Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 1681–1693
The Journal of Systems and Software
www.elsevier.com/locate/jss

A backtracking search tool for constructing combinatorial test suites ☆

Jun Yan, Jian Zhang *

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, 4# South Fourth Street, Zhong Guan Cun, P.O. Box 8718, Beijing 100190, PR China

Available online 4 March 2008

Abstract

Combinatorial testing is an important testing method. It requires the test cases to cover various combinations of parameters of the system under test. The test generation problem for combinatorial testing can be modeled as constructing a matrix which has certain properties. This paper first discusses two combinatorial testing criteria: covering array and orthogonal array, and then proposes a backtracking search algorithm to construct matrices satisfying them. Several search heuristics and symmetry breaking techniques are used to reduce the search time. This paper also introduces some techniques to generate large covering array instances from smaller ones. All the techniques have been implemented in a tool called EXACT (EXhaustive seArch of Combinatorial Test suites). A new optimal covering array is found by this tool.

Keywords: Software testing; Combinatorial testing; Optimal test set; Covering array; Orthogonal array

1. Introduction

Testing plays an important role in software development process. Due to the complexity of software systems, the number of possible tests can grow exponentially as a function of the number of parameters. Consider a black box testing approach. After category partitioning, we can assume that all the parameters of the system under test (SUT) take values from finite domains of sizes $v_1, v_2, \ldots, v_k$. In order to test the SUT thoroughly, we may need $\prod_{i=1}^{k} v_i$ test cases. This number is often too large in practice. However, a significant number of faults are caused by interactions of a small number of parameters in many applications. Kuhn and Reilly compared the historical records of reported bugs for the Mozilla web browser against results of interaction testing (Kuhn and Reilly, 2002). More than 70% of bugs were identified with 2-way interaction testing, and approximately 90% of bugs were identified with 3-way interaction testing. Thus, combinatorial criteria are reasonable trade-off between the cost of testing and the required degree of coverage.

A well-known combinatorial testing criterion is called CA (covering array). Consider a small number t. For every t parameters, each combination of valid values of these t parameters is covered by at least one test case. Then faults caused by interactions of t or fewer parameters can be found by the test suite. The value of t is often set as 2 or 3 in practice. This testing method is very efficient since for a system with k parameters, the minimum size of a test suite grows at most logarithmically in k (Godbole et al., 1996).

Let us first use an example to explain this coverage criterion. Suppose we have an E-commerce application which has four components, each with three configurations, as shown in Fig. 1. To test all possible interactions for this system we would need $3^4 = 81$ test cases.

But if we restrict ourselves to pairwise coverage (that is, $t = 2$), we only need to use 9 test cases (see Fig. 2) to test the system. Then for every two components, each combination will be tested just once.

☆ This work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 60673044 and 60633010.
* Corresponding author.
E-mail addresses: yanjun@ios.ac.cn (J. Yan), zj@ios.ac.cn (J. Zhang).
URLs: http://lcs.ios.ac.cn/~yanjun (J. Yan), http://lcs.ios.ac.cn/~zj (J. Zhang).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.02.034

Client	Web Server	Payment	Database
Firefox	WebSphere	MasterCard	DB/2
IE	Apache	Visa	Orade
Opera	.NET	AmEx	Access

Fig. 1. A system with four components.

Client	Web Server	Payment	Database
Firefox	WebSphere	MasterCard	DB/2
Firefox	.NET	AmEx	Orade
Firefox	Apache	Visa	Access
IE	WebSphere	AmEx	Access
IE	Apache	MasterCard	Orade
IE	.NET	Visa	DB/2
Opera	WebSphere	Visa	Orade
Opera	.NET	MasterCard	Access
Opera	Apache	AmEx	DB/2

Fig. 2. Test suite covering all pairs.

In general, combinatorial testing is widely believed to be a systematic, effective testing technique. Yilmaz et al. (2006) conclude that even low-strength (i.e., small t) covering arrays can provide up to 99% reduction in the number of configurations to be tested and have fault characterizations that were as reliable as those created through exhaustive testing. On the other hand, according to Schroeder's experiments of some controlled study (Schroeder et al., 2004), there is no significant difference in the fault detection effectiveness of t-way and random test suites. They use a test suite generation algorithm similar to the greedy algorithm AETG (refer to Section 7). Therefore we need new methods to design better combinatorial test suites.

There are also some other combinatorial test criteria, e.g., OA (orthogonal array), PPW (partly pairwise) (Grindal et al., 2005). The test data generation problem can be modeled as the problem of constructing a matrix, each row of which represents a test case, and each column represents a parameter of the SUT. Then the key problem of combinatorial testing is: how to find a test suite which has the minimum size, or how to construct the matrix such that it has the minimum number of rows.

Mathematicians have studied the combinatorial problems extensively and obtained many theoretical results (Colbourn and Dinitz, 2006). In addition, researchers in computer science have proposed some algorithms to generate arrays satisfying the CA criterion automatically. It is hard to say which approach is better. Due to the complexity of the problem, most algorithms are approximate, in the sense that they do not guarantee producing arrays of the minimum size. In contrast, exact algorithms have not received much attention. Our first attempt to apply backtrack search to the covering array problem was presented in (Yan and Zhang, 2006). The algorithm employs a number of search heuristics and can be used to find optimal solutions in many small cases. A tool called EXACT was also described there. In this paper, we make further improvements and extensions to the tool, so that it can find more arrays efficiently. Here we use two combinatorial criteria: CA (covering array) and OA (orthogonal array) to demonstrate the application of the tool.

This paper is organized as follows. We begin with an introduction of the CA and OA test criteria in the next section. Then in the following two sections, the basic algorithm with some useful heuristics are presented. In Section 5, we describe how to use some mathematical results to produce large CA instances from smaller ones found by our tool. Then we present some computational results in Section 6. Finally we discuss some related works and future research directions.

2. Background

In this section, we will briefly introduce some basic notation about two combinatorial testing criteria: CA (covering array) and OA (orthogonal array).

2.1. Covering array

Cohen et al. (2003) have given the following definitions of *covering array* and *mixed level covering array*:

Definition 2.1 (*Covering array*). A covering array, $CA(N;t,k,v)$, is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all ordered subsets from v symbols of size t at least once. In such an array, t is called *strength*, k the *degree* and v the *order*.

The covering array criterion requires that all parameters of the SUT have the same domain. If the parameters of the SUT have different domains, we can use mixed covering arrays to describe the test suite:

Definition 2.2 (*Mixed covering array*). A mixed covering array, $MCA(N;t,k,v_1v_2\cdots v_k)$ is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

(i) Each column i $(1 \leqslant i \leqslant k)$ contains only elements from a set S_i of size v_i.
(ii) The rows of each $N \times t$ sub-array cover all t-tuples of values from the t columns at least once.

Similarly, t is called *strength*, k the *degree* and v_i the *order* of the i'th column.

A covering array (or a mixed covering array) is *optimal* if it contains the minimum possible number of rows. We call the minimum number *covering array number*, $CAN(t,k,v)$

00000
01111
10011
11100
20101
21010

Fig. 3. $MCA(6;2,3^1 2^4)$.

(or *mixed covering array number*, $MCAN(t,k,v_1v_2\cdots v_k)$). A shorthand notation is used to describe covering arrays by combining equal entries in v_i.

Example 2.3 (*CA and MCA*). The example given in Section 1 is an instance of $CA(9;2,4,3)$ (or denoted as $CA(9;2,3^4)$). Fig. 3 lists an example of MCA. The mixed covering array, $MCA(6;2,5,32222)$ (shorthand notation $MCA(6;2,3^1 2^4)$), implies the SUT has five components, one of which has three possible configurations and each of the rest four has two possible configurations. Both these two arrays are optimal and have strength 2, thus we can use them to test the interaction of every two parameters.

Often we refer to a *CA* or *MCA* of strength t as a t-way covering array or t-covering array. When $t=2$, it is also called a pairwise covering array.

There are some mathematical results about estimating covering array numbers. For example, Chateauneuf et al. (1999) have proved that

$$CAN(3,k,3) \leqslant 6.1209(\log k)^2.$$

These results provide probabilistic bounds on the value of N, but they do not give us any method for constructing the arrays. Some people make use of notions in algebra, such as mutually orthogonal Latin squares, finite fields, etc. to construct covering arrays. Such methods often can only handle some specific cases. Some of the best results about CAs can be found on a website (Colbourn, 2007).

Seroussi and Bshouty (1988) show that a generalized version of the problem of finding a minimal t-covering array is NP-complete. Also Lei and Tai (1998) prove that the decision of whether a system has 2-covering of size N is NP-complete. Thus, it is unlikely for us to find an efficient algorithm which can always generate an optimal test suite.

2.2. Orthogonal array

Another kind of combinatorial arrays used in testing is orthogonal array:

Definition 2.4 (*Orthogonal array*). An orthogonal array, $OA(N;t,k,v)$, is an $N\times k$ array on v symbols such that for every $N\times t$ sub-array, each subset from v symbols of size t occurs equal number of times.

Similarly, if the parameters of the SUT have different domains, we use mixed orthogonal arrays:

Definition 2.5 (*Mixed orthogonal array*). A mixed orthogonal array, $MOA(N;t,k,v_1v_2\cdots v_k)$ is an $N\times k$ array on v symbols, where $v=\sum_{i=1}^{k} v_i$, with the following properties:

(i) Each column i $(1\leqslant i\leqslant k)$ contains only elements from a set S_i of size v_i.
(ii) For the rows of each $N\times t$ sub-array, each t-tuple of values from the t columns occurs equal times.

Similar to CA and MCA, t is called *strength*, k the *degree* and v (or v_i) the *order*. An orthogonal array (or mixed orthogonal array) is *optimal* if it contains the minimum possible number of rows. We call the minimum number *Orthogonal Array Number* $OAN(t,k,v)$ (or *Mixed Orthogonal Array Number* $MOAN(t,k,v_1v_2\cdots v_k)$).

Example 2.6 (*OA and MOA*). The example in Section 1 is also an instance of $OA(9;2,4,3)$. For each 9×2 sub-array, each combination of values occurs just once. Also consider the SUT, of which one component has three possible configurations and the rest four have two possible configurations. One of the optimal MOA for this SUT is illustrated in Fig. 4. Consider the two columns 1 and 3, there are 6 possible tuples: $\langle 0,0\rangle$, $\langle 0,1\rangle$, $\langle 1,0\rangle$, $\langle 1,1\rangle$, $\langle 2,0\rangle$, and $\langle 2,1\rangle$. Each tuple occurs twice. While for the column 2 and 3, there are 4 possible tuples and each occurs three times.

The theory of OA is well developed. Different from CA and MCA, the OA and MOA have the balancing property, that is, each t-tuple of a $N\times t$ sub-array appears a definite number of times. The beauty of these arrays attract researchers to study the theory. Some OAs and MOAs

00000
00011
01100
01111
10001
10110
11010
11101
20100
20111
21001
21010

Fig. 4. $MOA(12;2,3^1 2^4)$.

can be obtained from the website (Sloane, 2007). On the other hand, these arrays are widely employed in industrial experiments to study the effect of several control factors. It is beneficial to develop a tool that can construct OA and MOA automatically.

OA and MOA were first introduced to testing in 1985 (Colbourn, 2004). Evidently an orthogonal array $OA(N;t,k,v)$ is also a covering array. Therefore OA was once used to create CA and MCA. For example, an instance $OA(27;2,3^{13})$ can collapse to an $MCA(N;2,3^{10}2^3)$ (obviously $N \leqslant 27$) by replacing the undefined values by an arbitrarily defined values and removing the redundant rows. Williams and Probert (1996) thoroughly described this application of OA to testing. In general, the mixed orthogonal array number is much bigger than mixed covering array number (e.g., $MOAN(2,3^1 2^4) = 12$ and $MCAN(2,3^1 2^4) = 6$, refer to Figs. 3 and 4).

3. The search procedure

We can display a CA or OA as a 2-dimensional $N \times k$ matrix. Each row can be regarded as a test case and each column represents some parameter of the system under test. Each entry in the matrix is called a *cell*, and we use $ce_{r,c}$ to denote the cell at row r ($r > 0$) and column c ($c > 0$), or the value of parameter c in test case r.

We apply exhaustive search technique to this problem. Our algorithm is based on backtrack search. The main algorithm can be described as a recursive procedure in Fig. 5. For a given covering array number N, we try to assign each cell of the $N \times k$ matrix until all requirements of covering arrays are satisfied. The parameter *pSol* and *Fmla* represent the current assignments of cells and the constraints respectively. Each time a variable is assigned, the constraint propagation function BPropagate inspects the constraints to check if any constraint implies another value (returns TRUE) or contradiction (returns FALSE). The function Chk_Cons checks whether all the t-tuples have been covered in the columns $\{1,\ldots,c\}$ if all the cells in column c have been assigned values. The function CELL_Selection chooses an unassigned cell and function Val_Set_Gen generates the candidate value set S_x of a cell.

```
bool BSrh(pSol, Fmla) {
    if (!BPropagate(pSol, Fmla)) return FALSE;
    if (!Chk_Cons(pSol, Fmla)) return FALSE;
    if (pSol assigns a value to every cell) return TRUE;
    x = CELL_Selection(pSol);
    S_x = Val_Set_Gen(x, pSol, Fmla);
    for each value u in set S_x
        if (BSrh(pSol ∪ {x = u}, Fmla)) return TRUE;
    return FALSE;
}
```

Fig. 5. Exhaustive search algorithm.

4. Techniques for improving the efficiency

The worst time complexity of the naive exhaustive search for $MCA(N;t,k,v_1v_2\cdots v_k)$ is $\prod_{i=1}^{k} v_i^N$, but some improvements can actually lead to a practical algorithm. Generally speaking, there is a common problem in backtracking techniques, that is, the program repeats failure due to the same reason generated by older cell assignments. This problem can be partially solved by forming a proper order in assigning the cells. We will not try cells outside the columns $\{1,\ldots,c\}$ until all the constraints between these columns are satisfied. Another way is trying to discover inconsistency as early as possible.

In this section, we introduce some techniques to speed up the search process. These techniques include: preprocessing; symmetry breaking; imposing extra constraints; more powerful constraint propagation (to find contradiction as early as possible); and efficient strategies to select a proper value to assign to a cell.

4.1. Preprocessing

Without any loss of generality, assume we are solving a $MCA(N;t,k,v_1v_2\cdots v_k)$ problem with $v_1 \geqslant v_2 \geqslant \cdots \geqslant v_k$. Then consider the first t columns of the matrix. There are $mb = v_1v_2\cdots v_t$ possible combinations of values. We can make the first mb rows contain each value combinations once by swapping the rows of the matrix without changing the test set. Thus, we can fix this $mb \times t$ sub-array (which we call a *mini-block*) before search.

While for an $MOA(N;t,k,v_1v_2\cdots v_k)$ with $v_1 \geqslant v_2 \geqslant \cdots \geqslant v_k$, the mini-block has size $mb \times t$ where $mb = N$. Each t-tuple in the mini-block occurs $f = \frac{N}{v_1v_2\cdots v_t}$ times.

We use two matrices in Fig. 6 to illustrate the mini-block. The two sub-arrays in the boxes form the mini-blocks of the two arrays $CA(6;2,6,2)$ ($mb = 4$) and $OA(8;2,7,2)$ ($mb = 8$) respectively.

Theoretically, the time complexity will reduce to $(\prod_{i=1}^{k} v_i^N)/(\prod_{i=1}^{t} v_i^{mb})$ if the mini-block is fixed.

$CA(6;2,6,2)$	$OA(8;2,7,2)$
	0 0 0 0 0 0
	0 0 0 1 1 1
0 0 0 0 0 0	0 1 1 0 0 1 1
0 1 0 0 0 1	0 1 1 1 1 0 0
1 0 0 1 1 0	1 0 1 0 1 0 1
1 1 1 0 1 0	1 0 1 1 0 1 0
0 0 1 1 1 1	1 1 0 0 1 1 0
1 1 1 1 0 1	1 1 0 1 0 0 1

Fig. 6. Mini-blocks of arrays.

a	b
0 0 0	0 0 0
0 1 1	1 0 1
1 0 1	0 1 1
1 1 0	1 1 0

Fig. 7. Two isomorphic CAs of $CA(4;2,3,2)$.

4.2. Symmetry breaking

Two solutions are isomorphic if one can be obtained from the other by permuting element names. Because of the isomorphism, one solution may be represented in many ways, which results in much redundancy in the search space. We say that a problem has symmetries if it has isomorphic solutions.

Example 4.1 (*Two isomorphic CAs*). The two covering arrays of $CA(4;2,3,2)$ in Fig. 7 are isomorphic since we can get one from the other by swapping the first two columns of the matrix.

We can force our algorithm not to consider matrices which are isomorphic to those enumerated before by breaking symmetries. Symmetry breaking (SB) techniques have been studied via the symmetric property of variables. These techniques can be used by imposing extra constraints as part of the input or they can work dynamically. We use the latter method by adding the following function Sym_Eli (see Fig. 8) to our algorithm right after the function Val_Set_Gen.

Here we introduce two symmetry breaking methods. Both of them can be used in our algorithm to generate symmetry elimination rules.

4.2.1. Row and column symmetries

Flener et al. (2002) introduced an important class of symmetries in constraint programming, arising from matrices of decision variables where rows and columns can be swapped. To eliminate this type of symmetries, we can impose a partial order on the vectors of the matrix. Firstly, we give a recursive definition of the lexicographic order as follows:

```
void Sym_Eli(x, S_x, pSol) {
    generate SB rules;
    for all possible value u of S_x {
        try to assign u to x;
        if (the assignment conflicts with SB rules)
            remove u from S_x;
        restore the assignment;
    }
}
```

Fig. 8. The symmetry breaking function.

Definition 4.2 (*Lexicographic order*). Two vectors $X=[x_1,x_2,\ldots,x_n]$ and $Y=[y_1,y_2,\ldots,y_n]$ have a lexicographic order $X\leqslant_{\text{lex}}Y$ if

(i) If $n=1$, $x_1\leqslant y_1$;
(ii) If $n>1$, $x_1<y_1\vee(x_1=y_1\wedge[x_2,\ldots,x_n]\leqslant_{\text{lex}}[y_2,\ldots,y_n])$.

We say Y is lexicographically greater than X (denoted as $Y\leqslant_{\text{lex}}X$) if $X\leqslant_{\text{lex}}Y$ does not hold.

In general, an $m\times n$ matrix has $m!\cdot n!-1$ symmetries excluding the identity, and thus generates a super-exponential number of lexicographic ordering constraints. It is not practical to implement all the SB rules in our algorithm.

In this paper, we make use of two types of symmetries in the class. We can treat each row (column) as a vector and order these vectors lexicographically. The rows (columns) of a 2D-matrix are lexicographically ordered if each row (column) is lexicographically smaller than the next (if any). We add constraints that each column (row) is lexicographically smaller than the next column (row) as a symmetry breaking rule. Here is an example:

Example 4.3. For the matrix (b) of Fig. 7, the first column is not lexicographically smaller than the second one, so we will not encounter the second matrix during the search.

These row and column symmetry breaking techniques can be summarized as the following four rules. The scenarios of applying them are visualized in Figs. 9 and 10.

Rule 1. (Column SB rule) Let

$C_i=[ce_{1,i},ce_{2,i},\ldots,ce_{N,i}]$

denote the i'th column of the matrix. The constraint to break symmetries between columns is

if $v_i=v_{i+1}$ and $i>t$, then $C_i\leqslant_{\text{lex}}C_{i+1}$,

where v_i is the order of i'th column.

Rule 2. (Row SB rule) Let

$R_j=[ce_{j,1},ce_{j,2},\ldots,ce_{j,k}]$

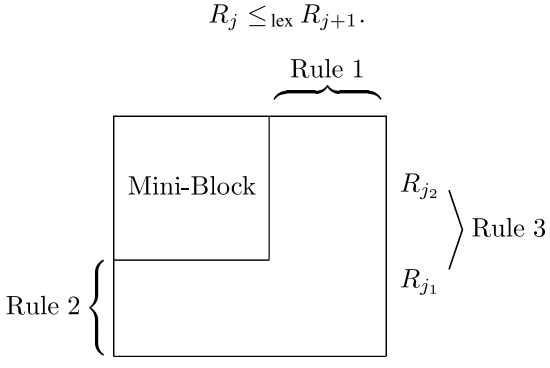

Fig. 9. The rules applied on CA or MCA.

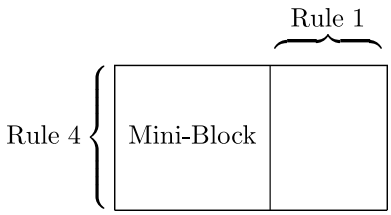

Fig. 10. The rules applied on OA or MOA.

denote the j'th row of the matrix. Then since we have a prefixed mini-block, we have

if $j > mb$, then $R_j \leqslant_{\text{lex}} R_{j+1}$.

Rule 3. There are also some symmetries between the rows in the mini-block and those outside the mini-block. Suppose there are two rows R_{j_1} $(j_1 > mb)$ and R_{j_2} $(j_2 \leqslant mb)$, we have

if $ce_{j_1,\ell} = ce_{j_2,\ell} (1 \leqslant \ell \leqslant t)$, then $R_{j_2} \leqslant_{\text{lex}} R_{j_1}$.

Rule 4. In the mini-block of MOA, each combination of values may occur f times (refer to Section 4.1). When trying to find an MOA, apart from the SB rules of MCA, we also need to add the following constraint to eliminate the symmetry between the rows in the mini-box:

$R_j \leqslant_{\text{lex}} R_{j+1}$.

Example 4.4. The two arrays of Fig. 6 satisfy these rules. We list some examples in Tables 1 and 2.

These SB constraints are checked on-the-fly by our tool. For example, consider $X = [1, 2, 3, 4]$ and $Y = [2, 3, ?, ?]$ where a "?" represents an unassigned cell. It is obvious that $X \leqslant_{\text{lex}} Y$.

4.2.2. LNH

For a column c, all the cells of the column share the same domain $D_c = \{0, 1, 2, \ldots, v_c - 1\}$, and these $|D_c|$ symbols can be permuted. Therefore a solution has $2^{|D_c|} - 1$ isomorphic matrices by swapping the symbols of column c. We can use the following strategy to prune some isomorphic ones.

The LNH (Least Number Heuristic) (Zhang, 2001) is a fairly simple but efficient strategy to eliminate some trivial symmetries. It is based on the observation that, at the early stages of the search, many symbols (which have not yet been used in the search) are essentially the same. Thus, when choosing a value for a new cell, we need not consider all the symbols of a given domain. It makes no difference to assign the symbol e or e' to the cell, if neither of them has been assigned to a cell. To make this concise, we say that two symbols e and e' are symmetric.

When processing a column c, all the unused values are the same to the unassigned cell. We use a program variable mdn_c to represent the largest value used in the assigned cells of column c. Then the candidate value set for the next unassigned cell is $\{0, 1, 2, \ldots, mdn_c + 1\}$. The initial value of mdn_c is -1 and it is dynamically updated during the search. Since in the first t columns, the mini-block has used all the symbols, we cannot apply LNH in these columns, thus we restrict $c > t$.

The CP method (Hnich et al., 2006) tries to use another technique to eliminate the isomorphism brought by permuting the symbols of a column. That technique adds the constraint $f_{c,1} \leqslant f_{c,2} \leqslant \cdots \leqslant f_{c,v_c}$ to the symbols, where $f_{c,u}$ is the number of times symbol u occurs in column c. We do not use this technique since it may be in conflict with our SCEH strategy (refer to Section 4.3).

Table 1
Rules on $CA(6; 2, 6, 2)$

Rule	Example
Rule 1	Column 3 $\leqslant_{\text{lex}}$ Column 4
Rule 2	Row 5 $\leqslant_{\text{lex}}$ Row 6
Rule 3	Row 1 $\leqslant_{\text{lex}}$ Row 5

Table 2
Rules on $OA(8; 2, 7, 2)$

Rule	Example
Rule 1	Column 3 $\leqslant_{\text{lex}}$ Column 4
Rule 4	Row 3 $\leqslant_{\text{lex}}$ Row 4

4.3. Additional constraints to reduce the search space

In general, we only need to find one solution. So we can add some constraints to reduce the search space. These constraints will greatly speed up the search. Similar to the symmetry breaking methods, these constraints are applied by a function Pruning inserted in our search algorithm after the set S_x is generated. We provide a novel pruning technique called SCEH (the Sub-Combination Equalization Heuristic) for this problem.

For OA or MOA, for every s $(s \leqslant t)$ columns, each s-tuple occurs the same times. For 2-Covering Arrays, Meagher and Stevens (2005) proposed a conjecture that it is always possible to find a CA where the occurrence numbers of symbols in a row are as equal as possible (which called Balanced Covering Arrays). We also observe that, for CA or MCA, in many instances, in any column of the matrix, each value appears *almost the same* number of times. This rule is also effective when we extend it to value-tuples. Here we introduce the definition of sub-combination first.

Definition 4.5 (*Sub-combination*). A sub-combination of size s is an array of s values from s columns, denoted as SC_C^V, where V is the vector of values and C the vector of columns. The frequency of the sub-combination, denoted by f_C^V, is defined as the times that the s-tuple V occurs in the columns C.

Then we can introduce some restrictions to prune the search tree (SCEH): for a given covering array problem $CA(N;t,k,v)$ or $MCA(N;t,k,v_1v_2\cdots v_k)$ and a given sub-combination of size s $(0<s\leqslant t)$, we have

$$|f_C^{V_1}-f_C^{V_2}|\leqslant 1 \quad \text{for all } V_1\neq V_2.$$

When N is the multiple of the size of set $\{V_i\}$, we can find a matrix such that $f_C^{V_1}=f_C^{V_2}$ for all V_1 and V_2 where $V_1\neq V_2$.

We can use an example to show this rule:

Example 4.6. A covering array $CA(11;2,5,3)$ is given in Table (a) of Fig. 11. Consider $s=1$, then we have $\binom{k}{s}=5$ column vectors and each has 3 value vectors. We can count the frequency of each column vector. The results are listed in Table (b). The frequencies of different vectors with the same column are almost the same.

While processing CA or MCA, we use the heuristic of $0<s\leqslant SL$ in practice, where SL is the strategy level specified by the user. We can often choose $SL=t-1$. The strategy is applied to our algorithm by estimating the upper-bound UB and lower-bound LB of the frequency of each sub-combination. When an assignment $x=u$ can cause the frequency of a sub-combination to go beyond the range $[LB,UB]$, the value u will be eliminated from the set S_x. This strategy can reduce the search time for almost all instances we have tried.

4.4. Can we use these strategies together?

We have introduced four techniques (mini-block, lexicographic order, LNH, SCEH) to decrease the search space. Can we use them at the same time when searching for an instance? The answer is yes. Here we give some proofs.

SCEH only restricts the frequency of each sub-combination, hence it does not affect the other three techniques.

a

a a a a a
a b b b b
a c b c c
b a a b c
b b c a c
b c a a b
c a b a c
c b a c b
c c c b a
a a c c b
b b b c a

b

The Value Vectors	⟨a⟩	⟨b⟩	⟨c⟩
Column 1	4	4	3
Column 2	4	4	3
Column 3	4	4	3
Column 4	4	3	4
Column 5	3	4	4

Fig. 11. The value vector's frequency.

Then, considering that the rule of LNH strategy can be regarded as a constraint

$$ce_{i+1,c}\leqslant MAX\{ce_{1,c},\ldots,ce_{i,c}\}+1$$

where $t<c<k$ and $0<i<N$, we have the following theorem:

Theorem 4.7. *If a case has a solution, then we can find at least one matrix such that it contains a mini-block and satisfies the rules of lexicographic order and LNH.*

Flener et al. (2002) proved that all the lexicographic order constraints can be used together to guide the searching without losing any non-isomorphic solution. Next we give a similar proof.

Proof. First, suppose we are processing a case of MCA (CA). Assume A is a solution, we swap the rows of A such that in the first mb rows, each t-tuple of the first t columns occurs just once. Then we sort these mb rows in non-descending lexicographic order (i.e., for all $0<i<mb$, $R_i\leqslant_{\text{lex}}R_{i+1}$). Therefore the new matrix, namely B, contains a mini-block. Obviously B is a solution. Then we repeat the following five operations to adjust B to satisfy the four rules of lexicographic order strategy (refer to Section 4.2.1) and LNH. All these operations preserve that matrix B is a solution:

Rule 1. Sort the columns outside the mini-block that have the same number of symbols in non-descending lexicographic order.

Rule 2. If we are processing a MCA or CA, sort the rows outside the mini-block in non-descending lexicographic order.

Rule 3. For each two rows $1<i\leqslant mb$ and $j>mb$ of a MCA or CA, if these two rows have the same first t values and $R_i>_{\text{lex}}R_j$, swap the two rows.

Rule 4. If we are processing a MOA or OA, sort the rows in non-descending lexicographic order.

LNH. For each column c $(t<c\leqslant k)$ and $0<i<N$, let $w_i=MAX\{ce_{1,c},\ldots,ce_{i,c}\}$ and $u_i=ce_{i+1,c}$. If $w_i+1<u_i$, permute the two symbols u_i and w_i+1.

If we regard the matrix B as a vector

$$B=[R_1,R_2,\ldots,R_N]=[ce_{1,1},\ldots,ce_{1_k},\ldots,ce_{N,1},\ldots,ce_{N,k}]$$

then all the operations make B lexicographically smaller. Since the number of isomorphic vectors is bounded, this loop will come to an end and result in the matrix C. C is a solution and satisfies all the additional constraints. □

4.5. Constraint propagation

For a cell, not all the candidate values will lead to a solution. The assignment of "bad" value should be avoided by estimating the future conflicts with this value. The domain of a variable can be modified consistently by any other variable. This technique is often called propagation in constraint processing.

The SCEH strategy and the covering requirements provide the upper- and lower-bounds for some value combinations. These domains may shrink to contain only one value if some cells are assigned. Then some unprocessed cells can also be assigned. For some columns including the current column C, if the frequency of all value combinations except for vc reach their upper-bound, the remaining unassigned cells of C should be assigned the value to fit for value combination vc.

We can also use a graph theoretic method to find future conflicts. Suppose we are processing column c. Consider a column combination C containing column c. We use X to represent the set of uncovered value combination of C and Y to denote the set of unassigned cells. We can build a bipartite graph G with these two sets as partite sets. The edges are defined as coverage relations, i.e., there is an edge between $x \in X$ and $y \in Y$ if the value combination x can be covered by assigning a value to cell y. The goal of our search is to find a matching that, each cell is linked to only one value combination since each cell can be assigned only one value, and each value combination must be linked to a cell. That is, we want to find a matching in G that saturates set X. P. Hall proved the following classical theorem in 1935:

Theorem 4.8 (Marriage theorem). *Let G be a bipartite (multi) graph with partite sets X and Y. Then there is a matching in G saturating X* iff $|N(S)| \geqslant |S|$ *for every $S \subseteq X$, where $N(S)$ is the neighbor set of S which is defined to be the set of vertices adjacent to vertices in S.*

Since there are $2^{|X|}$ subsets of X, we only use a few subsets for pruning. For example, consider $S = X$, since $|N(X)| \leqslant |Y|$, if $|Y| < |X|$, there is a contradiction.

We use an example to explain the usage of this strategy. Suppose we are processing a column c_3 of a CA, and c_3 has 3 cells unassigned. Assume $t = 3$. Consider 3 columns c_1, c_2 and c_3 ($c_1 < c_2 < c_3$). If there are three 3-tuple $X = \{\langle 2,1,0\rangle, \langle 1,1,1\rangle, \langle 1,1,0\rangle\}$ to be covered, then we can draw the bipartite graph in Fig. 12, where "?" denotes the unassigned cells. Let Y be the neighbor set of X, obviously $|Y| = 2 < |X| = 3$, so we cannot assign these 3 cells to cover X.

Unlike other strategies (symmetry breaking and pruning), this strategy does not remove any value that may lead to a solution from the set S_x.

4.6. Value selection strategy

When we are processing a cell that has more than one candidate values, usually we will choose the first value. But this is not necessarily the best choice. We may improve this situation by estimating which value is most likely to lead to a solution. We use a technique similar to the greedy strategy of AETG, that is, we assign the cell a value which covers the largest number of uncovered combinations. This strategy is applied by setting each possible value a priority, and we always choose the value u with the highest priority first to assign to the cell x.

Our experience shows this strategy is only efficient in some mixed covering arrays (especially for those instances whose parameter domains vary significantly). A most illustrating example is that, it can reduce the search time of $MCA(30; 2, 6^1 5^1 4^6 3^8 2^3)$ to 0.3%. But this strategy has little effect on many other instances.

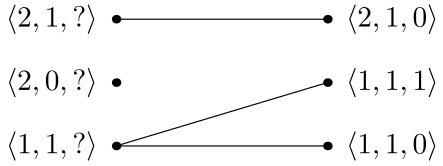

Fig. 12. Bipartite graph.

4.7. An example of the search process

Fig. 13 is an example of the search tree describing the process of constructing pairwise covering array $CA(5; 2, 4, 2)$. The notation "?" represents an unassigned cell. We use the SCEH strategy level 1 (i.e., $SL = 1$).

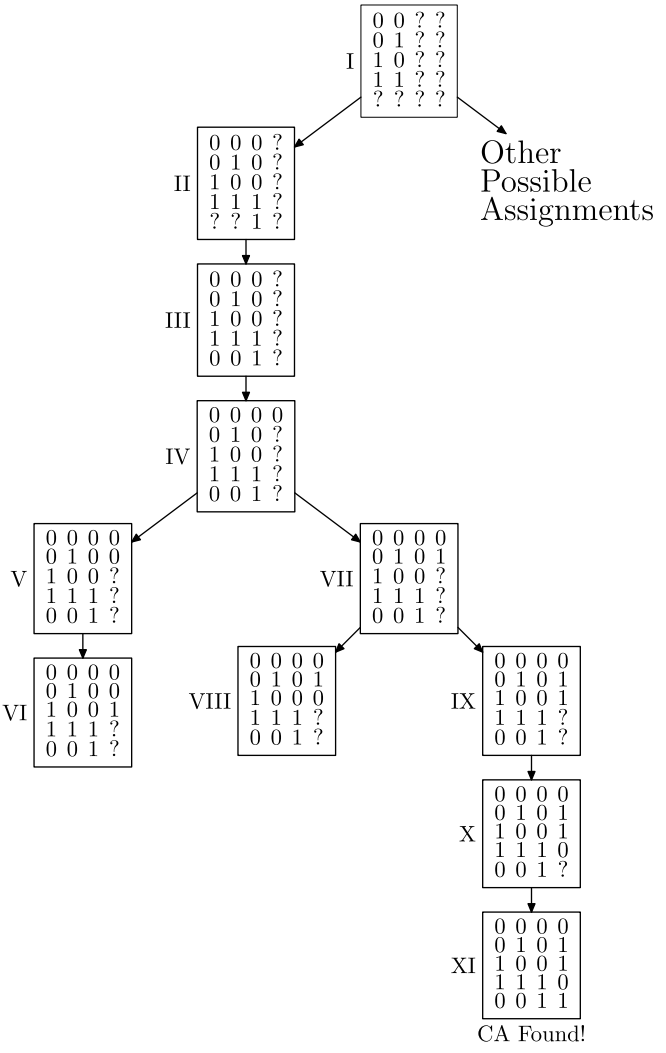

Fig. 13. Search tree.

The process begins at State I in which the cells of the mini-block are assigned. Then we try to assign possible values to the cells of column 3. The cells $ce_{4,3}$ and $ce_{5,3}$ are assigned the value 1 according to SCEH at State II. At State III, two cells $ce_{5,1}$ and $ce_{5,2}$ are assigned consecutively to cover all the pairs of the first 3 columns. At State IV, $ce_{1,4}$ can only get the value 0 according to LNH. Then there are 2 candidate values 0 and 1 for $ce_{2,4}$. At State V, we assign 0 to $ce_{2,4}$. At the next State VI, if $ce_{3,4}$ is assigned 0, then according to Marriage Theorem, there are 3 uncovered combinations of column 3 and 4 (the set X), and only 2 unassigned cells left (the set Y), $|Y| < |X|$. We cannot find a match and thus the value 0 for $ce_{3,4}$ is invalid. So we assign 1 to $ce_{3,4}$ and reach State VI. At the State VI, no candidate value for $ce_{4,4}$ is valid because of Marriage Theorem. So we backtrack to State IV. Similarly, we also find contradiction at State VIII. At last we find a covering array at State XI. If we need more than one covering arrays, we can backtrack and continue the search.

5. Producing large instances

The backtracking algorithm proposed in this paper is an exhaustive search method. We do not expect this algorithm to handle large instances. While as a software testing tool, EXACT should give at least one set of tests for a large instance. Fortunately, some mathematical results can help us produce some large arrays from smaller ones. In this section, we briefly describe some of these techniques for CA and MCA.

5.1. Large degree covering arrays

Firstly we consider pairwise testing. Stevens and Mendelsohn proved the following theorem (Stevens and Mendelsohn, 1999):

Theorem 5.1 (Products of strength-2 CAs). *If a* $CA(N; 2, k, v)$ *and a* $CA(M; 2, \ell, v)$ *both exist, then* $CA(N + M; 2, k\ell, v)$ *also exists.*

To be specific, let $A = (a_{ij})$ be $CA(N; 2, k, v)$ and $B = (b_{ij})$ be $CA(M; 2, \ell, v)$. Then the $(N + M) \times k\ell$ array $C = (c_{ij}) = A \otimes B$ in Fig. 14 is a $CA(N + M; 2, k\ell, v)$.

Consider $CAN(2, 30, 4)$. Our tool can generate A, which is an instance of $CA(16; 2, 5, 4)$, and B, which is a $CA(19; 2, 6, 4)$. Then we can produce a $CA(35; 2, 30, 4)$ instance $C = A \otimes B$. The IPO tool reports a $CA(46; 2, 30, 4)$ (Lei and Tai, 1998). The best result for this instance at present is 32 (Colbourn, 2007).

With this product method, for an instance $CAN(2, k, v)$, we can first use our tool to find a $CA(N; t, \lceil\sqrt{k}\rceil, v)$, where $\lceil x\rceil$ denotes the minimum integer not less than x. Then we can produce a $CA(2N; 2, k, v)$. This method can be used recursively.

There are similar product method for pairwise mixed covering arrays. Please refer to Colbourn et al. (2006) for details.

For covering arrays of strength 3, Chateauneuf and Kreher proved (Chateauneuf and Kreher, 2002):

Theorem 5.2. *If a* $CA(N; 3, k, v)$ *and a* $CA(M; 2, k, v)$ *both exist, then* $CA(N + (v - 1) \cdot M; 3, 2k, v)$ *also exists.*

For example, we can get two instances of $CA(125; 3, 5, 5)$ and one instance of $CA(25; 2, 5, 5)$. Then we can produce a $CA(225; 3, 10, 5)$. In contrast, the tool FireEye (Lei et al., 2007) reports that the covering array number is 308. The best result for this CA is 185 (Colbourn, 2007).

For the covering array strength 4, the product method is described in paper (Martirosyan and van Trung, 2004).

5.2. Large order covering arrays

Sometimes the SUT may have some components whose domains are not small. The following theorem is straightforward:

Theorem 5.3. $MCAN(t, k, v_1 \cdots (\ell v_i) \cdots v_k) \leqslant \ell \cdot MCAN(t, k, v_1 \cdots v_i \cdots v_k)$ *where* $0 < i \leqslant k$.

$$
\begin{array}{r|cccc|c|cccc|c|cccc}
N \text{ rows} & a_{11} & a_{12} & \cdots & a_{1k} & \cdots & a_{11} & a_{12} & \cdots & a_{1k} & \cdots & a_{11} & a_{12} & \cdots & a_{1k} \\
 & a_{21} & a_{22} & \cdots & a_{2k} & \cdots & a_{21} & a_{22} & \cdots & a_{2k} & \cdots & a_{21} & a_{22} & \cdots & a_{2k} \\
 & \multicolumn{4}{c|}{\vdots} & \cdots & \multicolumn{4}{c|}{\vdots} & \cdots & \multicolumn{4}{c}{\vdots} \\
 & a_{N1} & a_{N2} & \cdots & a_{Nk} & \cdots & a_{N1} & a_{N2} & \cdots & a_{Nk} & \cdots & a_{N1} & a_{N2} & \cdots & a_{Nk} \\
\hline
M \text{ rows} & b_{11} & b_{11} & \cdots & b_{11} & \cdots & b_{12} & b_{12} & \cdots & b_{12} & \cdots & b_{1\ell} & b_{1\ell} & \cdots & b_{1\ell} \\
 & b_{21} & b_{21} & \cdots & b_{21} & \cdots & b_{22} & b_{22} & \cdots & b_{22} & \cdots & b_{2\ell} & b_{2\ell} & \cdots & b_{2\ell} \\
 & \multicolumn{4}{c|}{\vdots} & \cdots & \multicolumn{4}{c|}{\vdots} & \cdots & \multicolumn{4}{c}{\vdots} \\
 & b_{M1} & b_{M1} & \cdots & b_{M1} & \cdots & b_{M2} & b_{M2} & \cdots & b_{M2} & \cdots & b_{M\ell} & b_{M\ell} & \cdots & b_{M\ell}
\end{array}
$$

Fig. 14. The product of $A \otimes B$.

For example, we can easily find an instance of *MCA* $(25; 2, 5^2 4^1 3^2 2^7)$. Then we can produce an *MCA* $(100; 2, 10^2 4^1 3^2 2^7)$. The tool FireEye (Lei et al., 2007) also reports the result 100 for this instance. Obviously this result is optimal.

Our tool can find optimal results for small cases. The above examples show that, for some large instances, the combination of mathematical method and our tool can lead to better results compared with direct search approaches. However, not all the large instances can be solved by the combined approach. For some instances, the greedy algorithms such as (Lei and Tai, 1998; Lei et al., 2007) give better results.

6. Experimental results

We have implemented the tool EXACT in the C programming language. Currently this tool supports the generation of CA (MCA) and OA (MOA). All the techniques mentioned in the previous section are implemented in the tool. The candidate value choosing strategy is disabled by default since it is not always efficient. All the parameters of the CA, MCA, OA or MOA are given by the user. The configurations of the strategies have the following differences for the two types of problems:

(i) The mini-blocks of CA and OA are different;
(ii) For OA, we use different SB constraints to eliminate the row and column isomorphism;
(iii) For OA, the SCEH strategy is enabled by default and $SL = t$; while for CA the SL values are given by the user.

We list some experimental results in this section. We first use some instances to illustrate the runtime performance of EXACT, then list a new CA instance found by the tool. At last, two examples are used to demonstrate how to apply our tool to real test scenarios. In this section except for Section 6.2, we run our program EXACT on a Pentium IV 3.2 GHz PC with Gentoo Linux operating system and all the times are measured in seconds.

6.1. Some instances of OA and MOA

We have applied our tool EXACT to some instances of OA and MOA with strength 2, 3, 4. Some results are listed in Table 3. The instances of strength 3 come from Brouwer et al. (2006).

6.2. Some instances of CA and MCA

Since our previous work (Yan and Zhang, 2006) has compared EXACT with other tools or methods on some CA and MCA instances, we only pick out some results of the comparison between EXACT and other two methods: AETG (Cohen et al., 1997) and SA (Cohen et al., 2003) (these two methods will be discussed in Section 7), and list them in Table 4. These small instances with $N < 100$ and $k < 30$ and the data of SA and AETG are obtained from paper (Cohen et al., 2003). EXACT were run on a Pentium IV 2.4 GHz PC and the times are measured in seconds. We can see from the table that our method outperforms AETG and SA in some small cases. In fact, the covering array numbers calculated by EXACT in Table 4 are all the best results we know till now.

In general, the time complexity (or the size of search space) $\prod_{i=1}^{k} v_i^N$ determines the time costs for these different instances, and that is the reason why we restrict our tool to small instances (for CA and MCA, $N < 100$ and $k < 30$). However, some other factors may reduce this search space and remarkably affect the time consumptions. For instance, the higher strength t leads to tighter constraints on the matrix and reduces more search spaces. Also if an instance has many isomorphic solutions, our symmetry breaking techniques will decrease much time consumption. It is not easy to say which factor dominates the time costs.

Table 3
Some orthogonal arrays

N	t	Parameters	Time (s)
24	2	$6^1 4^1 2^{11}$	0.02
36	2	$6^2 2^{10}$	159.43
40	2	$10^1 4^1 2^{11}$	635.61
48	2	$8^1 6^1 2^{11}$	401.14
60	2	$15^1 2^{12}$	468.33
40	3	$5^1 2^6$	0.83
48	3	$4^1 2^{11}$	0.25
54	3	$3^5 2^1$	1.08
56	3	$7^1 2^6$	30.23
64	3	$4^4 2^6$	28.23
64	3	2^{32}	61.54
64	4	$4^1 2^6$	0.03
96	4	$3^1 2^7$	27.34
128	4	$4^3 2^3$	0.41
243	4	3^7	2.48

Table 4
Some CA and MCA instances

Instance	AETG	SA	EXACT		
			N	Time (s)	SL
$CAN(2, 3^4)$	9	9	9	0.001	1
$CAN(2, 3^{13})$	17	16	15	0.188	1
$CAN(2, 2^{100})$	–	12[a]	10	0.062	1
$CAN(3, 4^6)$	77	64	64	0.016	2
$MCAN(2, 5^1 3^8 2^2)$	20	15	15	0.016	1
$MCAN(2, 7^1 6^1 5^1 4^5 3^8 2^3)$	42	42	42	0.500[b]	1
$MCAN(2, 5^1 4^4 3^{11} 2^5)$	28	21	21	20.297	1
$MCAN(2, 6^1 5^1 4^6 3^8 2^3)$	35	30	30	3.656[b]	1
$MCAN(2, 4^1 3^{39} 2^{35})$	27	21	21	73.734	1

[a] This result is generated by GA algorithm since the SA data is not available in the paper.
[b] We use candidate value choosing strategy only on these two instances.

6.3. New CA instance

The lexicographic order Rule 3 was not given in our previous work (Yan and Zhang, 2006). Recently we implemented this rule in EXACT. The new version of EXACT performs much better on some instances. Using the tool, we found a new instance $CA(24;4,12,2)$ which has not been found by other researchers, and it has been included in Colbourn (2007).[1] The previous best covering array number from Colbourn (2007) is 37.

In fact, our result is optimal. According to Chateauneuf et al. (1999),

$$CAN(t,k,v) \geqslant v \cdot CAN(t-1,k-1,v)$$

The experimental results of Hnich et al. (2006) showed that $CAN(3,11,2) = 12$. Therefore $CAN(4,12,2) \geqslant 24$.

6.4. Two test scenarios

In this subsection, we show how to use our tool EXACT to generate test cases satisfying the CA criterion. Since we may not know the minimum covering number, we use our tool to try different covering array number N. We first try $N = mb$ (mb is the number of rows in mini-block, see Section 4.1). If we cannot find a covering array, then we try $N = mb + 1, mb + 2, \ldots$ until we can find a solution. To find a test suite quickly, we can put a limit on the processing time of each try. In the following, we set the time limit to be 30 s.

6.4.1. $LaTeX$ Font System

Suppose we have a LaTeX system. We want to test whether we have properly installed the font component. For the CM (Computer Modern) fonts provided with TeX and LaTeX, the following attributes and values exist (Kopka and Daly, 2004):

- Size: the size of the font. There are totally 10 available font sizes from `tiny` to `Huge`;
- Family: the general overall style. The standard LaTeX installation provides three families: `rmfamily`, `ttfamily` and `sffamily`;
- Shape: the shape of the font. The shape declarations available with the standard installation are: `upshape`, `itshape`, `slshape` and `scshape`;
- Series: the width and weight of the font. The declarations possible are `mdseries` and `bfseries`.

Then our system has 10 available font sizes, 3 families, 4 font shape styles and 2 Series. To completely test the fonts with these attributes, we need 240 tests. But if we use pairwise testing, this test case generation problem can be formulated as $MCA(N;2,10^1 4^1 3^1 2^1)$. For this problem, $mb = 40$. With our tool EXACT and setting $SL = 2$, we can find an instance of $MCA(40;2,10^1 4^1 3^1 2^1)$ in 0.001 s. Thus, we only need 40 test characters to test our font system. Obviously, this is the optimal number of test cases.

6.4.2. Unix `sort` utility

Chapter 12 of Mathur (2007) lists all the 20 parameters of the `sort` command for Unix and their corresponding levels. All the levels of these factors lead to a total of approximately 1.9×10^8 combinations. But if we apply pairwise testing method to this function, we can formulate the problem as finding a mixed level covering array $MCA(N;2,4^4 3^6 2^{10})$ and $mb = 16$. The tool EXACT reports a covering array $MCA(18;2,4^4 3^6 2^{10})$ with $SL = 1$. The running time for $N = 18$ is 1.53 s.

7. Related work

Most combinatorial test generation tools focused on generation a test set for CA criterion. Among these approaches, most of the algorithms are approximate since the problem is NP-complete. Among the approximate methods, the greedy algorithms provide the fastest solving method and are well-researched. The IPO algorithm (In-Parameter-Order) (Lei and Tai, 1998) and its improved version IPOG (Lei et al., 2007), repeat the two steps to find a test set:

- Horizontal growth, which extends each existing test by adding one value for the new parameter;
- Vertical growth, which adds new tests (if needed) to the test set produced by horizontal growth.

To handle very large scale of covering arrays, Kuhn proposed an algorithm Paintball (Kuhn, 2006) that can be parallelized. This algorithm processes faster than IPOG, but produces more test cases. The commercial tools AETG (Automatic Efficient Test Generator) (Cohen et al., 1997) and TCG (Test Case Generator) (Tung and Aldiwan, 2000), use one-test-at-a-time methods. The main idea is, the algorithm constructs a test set by repeatedly adding one test (this test covers the most uncovered combinations) at a time until all the combinations are covered.

Some recent stochastic algorithms in artificial intelligence, such as tabu search (Nurmela, 2004), SA (Simulated Annealing) (Cohen et al., 2003), GA (Generic Algorithm) and ACA (Ant Colony Algorithm) (Shiba et al., 2004) provide an effective way to find approximate solutions. Shiba et al. (2004) showed that the SA is the most effective probabilistic search technique. This type of methods can reach more optimal results than the greedy methods.

For the exact algorithms, Hnich et al. (2006) applied constraint programming (CP) techniques to this problem, but the performance degrades significantly as the problem size increases. Compared with the CP approach, our tool uses more constraint satisfaction techniques, and is much more efficient (Yan and Zhang, 2006).

[1] In fact, we reported two new results to C. J. Colbourn in June 2007, and he confirmed them. Later it occurred to him that $CAN(5,7,2) = 42$ has been established by other researchers.

Mathematicians use some algebraic and combinatorial methods to construct covering arrays. Colbourn reviews these methods on CA (Colbourn, 2004). Only a few works aim to construct MCAs. The paper (Colbourn et al., 2006) studied the producing method on strength 2 MCAs.

Most constructing method for OA and MOA are based on mathematical results. For example, Hadamard construction, juxtaposition, etc. (Colbourn and Dinitz, 2006; Brouwer et al., 2006).

8. Conclusion

We proposed an exhaustive search based tool EXACT to generate combinatorial test cases in this paper. Some techniques are employed to improve the backtrack search. With all these techniques, the tool can process small instances effectively for CA and OA testing criteria. Since the backtracking algorithm may cost much time when processing large scale instances, we also introduced some mathematical method to produce large instances from small cases generated by EXACT.

Future works include some expansion on the scope of our tool. Currently our tool only supports the basic CA and OA test criteria. Some researchers introduced some modified combinatorial test criteria. The AETG system supports complex relation between the fields of different components (Dalal et al., 1999). For example, the following test specification of AETG describes the relation between the components `b`, `c`, and `d`.

```
if b < 9 then c >= 8 and d <= 3
```

Cheng et al. (2003) proposed a weakened CA criterion that the test suites should cover the input–output relations. These criteria are useful for some special test scenarios. To adapt to these criteria, the tool EXACT may need some well designed strategies.

References

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

Chateauneuf, M., Kreher, D., 2002. On the state of strength-three covering arrays. Journal of Combinatorial Designs 10 (4), 217–238.

Chateauneuf, M.A., Colbourn, C.J., Kreher, D.L., 1999. Covering arrays of strength three. Designs, Codes and Cryptography 16 (3), 235–242.

Cheng, C.T., Dumitrescu, A., Schroeder, P.J., 2003. Generating small combinatorial test suites to cover input–output relationships. In: Proceedings of 3rd International Conference on Quality Software (QSIC'03), pp. 76–83.

Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C., 1997. The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23 (7), 437–443.

Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J., 2003. Constructing test suites for interaction testing. In: Proceedings of the 25th International Conference on Software Engineering (ICSE'03), pp. 38–48.

Colbourn, C.J., 2004. Combinatorial aspects of covering arrays. Le Matematiche (Catania) 58, 121–167.

Colbourn, C.J., Dinitz, J.H. (Eds.), 2006. Handbook of Combinatorial Designs, second ed.. Discrete Mathematics and its Applications Chapman & Hall/CRC.

Colbourn, C.J., Martirosyan, S.S., Mullen, G.L., Shasha, D., Sherwood, G.B., Yucas, J.L., 2006. Products of mixed covering arrays of strength two. Journal of Combinatorial Designs 14 (2), 124–138.

Colbourn, C.J., 2007. CA tables for $t = 2, 3, 4, 5, 6$. <http://www.public.asu.edu/~ccolbou/src/tabby/catable.html>.

Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M., 1999. Model-based testing in practice. In: Proceedings of the 21th International Conference on Software Engineering (ICSE'99), pp. 285–294.

Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T., 2002. Breaking row and column symmetries in matrix models. In: Proceedings of 8th International Conference on Principles and Practice of Constraint Programming, LNCS 2470, pp. 462–477.

Godbole, A.P., Skipper, D.E., Sunley, R.A., 1996. t-covering arrays: upper bounds and Poisson approximations. Combinatorics, Probability and Computing, 105–118.

Grindal, M., Offutt, J., Andler, S.F., 2005. Combination testing strategies: a survey, software testing. Verification and Reliability 15 (3), 167–199.

Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M., 2006. Constraint models for the covering test problem. Constraints 11 (2–3), 199–219.

Kopka, H., Daly, P.W., 2004. Guide to $\LaTeX$, fourth ed. Addison–Wesley, Boston, MA.

Kuhn, D.R., 2006. An algorithm for generating very large covering arrays. Technical report NISTIR 7308.

Kuhn, D.R., Reilly, M.J., 2002. An investigation of the applicability of design of experiments to software testing. In: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW'02), pp. 91–95.

Lei, Y., Tai, K.C., 1998. In-Parameter-Order: a test generation strategy for pairwise testing. In: Proceedings of 3rd IEEE International Symposium on High Assurance Systems Engineering, pp. 254–261.

Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J., 2007. IPOG: a general strategy for t-way software testing. In: Proceedings of 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), pp. 549–556.

Martirosyan, S., van Trung, T., 2004. On t-covering arrays. Designs, Codes and Cryptography 32 (1), 323–339.

Mathur, A.P., 2007. Foundations of software testing, draft v4.x Edition. <http://www.cs.purdue.edu/homes/apm/foundationsBook/>.

Meagher, K., Stevens, B., 2005. Covering arrays on graphs. Journal of Combinatorial Theory, Series B 95 (1), 134–151.

Nurmela, K.J., 2004. Upper bounds for covering arrays by tabu search. Discrete Applied Mathematics 138 (1–2), 143–152.

Schroeder, P.J., Bolaki, P., Gopu, V., 2004. Comparing the fault detection effectiveness of n-way and random test suites. In: Proceedings of International Symposium on Empirical Software Engineering (ISESE'04), pp. 49–59.

Seroussi, G., Bshouty, N.H., 1988. Vector sets for exhaustive testing of logical circuits. IEEE Transactions on Information Theory 34, 513–522.

Shiba, T., Tsuchiya, T., Kikuno, T., 2004. Using artificial life techniques to generate test cases for combinatorial testing. In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), pp. 72–77.

Sloane, N.J.A., 2007. A library of orthogonal arrays. <http://www.research.att.com/~njas/oadir/index.html>.

Stevens, B., Mendelsohn, E., 1999. New recursive methods for transversal covers. Journal of Combinatorial Designs 7 (3), 185–203.

Tung, Y.-W., Aldiwan, W.S., 2000. Automating test case generation for the new generation mission software system. In: Proceedings of IEEE Aerospace Conference, pp. 431–437.

Williams, A.W., Probert, R.L., 1996. A practical strategy for testing pairwise coverage of network interfaces. In: Proceedings of 7th Interna-

tional Symposium on Software Reliability Engineering (ISSRE'96), pp. 246–254.

Yan, J., Zhang, J., 2006. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In: Proceedings of 30th Annual International Computer Software and Applications Conference (COMPSAC'06), pp. 385–394.

Yilmaz, C., Cohen, M.B., Porter, A.A., 2006. Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Transactions on Software Engineering 32 (1), 20–34.

Zhang, J., 2001. Automatic symmetry breaking method combined with SAT. In: Proceedings of the 2001 ACM Symposium on Applied Computing (SAC'01), pp. 17–21.

Jun Yan received his bachelor's degree in E.E. from University of Science and Technology of China in 2001 and his Ph.D. degree in computer science from Institute of Software, Chinese Academy of Sciences (ISCAS) in 2007. He is now an assistant research professor at ISCAS. His research interests include constraint processing, program analysis and software testing.

Jian Zhang received his Ph.D. degree in computer science from Institute of Software, Chinese Academy of Sciences (ISCAS) in 1994. He is currently a research professor and an assistant director of ISCAS. His research interests include automated reasoning, formal methods, program analysis and software testing. He is a member of ACM and IEEE.