267	794.26	0.03015	0.02724	0.03306	1.98565
$OA(24, 12^1 2^{12})$	6000	6058	20.58	0.99043	0.98797	0.99288	0.00343
$OA(24, 4^1 3^1 2^{13})$	400	8724	438.81	0.04585	0.04146	0.05024	1.09703
$OA(24, 6^1 4^1 2^{11})$	6000	57296	1916.43	0.10472	0.10221	0.10723	0.31941
$OA(25, 5^6)$	6000	59946	1521.9	0.10009	0.09768	0.10249	0.25365
$OA(27, 9^1 3^9)$	6000	6394	140.16	0.93838	0.93249	0.94427	0.02336
$OA(27, 3^{13})$	100	344097	12903	0.00029	0.00023	0.00035	129.03000
$OA(28, 2^{27})$	400	219767	22524	0.00182	0.00164	0.00200	56.31000
$OA(32, 16^1 2^{16})$	6000	6000	144.15	1.00000	1.00000	1.00000	0.02403
$OA(32, 8^1 4^2 2^{18})$	6000	22865	2963	0.26241	0.25671	0.26811	0.49383
$40, 20^1 2^{20})$	6000	10748	1579.52	0.55824	0.54885	0.56763	0.26325

Chapter 5

Summary

Many experiments investigate the effects of two or more factors, in which case factorial designs are frequently used. In many situations, a fractional factorial design can be used to run an experiment at a fraction of the runs needed for a full factorial design. These designs are particularly useful in industry as screening experiments, to identify important factors. Orthogonal arrays can be used as factorial designs with desirable statistical properties. If an orthogonal array does not exist, a nearly-orthogonal array can be used, where near-orthogonality can be measured by a number of criteria, some of which were introduced in Chapter 2.

When an experimenter needs an orgthogonal or nearly-orthogonal array for an experiment, one may not be readily available and may not be easy to find. In such situations, we wish to have an algorithm to construct an orthogonal array or a nearly-orthogonal array optimal according to some criterion. Chapter 3 discussed an algorithm by Xu (2002) and introduced a new algorithm.

As orthogonal arrays of higher strength are desirable, Chapter 3 extended the J_2 and χ^2 criteria to higher strength. The extension to higher strength was applied to the two algorithms from Chapter 3.

In Chapter 4, we compared Xu's algorithm and the new algorithm in terms of speed and efficiency for finding an orthogonal array. In comparing the algorithms, we also looked at the impact of the number of restarts on each of the algorithms. For orthogonal arrays, Xu's algorithm performs best with a small number of restarts, around 50 or 100. The new algorithm performs best with 300 to 500 restarts for very small run sizes and around 1000 for moderate run sizes. While there was no definitive winner between the two algorithms,

the new algorithm generally performs better when the number of factors is small relative to the run size. In constructing nearly-orthogonal arrays, using A_2 as a measure, the new algorithm performed similar to Xu's algorithm. For nearly-orthogonal arrays, the results suggest that in some situations, finding an optimal A_2 design is difficult for both algorithms and it may be worthwhile to increase the number of restarts.

Future work would include trying to further examine the connection between the J-criteria of higher strength in terms of the generalized minimum aberration criterion. Using the algorithms for larger run size would be desirable. Further study could also be done to look at the unification of more of the near-orthogonality criteria.

Bibliography

Chipman, H., Hamada, M., and Wu, C.F.J. (1997). "A Bayesian Variable-Selection Approach for Analyzing Designed Experiments With Complex Aliasing", *Technometrics*, **39**, 372-381.

Cramer, H. (1946). *Mathematical Methods of Statistics.* Princeton University Press, Princeton.

DeCock, D., and Stufken, J. (2000). "On Finding Mixed Orthogonal Arrays of Strength 2 With Many 2-level Factors", *Statistics and Probability Letters*, **50**, 383-388.

Deng, L. Y., and Tang, B. (1999). "Generalized Resolution and Minimum Aberration Criteria for Plackett-Burman and Other Nonregular Factorial Designs", *Statistica Sinica*, **9**, 1071-1082.

Hamada, M. and Wu, C. F. J. (1992). "Analysis of Designed Experiments with Complex Aliasing", *Journal of Quality Technology*, **24**, 130-137.

Hedayat, A. S., Sloane, N. J., and Stufken, J. (1999). *Orthogonal Arrays: Theory and Applications.* Springer, New York.

Li, W W., and Wu, C. F. J. (1997). "Columnwise-Pairwise Algorithms With Applications to the Construction of Supersaturated Designs", *Technometrics*, **39**, 171-179.

Ma, C.-X., and Fang, K.-T. (2001). "A Note on Generalized Aberration in Factorial Designs", *Metrika*, **53**, 85-93.

Ma, C.-X., Fang, K.-T., and Liski, E. (2000). "A New Approach in Constructing Orthogonal and Nearly Orthogonal Arrays", *Metrika*, **50**, 255-268.

Miller, A. J., and Nguyen, N.-K. (1994). "A Federov Exchange Algorithm for D-optimal Design", *Applied Statistics*, **43**, 669-677.

Montgomery, D.C.(1997). *Design and Analysis of Experiments*, 5th edition. Wilery, New York.

Nguyen, N.-K. (1996). "A Note on the Construction of Near-Orthogonal Arrays With Mixed Levels and Economic Run Size", *Technometrics*, **38**, 279-283.

Rao, C. R. (1947). "Factorial Experiments Derivable from Combinatorial Arragements of Arrays", *Journal of the Royal Statistical Society*, Supplement, **9**, 128-139.

Tang, B., and Deng, L. Y. (1999). "Minimum G_2-Aberration for Non-regular Fractional Factorial Designs", *The Annals of Statistics*, **27**, 1914-1926.

Wang, J. C, and Wu, C. F. J. (1992). "Nearly Orthogonal Arrays With Mixed Levels and Small Runs", *Technometrics*, **34**, 409-422.

Wu, C. F. J. and Hamada, M. (2000). *Experiments: Planning, Analysis and Parameter Design Optimization.* Wiley, New York.

Xu, H. (2002). "An Algorithm for Constructing Orthogonal and Nearly-orthogonal Arrays With mixed levels and small runs ", *Technometrics*, **44**, 356-368.

Xu, H. (2003). "Minimum Moment Aberration for Nonregular Designs and Supersaturated Designs", *Statistica Sinica*, **13**, 691-708.

Xu, H. and Lau, S. (2006). "Minimum Aberration Blocking Schemes for Two- and Three-Level Fractional Factorial Designs", *Journal of Statistical Planning and Inference*, **136**, 4088-4118.

Xu, H., and Wu, C.F.J. (2001). "Generalized Minimum Aberration for Asymmetrical Fractional Factorial Designs", *The Annals of Statistics*, **29**, 549-560.

Yamada, S., Ikebe, Y. T., Hashiguchi, H. and Niki, N. (1999). "Construction of Three-level Supersaturated Designs ", *Journal of Statistical Planning and Inference*, **81**, 183-193.

Yamada, S. and Lin, D. K. J. (1999). "Three-level Supersaturated Designs", *Statistics and Probability Letters*, **45**, 31-39.

Yamada, S. and Matsui, T. (2002). "Optimality of Mixed-level Supersaturated Designs", *Journal of Statistical Planning and Inference*, **104**, 459-468.

Ye, K. and Sudjianto, A. (2003). "The Use of Cramer V^2 Optimality for Experiments with Qualitative Levels", under revision, submitted to IIE Transactions.