Complete Enumeration of Pure-Level and Mixed-Level Orthogonal Arrays

Eric D. Schoen[*]

University of Antwerp, Belgium, and TNO Science and Industry, Delft, Netherlands

Pieter T. Eendebak[†]

TNO Science and Industry, Delft, Netherlands

Man V.M. Nguyen[‡]

University of Technology, Ho Chi Minh City, Vietnam

June 17, 2009

Abstract

We specify an algorithm to enumerate a minimum complete set of combinatorially non-isomorphic orthogonal arrays of given strength t, run-size N, and level-numbers of the factors. The algorithm is the first one handling general mixed-level and pure-level cases. Using an implementation in C, we generate most non-trivial series for $t = 2, N \leq 28$, $t = 3, N \leq 64$, and $t = 4, N \leq 168$. The exceptions define limiting run sizes for which the algorithm returns complete sets in a reasonable amount of time.

KEY WORDS: Experimental Design; Isomorphism; Asymmetric Array; Symmetric Array

[*]Corresponding author. Address: Dept. of Mathematics, Statistics, and Actuarial Sciences, Prinsstraat 13, 2000 Antwerp, Belgium. E-mail: eric.schoen@ua.ac.be

[†]Address: Dept. of Computer Vision and Statistics, PO Box 155, 2600 AD Delft, Netherlands. E-mail: pieter.eendebak@tno.nl

[‡]Address: Faculty of Information Technology, 268 Ly Thuong Kiet Street, Ho Chi Minh City, Vietnam. E-mail: mnguyen@cse.hcmut.edu.vn

1 Introduction

In scientific experimentation, researchers may want to investigate the joint effect of several factors on the properties of some product or process. The experiments are frequently conducted according to an orthogonal array (OA) of strength t. Formally, an OA of strength t is an $N \times n$ matrix whose ith column contains s_i different factor-levels in such a way that, for any t columns, every t-tuple of levels appear equally often in the matrix [19]. The N rows thus specify the different experiments to be performed, and the n columns specify the factors whose action is to be investigated.

For designed experiments, the strength t of an OA is usually chosen in the range $2 \leq t \leq 4$. OAs with $t = 2$ result in main effect estimates that are uncorrelated with each other. These estimates are, however, correlated with those of two-factor interactions. For OAs with $t = 3$, the estimates of the main effects are uncorrelated with interactions between any two other factors. However, there could be a problem of interpreting an active interaction component. Finally, OAs with $t > 3$ have orthogonal components of two-factor interactions. Thus, they could be employed when many such interactions are expected, at the cost of an increased run size. We refer to [13] for a comprehensive account on properties and explicit constructions of orthogonal arrays.

At this point, it is convenient to introduce more terminology. We use the notation $\mathrm{OA}(N; s_1, \ldots, s_n; t)$ for an OA of run size N, strength t, n factors, and level numbers $s_1, \ldots, s_n$. These features are collectively called the parameters of an array. A mixed, or asymmetrical, array has not all its s_i equal. Pure, or symmetrical arrays have equal level numbers for all the factors. It is convenient to use the notation $s_1^a s_2^b$ for arrays with a s_1-level factors and b s_2-level factors. Finally, regular arrays have $N = s^p$ for some positive integer p, and prime number s. There are p s-level basic factors, and the settings of the remaining factors can be calculated by modular arithmetic from those of the basic factors.

Two arrays are said to be combinatorially isomorphic if one array can be

obtained by permuting rows, columns, or factor levels of the other array. An *isomorphism class* is the collection of all arrays that can be obtained from a parent array by permuting rows, columns, or factor levels. Each array of such a class can be considered as a representative of the class, because one can obtain the remaining arrays by the permutations just mentioned. It would be highly desirable to have an enumeration method obtaining, for given parameters, a minimum complete set of OAs. Such a set has a single representative for each isomorphism class. If the factors are all qualitative, one would then have to choose the best array according to some optimality criterion, assign the factors to the columns such that prior knowledge on the factor's activities is incorporated as good as is feasible, and to assign the factor levels to the symbols in a column at random.

There is an extensive literature on enumeration methods producing pure-level OAs, with a bias towards strength-2 arrays. One of the most time consuming steps in all of these methods is the pairwise testing of arrays to decide their equivalence. Only non-equivalent arrays are considered for extension with further columns. The present paper has two main contributions. The first one is the replacement of the pairwise testing with a very fast test to decide whether a single array is of some special form. Only if this is the case, the array is kept for possible further extension. The second main contribution is to propose for the first time an algorithm for constructing a minimum complete set of mixed-level OAs of given run-size, numbers of factor-levels and strength. When using such an algorithm, one would like to obtain results in a reasonable amount of computing time. We generate a wide range of example arrays to show its potential to produce practically meaningful series of arrays in a limited amount of time. More in particular, we generate all but some non-trivial cases of $t = 2$ and $N \leq 28$, $t = 3$ and $N \leq 64$, and $t = 4$ and $N \leq 144$. The exceptions are used to discuss the limitations of our algorithm.

The rest of this paper is organized as follows. In Section 2, we give an overview of existing literature on enumeration of pure OAs. Our algorithm for

mixed arrays is discussed in detail in Section 3. Next, we present series of arrays of strength 2, 3, and 4, produced with the algorithm, along with computing times (Section 4). Finally, in Section 5 there is a brief discussion.

2 Literature review

An essential element in establishing a minimum complete set of OAs is in proving that all of the arrays are indeed non-isomorphic to each other. The problem is to avoid calculation of all possible permutations of rows, columns, and factor-levels to map one array on another array. Clark and Dean [6] present an algorithm to test the equivalence of two-level arrays without having to calculate all these permutations. The algorithm can be extended to cases with factors at more than two levels, and indeed was used a such by [10]. A survey of further equivalence tests is given by [16].

The enumeration in a minimum complete set of OAs was addressed by [5], [26], [25], [1], [11], and [4]. These papers differ in the specific arrays considered and in the test of design isomorphism. Chen et al. [5] is on symmetrical regular designs. The authors were able to obtain a complete catalogue of all regular two-level designs of up to 64 runs. For given run-size N and number of factors n, the authors start with a minimum complete set of designs with $n-1$ factors and they consider all possible extensions with one additional column. The resulting designs are then classified with the word-length pattern. Those of equal word-length pattern are classified further according to their letter pattern. Finally, among designs with equal letter pattern, exhaustive isomorphism testing is performed. The ideas from this paper were used by later authors to obtain two-level designs with run-sizes up to 128 [2], or three-level designs with run-sizes up to 729 [28].

Tsai et al. [26] study designs for quantitative factors. They present a column-wise algorithm to obtain three-level designs of strength 2. There is no combinatorial isomorphism testing, but the designs are classified through measures

derived from A-efficiency.

Sun et al. [25] present an algorithm to construct all non-isomorphic two-level designs of specified run-size and numbers of factors. Like [5], the authors start with a minimum complete set of designs with $n-1$ factors and they consider all possible extensions with one additional column. The resulting designs are then classified with the extended word-length pattern [7]. Those in the same class are further tested with algebraic techniques.

Angelopoulos et al. [1] also considered two-level designs. These authors use initial classification with Hamming distances and D-efficiencies. Designs in the same class are then tested with the so-called max-int algorithm of [8]. Evangelaras et al. [9] used a column-wise extension method earlier described in [26]) for the purpose of generating all pure nonisomorphic 18-run arrays with three-level factors. Isomorphism tests were carried out directly, without intermediate classification.

Finally, [4] do not use extensions from a minimum complete set of $n-1$ factor arrays. Instead, they proposed a powerful ILP programming technique to construct minimum complete sets of pure-level arrays of given run-size, strength, and level number of all the factors. Isomorphism testing was carried out with computer-algebraic techniques.

In our algorithm, to be described shortly, we also use the idea of extending a minimum complete set of arrays for $n-1$ factors with an additional column. Like [11], we start each column with a specific element. Previous authors added complete columns and checked the orthogonality to existing columns afterwards. Instead, we use element-wise addition of symbols and registers book-keeping all t-tuples of symbols to abort additions that violate the strength requirement in an early stage. We avoid pairwise preliminary isomorphism testing [16] by retaining only arrays of a specific form.

3 Minimum Complete Set Algorithm

The enumeration problem entails constructing a minimum complete set of OAs with given run-size, strength, and level numbers of the factors. No array in such a set can be obtained from another array in the set by the joint operations of permuting columns, permuting rows, and exchanging factor-levels. The set thus does not contain combinatorially isomorphic arrays. Note that an OA can be considered as a specific kind of combinatorial object. In this section, we apply general principles for algorithms that exhaustively generate all combinatorial objects of some specified class to formulate a Minimum Complete Set Algorithm, or MCS Algorithm, for enumerating orthogonal arrays. We illustrate with a worked example. We end this section with a comparison of the run-time of the algorithm with previously published results. A possible concern with the algorithm is that it might not return a minimum complete set. In Appendix 1, we prove that it does return such a set.

3.1 Description

The Minimum Complete Set algorithm, or MCS Algorithm, implements general principles from [15] to the specific case of generating a minimum complete set of OA$(N; s_1, \ldots, s_n; t)$. We assume without loss of generality that $n > t$, and $s_1 \geq s_2 \geq \cdots \geq s_n$. We also assume that an s_i-level factor uses symbols $0, 1, \ldots, (s_i - 1)$. The task at hand is called isomorph-free exhaustive generation. A key idea of the algorithm is to retain arrays only if they are of a specific form that can be tested easily. This form is called lexicographically minimum in columns, according to Definition 1.

Definition 1. Consider two OA$(N; s_1, \ldots, s_n; t)$, say, D_1 and D_2. Write d_1 (d_2) for the $N.n$-tuples obtained by concatenating the columns of D_1 (D_2). Let the elements of these tuples be denoted with $d_{ij} (i = 1, 2; j = 1 \cdots Nn)$. D_1 is lexicographically less than D_2, denoted $D_1 < D_2$, if there is a $k \leq N.n$ such that

$d_{1j} = d_{2j}$ for $j = 1 \cdots k-1$ and $d_{1k} < d_{2k}$. D_1 is lexicographically minimum in columns (LMC) if no other array from its isomorphism class is smaller.

Each array considered by the MCS ALGORITHM will be called a node. The input is the set $N, n, t, s_1, \ldots, s_n$. The output is a set of $n - t$ minimum complete sets of OAs in LMC form. A global description of the algorithm follows.

1. Define the root node R as the unique LMC OA$(N; s_1, \ldots, s_t; t)$. Here, λ copies of all combinations of the first t factors are generated, with the index λ defined as $\lambda = N/(\prod_i^t s_i)$. These are subsequently written down in the unique LMC order, the first factor varying slowest, then the second one, etc.

2. Call the EXTENSION ALGORITHM. This algorithm extends a particular LMC OA$(N; s_1, \ldots, s_q; t)$ to a series of OA$(N; s_1, \ldots, s_{q+1}; t)$. The set of all extensions of a minimum complete set in q factors is guaranteed to contain all LMC OA$(N; s_1, \ldots, s_{q+1}; t)$.

3. For each resulting array, call the LMC TEST to check whether the array is LMC.

4. For the new set of LMC arrays, if $q < n$, then return to step (2).

We will denote the collection of all child nodes from a parent X with $C(X)$. In Step 3, it is tested whether a child node $C_i(X)$ is LMC. Extensions of non-LMC nodes are not considered. LMC nodes are further extended and tested. We will now look more closely at the components of the algorithm.

3.2 Extension Algorithm

The EXTENSION ALGORITHM extends an LMC array X with $q \geq t$ columns to arrays $C_i(X)$ with $c = q + 1$ columns by element-wise addition. Denote the additional column with $X(:, c)$, and its elements with $X(r, c)$, with r indexing the rows. Denote the number of different symbols for the new column with

s_c, and denote the symbols themselves with $0, \ldots, (s_c - 1)$. The EXTENSION ALGORITHM then runs as follows.

1. Set $X(1, c) = 0$. Note that all other choices result in a lexicographically larger array.

2. Consider the next row. Choose the next element subject to all of the following conditions.

   (a) If $s_c = s_{c-1}$, and $X(z, c) = X(z, (c-1))$ for all $z < r$, then we require $X(r, c) \geq X(r, (c-1))$.

   (b) If, apart from the new column, $X(r, :) = X((r-1), :)$, choose the new symbol from $\{X(r-1, c), \ldots, s_c - 1\}$. Otherwise, choose from $\{0, \ldots, (s_c - 1)\}$.

   (c) Introduce a symbol for the first time only if all smaller symbols have been used at least once before in the current column.

3. For each possible element addition $X(r, c)$, test compatibility of the the column constructed so far with the strength of the array.

4. If the strength condition is violated, stop the current extension.

5. If the strength condition is met, and if the column is not completed, return to step (2).

6. If the strength condition is met, and if the column is completed, return the extended array to the main algorithm.

Condition 2a uses the fact that a new column in a section of the array must be lexicographically larger than the previous one. Condition 2b prevents getting an array that was generated earlier. For example, suppose that we have to extend a row 1 1 1 by an element 0, 1, or 2. We first try the extension with 0 and proceed with further extensions. Later on we return to this point and try an extension with 1. Suppose that the next row starts off with 1 1 1. Than

it would not make sense to try an extension of that row to 1 1 1 0, because switching the rows results in an array that has already been considered.

3.3 LMC test

It remains to be discussed how to test whether a node is LMC. We will explain the test for pure arrays. The test for mixed arrays is slightly more complicated because permutations of columns can only be carried out among those with equal level-numbers.

A naive LMC test would perform all possible level permutations, row permutations, and column permutations to find an image that is lexicographically smaller than the original. As soon as we find such an image, we conclude that the original is not LMC. If there is no such image, we conclude that the original is LMC. For a pure array with s-level factors, the total number of permutations is $n!(s!)^n N!$. However, subsequent to permutations of levels and columns, the row permutations giving the lexicographically smallest image can be determined by sorting the resulting array. This reduces the number of permutations to be checked to $n!(s!)^n$. The sorting operation is computationally quite intensive, but we will show below how to reduce the sorting time. In addition, we show how to reduce the number of column and level permutations further by exploiting the combinatorial properties of OAs.

The sorting time can be reduced by using the fact that the root of an OA, subsequent to any combination of column permutations and level permutations, and after sorting the rows, is always the same. For all $s!^t$ level permutations in the first t columns of an array, we can calculate in advance what row permutations are needed to return the root. For each of the level permutations, there may be several row permutations restoring the root. For each of the level permutations, we store just one of these row permutations. We can use the stored information repeatedly, because there are $\binom{n}{t}t!$ possible choices for the first t columns that may need to be checked.

The above reasoning led us to split the LMC test in two stages called the

Algorithm 1 Root stage of the LMC test. Input: array O, strength t. Output: scalar $y(O)$: 0=fail, 1=pass

```
1: for all choices for the first t columns do
2:    Sort the resulting array using the first t columns.
3:    for all possible level permutations of the first t columns do
4:       Sort the array using a precalculated row permutation such that the root
         is restored.
5:       Pass the array on to the non-root stage at level t + 1.
6:       if the non-root stage returns a 0 then
7:          Set y(O) = 0. Terminate the test.
8:       else
9:          Continue with the containing for-loop.
10:      end if
11:   end for
12: end for
13: if the test so far has not been terminated then
14:    Set y(O) = 1. Terminate the test.
15: end if
```

root stage and the non-root stage. Algorithm 1 shows the root-stage of the test. There, we consider all possible choices for the first t columns. The first sorting operation results in the first t columns having the root form. The array thus obtained can be subjected to further level permutations in the first t columns; the sorting of the rows can use the precalculated row permutations. The resulting array is then passed on to the non-root stage. If the latter stage finds an array that is lexicographically less than the original, the original is discarded and the test is terminated.

Algorithm 2 details the non-root stage of the LMC test. The required input is an array A and a level p. The non-root stage starts with replacing column p with any of the $n - p + 1$ non-root columns $A(:, p)$ up to $A(:, n)$. All level permutations of the replacing column are checked. If needed, further column permutations are checked by calling the non-root stage recursively in line 12.

The row sorting operations needed in the non-root stage are simple to perform, because the root stage always produces arrays that are already sorted in the root. So the additional sorting needs only to be carried out in blocks of the size of the index λ . This reduces the complexity of sorting from $\mathcal{O}(N \log N)$ to $\mathcal{O}((N/\lambda)(\lambda \log \lambda)) = \mathcal{O}(N \log \lambda)$.

Algorithm 2 Non-root stage of the LMC test. Input: array A, level p. Output scalar $y(A)$: 0=decision taken (not LMC), 1=no decision taken

```
1: At level p do:
2: for i = p ... n do
3:    Let B be the array resulting from swapping column A(:,i) and column
      A(:,p).
4:    for all level permutations of B(:,p) do
5:       Sort B on the rows.
6:       Compare B(:,p) with O(:,p).
7:       if B(:,p) < O(:,p) then
8:          O failed the LMC test; return y(A) = 0.
9:       else if B(:,p) > O(:,p) then
10:         Continue with the containing for-loop.
11:      else
12:         Pass B to the non-root stage at level p + 1.
13:         if y(B) = 0 then
14:            Set y(A) = 0. Terminate non-root stage for A.
15:         end if
16:      end if
17:   end for
18: end for
19: Return y(A) = 1.
```

A key feature of the non-root stage is to skip in an early stage the column permutations that cannot make the array lexicographically smaller; see line 10 of Algorithm 2. Indeed, only if the test at level p is not conclusive, further permutations are invoked.

4 Generation of minimum complete sets

Our LMC test operates on single arrays. So there is no need for a pairwise comparison of arrays. This feature greatly facilitates distribution of calculation tasks over several processors. Accordingly, we implemented the algorithm in a C program that can be run on a single-core computer as well as on a multi-core computer. Even in the single-core version, we believe that the LMC test and the MCS ALGORITHM in general is very fast. The exception to this rule bears on OAs with at least one factor with $s_i \geq 10$. These OAs are hard cases, because the root stage of our LMC test calculates in advance what row permutations are needed to return the root after level permutations in the first t columns of

an array. This amounts to storing $\prod_{i=1}^{t}(s_i)!$ row permutations. Fortunately, all of these cases within the range of OAs generated for this paper can be handled in a different way.

In the next subsection, we compare computing times with the few computing times we were able to find in the literature. We then present enumeration results for a large number of series with $2 \leq t \leq 4$. Our results are either new, or they corroborate enumerations from the literature. We include a detailed discussion that identifies the new results and those obtained earlier.

4.1 Some run-time comparisons

We were able to locate only two papers permitting a comparison of computing times with our method. In the first one, [16], the authors state a computing time of about 365 sec for the comprehensive *Deseq2* method to decide that two specific OA(18; 3^7; 2) were equivalent. We compute all OA(18; 3^a; 2) for $a = 3 \ldots 7$ in less than 1 second. This includes discarding equivalent arrays that are not of LMC form. This is not to be taken as a criticism of [16]. Indeed, the authors capitalized on a thorough overview of test methods rather than on computing speed.

The second reference to compare our computing times with is [4]. The paper presents a clever method to compute complete enumerations of non-isomorphic pure-level orthogonal arrays using integer linear programming. For 36 out of 37 cases in that paper, our program yields the same number of non-isomorphic arrays in much less computing time. The speed gain ranged from a factor of 7 up to a factor of 2122. In addition, the stated computing time of our MCS ALGORITHM for a factor set s^{a_*} also covers MCS generation for all series with $a \leq a_*$. In [4], the computing times bear on $a = a_*$ only.

For OA$(162; 3^6; 4)$, [4] state a computing time of 13.9 sec, where our computing time was 14.2 sec. Our algorithm returned all OA$(162; 3^5; 4)$ (14 solutions) in a few seconds. For the extension to 3^6 arrays, the LMC test was not called, because there were no solutions at all. So the extension algorithm dominates

the search. This may explain the more or less equal computing times for the two approaches.

We like to stress the fact that run-time comparisons are always risky. Indeed, the speed gain of our algorithm may be explained partly by improved computer speed. Nevertheless, our comparisons suggest that our algorithm performs well over a large range of cases.

4.2 Strength-2 arrays

Table 1: Strength-2 results

N	Factor set	$a_{\max}$	Time [min]	Isomorphism classes
4	2^a	3	-	1
8	$4^1 2^a$	4	-	1, 1, 1
8	2^a	7	-	2, 2, 1, 1, 1
9	3^a	4	-	1, 1
12	$6^1 2^a$	2	-	1
12	$3^1 2^a$	4	-	2, 3, 1
12	2^a	11	-	2, 1, 2, 2, 1, 1, 1, 1, 1
16	$8^1 2^a$	8	-	1, 1, 2, 1, 1, 1, 1
16	4^a	5	-	2, 1, 1
16	$4^4 2^a$	3	-	1, 1, 1
16	$4^3 2^a$	6	-	2, 4, 4, 4, 2, 2
16	$4^2 2^a$	9	-	2, 7, 17, 27, 30, 25, 14, 6, 3
16	$4^1 2^a$	12	-	3, 10, 28, 65, 110, 123, 110, 72, 38, 15, 8
16	2^a	15	0.2	3, 5, 11, 27, 55, 80, 87, 78, 58, 36, 18, 10, 5
18	$6^1 3^a$	6	-	2, 3, 1, 1, 1
18	3^a	7	-	4, 12, 10, 8, 3
18	$3^a 2^1$	7	-	3, 15, 48, 19, 12, 3
20	$10^1 2^a$	2	0.2	1
20	$5^1 2^a$	8	-	3, 10, 15, 38, 30, 4, 1
20	2^a	19	1.6	3, 3, 11, 75, 474, 1603, 2477, 2389, 1914, 1300, 730, 328, 124, 40, 11, 6, 3
24	$12^1 2^a$	12	†	1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1
24	$6^1 4^1 2^a$	11	7	3, 38, 400, 2060, 3911, 2200, 1357, 689, 283, 64, 14
24	$6^1 2^a$	14	133	4, 25, 226, 2663, 19323, 58112, 65679, 26454, 12243, 4882, 1543, 277, 45
24	$4^1 3^1 2^a$	13	856	5, 131, 6412, 226330, 2360583, 6467858, 5404183, 2341404, 963413, 350559, 94228, 16238, 1282

continued on next page

Table 1 (continued)

N	Factor set	$a_{\max}$	Time [min]	Isomorphism classes
24	$4^1 2^a$	20	6086	4, 25, 552, 21757, 457768, 3113669, 8168256, 12605571, 15119461, 14961206, 12096092, 7855020, 4066838, 1665918, 532484, 129122, 22880, 2758, 238
24	$3^1 2^a$	16	1391	4, 29, 573, 28745, 1089168, 14576216, 57436095, 71157023, 33893515, 10266252, 3305030, 981180, 220993, 32567, 2282
24	2^a	23	66369	4, 10, 63, 1350, 57389, 1470157, 12952435, 38592861, 52912678, 51154497, 43092737, 31833387, 19960039, 10351396, 4385567, 1502242, 409478, 86725, 13833, 1604, 130
25	5^a	6	0.7	2, 1, 1, 1
27	$9^1 3^a$	9	47	2, 7, 10, 13, 12, 9, 5, 4
27	3^a	13	323	9, 711, 187188, 922548, 157829, 21688, 9793, 3766, 1252, 341, 68
28	$14^1 2^a$	2	†	1
28	$7^1 2^a$	12	10918	4, 25, 371, 21502, 395598, 1422094, 1005490, 135569, 4296, 104, 21
28	2^a	27	?	4, 7, 127, 17826, 5882186, . . . , 7570

NOTES: -: computing times less than one second; ?: series are not completed; †: series not obtained by direct calculation.

For $t = 2$, we obtained all mixed-level series with $N \leq 28$, and all pure-level series with $N \leq 27$. In addition, we obtained all $\mathrm{OA}(28; 2^a; 2)$ for $a \leq 7$. Table 1 presents the results. There are 33 'cases', identified by their run sizes N, and their factor sets. There can be one or more series per case; these differ in the number of factors for one of the levels in the factor set. The tabulated results are the maximum numbers of factors with that factor-level ($a_{\max}$), the computing times of the complete case, and the numbers of isomorphism classes. The numbers are ordered from right to left, the right most number bearing on $a_{\max}$, the previous one on $a_{\max} - 1$, etc.

$\mathrm{OA}(24; 12^1 2^a; 2)$ and $\mathrm{OA}(28; 14^1 2^2; 2)$ were not obtained by direct enumeration. For the first case, see Appendix 2. Regarding the second case, the single isomorphism class as well as the value $a_{\max} = 2$, follows from [27].

The computing times in Table 1 will decrease with growing computer power. They were included to distinguish easy and hard cases. At this moment, the

implementation performs well for mixed orthogonal arrays of strength 2 and up to 28 runs. The $\mathrm{OA}(28; 2^a; 2)$ for $a \geq 8$ were not obtained, however. The estimated time for the complete case is roughly 24,000 years. Note that some series with $N > 28$ may still be obtained for small numbers of factors.

For the 24-run cases with factor sets $6^1 4^1 2^a, 6^1 2^a, 4^1 3^1 2^a$, and $3^1 2^a$, the values of $a_{\max}$ were previously unknown [18]. Therefore, the number and nature of the so-called dual atoms of the lattice Λ_{24} of 24-run orthogonal arrays [18] were also unknown. Dual atoms indicate factor sets such that, taking one array of each of these sets, an array of all remaining factor sets can be obtained by deleting columns or replacing the four-level factor with three two-level factors [18]. We now establish that the values of $a_{\max}$ are 11, 14, 13, and 16. This implies that the dual atoms are $12^1 2^{12}, 6^1 4^1 2^{11}, 4^1 3^1 2^{13}$, and $4^1 2^{20}$.

For the $\mathrm{OA}(28; 7^1 2^a)$, we establish for the first time that $a_{\max} = 12$. This results makes the lattice Λ_{28} completely known. Its dual atoms are $14^1 2^2, 7^1 4^1, 7^1 2^{12}$, and 2^{27}.

For all values of $a_{\max}$ in Table 1, the web-site [22] gives one explicit array and a reference to papers where this array was obtained.

We will now discuss the number of isomorphism classes stated in Table 1. All classes for the factor sets 2^3 and 2^4 and any N are given by [21] and [24], respectively. The results on $N \leq 9$ and tabulated in Table 1 are well-known. The $\mathrm{OA}(4; 2^2; 2)$ and $\mathrm{OA}(8; 2^7; 2)$ are unique; see Theorem 7.37 of [13]. The latter result explains one of the isomorphism classes for each of $\mathrm{OA}(8; 2^a; 2)$ with $a \leq 7$. The two isomorphism classes for $\mathrm{OA}(8; 2^3; 2)$ are the full factorial and the replicated $\mathrm{OA}(4; 2^3; 2)$. The two isomorphism classes for $\mathrm{OA}(8; 2^4; 2)$ are the half fraction of a full factorial, and the replicated $\mathrm{OA}(4; 2^3; 2)$ with one additional factor. The unique $\mathrm{OA}(8; 4^1 2^4; 2)$ can be constructed from $\mathrm{OA}(8; 2^7; 2)$ by replacing two columns and their sum modulo 2 with a four-level factor. Removing two-level columns gives the complete case of $\mathrm{OA}(8; 4^1 2^a; 2)$. Finally, the single isomorphism class of $\mathrm{OA}(9; 3^3; 2)$ follows form the uniqueness of a Latin Square of order 3; see [13], Table 8.47. The unique $\mathrm{OA}(9; 3^4; 2)$ follows form

Theorem 8.3 in the same reference.

There has been much recent interest in cases with $12 \leq N \leq 20$. The single isomorphism class for $\mathrm{OA}(4k; 2k \times 2^{a_{\max}}; 2)$ for odd k, as well as the value $a_{\max} = 2$, follows from [27]. The pure two-level arrays have been enumerated completely by [25]. A classification of all 18-run orthogonal arrays with statistical criteria is given in [20]. The $3^1 2^a$ series with $N = 12$, the $3^a 2^1$ cases with $N = 18$, and the $5^1 2^a$ cases with $N = 20$ were established independently by Ye et al. [29]. The 3^a series was established by [9]. As far as we know, the results on the $4^a 2^b$ cases with $a \geq 1, b \geq 1$ are new. For the 4^a case, see [13].

Until now, the numbers of isomorphism classes for the mixed-level cases with $24 \leq N \leq 28$ were not known, except for the trivial $\mathrm{OA}(28; 14^1 2^2)$. For $\mathrm{OA}(24; 2^a; 2)$, [1] presented enumeration results for $a \leq 7$. The 130 isomorphism classes for $a = 23$ were first given in [13], Theorem 7.37. For $\mathrm{OA}(28; 2^a; 2)$, [1] give the number of isomorphism classes for $a \leq 6$. The 7570 classes for $a = 27$ were obtained by Sloane; see sequence A048885 of [23]. The $\mathrm{OA}(25; 5^a; 2)$ case for $a = 2$ follows from Table 8.47 of [13]. The results for $a = a_{\max}$ follows form Theorem 8.3 and 8.28 of the same reference. Finally, the 68 $\mathrm{OA}(27; 3^{13}; 2)$ were first given by [17].

4.3 Strength-3 arrays

For $t = 3$, we obtained all but a few cases with $N \leq 64$. Exceptions are the $\mathrm{OA}(56; 2^a; 3)$, $\mathrm{OA}(64; 2^a; 3)$, and $\mathrm{OA}(64; 4^1 2^a; 3)$. These cases were obtained only for a up to 8, 6, and 6, respectively. The incompleteness of these results comes as no surprise, because a complete enumeration of the the derived $\mathrm{OA}(28; 2^a; 2)$, $\mathrm{OA}(32; 2^a; 2)$, and $\mathrm{OA}(32; 4^1 2^a; 2)$ was not reached either. The estimated time to completion for these cases is roughly 50,000 years (both pure two-level cases) or 1000 years (mixed-level case).

Table 2 presents the enumeration results. There are 38 cases in total, dis-

Table 2: Strength-3 results

N	Factor set	$a_{\max}$	Time [min]	Isomorphism classes
8	2^a	4	-	1
16	$4^1 2^a$	3	-	1
16	2^a	8	-	2, 2, 1, 1, 1
24	$6^1 2^a$	3	-	1
24	$3^1 2^a$	4	-	2, 3
24	2^a	12	-	2, 1, 2, 1, 1, 1, 1, 1, 1
27	3^a	4	-	1
32	$8^1 2^a$	3	-	1
32	$4^2 2^a$	4	-	2, 2, 2
32	$4^1 2^a$	7	-	3, 7, 7, 11, 8
32	2^a	16	4.9	3, 5, 10, 17, 33, 34, 32, 22, 23, 12, 10, 5, 5
36	$3^2 2^a$	2	-	3
40	$10^1 2^a$	3	2.0	1
40	$5^1 2^a$	6	-	3, 7, 1, 1
40	2^a	20	4.8	3, 3, 9, 25, 105, 213, 353, 260, 235, 132, 96, 36, 26, 7, 6, 3, 3
48	$12^1 2^a$	3	†	1
48	$6^1 4^1 2^a$	2	-	3
48	$6^1 2^a$	7	0.6	4, 14, 30, 74, 45
48	$4^1 3^1 2^a$	4	-	5, 35, 19
48	$4^1 2^a$	11	22	4, 4, 24, 98, 387, 1362, 2595, 2217, 560
48	$3^1 2^a$	9	727	4, 21, 134, 938, 3056, 5018, 3
48	2^a	24	74232	4, 10, 45, 397, 8383, 166081, 1310006, 3528089, 4460865, 3980095, 3139653, 2165144, 1288460, 629705, 259346, 84495, 24012, 4919, 1129, 130, 60
54	$6^1 3^a$	3	-	2
54	$3^a 2^1$	5	0.2	5, 33, 4
54	3^a	5	-	7, 4
56	$14^1 2^a$	3	†	1
56	$7^1 2^a$	6	0.6	4, 14, 7, 10
56	2^a	28	?	4, 7, 86, 4049, 757190, ...
60	$5^1 3^1 2^a$	2	-	6
64	$16^1 2^a$	3	†	1
64	$8^1 4^1 2^a$	2	0.3	4
64	$8^1 2^a$	7	727	5, 26, 192, 1146, 924
64	4^a	6	0.5	5, 1, 1
64	$4^5 2^a$	2	0.4	1, 1
64	$4^4 2^a$	6	0.4	3, 5, 3, 3, 1, 1
64	$4^3 2^a$	8	1.9	10, 107, 237, 255, 126, 35, 12, 2
64	$4^2 2^a$	12	873	12, 267, 13903, 104949, 175297, 151708, 138825, 83409, 35807, 10030, 2159
64	$4^1 2^a$	15	?	7, 75, 4101, 251617, ...
64	2^a	32	?	5, 19, 358, 91789, ...

played in the same way as in Table 1. The OA$(48; 12^1 2^a; 3)$, and the OA$(64; 16^1 2^a; 3)$ were not obtained with our algorithm; see Appendix 2.

The computing times in Table 2 shows that the current implementation performs well for orthogonal arrays of $t = 3$ and $N \leq 64$, with the exception of the missing cases mentioned earlier. Note that some series with $N > 64$ may still be obtained for moderate numbers of factors.

For all cases mentioned in Table 2, [3] derive the value of $a_{\max}$ and give one explicit construction for arrays with $a = a_{\max}$. The same authors give the number of non-isomorphic OA$(48; 4^1 3^1 2^4; 3)$ and OA$(54; 3^5 2^1; 3)$. The four OA$(54; 3^5; 3)$ were derived by [12]. All classes for the factor sets 2^4 and 2^5 and any N are given by [21] and [24], respectively. The uniqueness of OA$(27; 3^4; 3)$ was proven by [14]. Results on OA$(4s; s^1 2^a; 3)$ are trivial. Bulutoglu and Margot [4] obtained all OA$(N; 2^a; 3)$ for $N = 24$, 32, 40, and 48, and values of a from 6 up to 11, 11, 10, and 8, respectively. Finally, the classes for OA$(16; 2^a; 3)$ and $6 \leq a \leq 8$ follow from folding over the unique OA$(8; 2^{a-1}; 2)$.

As far as we can see, all numbers of isomorphism classes in Table 2 not covered with the above paragraph are new.

4.4 Strength-4 arrays

Table 3: Strength-4 results

N	Factor set	$a_{\max}$	Time [min]	Isomorphism classes
16	2^a	5	-	1
32	$4^1 2^a$	4	-	1
32	2^a	6	-	2, 2
48	$6^1 2^a$	4	-	1
48	$3^1 2^a$	5	-	2, 3
48	2^a	5	-	2
64	$8^1 2^a$	4	0.1	1
64	$4^2 2^a$	3	-	2
64	$4^1 2^a$	6	-	3, 7, 2
64	2^a	8	-	3, 5, 7, 3
72	$3^2 2^a$	3	-	3
80	$10^1 2^a$	4	-	1
80	$5^1 2^a$	5	-	3, 7

continued on next page

Table 3 (continued)

N	Factor set	$a_{\max}$	Time [min]	Isomorphism classes
80	2^a	6	-	3, 1
81	3^a	5	-	1
96	$12^1 2^a$	4	†	1
96	$6^1 4^1 2^a$	3	-	3
96	$6^1 2^a$	5	0.5	4, 14
96	$4^1 3^1 2^a$	4	-	5, 35
96	$4^1 2^a$	5	-	4, 1
96	$3^1 2^a$	7	0.3	4, 21, 64, 5
96	2^a	7	-	4, 9, 4
108	$3^3 2^a$	2	-	5
112	$14^1 2^a$	4	†	1
112	$7^1 2^a$	5	3.7	4, 14
112	2^a	6	-	4, 3
120	$5^1 3^1 2^a$	3	-	6
128	$16^1 2^a$	4	†	1
128	$8^1 2^a$	8	239	5, 26, 19, 11, 10
128	$8^1 4^1 2^a$	3	2.9	4
128	$4^3 2^a$	3	0.2	10, 6
128	$4^2 2^a$	6	4.5	12, 198, 202, 77
128	$4^1 2^a$	9	244	7, 75, 870, 2649, 1014, 275
128	2^a	15	4372	5, 17, 123, 1153, 3632, 1479, 400, 2, 1, 1, 1
144	$18^1 2^a$	4	†	1
144	$9^1 2^a$	5	709	5, 26
144	$6^2 2^a$	3	4.3	6
144	$6^1 2^a$	4	-	6
144	$6^1 3^1 2^a$	4	16	10, 265
144	$4^1 3^2 2^a$	2	-	18
144	$3^2 2^a$	5	8333	9, 319, 11232
144	$3^1 2^a$	6	181	6, 78, 130
144	2^a	8	229	5, 7, 35, 20
160	$20^1 2^a$	6	†	1
160	$10^1 2^a$	6	†	6, 45
160	$5^1 4^1 2^a$	5	291	26, 2277, 28
160	$4^1 2^a$	7	14642	9, 43, 1965, 534
160	$5^1 2^a$	7	29648	9, 230, 11016, 19153
160	2^a	≤ 17	?	6, 29, 450, 99618, ...
162	3^a	5	0.2	14
162	$3^5 2^a$	1	1.6	14, 46
162	$3^4 2^a$	1	-	13
168	$7^1 3^1 2^a$	3	0.3	12

NOTES: -: computing times less than one second; ?: series are not completed; †: series not obtained by direct calculation.

For $t = 4$, we completed all cases but one with $N \leq 168$. The case $\mathrm{OA}(160; 2^a; 4)$ was obtained only for $a \leq 8$. Table 3 presents the results. There are 48 cases in total, displayed in the same way as in Table 1. The cases of $\mathrm{OA}(8s; s^1 2^a; 4)$ and $\mathrm{OA}(160; 10^1 2^a; 4)$ were derived in a different way, detailed in Appendix 2.

The computing times in Table 3 shows that the current implementation performs well for orthogonal arrays of strength $t = 4$ and up to 168 runs, with the exception of $\mathrm{OA}(160, 2^a; 4)$. The estimated time to completion of this case is roughly 10,000 days. The last case in the table illustrates that some series with $N > 160$ may still be obtained for moderate numbers of factors.

The values of $a_{\max}$ for pure two-level arrays of strength $t = 4$ and index $1 \leq \lambda \leq 5$ were established by [21]. The $a_{\max}$ for $\lambda = 8$ is given in [13]. The values for $\lambda = 6, 7$, and 9 were established by [4]. The first open case is for $\lambda = 10$. The stated upper bound $a_{\max} \leq 17$ is the Rao bound.

For the pure three-level arrays with $\lambda = 1$ or 2, the values of $a_{\max}$ follow from Table 12.2 in [13].

As to the numbers of isomorphism classes, the factor sets 2^5 and 2^6 were enumerated by [21] and [24], respectively. Bulutoglu and Margot [4] enumerated all pure two-level arrays of run-sizes $N = 64, 80, 96, 112$, and 144. The 81-run case was derived in [14]. Apart from these cases, the results in Table 3 are new.

5 Discussion

To our knowledge, the algorithm proposed in this paper is the first one to handle general orthogonal arrays of given strength, run-size and level-numbers of the factors. It features element-wise addition of symbols in a column, and isomorphism testing performed on single arrays.

For growing run-size, element-wise addition has advantages over addition of a new column balanced for the first $t - 1$ columns ([26], [25], [9]). For example, the addition of two-level columns to $\mathrm{OA}(24;\ 3^1 2^2;\ 3)$ requires evaluation of 46656 columns. For the $\mathrm{OA}(48;\ 3^1 2^2;\ 3)$, this number grows to more than 117 billion. For adding a three-level column to $\mathrm{OA}(54;\ 3^3;\ 3)$ the number grows to 3.87×10^{17}. Element-wise addition is clearly useful in these cases.

The isomorphism testing performed on single arrays has the advantage that it permits a distribution of the calculation over several processors. Given a set

of, say, m arrays that have to be subjected to isomorphism testing, a pairwise testing procedure could involve $m-1 \leq c \leq 0.5m(m-1)$ tests. For the single-array testing, $c = m$. On the other hand, the single-array testing requires that the testing is comprehensive, while the pairwise testing need only to be applied within groups that are homogeneous with respect to initial classification criteria.

The enumeration results in the Tables 1, 2, and 3 were set up to contain the limiting run-size N_l for $t = 2, 3$, and 4. N_l is such that the isomorphism classes of all series with $N < N_l$ can all be generated, while this is not the case for the limiting run size itself. For $t = 2, 3$, and 4, $N_l = 28, 56$, and 160, respectively. These run-sizes are limiting in view of the very large number of possible pure two-level arrays. The tables also show that it is well possible to obtain complete series for run sizes $N \geq N_l$, especially if the factor set contains multi-level factors. To identify critical sections in the current implementation, we break down the running time of a series into (1) the extension algorithm, (2) the LMC check for arrays not of LMC form, and (3)LMC check for arrays of LMC form. As an example, Table 4 presents the distribution of computing times for OA$(27; 3^a; 2)$. The table has computing times for the three main components, and the total generation. The total is larger than the sum for the main components because of input/output and communication tasks. All the

Table 4: Distribution of computing times in OA$(27; 3^a; 2)$

a	Time (sec)				# arrays	
	LMC+	LMC-	extension	total	n+	n-
3	0.00	0.00	0.00	0.01	9	378
4	0.59	2.34	0.99	3.95	711	146589
5	373.53	203.48	62.43	646.19	187188	1071410
6	3516.03	1451.97	4380.38	9394.34	922548	2023895
7	1059.08	357.86	5958.09	7413.85	157829	229279
8	291.09	102.21	495.94	895.85	21688	46180
9	237.72	87.55	39.17	365.82	9793	22676
10	156.14	64.13	11.81	232.69	3766	9617
11	92.60	26.24	2.86	122.01	1252	3583
12	41.76	4.63	0.63	47.10	341	565
13	29.57	0.65	0.11	30.35	68	52
14	0.00	0.00	0.02	0.02	0	0

times bear on extension and testing from arrays with one column less. The last two columns count the number of LMC and non-LMC arrays generated. We see that even within one case the distribution of the total computing time over the components can vary substantially.

We have three main candidates for improving the N_l of our MCS ALGORITHM. Firstly, we may be able to use a partial LMC test after completing half of the column (or some other fraction). If the test is negative, further extension is blocked. This could speed up the extension itself and the LMC check for non-LMC arrays. Secondly, results of previous LMC rejections may improve the next array or block further generation or testing. Finally, we feel that an improved sorting time should be possible by using the structure present in the array. All these improvements can help to reduce the time for generating the arrays and testing non-LMC arrays. For the testing of arrays of LMC form, there are no major ideas for improvement. Indeed, there is no help for the combinatorial explosion of arrays of given strength when run-sizes increase.

Using an algorithm to generate series of arrays gives little information on succinct ways to construct individual arrays. Knowledge of an explicit construction method may well lead to more insight into the best way to analyse the results of an experiment. We would therefore welcome research on such methods.

As a final note, our LMC test can be changed such that it transforms an array to the LMC representative of its class. This may provide a quick isomorphism test for any pair of arrays. We investigated this idea by switching each of the 130 LMC OA$(24, 2^2 3; 2)$ to a randomly chosen array from its isomorphism class. It took 433 sec to transform the 130 arrays back to LMC form. It takes 21 sec to check whether the 130 LMC arrays are indeed LMC. So the reduction to LMC form seems to take about 20 times longer than the LMC check. As the LMC check is quite fast, we consider the isomorphism test based on the LMC test as a promising issue for further research.

Acknowledgements

The research leading to this paper was started when EDS and MVMN were at Eindhoven University of Technology in the Netherlands. The authors are grateful to Andries Brouwer for an original computer implementation of the algorithm for specific strength-3 cases, to Arjeh Cohen and the late Jan Dijkstra for their encouragement, and to Ruben Snepvangers and Vincent Brouérius van Nidek for a parallel-computer implementation handling general cases. Further research of the first author was supported by a grant from the Flemish Foundation for Scientific Research FWO.

Appendix 1: Proof of correctness

Using the terminology of [15], henceforth KO, we define the domain of the search to be the root node joined with all child nodes from an LMC parent. It is convenient to define a search tree as a rooted tree whose nodes are objects in the domain of the search. Two nodes are joined by an edge if and only if they are related by one search step. The following theorem ensures that $\rho(X) \in C(p(X))$, where $\rho(X)$ is the LMC representative of the isomorphism class to which X belongs, $p(X)$ is the parent of node X, and $p(X)$ is in LMC form.

Theorem 1 *The* Extension Algorithm *produces nodes X for which $\rho(X) \in C(p(X))$, if $p(X)$ is in LMC form.*

Proof. Consider the parent node $p(X)$. An LMC child node must have $v_{1c} = 0$. This is ensured by Step 1 of the algorithm. Step 2 prevents generation of an array that is lexicographically higher than is necessary. Otherwise, the algorithm permits all possible choices of elements that are compatible with the required strength. □

The MCS Algorithm considers $C(X)$ only if X is of LMC form. So an

array is augmented only if its parent is LMC. KO call this way of generating the arrays orderly generation. Their Theorem 4.20 states that if the orderly generation is implemented on a search tree that satisfy two conditions, it reports exactly one node from each isomorphism class of nodes, which is what we want to prove. The conditions are (1) the search tree has nodes X for which $\rho(X) \in C(p(X))$, and (2) for every LMC node X in the search tree, $p(X)$ is also LMC. Condition (1) is ensured by Theorem 1 above. For condition (2), we note that the objects in the search domain can be considered as $N.n$ tuples over an alphabet of $\max\{s_i\}$ symbols and the additional symbol $\diamond$ for entries in columns yet undefined. Now, define the lexicographical order of symbols from the smallest to the largest element to be $0, 1, \ldots, (\max\{s_i\} - 1), \diamond$. Further, define the lexicographic significance of a position in the tuple as its order number. We rephrase Theorem 4.24 of KO as

Theorem 2 *For every nonroot node x and for a search tree algorithm constructing $N.n$-tuples in order of lexicographic significance: if $p(x) \neq \rho(p(x))$ then $x \neq \rho(x)$.*

By noting that our algorithm does construct the tuples in increasing order of lexicographical significance, we see that Theorem 2 applies. Condition (2) above is just the reversal of the implication statement in the theorem. We conclude that the algorithm reports exactly one node from each isomorphism class of nodes, and therefore, exactly one array of each isomorphism class.

Appendix 2: Mixed series not covered by the implementation

The implementation of our algorithm cannot handle level-numbers larger than 9. So some series in Tables 1, 2, and 3 were enumerated in an indirect way.

First, the series $\mathrm{OA}(24; 12 \times 2^a; 2)$ can be enumerated by noting the fact that each derived $\mathrm{OA}(24; 2^a; 2)$ must be a fold-over array. As a fold-over array

must have $t = 3$, we can construct all arrays for $a \geq 2$ by checking which of the OA$(24; 2^a; 3)$ have a fold-over structure. For $a = 2$ an $a = 3$ this is trivial, because these arrays are six or 3 copies of a full factorial design and these are fold-over designs. These designs are unique. For $4 \leq a \leq 12$, all OA$(24; 2^a; 3)$ from Table 2 are fold-over arrays. There are two isomorphism classes for $a = 4$ and $a = 6$; there is one class for each of the remaining values of a. We conclude that the numbers of isomorphism classes for OA$(24; 12 \times 2^a; 2)$ and $a \geq 2$ are given by 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, and 1.

The second case not completely covered by our implementation is $OA(4s; s \times 2^a; 3)$ for $4s \geq 48$. For each level of the s-level factor, these arrays have one copy of an OA$(4; 2^a; 2)$. There is a single such array for $a = 3$. So the two-level part of the required series must consist of an even number of copies of this design an the same number of fold-over copies. The number of isomorphism classes is thus 1.

The third cases is OA$(8s; s \times 2^a; 4)$. The number of isomorphism classes is found by an argument similar to the above one.

Finally, generation of OA$(160; 10^1 2^5; 4)$ was performed as usual. However, we made the root used in the LMC check consisting of two columns only. In this way, we could store all row permutations as required for the root stage. There were 6 isomorphism classes. We generated extensions with a sixth two-level column without invoking the LMC check. This resulted n 4378 arrays. We split the 10-level column in a two-level and a five-level column to obtain OA$(160, 2 \times 5 \times 2^6; 4)$. If any of these arrays is not LMC, then neither is the parent array with the 10-level factor. After the LMC check 47 arrays were retained. These were subjected to a full LMC test, using a root of just the first two columns for the root stage. A total of 45 arrays remains. None of these are extendible.

References

[1] P. Angelopoulos, H. Evangalaras, C. Koukouvinos, and E. Lappas. An effective step-down algorithm for the construction and the identification of nonisomorphic orthogonal arrays. *Metrika*, 66:139–149, 2007.

[2] R. Block and R. Mee. Resolution iv designs with 128 runs. *Journal of Quality Technology*, 37:282–293, 2005.

[3] A.E. Brouwer, A.M. Cohen, and M.V.M. Nguyen. Orthogonal arrays of strength 3 and small run sizes. *Journal of Statistical Planning and Inference*, 136:3268–3280, 2006.

[4] D.A. Bulutoglu and F. Margot. Classification of orthogonal arrays by integer programming. *Journal of Statistical Planning and Inference*, 138:654–666, 2008.

[5] J. Chen, D.X. Sun, and C.F.J. Wu. A catalogue of two-level and three-level fractional factorial designs with small runs. *International Statistical Review*, 61:131–145, 1993.

[6] J.B. Clark and A.M. Dean. Equivalence of fractional factorial designs. *Statistica Sinica*, 11:537–547, 2001.

[7] L.Y. Deng and B. Tang. Generalized resolution and minimum aberration criteria for plackett-burman and other nonregular factorial designs. *Statistica Sinica*, 9:1071–1082, 1999.

[8] H. Evangelaras, C. Koukouvinos, and E. Lappas. An efficient algorithm for the identification of isomorphic orthogonal arrays. *Journal of Discrete Mathematical Science and Cryptography*, 9:125–132, 2006.

[9] H. Evangelaras, C. Koukouvinos, and E. Lappas. 18-run nonisomorphic three level orthogonal arrays. *Metrika*, 66:31–37, 2007.

[10] H. H. Evangelaras, C. Koukouvinos, A.M. Dean, and C.A. Dingus. Projection properties of certain three level orthogonal arrays. *Metrika*, 62:241–257, 2005.

[11] H. H. Evangelaras, C. Koukouvinos, and E. Lappas. Further contributions to nonisomorphic two level orthogonal arrays. *Journal of Statistical Planning and Inference*, 137:2080–2086, 2007.

[12] A.S. Hedayat, E. Seiden, and J. Stufken. On the maximal number of factors and the enumeration of 3-symbol orthogonal arrays of strength 3 and index 2. *Journal of Statistical Planning and Inference*, 58:43–63, 1997.

[13] A.S. Hedayat, N.J.A. Sloane, and J. Stufken. *Orthogonal arrays : theory and applications*. Springer, 1999.

[14] A.S. Hedayat, J. Stufken, and G. Su. On the construction and existence of orthogonal arrays with three levels and indexes 1 and 2. *Annals of Statistics*, 25:2044–2053, 1997.

[15] P. Kaski and P.R.J. Östergård. *Classification Algorithms for Codes and Designs*. Springer-Verlag, New York, NY, USA, 2005.

[16] T.I. Katsaounis and A.M. Dean. A survey and evaluation of methods for determination of combinatorial equivalence of factorial designs. *Journal of Statistical Planning and Inference*, 138:245–258, 2008.

[17] C. Lam and V.D. Tonchev. Classification of affine resolvable 2-(27, 9, 4) designs. *Journal of Statistical Planning and Inference*, 56:187–202, 1996.

[18] E.M. Rains, N.J.A. Sloane, and Stufken J. The lattice of n-run orthogonal arrays. *Journal of Statistical Planning and Inference*, 102:477–500, 2002.

[19] C.R. Rao. Factorial experiments derivable from combinatorial arrangements of arrays. *Journal of the Royal Statistical Society Supplement*, 9:128–139, 1947.

[20] E.D. Schoen. All orthogonal arrays with 18 runs. *Quality and Reliability Engineering International*, 25:467–480, 2009.

[21] E. Seiden and R. Zemach. On orthogonal arrays. *Annals of Mathematical Statistics*, 37:1355–1370, 1966.

[22] N.J.A. Sloane. A library of orthogonal arrays. `http://www.research.att.com/~njas/oadir/`.

[23] N.J.A. Sloane. The on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/`.

[24] J. Stufken and B. Tang. Complete enumeration of two-level orthogonal arrays of strength d with $d + 2$ constraints. *Annals of Statistics*, 35:793–814, 2007.

[25] D.X. Sun, W. Li, and K.Q. Ye. An algorithm for sequentially constructing nonisomorphic orthogonal designs and its applications. Technical report, Department of Applied Mathematics and Statistics, SUNY at Stony Brook, 2002.

[26] P.W. Tsai, S.G. Gilmour, and R Mead. Projective three-level main effects designs robust to model uncertainty. *Biometrika*, 87:467–475, 2000.

[27] J.C. Wang and C.J.F. Wu. Nearly orthogonal arrays with mixed levels and small runs. *Technometrics*, 34:409–422, 1992.

[28] H. Xu. A catalogue of three-level regular fractional factorial designs. *Metrika*, 62:259–281, 2005.

[29] K.Q. Ye, D. Park, W. Li, and A.M. Dean. Construction and classification of orthogonal arrays with small numbers of runs. *Statistics and Applications*, 6:1–9, 2008.