Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing*

Jun Yan[1,2] and Jian Zhang[1]
[1]Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2]Graduate University, Chinese Academy of Sciences
E-Mail: {yanjun,zj}@ios.ac.cn

Abstract

Combinatorial covering arrays have been used in several testing approaches. This paper first discusses some existing methods for finding such arrays. Then a SAT-based approach and a backtracking search algorithm are presented to solve the problem. A novel pruning strategy called SCEH is proposed to increase the efficiency of the methods. Several existing search heuristics and symmetry breaking techniques are also used in the backtracking search algorithm. Lastly, this paper introduces a tool called EXACT (EXhaustive seArch of Combinatorial Test suites) which implements all the above techniques to construct the covering arrays automatically. The experimental results show that our backtracking search method outperforms other methods in many small size cases.

1 Introduction

Combinatorial testing is a kind of black box testing. Suppose the system under test (SUT) has k parameters, each of which has v possible values. In order to test the SUT completely, we need v^k test suites. This number is often too big in practice. One trade off between the cost of testing and the required degree of coverage is to perform combinatorial testing. This approach is motivated by the observation that a significant number of faults are caused by interactions of a small number of parameters in many applications. So we can generate a test suite for the system, such that for every t parameters, each combination of valid values of these t parameters is covered by at least one test case. 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 [5].

*Supported by National Natural Science Foundation of China (No. 60125207, 60421001).

Example 1.1 *(Combinatorial Testing Configuration).* Suppose we have an e-commerce application which has four components each with three configurations, as shown in Figure 1. To test all possible interactions for this system we would need $3^4 = 81$ test cases.

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

Figure 1. A System With Four Components

But if we restrict ourselves to pairwise coverage (that is, $t = 2$), we can use only 9 test cases (see Figure 2) to test the system.

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

Figure 2. Test Suite Covering All Pairs

The key problem of combinatorial testing is: how to find a test suite which has the minimum size? This problem is an instance of the covering array problem. A covering array is a matrix, each row of which represents a test case. There are many applications of covering arrays. For example, we can use them in interaction software testing, hardware testing, advanced materials testing, and so on.

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.

In general, combinatorial testing is widely believed to be a systematic, effective testing technique. Yilmaz et al. [15] concluded that even low-strength covering arrays can provide up to 99% reduction in the number of configurations to be tested and had 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 [10], 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 4). Therefore we need new methods to design better combinatorial test suites.

Over the past two decades, mathematicians have studied covering arrays extensively and obtained many theoretical results. In addition, researchers in computer science have proposed some algorithms to generate the arrays automatically. It is hard to say which approach is better. Most algorithms are approximate, in the sense that they do not guarantee producing covering arrays of the minimum size. In contrast, the exact algorithms have not received much attention. In this paper, we compare these existing techniques and apply backtrack search to this problem. Our algorithm employs a number of search heuristics to enhance the efficiency. Our experiences show that the improved exhaustive search can be very efficient for some test sets.

This paper is organized as follows. We begin with an introduction of the covering array problem in the next section. Then in the following two sections we will discuss two classes of techniques, translation methods and constraint satisfaction methods, to build the test suites. Our method with some useful heuristics are presented in section 5. Then we present some computational results and compare our algorithm with some existing methods. The last section concludes.

2 Covering Arrays

Cohen et.al [3] 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**. A covering array is **optimal** if it contains the minimum possible number of rows. We call the minimum number **covering array number**, $CAN(t, k, v)$.

Definition 2.2 (Mixed level Covering Array). A mixed level covering array, $MCA(N; t, k, (v_1 v_2 \dots v_k))$ is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

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

We use a shorthand notation to describe mixed covering arrays by combining equal entries in v_i. For example three entries which are equal to 2 can be written as 2^3.

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.

We can display a covering array 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 test cases. Each entry in the matrix is called a *cell*, and we use $ce_{r,c}$ to denote the cell at row $r (r \geq 0)$ and column $c (c \geq 0)$, or the value of parameter c in test case r.

There are some mathematical results about Covering Array Numbers. 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, and will not be discussed in this paper. Instead, we focus on algorithms that can generate covering arrays automatically.

Seroussi and Bshouty [12] show that a generalized version of the problem of finding a minimal t-covering array is NP-complete. Also Lei *et al.* [8] 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.

3 Translation to SAT

The satisfiability (SAT) problem is the first proved NP-complete problem and there are many efficient SAT solvers nowadays. Generally speaking, all the NP-complete problems can be translated to SAT problems, and solved by a SAT solver. There are two types of SAT solvers, which are based on stochastic local search and systematic backtracking search, respectively.

Hnich *et al.* [6] have tried Walksat[11], a local search tool, to solve the problem. Their experimental results show that Walksat can handle some small cases in several minutes. Their results may not be the best covering array numbers. Some results by Walksat are listed in Table 3.

We use the SAT-encoding method similar to the method of [6] and tried to apply another SAT solver zChaff [9], which is one of the most efficient systematic search based SAT solvers, to this problem. Generally speaking, the original clause sets are too hard for zChaff to solve. To reduce

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.

the search time, we use the tool Shatter [1] to break symmetries in the SAT clauses before calling zChaff (see Section 5.3 for more about symmetries.) Shatter reads a set of clauses and detects the symmetries in it. Some experimental results are shown in Table 4. We can see that the results are disappointing. It costs much time in some small cases.

4 Existing Direct Search Methods

In the method of SAT-translation, the SAT clause sets of covering array problem may be very huge even for some small instances and this restricts the use of SAT methods. Some researchers make use of the classic algorithms for solving constraint satisfaction problems (CSP) and optimization problems to search covering arrays directly. Most of the algorithms are approximate since the problem is NP-complete. The greedy algorithms, for example, the IPO algorithm (IN Parameter-Order) [8], the commercial tools AETG (Automatic Efficient Test Generator) [2] and TCG (Test Case Generator) [14], provide the fastest solving methods. Some recently stochastic algorithms, such as SA(Simulated annealing) [3], GA (Generic Algorithm) and ACA (Ant Colony Algorithm) [13] provide an effective way to find approximate solutions. This type of methods can reach more optimal results than the greedy methods. For the exact algorithms, Hnich et al. [6] applied constraint programming(CP) techniques to this problem, but the performance degrades significantly as the problem size increases.

5 The Backtrack-Based Search Approach

We apply exhaustive search technique to this problem. Our algorithm is based on backtrack search. Compared with the CP approach of [6], more constraint satisfaction techniques are used, and our tool is much more efficient than CP.

5.1 The Basic Procedure

Our main algorithm is written in Figure 3 as a recursive procedure. For a given covering array number N, we try to assign each cell of the $N \times k$ matrix until all requirments of covering array 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 or contradiction. The function `CELL_Selection` chooses an unassigned cell and function `Val_Set_Gen` generates the candidate value set S_x of a cell.

The time complexity of exhaustive search of $MCA(N; t, k, v_1 v_2 \dots v_k)$ 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 improved by forming a proper order in assigning the cells. We will not try cells outside the columns $\{0, 1, \dots, c\}$ until all the constraints between these columns are satisfied. Another way to improve it is discovering the inconsistencies as early as possible.

```
bool BSrh(pSol, Fmla) {
    BPropagate(pSol, Fmla);
    if (contradiction results) 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;
}
```

Figure 3. Exhaustive Search

5.2 Preprocessing

Without loss of generality, assume we are processing a $MCA(N; t, k, v_1 v_2 \dots v_k)$ problem with $v_1 \geq v_2 \geq \dots \geq v_k$. Then consider the first t columns of the matrix. There are $mb = v_1 \cdot v_2 \cdots v_t$ 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. This simple technique reduces much wasted backtracking.

Example 5.1. The 4×2 sub-array in the upper left corner of the covering array of Figure 7 forms a mini-block.

5.3 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 5.2 *(Two Isomorphic CAs).* The two covering arrays of $CA(4; 2, 3, 2)$ of Figure 4 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 matries which are isomorphic to those enumerated before by break-

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.

(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

Figure 4. Two Isomorphic CAs of $CA(4; 2, 3, 2)$

ing symmetries. Symmetry breaking 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 to our algorithm right after the the function `Val_Set_Gen`.

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

Figure 5. The Symmetry-Breaking Function

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

5.3.1 Row and Column Symmetries

Flener *et al.* [4] introduced an important class of symmetries in constraint programming, arising from matrices of decision variables where rows and columns can be swapped. In general, an $m \times n$ matrix has $m! \cdot n! - 1$ symmetries excluding the identity, and generates thus a super-exponential number of lexicographic ordering constraints. Hence this approach is not always practical.

We make use of two types of symmetries of 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 lexigraphically smaller (denoted by $\leq_{lex}$) than the next (if any). We can give a recursive definition of the lexigraphic order as follows:

Definition 5.3 (Lexigraphical Order). Two vectors $X = [x_1, x_2, \ldots x_n]$ and $Y = [y_1, y_2, \ldots y_n]$ have a lexigraphical order $X \leq_{lex} Y$ **iff**

1. If $n = 1$, $x_1 \leq y_1$;
2. If $n > 1$, $x_1 < y_1 \vee (x_1 = y_1 \wedge [x_2, \ldots, x_n] \leq_{lex} [y_2, \ldots, y_n])$.

We add constraints that each column(row) is lexigraphically smaller than the next column(row) as symmetries elimination rule. Here is an example:

Example 5.4. For the matrix (b) of Example 5.2, the first column is not lexigraphically smaller than the second one, so we will not encounter the second matrix during the search.

When handling the mixed level covering array, the column lexicographical order can only be used between the columns with the same number of valid values. In addition, since we have a pre-fixed mini-block, we only need to use the lexicographical order on those rows and columns outside the mini-block.

5.3.2 LNH

The LNH (Least Number Heuristic) [16] 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 element names(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 elements of a given domain. It makes no difference to assign the element e or e' to the cell, if neither of them has been assigned to a cell. To make this concise, we say that two elements e and e' are symmetric.

When processing a column c, since all the cells of the column share the same domain $D_c = \{0, 1, 2, \ldots, v_c - 1\}$, all the unused values are the same to the unassigned cell. We use a program variable mdn to represent the largest value used in the assigned cells. Then the candidate value set for the next unsigned cell is $\{0, 1, 2, \ldots, mdn + 1\}$. The initial value of mdn is -1 and it is dynamically updated during the search.

5.4 Pruning

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 symmetries breaking methods, these constraints are applied by a function `Pruning` inserted in our search algorithm after set S_x is generated. We provide a novel pruning technique called SCEH (The Sub-Combination Equalization Heuristic) for this problem.

The motivation of the sub-combination equalization heuristic comes from the observation that in many covering arrays, for any column of the matrix, each value appears almost the same number of times. This rule is also effective

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.

when we extend it to value-tuples. Here we introduce the definition of Sub-combination first.

Definition 5.5 (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. We use $ce_{r,C}$ to denote a cell vector of size s, the k'th element of $ce_{r,C}$ is ce_{r,c_k} where c_k is the k'th element of C. The frequency of the sub-combination, denoted by f_C^V, is defined as $f_C^V = \sum_{ce_{r,C}=V} 1$ which means there are f_C^V rows (cases) containing the sub-combination SC_C^V in the array.

Then we can summarize the rule as a conjecture here:

Conjecture 5.6 (SCEH). *Given a covering array problem $CA(N; t, k, v)$ or $MCA(N; t, k, (v_1 v_2 \dots v_k))$, we can find at least one solution such that for a Sub-combination of size $s(0 < s \leq t)$, we have $|f_C^{V_1} - f_C^{V_2}| \leq 1$ for all V_1 and V_2 where $V_1 \neq V_2$.*

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 $V_1 \neq V_2$. This conclusion can be easily deduced from the conjecture.

Till now, we cannot prove this conjecture; but on the other hand, we have not found any counter-example. We can use an example to show this rule:

Example 5.7. There is a covering array $CA(11; 2, 5, 3)$ in Table (A) of Figure 6. Consider $s = 1$, then we have $\binom{k}{s} =$

(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	$\langle a \rangle$	$\langle b \rangle$	$\langle c \rangle$
Column 0	4	4	3
Column 1	4	4	3
Column 2	4	4	3
Column 3	4	3	4
Column 4	3	4	4

Figure 6. The Value Vector's Frequency

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 frenquencies of different vectors with the same column are almost the same.

We use the heuristic of $0 < s \leq SL$ in practice, where SL is the strategy level specified by users. In practice, we can choose $SL = t - 1$. The strategy is applied to our algorithm by estimating the upper-bound UP and lower-bound LB of the frequency of each sub-combination. When an assignement $x = u$ can cause the frequency of a sub-combination to overstep the range $[LP, UP]$, the value u will be eliminated from set S_x.

We use this strategy to reduce the search space. It can speed up almost all instances we have tried. The efficiency of this strategy is shown in Table 1.

5.5 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 combintions. 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 processing column C, if the frequency of all value-combinations except vc reached their upper-bound, the remain unsigned cells of C should be assigned to the value to fit for vc.

We can also use a graph 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 to 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 classic theorem of matching problem in 1935:

Theorem 5.8 (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 *$|N(S)| \geq |S|$ for every $S \subseteq X$, where $N(S)$ is the neighbor set of S which is defined to 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 $N(X) \leq |Y|$, if $|Y| < |X|$, there is a contradiction.

Unlike other strategies of symmetry breaking and pruning, this strategy doesn't remove any value that may lead to a solution from the set S_x.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.

5.6 Candidate Value Choosing Strategy

When we are processing a cell which has more than one candidate values, usually we will choose the first value, but often it is not the best choice. We may improve this situation by estimating which value is most likely to lead to a solution. We use a strategy 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 level covering arrays (especially those instances whose domains of parameters vary significantly). A most illustrating example is that, it can reduce the search time of $MCA(30; 2, 6^1 5^1 4^6 3^8 2^3)$ from 1161.593s to 3.656s. But this strategy has little effect on other instances.

5.7 An Example of the Search Process

Figure 7 is an example of a search tree describing the process of constructing pairwise covering array $CA(5; 2, 4, 2)$. The notation "?" represents an unsigned cell. We use the SCEH strategy level 1. The process begins

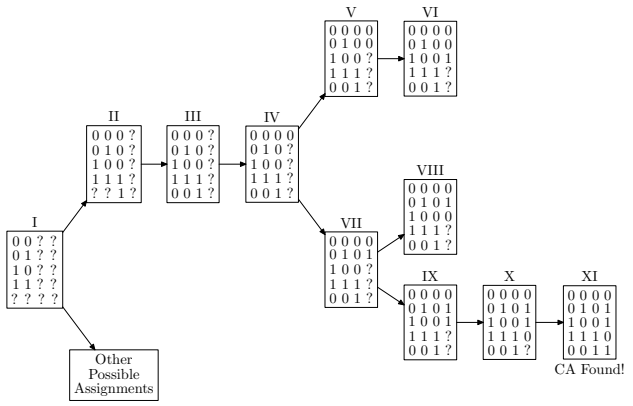

Figure 7. Search Tree

at State I with the cells of mini-block are assigned. Then we try to assign possible values to the cells of column 2. The cell $ce_{3,2}$ and $ce_{4,2}$ are assigned to 1 according to SCEH at State II. At State III, two cells $ce_{4,0}$ and $ce_{4,1}$ are assigned consequently to cover all the pairs of the first 3 columns. At State IV, $ce_{0,3}$ can only be assigned to 0 according to LNH. Then there are 2 candidate values 0 and 1 for $ce_{1,3}$. At State V, we assign 0 to $ce_{1,3}$. Then at the next State VI, if $ce_{2,3}$ is assigned to 0, then according to Marriage Theorem, there are 3 uncovered combinations of column 2 and 3(the set X), and only 2 unsigned cells left(the set Y), $|Y| < |X|$, we can not find a match and thus the value 0 for $ce_{2,3}$ is invalid. So we assign $ce_{2,3}$ with 1 and reach State VI. At the State VI, no candidate value for $ce_{3,3}$ are 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.

6 Experimental Results

We developed a covering array generator EXACT (EXhaustive seArch of Combinatorial Test suites) in the C programming language. All the techniques mentioned in the previous section are implemented in the program. Only the candidate value choosing strategy is disabled by default since it's not always efficient for all instances. All the parameters of a CA or MCA, as well as the SL values, are given by the user. We run our program EXACT on a Pentium IV 2.4GHz PC with Gentoo Linux operating system, and compare our algorithm with other existing algorithms. The results are summarized as follows. All the execution times are measured in seconds and the times less than 1 millisecond are treated as 0.001s.

6.1 A Real Instance

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, there are four attributes, **Size**, **Family**, **Shape** and **Series**[7]. Suppose our system have 10 available font sizes, 3 families, 4 font shape styles and 2 Series. We can use several characters with different font attributes to test our font component. To completely test the fonts with these attributes, we need 240 tests. But if we use pairwise test, we will soon get an instance of $MCA(40; 2, 10^1 4^1 3^1 2^1)$ in 0.001s with $SL = 2$ by our tool EXACT. Thus we only need 40 test characters to test our font system.

6.2 The Efficiency of the Strategies

The efficiency of a pruning strategy lies in two ways: It should decrease the searching time of the first solution and it should find at least one solution. We use some experiments to show the efficiency of two strategies: LNH and SCEH, since this is the first time that they are applied to the covering array problem.

We use our tool to find all the solutions of two small instances with different configurations of these two strategies. All the other symmetry-breaking and pruning techniques except the candidate value choosing strategy are used in each experiment. The number of loops for finding the first solution and the total number of solutions are listed in Table 1 and Table 2, respectively. In our program of EXACT, one

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.

loop means assigning a value to a cell once or backtracking one level. Both these two techniques can decrease the

Table 1. The Efficiency of SCEH

Covering Array	SL		
	0	1	2
$CA(6;2,2^5)$	47/120	15/56	-
$CA(12;3,2^6)$	54649/3180	67/3180	818/448

Table 2. The Efficiency of LNH

Covering Array	Use LNH?		SL
	Yes	No	
$CA(6;2,2^5)$	15/56	21/2912	1
$CA(12;3,2^6)$	818/448	1549/23296	2

search space. We can see from these two tables that SCEH can reduce much useless search and LNH can prune many redundant solutions.

6.3 Comparison with SAT-based Methods

Paper [6] applied the Walksat local search algorithm to the CP model. They ran Walksat on decreasing values of N until no solution was found after several minutes on a Pentium III 733 MHz PC. Table 3 shows comparison with Walksat. We only want to compare with Walksat, so the results may not be the best since searching for an optimal solution may cost much time. Walksat is a very efficient

Table 3. Comparison With Walksat

Covering Array	Walksat Result	EXACT		
		Result	Runtime	SL
$CAN(3,2^{13})$	16	16	0.430	2
$CAN(3,2^{14})$	17	16	0.440	2
$CAN(3,2^{15})$	18	18	0.050	2
$CAN(3,2^{16})$	18	18	1.460	2
$CAN(4,2^9)$	24	24	0.609	3
$CAN(4,2^{10})$	24	24	0.610	3
$CAN(4,3^5)$	81	81	0.656	1

algorithm for SAT problems. EXACT can easily solve most instances Walksat can solve.

We applied zChaff to this problem and our method outperforms zChaff in processing speed for all the instances we tried. We ran Shatter (version 0.3) and zChaff (version 2004.11.15 Simplified) and the maximum execution times of these two tools are limited to 2 hours. Table 4 lists the time cost of Shatter, zChaff and EXACT for some small instances which are very "easy" for our algorithm.

Table 4. Runtime Comparison With zChaff

Covering Array	zChaff Methods		EXACT	
	Shatter	zChaff	Runtime	SL
$CA(9;2,3^4)$	0	0.009	0.001	1
$CA(12;3,2^{11})$	341	3.328	0.016	2
$CA(15;2,3^{13})$	198	T/O	0.188	1
$CA(24;4,2^7)$	150	54.593	0.594	3
$MCA(15;2,5^1 3^8 2^2)$	88	19.377	0.016	1

6.4 Comparison with Direct Search Methods

We compared our method with two existing direct search appraoch, CP(an exact method) and SA(approximate local search), in this section. The problem instances and data of CP were taken from [6] and the tool was run on a Pentium IV 1.8GHz machine. Table 5 shows the runtime comparison of our algorithm with CP.

Table 5. Runtime Comparison with CP by Some Instances Of $CA(N;3,k,2)$

k	N	CP Runtime	EXACT	
			Runtime	SL
6	12	3.92	0.016	2
7	12	13.11	0.016	2
8	12	21.86	0.016	2
9	12	75.47	0.016	2
10	12	108.83	0.016	2
11	12	140.90	0.016	2

We can see that our technique outperforms the CP approach significantly in terms of processing time, since we have used more efficient strategies to decrease the search space.

The paper [13] showed that the SA(annealing-based algorithm) is the most effective probabilistic search technique. Similar to our approach, the SA algorithm chooses a fixed size array and tries to manipulate the space until all constraints are met. The instances and data of SA were obtained from paper [3] in which those instances were tested on a Pentium IV 1.8GHz computer. Table 6 reports the execution times on instances with $N < 100$ and $k < 30$.

We can see from the table that our method outperforms SA in some small cases. In fact, the covering array number N caculated by EXACT in the Table 6 are all the best results we know till now. But when the size of the instances grows, the running time increases notably. We have some difficulties with larger size instances. For example, our program can not solve a frequently used benchmark $MCA(N;2,4^{15}3^{17}2^{29})$.

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.

Table 6. Comparison With SA

Covering Array	SA Method		EXACT		
	N	Runtime	N	Runtime	SL
$CAN(2, 3^4)$	9	-	9	0.001	1
$CAN(2, 3^{13})$	16	-	15	0.188	1
$CAN(2, 2^{100})$	12[a]	-	10	0.062	1
$CAN(3, 4^6)$	64	-	64	0.016	2
$MCAN(2, 5^1 3^8 2^2)$	15	214	15	0.016	1
$MCAN(2, 7^1 6^1 5^1 4^5 3^8 2^3)$	42	874	42	0.500[b]	1
$MCAN(2, 5^1 4^4 3^{11} 2^5)$	21	379	21	20.297	1
$MCAN(2, 6^1 5^1 4^6 3^8 2^3)$	30	579	30	3.656[b]	1
$MCAN(2, 4^1 3^{39} 2^{35})$	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.

7 Conclusion

We summarized some automatic methods to construct the covering arrays. In the SAT-translation methods, Walksat can give some approximate results and costs little time. The systematic SAT solvers may perform better if some special symmetry breaking constraints are added to the clause set. For direct search methods, heuristic search techniques can handle some larger scale instances in acceptable time, but these techniques can not assuredly give the optimal solutions. The exhaustive search methods including our approach can only handle some instances efficiently till now, but they can give the best solution if we have enough time. There is a trade off between the time cost and the precision of the required covering array numbers.

We use some techniques to improve our backtrack search. We proposed a novel pruning technique SCEH. There are also some other techniques such as LNH which are first used in this problem. With all these techniques, our method performs better than other exhaustive search methods. In some small cases, our technique outperforms all other methods. But when processing some large scale instances, our method will cost much time. There may be some techniques to improve our work in the future. For example, most techniques we use during the search except for the column lexicographical order are based on the constraints between cells in the same column. So if we can find some more inter-column constraints, we may speed up the whole processing. Also some CSP techniques, such as conflict analysis and backjumping, will improve efficiency. We also need some efficient heuristic techniques in choosing the candidate values.

References

[1] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Design Automation Conference(DAC'03)*, 2003.

[2] D. Cohen, S. Dalal, M. Fredman, and G. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Software Eng.*, 23(7):437–443, 1997.

[3] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th international Conference on Software Engineering(ICSE'03)*, 2003.

[4] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *CP2002, LNCS 2470*, pages 462–477, 2002.

[5] A. Godbole, D. Skipper, and R. Sunley. t-covering arrays: upper bounds and Poisson approximations. *Combinatorics, Probability and Computing*, pages 105–118, 1996.

[6] B. Hnich, S. Prestwich, and E. Selensky. Modeling the covering test problem. In *CSCLP 2004: Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming*, 2004.

[7] H. Kopka and P. W. Daly. *Guide to* LaTeX. Addison–Wesley, Boston, MA, fourth edition, February 2004.

[8] Y. Lei and K. Tai. In-Parameter-Order: A test generation strategy for pairwise testing. In *Proceedings of 3rd IEEE Intl. Symp. on High Assurance Systems Engineering*, pages 254–261, 1998.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th conference on Design automation table of contents*, pages 530–535, 2001.

[10] P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *2004 International Symposium on Empirical Software Engineering (ISESE'04)*, 2004.

[11] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *AAAI-92: Proceedings 10th National Conference on AI*, 1995.

[12] G. Seroussi and N. Bshouty. Vector sets for exhaustive testing of logical circuits. *IEEE Trans. Information Theory*, 34:513–522, 1988.

[13] T. Shiba, T. Tsuchiya, and T. Kikuno. 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)*, 2004.

[14] Y. Tung and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *Proceedings of IEEE Aerospace Conference*, pages 431–437, 2000.

[15] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1), January 2006.

[16] J. Zhang. Automatic symmetry breaking method combined with SAT. In *Proceedings of the 2001 ACM Symposium on Applied Computing(SAC'01)*, 2001.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 05:40:35 UTC from IEEE Xplore. Restrictions apply.