

Finding Orthogonal Arrays Using Satisfiability Checkers and Symmetry Breaking Constraints*

Feifei Ma^{1,2} and Jian Zhang¹

¹ State Key Laboratory of Computer Science

Institute of Software, Chinese Academy of Sciences

² Graduate University, Chinese Academy of Sciences

{maff,zj}@ios.ac.cn

Abstract. Orthogonal arrays are very important combinatorial objects which can be used in software testing and other areas. Mathematical methods for constructing such arrays have been studied extensively in the past decades. In contrast, computer search techniques, in particular exhaustive search methods, are rarely used to solve the problem. In this paper, we present an algorithm which finds orthogonal arrays of given sizes or shows their non-existence. The algorithm is essentially a backtrack search procedure, but enhanced with some novel symmetry breaking (isomorphism elimination) techniques. The orthogonal array is generated column by column, and the constraints are checked by an efficient SAT solver or pseudo-Boolean constraint solver. We have implemented a tool called BOAS (Backtrack-style OA Searcher) using MiniSat and PBS. Experimental results show that our tool can find many orthogonal arrays quickly, especially those with strength higher than 2.

1 Introduction

The concept of **orthogonal arrays**, since introduced by C. R. Rao (1947), plays an important role in factorial designs in which each treatment is a combination of factors at different levels. Roughly speaking, an orthogonal array (OA) of strength t is an array with the property that all the ordered combinations of t symbols from different columns occur equally often in rows. As a combinatorial design with beautiful balancing property, OA has long been the interest of mathematicians. In addition, OAs are also frequently used in industrial experiments for quality and productivity improvement.

In recent years, OAs have also attracted the attention of researchers and engineers in software testing [3]. The Orthogonal Array Testing Strategy (OATS), a technique employing OA to represent uniformly distributed variable combinations, is very useful for integration testing of software components. It has been adopted by big companies like AT&T [12] and Motorola [9].

There are many mathematical results about OAs, either dealing with their construction or proving their non-existence given some parameters. However,

* This work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 60673044 and 60633010.

these mathematical results do not cover all cases. Sometimes we can only resort to computer search. So far, computer search methods for finding OAs have not received much attention. In this paper, we propose an exhaustive search method for finding OAs of given sizes. It integrates an efficient solver for the Boolean SATisfiability problem (SAT solver) and a solver for pseudo-Boolean constraints. In addition, it employs several novel symmetry breaking (isomorphism elimination) techniques. Experimental results show that the algorithm performs quite well in many cases, especially for finding OAs with high strengths.

The paper is organized as follows. In Section 2, we give a brief introduction to OAs. Then the search algorithm is presented in Section 3. In Section 4, we propose several heuristics for eliminating isomorphism. In Section 5, we give some experimental results. Finally, we discuss related works and possible improvements to our approach. Due to space limitation, we omit the detailed proofs for theorems and propositions. They can be found in our technical report [11].

2 Orthogonal Arrays

An **orthogonal array** (OA) of **run size** N , **factor number** k , **strength** t can be denoted by $\text{OA}(N, s_1, s_2, \dots, s_k, t)$. It is an $N \times k$ matrix satisfying:

1. There are exactly s_i symbols appearing in each column i , $1 \leq i \leq k$.
2. In every $N \times t$ sub-array, each ordered combination of symbols from the t columns appears equally often in rows.

We refer to s_i as the **level** of factor i . When the levels of an OA are not all equal, we also call it a **mixed orthogonal array** (MOA). By combining equal entries in $\{s_i\}$, we may represent an $\text{OA}(N, s_1, s_2, \dots, s_k, t)$ in the shortened form $\text{OA}(N, s_{i_1}^{a_1} \cdot s_{i_2}^{a_2} \dots, t)$, where the exponents a_1, a_2, \dots indicate the number of factors at level s_{i_1}, s_{i_2}, \dots , respectively. For instance, an $\text{OA}(16, 4, 2, 2, 2, 3)$ can also be written as $\text{MOA}(16, 4.2^3, 3)$. In the following sections, we use the terms “run” and “row”, “factor” and “column” interchangeably.

The symbols in an OA can be chosen arbitrarily, for example, a, b, c, \dots . But conventionally, we often use the natural numbers $0, 1, 2, \dots$. A simple example is given in Figure 1. It is an instance of $\text{OA}(8, 2^4, 3)$. In each 8×3 sub-array, each ordered combination of symbols (e.g., $(1, 0, 1)$) occurs exactly once.

OAs could be constructed by various mathematical means, e.g., Hadamard construction, juxtaposition, splitting, zero-sum, etc. There are also some bounds of parameters concerning the existence, the most famous among those being Rao’s generalized bound. For more details, one could refer to [5] or [7]. OAs of strength 2 have been studied extensively mainly through such methods. Brouwer et al. [4] discussed OAs of strength 3 and small run sizes in a similar way. As for OAs of higher strengths, there seem to be much fewer results.

3 The Search Procedure

We apply exhaustive search techniques to find (M)OAs of given sizes. The framework of our approach is described in this section.

3.1 The Basic Procedure

The approach is based on backtracking search, finding the orthogonal array column by column. It can be described as a recursive procedure like the following.

```
bool Colsch(j){
  cons = CONS_GEN(j, oaSol);
  while(true){
    column = SOLVE(cons);
    if(column == null) return FALSE;
    APPEND(column, oaSol);
    if(j == k) return TRUE;
    if(Colsch(j+1)) return TRUE;
    add NEGATE(column) to cons;
  }
}
```

Suppose we are processing column j and represent the obtained partial solution by `oaSol`. The function `CONS_GEN` obtains some constraints (denoted by `cons` in the pseudo-code) which are both necessary and sufficient for the current column to be consistent with the OA definition. Then the function `SOLVE` resorts to a pseudo-Boolean constraint solver or a SAT solver to find a solution (denoted by `column`). If there is no solution for the current column, the algorithm has to backtrack to the previous column and try another solution; otherwise the recently found column is added to the partial array `oaSol` by the function `APPEND`. When all the k columns are generated, a solution is obtained and the function returns `TRUE`. If the array is not completed, the procedure is executed recursively.

Our algorithm integrates a satisfiability checker as a search engine for solving part of the problem. The checker might be called a number of times, to find different solutions to a column. In the pseudo-code, the negation of the current solution, obtained by the function `NEGATE`, is added to the input file `cons` to guide the checker to find new solutions to the column.

One might wonder why we do not ask the satisfiability checker to do the whole search. The reason is that if an OA specification is directly encoded into SAT, there would be too many variables and clauses. It is difficult to describe the unique distribution of different symbol combinations in every t columns. Of course, if each symbol combination appears exactly once, there is an easy way. For example, k mutually orthogonal Latin squares of order N are equivalent to $OA(N^2, N^{k+2}, 2)$ and SAT solvers have been applied directly to the problem in the literature [2]. Efficiency is another issue. In Section 3 of reference [17], a direct SAT encoding of covering arrays is discussed, and it is reported that much time is needed to solve some small cases even after symmetry breaking.

In the next subsection, we shall discuss a trick which allows us to determine the first t columns of `oaSol`. Thus the search procedure begins at column $t + 1$, and the first call of the recursive function is `Colsch(t+1)`.

3.2 Preprocessing

Without loss of generality, we can assume in an OA problem $(N, s_1, s_2, \dots, s_k, t)$, the levels s_1, s_2, \dots, s_k are sorted in non-increasing order. Noticing the fact that in the first t columns all combinations of s_1, \dots, s_t symbols should appear $\lambda = \frac{N}{s_1 \times \dots \times s_t}$ times, we can generate the first t columns directly by enumerating all possibilities, each of which λ times. We call the generated partial array **init-block**. For example, the 8×3 sub-array formed by the first three columns of Figure 1 is an init-block.

3.3 Constraint Generation

Assume that we have constructed m columns ($m \geq t$). How can we generate the constraints for column $m + 1$? Our method is based on an observation in [7]:

Remark 1. An $OA(N, s_1, s_2, \dots, s_k, t)$ is also an $OA(N, s_1, s_2, \dots, s_k, t - 1)$.

If we extract $t - 1$ columns (denoted by $C_{i_1}, C_{i_2}, \dots, C_{i_{t-1}}$) from the matrix and partition the row numbers by the row vectors, i.e. putting the row numbers into the same set if the row vectors are identical, we can get $s_{i_1} \times \dots \times s_{i_{t-1}}$ mutually exclusive sets of equal size. We call each set of the partition a **p-set** induced by the sub-array.

Example 1. Consider the OA in Figure 1. The state of p-set stack after the init-block is constructed is shown in Figure 2. For each 8×2 sub-array in the init-block, there are four p-sets induced. More specifically, the p-set $\{4, 8\}$ is induced by the sub-array of column 2 and column 3 because row 4 and row 8 in the sub-array share the same row vector $\langle 1, 1 \rangle$, and so on. Similarly, for column 1 and column 2, if we examine the rows of the 8×2 sub-matrix, we can get the partition $\{\{1, 2\}, \{3, 4\}, \{5, 6\}, \{7, 8\}\}$.

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Fig. 1. $OA(8, 2^4, 3)$

Column	P-set
2 3	$\{4, 8\} \{3, 7\}$ $\{2, 6\} \{1, 5\}$
1 3	$\{6, 8\} \{5, 7\}$ $\{2, 4\} \{1, 3\}$
1 2	$\{7, 8\} \{5, 6\}$ $\{3, 4\} \{1, 2\}$

Fig. 2. Stack of p-sets

Theorem 1. An $N \times (m + 1)$ matrix is an $OA(N, s_1, \dots, s_{m+1}, t)$ **iff** the matrix formed by its first m columns is an $OA(N, s_1, \dots, s_m, t)$, and for each $N \times (t - 1)$ sub-array of the first m columns, the s_{m+1} symbols in column $m + 1$ are equally distributed within the rows in each p-set induced by the sub-array.

According to Theorem 1, firstly we should calculate all p-sets induced by all $N \times (t - 1)$ sub-arrays from the first m columns, then the constraints for column

$m + 1$ can be obtained directly. At each search step we may add the p-sets incrementally to a stack so as to avoid recomputation.

3.4 Translation to Boolean and Pseudo-boolean Constraints

Once all the p-sets are computed, it's easy to translate the constraints to CNFs as an input file for the SAT solver. Suppose we are processing column $m + 1$. For an arbitrary p-set T , each of the s_{m+1} symbols from column $m + 1$ should appear $\frac{|T|}{s_{m+1}}$ times in the rows of T , where $|T|$ is the cardinality of T . In other words, each symbol must not appear $\frac{|T|}{s_{m+1}} + 1$ times or more. This constraint can be directly translated to logic formulas. For each $\mathcal{X} \subseteq T$, $|\mathcal{X}| = \frac{|T|}{s_{m+1}} + 1$, we add:

$$\bigwedge_{0 \leq i < s_{m+1}} \left(\bigvee_{j \in \mathcal{X}} v_j \neq i \right)$$

where v_j denotes the symbol in row j , column $m + 1$. The constraints are then expressed by the conjunction of all the CNFs.

For a p-set T , the above translation would generate $\left(\frac{|T|}{s_{m+1}}\right) \times s_{m+1}$ CNFs. When the factor level s_{m+1} is small and T is large, the input file for SAT solver can be huge. So we present another encoding of the constraints which is more compact: translating to pseudo-Boolean constraints.

A pseudo-Boolean (PB) constraint is an equation or inequality between polynomials in 0-1 variables. A linear PB clause has the form: $\sum c_i \cdot L_i \sim d$, where $c_i, d \in \mathbb{Z}$, $\sim \in \{=, <, \leq, >, \geq\}$, and L_i s are literals. The constraint of a p-set is naturally represented by the following PB clauses:

$$\bigwedge_{0 \leq i < s_{m+1}} \sum_{j \in T} P_{j \sim i} = \frac{|T|}{s_{m+1}}$$

where $P_{j \sim i}$ denotes the proposition $v_j = i$. In this way a p-set would generate only s_{m+1} PB clauses.

4 Exploiting Symmetries

For a constraint solving problem, a **symmetry** is a one to one mapping (bijection) on variables that preserves solutions and non-solutions. We say two solutions are **isomorphic** if there is a symmetry mapping one of them to the other. Since isomorphism is caused by symmetry, in the remainder of the paper, we use symmetry breaking and isomorphism elimination without distinction.

Obviously two orthogonal arrays with the same parameters are isomorphic if one can be obtained from the other by a finite sequence of row permutations, column permutations and permutations of symbols in each column. For example, Figure 3 illustrates two isomorphic OAs. It is better to eliminate isomorphism while searching for solutions as the whole search space is too redundant.

To eliminate isomorphism caused by permutations of rows and columns, a direct way is to introduce some order to fix the rows and columns. In a 2D-matrix,


<u>0 0 0 0</u>		<u>0 0 1 0</u>
0 0 0 1	1.Swap column 3 with column 4;	0 0 0 0
0 1 1 0	2.Permute symbol '0' and '1' in column 3	0 1 1 1
0 1 1 1		0 1 0 1
1 0 1 0		1 0 1 1
1 0 1 1		1 0 0 1
1 1 0 0		1 1 1 0
<u>1 1 0 1</u>		<u>1 1 0 0</u>

Fig. 3. Two isomorphic instances of $OA(8, 2^4, 2)$

each row (column) can be viewed as a vector. The rows (columns) are *lexicographically ordered* if each row (column) is lexicographically smaller (denoted by \leq_{lex}) than the next (if any). Flener et al.[8] found that lexicographically ordering both the rows and the columns can break symmetries efficiently, while bringing in only a linear number of constraints. In our approach, we take this conclusion, adding the constraints that the matrix should be lexicographically ordered along both rows and columns. For MOAs, column lexicographic order is only imposed on the columns with the same factor levels.

Example 2. Figure 4 demonstrates three solutions of $MOA(12, 3, 2^4, 2)$. A and B are lexicographically ordered along both the rows and the columns. Matrix C does not satisfy the lexicographic order since the third column $\not\leq_{lex}$ the fourth column, thus C would not be encountered in the search process.

In the following we will introduce two other symmetry breaking techniques. It can be easily proved that all these techniques could be combined to guide searching without losing any non-isomorphic solution [11].

4.1 Symbol Symmetries

There are symbol symmetries in the OA problem too. Assume that we are processing the column $X = \langle x_1, x_2, \dots, x_N \rangle$ and the column has s distinct symbols: $\{0, 1, \dots, s-1\}$. Permutations of these symbols would generate $s! - 1$ symmetries. In constraint programming symbol symmetries are also called symmetries of indistinguishable values, which can be tackled by imposing value precedence [14,10]. Our idea to break symbol symmetry is essentially the same. For each column vector in the isomorphic class, we can transform it to be lexicographically smallest by relabeling all the symbols, forcing their first appearances to be in an increasing order. In fact, the scope of their first appearances can be narrowed from $\{1, 2, \dots, N\}$ to $\{1, \dots, l\}$, where $l = \frac{N}{s_1 \times \dots \times s_{t-1}}$. Because after the preprocessing, in the first $t-1$ columns, the zero vector appears from row 1 to row l and the p-set $\{1, \dots, l\}$ is induced. All symbols in X would occur in the rows of a p-set. Formally, for all $1 \leq j < l$, $0 \leq p < q \leq s-1$, we add

$$\left(\bigwedge_{1 \leq i < j} (x_i \neq p \wedge x_i \neq q) \right) \rightarrow x_j \neq q \quad (1)$$

The formulas imply that for any two symbols p, q ($p < q$), the first appearance of q must not precede that of p in a column.

This trick is similar to the Least Number Heuristic (LNH) [18] essentially, except that, while LNH just needs to keep a variable to represent the largest value used in the assignments, here we add some propositional formulas.

Proposition 1. *A column of OA can be transformed to satisfy Formula (1).*

4.2 Filter

In this subsection we introduce a technique called Filter that can eliminate a class of symmetries, which arise from the automorphisms of init-block. If we permute the symbols in the init-block of an OA, reconstruct the init-block by swapping rows, we can always get another solution by performing some other isomorphic operations outside the init-block. We call this transformation procedure **init-block reconstruction**. The newly obtained array has the same init-block, satisfies all constraints of lexicographic order and symbol symmetry breaking, hence would be a solution to be discovered by the program.

We say two OAs are **symmetric with respect to (w.r.t.) an init-block reconstruction** if one is obtained from the other through this reconstruction.

Example 3. In Figure 4, we can obtain Matrix B from A by performing the following init-block reconstruction:

1. Permute symbol ‘0’ and ‘1’ in the first column.
2. Swap rows 1, 2, 3, 4 with rows 5, 6, 7, 8 respectively.
3. Permute symbol ‘0’ and ‘1’ in the last column.

After step 2, the init-block is unchanged. However, the last column of Matrix A is converted to $\langle 1001011010 \rangle$, contradicting Formula (1) which specifies that symbol ‘0’ must precede ‘1’. Step 3 is then performed to adjust the matrix and

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

(A)
(B)
(C)

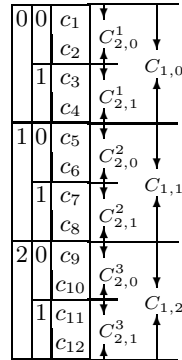


Fig. 4. Three Instances of $\text{MOA}(12, 3.2^4, 2)$

Fig. 5. Filter

we get Matrix B, which satisfies all symmetry breaking constraints. Therefore, A and B are symmetric w.r.t. the init-block reconstruction.

For an $OA(N, s_1, s_2, \dots, s_k, t)$, there are $s_1! \times s_2! \cdots s_t! - 1$ symmetries caused by init-block reconstructions except for the identity mapping. To break all these symmetries, it can be costly, because there are too many swappings and permutations to perform. We will introduce an approximate way to solve this problem. It is based on the following observation:

Suppose two matrices M_a and M_b are both OAs of $(N, s_1, s_2, \dots, s_k, t)$. We only focus on column $t + 1$. If $s_{t+1} > s_{t+2}$, column $t + 1$ cannot be exchanged with the following columns; For cases $s_{t+1} = s_{t+2}$, we can still fix column $t + 1$ by supposing the symbols in column $t + 1$ are special. Hence if column $t + 1$ of M_a and column $t + 1$ of M_b can't be obtained from each other by any init-block reconstruction, then the whole matrices M_a and M_b are not symmetric w.r.t. init-block reconstruction.

Once a symmetry caused by init-block reconstruction has been broken in column $t + 1$, it is prevented from spreading to the following columns. Breaking up symmetries in the column beyond the init-block is like setting a filter. To break such symmetries, we just add to column $t + 1$ a few constraints which are obtained according to the regularity of the init-block. We name this set of constraints **Filter**. Filter can not break all symmetries w.r.t. init-block reconstruction, but it can eliminate enough isomorphisms, yet the extra cost is negligible.

Now we describe how to set a Filter. For $1 \leq i \leq t$, denote the sub-vector of column $t + 1$ corresponding to symbol j in column i by $C_{i,j}$. To prevent symbol permutations in column i in the init-block, we can force the sub-vectors of column $t + 1$ corresponding to the symbols lexicographically ordered, i.e.,

$$\bigwedge_{0 \leq j < s_i - 1} C_{i,j} \leq_{lex} C_{i,j+1} \quad (2)$$

Take the case $MOA(12, 3.2^4, 2)$ for example. Suppose after preprocessing, the init-block is constructed and column $t + 1$ is represented by $\langle c_1, c_2, \dots, c_{12} \rangle$, as illustrated in Figure 5 ($C_{i,j}^k$ represents the k th segment of $C_{i,j}$). To prevent symbol permutations in the first column, we add $\langle c_1, c_2, c_3, c_4 \rangle \leq_{lex} \langle c_5, c_6, c_7, c_8 \rangle \leq_{lex} \langle c_9, c_{10}, c_{11}, c_{12} \rangle$. Similarly, for the 2nd column, we have $\langle c_1, c_2, c_5, c_6, c_9, c_{10} \rangle \leq_{lex} \langle c_3, c_4, c_7, c_8, c_{11}, c_{12} \rangle$. The third column of matrix A does not satisfy the first constraint since $\langle 0101 \rangle \not\leq_{lex} \langle 0011 \rangle$, thus would not be encountered during the search. The third column of B satisfies the two constraints.

Proposition 2. *An OA can be transformed so that column $t + 1$ satisfies the constraints of the Filter.*

Proof. For each i, j ($1 \leq i \leq t, 0 \leq j < s_i - 1$) contradicting Formula (2), adjust the matrix in this way: swap all those rows intersecting with the vector $C_{i,j}$ with those intersecting with $C_{i,j+1}$ accordingly, i.e., for each r that appears as a subscript in the vector $C_{i,j}$, exchange row r with row $r + \frac{N}{t_i \times s_i}$, then exchange symbol j with symbol $j + 1$ in column i to reconstruct the init-block. Each adjusting step makes column $t + 1$ lexicographically smaller while preserving the init-block. Since the

column vector has a lexicographic lower bound, the procedure will come to an end. Finally all the constraints of the Filter are satisfied. \square

5 Experimental Results

The OA searching tool BOAS (Backtrack-style OA Searcher) was implemented in the C programming language and integrated with the SAT solver MiniSat 2.0 beta [6] and the pseudo-Boolean constraint Solver PBS v2.1 [1]. We ran the program on an Intel 1.86GHZ Core Duo 2 PC with Fedora 7 OS.

To study the benefits of the symmetry breaking strategies, we carried out some experiments and asked BOAS to search the whole space to find all solutions. The numbers of loops and solutions are then compared. One loop means a successful assignment of values to the current column or concluding that there’s no solution to the column. The efficiency of symbol symmetry breaking constraints is shown in Table 1. We can see that these constraints perform well for all level factors. For most of these cases, the program runs faster when using MiniSat as the search engine than using PBS, as can be revealed in Table 1. Therefore in Table 2 only the running times for MiniSat are listed, so as to make the table clear. Since there are always too many symmetries, when one technique is being tested, the others are enabled by default.

Table 1. Efficiency of Symbol Symmetry Breaking

OA	Breaking Symbol Symmetry?							
	No				Yes			
	Loop	Solution	Time (s)		Loop	Solution	Time (s)	
			MiniSat	PBS			MiniSat	PBS
(18, 3 ⁶ , 2)	71237	41239	81.88	87.35	4116	922	14.90	18.06
(32, 4.2 ⁷ , 3)	17503	2958	9.98	13.85	2841	199	2.14	3.13
(64, 4 ⁴ .2 ⁶ , 3)	5312	192	1.57	3.25	707	3	1.08	1.07
(64, 4 ⁶ , 3)	800	576	1.38	3.15	133	1	0.76	1.11

Table 2. Efficiency of Filter

OA	Use Filter?					
	No			Yes		
	Loop	Solution	Time (s)	Loop	Solution	Time (s)
(16, 2 ¹⁵ , 2)	327396	74	111.76	319776	74	90.73
(32, 4.2 ⁷ , 3)	5517	597	7.40	2841	199	2.14
(40, 5.2 ⁶ , 3)	6522	504	73.59	2976	144	27.88
(56, 14.2 ⁴ , 3)	3432	0	252.78	2	0	0.12
(60, 5.3.2 ² , 3)	720	719	19.14	11	10	0.00
(64, 4 ⁴ .2 ⁶ , 3)	25452	108	201.33	707	3	1.08
(128, 8.4.2 ³ , 4)	-	> 8000	-	11	10	0.00

Table 2 shows the influence of Filter. From the table we can see that this strategy is more effective for OAs with large factor levels in the init-block, for example, the case $(56, 14.2^4, 3)$. When factor levels are large, there are more symbol permutations, resulting in more symmetries w.r.t. init-block reconstructions. The Filter strategy can eliminate enough of them.

Finding a Single OA

Now we see how our tool BOAS performs when we just want to find one solution. We concentrate on the OA cases with strengths no smaller than 3. As a result of space limitation, we just list some of them in Table 3. Many smaller cases which can be constructed in less than 0.01 second are omitted. The strength 3 cases with run sizes no more than 64 are from [4], and most of the non-existence results discussed in the paper can also be verified. The cases listed in Table 3 (A) all have solutions. For cases with strengths higher than 3, we have not seen any survey so far, so we also provide some conclusions for non-existence cases. “Para” stands for “Parameters”, and ‘-’ means no result was obtained within the time limit of 2 hours.

From the table we can see that for most cases that have solutions, BOAS can find one solution in about one minute, either with MiniSat or PBS, thus would be helpful in practice. PBS outperforms MiniSat in finding first solutions generally, however, when exploring the whole space, MiniSat may do better. This is probably because the efficiency of MiniSat is more stable as the number of clauses grows.

Table 3. Running Time to Find First Solution

t	N	Para	Time (s)	
			MiniSat	PBS
3	48	6.2^7	0.27	0.01
3	48	$4.3.2^4$	0.02	0.00
3	48	4.2^{11}	42.33	0.02
3	56	14.2^3	0.03	0.03
3	56	7.2^6	2.47	1.03
3	64	16.2^3	0.15	1.14
3	64	$8.4.2^4$	8.15	0.00
3	64	4^6	0.09	1.11
3	64	$4^5.2^2$	0.09	1.10
3	64	$4^4.2^6$	0.01	0.02
3	64	$4^3.2^8$	150.47	510.71
3	72	$3^2.2^{12}$	-	89.04
3	81	3^{10}	-	12.14
3	81	9.3^4	0.00	0.00
3	96	$6.4^2.2^6$	-	1150.00
3	108	3^5	1.17	3.06

(A)

t	N	Para	Exists	Time (s)	
				MiniSat	PBS
4	64	4.2^6	YES	0.02	0.00
4	64	4.2^7	NO	3.21	11.65
4	64	2^8	YES	0.05	0.01
4	64	2^9	NO	99.66	97.58
4	80	2^7	NO	9.00	3.08
4	128	8.2^6	YES	20.15	0.31
4	128	$8.4.2^4$	NO	5.13	1.02
4	128	$4^3.2^3$	YES	0.00	0.05
4	128	$4^3.2^4$	NO	12.30	23.84
4	162	$3^5.2$	YES	0.01	0.00
4	162	3^6	NO	4003.05	4257.00
4	243	3^7	YES	227.03	110.28
5	128	2^9	YES	0.21	0.02
5	128	2^{10}	NO	870.06	868.73
5	160	5.2^6	YES	0.06	0.01
5	160	5.2^7	NO	645.59	1369.28

(B)

6 Related Works

In the last ten years, a few efforts have been made to develop algorithms for constructing OAs. Yamamoto et al. [16] proposed a constructive methodology. It is based on an algorithm obtaining modular representation vectors of an OA. The methodology is complex and restricted to pure-level OAs. It lacks implementation details. Xu [15] gave an approximate algorithm for constructing OAs and NOAs (nearly orthogonal arrays) through optimization of certain combinatorial criteria. His program can find solutions to some OAs of strength 2 and small runs with high probability, but for higher strengths, the chances seem slight.

Another trend in OA searching is employing global optimization algorithms such as simulated annealing, thresholding accepting and genetic algorithms. However, as discussed in [15], they are slow in convergence for the OA problem, failing to produce some OAs with quite moderate parameters. What is more, most of the algorithms mentioned above are inexact in the sense that they do not guarantee to provide a solution or draw the negative conclusion. In contrast, an exhaustive searching algorithm can always give a definite answer provided that there is enough memory and running time.

Snepvengers [13] developed a parallel implementation of an enumeration algorithm. In his program, isomorphisms are eliminated dynamically. Each time a new column is generated, all isomorphic operations are examined to get the lexicographically smallest solutions, therefore all isomorphisms can be eliminated. However, it may take a long time to perform isomorphism checking, and the largest factor level is restricted to 10. The parallel program was executed on a CRAY super-computer, and the running times were measured in hours. It is not convenient to compare our tool with that program directly. But it seems that we can solve some cases that are quite difficult. For example, to solve the case $(96, 6.4^2.2^6, 3)$, the parallel program ran on 16 processors for 6 days, but no result was obtained. Our program BOAS can find one solution in twenty minutes. In [13] $OA(64, 4^2.2^5, 3)$, $OA(64, 4.2^8, 3)$, $OA(108, 4.3^4, 3)$ and $OA(108, 3^5, 3)$ are claimed to be new results. It took their parallel program from 26.76 hours to 934.10 hours on a dozen of processors to obtain the results, while we can find these OAs within seconds. However, we feel it a bit unfair to perform such comparison since they aim at finding all non-isomorphic solutions (although they failed in three cases), while we try to find only one solution.

The cell-by-cell method for finding Covering Arrays (CAs) presented in [17] can also be extended to search for OAs. Currently we are investigating how to incorporate some of our symmetry breaking techniques into the framework.

7 Conclusion

As we mentioned in the Introduction, OAs (and MOAs) are quite important in combinatorics, and they also find applications in various areas such as software testing. It is desirable to have efficient tools for finding them. On the other hand, searching for such combinatorial objects poses interesting and challenging questions for automated reasoning research.

In this paper we proposed a framework for finding (M)OAs, which uses automated reasoning and constraint solving technology. And we also proposed some new symmetry breaking techniques for speeding up the search process. Our approach is a universal method in the sense that there is no restriction on the given parameters and that we can always get a conclusion after the search is completed. The experimental results show that our program is effective in solving cases with strengths higher than 2. The reason is that, in general the problems with more constraints are easier to solve for constraint solvers. They can take full advantage of constraint propagation techniques. But these cases (with high strengths) are more difficult for traditional mathematical methods.

The symmetry breaking techniques are quite effective in general. The SAT solver also contributes to the high efficiency of our program. But when the factor levels s_1, s_2, \dots, s_k are too small as compared with the run size N , we may get a large number of propositional clauses. Using PBS adds a level of modularity and makes it easier to program.

In the future, we will look further into various techniques, including more complete symmetry breaking, stronger constraint propagation, parallelization, direct search methods as well as mathematical results. We believe that they can be combined effectively. And we plan to implement a more powerful tool for finding (M)OAs.

References

1. Aloul, F.: The PBS Page, <http://www.eecs.umich.edu/~faloul/Tools/pbs/>
2. Bennett, F.E., Zhang, H.: Latin Squares with Self-Orthogonal Conjugates. *Discrete Mathematics* 284(1–3), 45–55 (2004)
3. Black, R.: *Pragmatic Software Testing*. Wiley, Chichester (2007)
4. Brouwer, A.E., Cohen, A.M., Nguyen, M.V.M.: Orthogonal Arrays of Strength 3 and Small Run Sizes. *J. Statistical Planning and Inf.* 136(9), 3268–3280 (2006)
5. Colbourn, C.J., Dinitz, J.H.: *Handbook of Combinatorial Designs*, 2nd edn. CRC Press, Boca Raton (2007)
6. Eén, N., Sorensson, N.: The MiniSat Page, <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/>
7. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: *Orthogonal Arrays: Theory and Applications*. Springer Series in Statistics (1999)
8. Flener, P., et al.: Breaking Row and Column Symmetries in Matrix Models. In: Van Hentenryck, P. (ed.) *CP 2002*. LNCS, vol. 2470, pp. 462–476. Springer, Heidelberg (2002)
9. Krishnan, R., et al.: Combinatorial Testing: Learnings from Our Experience. *ACM SIGSOFT Softw. Eng. Notes* 32(3), 1–8 (2007)
10. Law, Y.C., Lee, J.H.: Symmetry Breaking Constraints for Value Symmetries in Constraint Satisfaction. *Constraints* 11(2–3), 221–267 (2006)
11. Ma, F., Zhang, J.: Seaching for Orthogonal Arrays. *ISCAS-LCS-07-05* (2007)
12. Robert, B., Prowse, J., Phadke, M.S.: Robust Testing of AT&T PMX/Star MAIL using OATS. *AT&T Technical Journal* 71(3), 41–47 (1992)
13. Snepvangers, R.: *Statistical Designs for High-Throughput Experimentation*. Technical report, Stan Ackermans Institute (August 2006)

14. Walsh, T.: Symmetry Breaking using Value Precedence. In: Proc. ECAI 2006, pp. 168–172 (2006)
15. Xu, H.: An Algorithm for Constructing Orthogonal and Nearly-orthogonal Arrays with Mixed Levels and Small Runs. *Technometrics* 44, 356–368 (2002)
16. Yamamoto, S., et al.: Algorithm for the Construction and Classification of Orthogonal Arrays and Its Feasibility. *J. Combin. Inform. System Sci.* 23, 71–84 (1998)
17. Yan, J., Zhang, J.: Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing. In: Proceedings of COMPSAC, pp. 385–394 (2006); Extended version, to appear *J. of Systems and Software*
18. Zhang, J., Zhang, H.: SEM: A System for Enumerating Models. In: Proceedings of IJCAI 1995, pp. 298–303 (1995)