# A New Backtracking Algorithm for Constructing Binary Covering Arrays of Variable Strength*

Josue Bracho-Rios, Jose Torres-Jimenez, and Eduardo Rodriguez-Tello

CINVESTAV-Tamaulipas, Information Technology Laboratory.
Km. 6 Carretera Victoria-Monterrey, 87276 Victoria Tamps., MEXICO
{jbracho,jtj,ertello}@tamps.cinvestav.mx

**Abstract.** A Covering Array denoted by CA$(N; t, k, v)$ is a matrix of size $N \times k$, in which each of the $v^t$ combinations appears at least once in every $t$ columns. Covering Arrays (CAs) are combinatorial objects used in software testing. There are different methods to construct CAs, but as it is a highly combinatorial problem, few complete algorithms to construct CAs have been reported. In this paper a new backtracking algorithm based on the Branch & Bound technique is presented. It searches only non-isomorphic Covering Arrays to reduce the search space of the problem of constructing them. The results obtained with this algorithm are able to match some of the best known solutions for small instances of binary CAs.

**Key words:** Software testing, Covering Arrays, Branch & Bound.

## 1 Introduction

Within the recent years the use of software has become more and more important in our society. Most of the businesses nowadays depend on computer software, thus a failure on it can be catastrophic. As mentioned in a NIST (National Institute on Standards and Technology) report [1], the sales of software reached approximately $180 billion dollars generating a significant and high-paid workforce, composed of 697,000 software engineers and 585,000 computer programmers. It shows clearly that the software industry is a really important part of the economy, moreover if the software present errors, millionaire losses can occur affecting the economy. According to Hartman [2], the quality of the software strongly depends on the use of appropriate software testing techniques.

Software systems at the moment are numerous times more complex than before. Moreover, they usually have a lot of possible configurations that are produced by the combination of their input parameters. To totally guarantee the quality of these software products all the possible configurations must be tested, but this exhaustive approach is not a viable option. For instance, suppose

* This research was partially funded by the following projects: CONACyT 58554-Cálculo de Covering Arrays, 51623-Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas.

that we have a simple program that takes 12 arguments and each one of the arguments take only 4 possible values. In order to verify it totally we need to make $4^{12} = 16,777,216$ tests. But, previous works have demonstrated that testing it with an interaction of level 6 (all the combinations of 6 parameters) will only require $4^6 = 14,888$ tests [3] (each test is a combination of values taken by all the parameters). This approach based on constructing economical sized test suites that provide coverage of the most prevalent configurations is known as *software interaction testing*. Covering arrays (CAs) are combinatorial structures used to represent these test suites when exhaustive testing is not feasible [4].

A Covering Array $\mathrm{CA}(N; t, k, v)$ of size $N$ is an $N \times k$ array consisting of $N$ vectors of length $k$ (degree) with entries from an alphabet of size $v$, i.e., $\{0, 1, \ldots, v-1\}$, such that every one of the $v^t$ possible vectors of size $t$ (t-wise) occurs at least once in every possible selection of $t$ elements from the vectors. The parameter $t$ is referred to as the strength or level of interaction. The minimum $N$ for which a $\mathrm{CA}(N; t, k, v)$ exists is known as the *covering array number* and it is defined according to (1).

$$\mathrm{CAN}(t, k, v) = min\{N : \exists\, \mathrm{CA}(N; t, k, v)\} \tag{1}$$

A $\mathrm{CA}(N; t, k, v)$ can be mapped to a software test suite as follows. In a software test we have $k$ components, each of these has $v$ configurations. A test suite is an $N \times k$ array where each row is a test case. Each column represents a component and the value in the column is the particular configuration.

Lei and Tai [5] demonstrated that the problem of generating the minimum pairwise test set belongs to the NP class (for non-binary alphabets). Then by reduction of the problem of vertex cover, they proved that the *pair-cover problem* is NP-complete. Colbourn [6] also showed that this problem is NP-complete by reducing it to the SAT problem.

Even though the general problem of finding a combinatorial test suite is NP [7], there are some isolated cases that can be solved in polynomial time [6]:

1. When the strength is 2 ($t = 2$) and the alphabet is 2 ($v = 2$). Sloane stated that this case was solved by Rényi with an even value of $N$ and by Katona independently. Then, it was completely solved by Kleitman and Spencer for all $N$ [8].
2. When the alphabet is a power of prime $v = p^\alpha$, $k \leqslant (p^\alpha + 1)$, and $p^\alpha > t$. This construction was proposed by Bush [9], he used Finite Galois Fields in order to solve this case.

Given the complexity of the optimal construction of CAs many of the algorithms developed to solve this problem are approximate methods. They try to reach solutions as close as possible to $\mathrm{CAN}(t, k, v)$, given some values for $k, v$, and $t$ [10].

In this paper we introduce a new backtracking algorithm for constructing binary CAs of variable strength which is based on the Branch & Bound (B&B) technique. It incorporates some distinguished features for improving the efficiency of the search process, including: symmetry breaking techniques, partial

$t$-wise verification and fixed blocks. This backtracking algorithm guarantees to discover optimal CAs if they exist or prove their nonexistence, for small instances, given that no computer time restrictions are imposed.

The effectiveness of the proposed backtracking algorithm is assessed using a benchmark, conformed by 14 binary covering arrays of strength $3 \leq t \leq 5$, taken from the literature. The computational results are reported and compared against those reached by some state-of-the-art methods, showing that our algorithm is able to match some of the best-known solutions for small instances of binary CAs expending in some cases less computational time.

The remainder of this work is organized as follows. In Sect. 2, a brief review is given to present some representative solution procedures for constructing binary covering arrays of variable strength. Then, the components of our backtracking algorithm are detailed in Sect. 3 and 4. Section 5 presents computational experiments and comparisons of our backtracking algorithm with respect to other previously published algorithms. Last section is dedicated to summarize the main contributions of this work.

## 2 Relevant Related Work

The objective of constructing a CA is to minimize the number of rows given the parameters $v, k$, and $t$. There are several reported methods for constructing these combinatorial models. Among them are: a) recursive methods [8, 11], b) algebraic methods [12], c) greedy methods [13] and d) meta-heuristics such as Simulated Annealing [14] and Tabu Search [15]. These algorithms have a point in common, all of them are approximate methods.

To the best of our knowledge, there exists only one method reported in the literature which tries to make a systematic enumeration of the candidate solutions in the search space in order to guarantee to discover optimal CAs. This method, that we used in our experimental comparisons, is called EXACT (Exhaustive search of combinatorial test suites) and was proposed by Yan and Zhang [10].

EXACT is a backtracking algorithm that employs certain rules in order to eliminate isomorphic CAs. Two CAs are isomorphic if they have the same number of rows, columns and alphabet and one can be transformed into the other via permutations of rows, columns, and/or symbols [16]. Moreover, it introduces the concept of *miniblock*. A miniblock is a $mb \times t$ sub-array which values are fixed before starting the construction of a CA. It allows to reduce the size of the search space to the value given in (2). Please note that in this expression the total size of search space equals the numerator, and the denominator indicates the size of the fixed miniblock.

$$\frac{(\prod_{i=1}^{k} v_i^N)}{(\prod_{i=1}^{t} v_i^{mb})} \tag{2}$$

A novel pruning technique, called SCEH (Sub-Combination Equalization Heuristic), is also integrated to EXACT. The authors decided to use this tech-

nique because they noted that for many CAs each symbol appears almost the same number of times in each column of the CA.

In 2008, Yan and Zhang [17] reported an improvement to EXACT. In this new version of EXACT they added a new rule: for each two rows $i, j$ ($1 < i \leq mb$ and $j > mb$) of a CA, if these two rows have the same first $t$ values and $R_i >_{lex} R_j$ (where $>_{lex}$ refers to a lexicographical order), then these rows are exchanged. This work has improved the best-known solution for only one instance (CA$(24; 4, 12, 2)$).

## 3 A New Backtracking Algorithm

We can represent a CA 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 $M_{ij}$ to denote the cell at row $i$ ($i > 0$) and column $j$ ($j > 0$), i.e., the value of parameter $j$ in test case $i$.

We apply an exhaustive search technique to this problem. Our algorithm is based on the Branch & Bound technique. The main algorithm can be described as an iterative procedure as follows.

For a given matrix $N \times k$, and a strength $t$, we construct the first element $l$ belonging to the set of all possible columns with $\lfloor \frac{N}{2} \rfloor$ zeros and is inserted in the first column of the partial solution M. Then the next element $l_{i-1} + 1$ is constructed and if the row $i$ is smaller than the row $i + 1$ and it is a partial CA then the element is inserted, otherwise it tries with the next element. If no elements could be inserted in the current column of $M$, it backtracks to the last column inserted and tries to insert a new element. When $k$ columns are inserted then the procedure finishes and the CA is generated. A flow chart of this algorithm is shown in Figure 1.

## 4 Techniques for Improving the Efficiency of the Search

The worst time complexity of the naive exhaustive search for CA$(N, k, v, t)$ is shown in (3).

$$\binom{\binom{N}{\lfloor \frac{N}{2} \rfloor}}{k} \tag{3}$$

This worst time complexity is due mainly because there exist so many isomorphic CAs. There are 3 types of symmetries in a CA: row symmetry, column symmetry and symbol symmetry. The row symmetry refers to the possibility to alter the order of the rows without affecting the CA properties. There are $N!$ possible row permutations of a CA. The column symmetry refers to permuting columns in the CA without altering it. There exist $k!$ possible column permutations of a CA. In the same way thanks to the symbol symmetry if we make one of the $(v!)^k$ possible permutations of symbols it results in an isomorphic CA. By

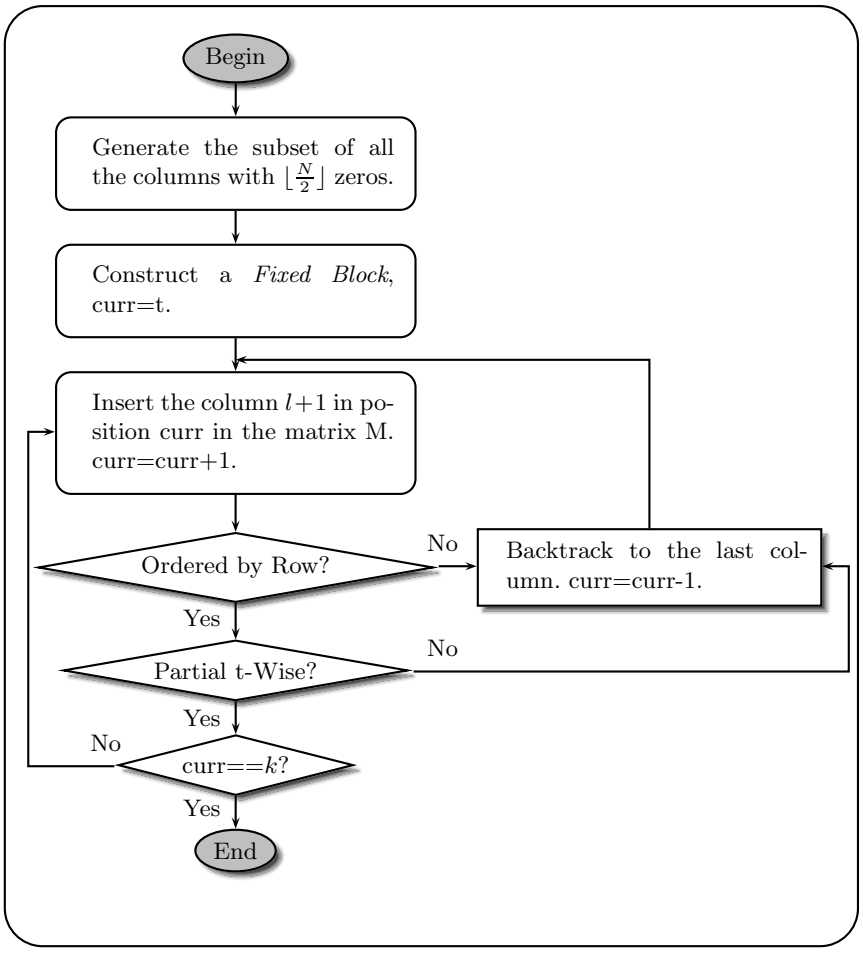


**Fig. 1.** A new backtracking algorithm.

the previous analysis we can conclude that there are a total of $N! \times k! \times (v!)^k$ number of symmetries in a CA. An example of two isomorphic CAs is shown in Table 1.

The covering array in Table 1(d) can be produced by the following steps over the covering array in Table 1(a): exchanging the symbols of the first column (Table 1(b)), then exchanging the first and second rows (Table 1(c)), finally exchanging the third and fourth columns.

Searching within the non-isomorphic CAs can significantly reduce the search space, these symmetry breaking techniques have been previously applied in order to eliminate the row and column symmetries in [17] by Yan and Zhang. However, they only proposed an approach to eliminate row and column symmetries. In the following sections we will describe the symmetry breaking techniques that we ap-

**Table 1.** Isomorphic Covering Arrays

(a)

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

(b)

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |

(c)

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |

(d)

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 |

plied within our backtracking algorithm to search only over the non-isomorphic CAs.

### 4.1 Symmetry Breaking Techniques

In order to eliminate the row and column symmetries in our new backtracking algorithm the restriction that within the current partial solution $M$ the column $j$ must be smaller than the column $j+1$, and the row $i$ must be smaller than the row $i+1$.

As we have mentioned above a CA$(N; t, k, v)$ has $N! \times k!$ row and column symmetries. This generates an exponential number of isomorphic CAs. Adding the constraints mentioned above we eliminate all those symmetries and reduce considerably the search space. Another advantage of this symmetry breaking technique is that we do not need to verify that the columns are ordered, we only need to verify that the rows are still ordered as we insert new columns. This is because we are generating an ordered set of columns such that the column $l$ is always smaller than the column $l+1$.

Moreover, we propose a new way of breaking the symbol symmetry in CAs. We have observed, from previously experimentation, that near-optimal CAs have columns where the number of 0's and 1's are balanced or near balanced. For this reason we impose the restriction that through the whole process the symbols in the CAs columns must be balanced. In the case where $N$ is not even, the number of 0's must be exactly $\lfloor \frac{N}{2} \rfloor$ and the number of 1's $\lfloor \frac{N}{2} \rfloor + 1$. As long as we know this is the first work in which the symbol symmetry breaking is used. In Table 2 an example of our construction is shown.

In Table 2 we can see clearly that the next column generated is automatically lexicographically greater than the previous one, so we do not need to verify for the column symmetry breaking rule. Even though that in Table 2 the rows are ordered as well, it does not happen in general so we still have to check that the rows remain ordered for each element that we try to insert in the partial solution $M$.

### 4.2 Partial t-Wise Verification

Since a CA of strength $t-1$ is present within a CA of strength $t$ we can bound the search space more quickly and efficiently if we partially verify for a CA

**Table 2.** Example of the current partial solution $M$ after 4 column insertions.

| $l$ | $l+1$ | $l+2$ | $l+3$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

instead of waiting until all $k$ elements are inserted. We can test the first $t-1$ columns with strength $i$, where $i$ is the current element inserted, and when $i$ is greater or equal to $t$ then is tested with strength $t$. It is important to remark that a complete CA verification is more expensive in terms of time,[1] than making the partial evaluation described above due to the intrinsic characteristics of the backtracking algorithm.

### 4.3 Fixed Block

The search space of this algorithm can be greatly reduced if we use a *Fixed Block* (FB). We define a FB as matrix of size $N$ and length $t$ in which the first $\lfloor \frac{(N-v^t)}{2} \rfloor$ rows are filled with 0's, then a CA of strength $t$, $k=t$ and $N=v^t$ is inserted. This CA can be easily generated (in polynomial time) by creating all the $v^t$ binary numbers and listing them in order. An example of a CA$(v^t;t,t,v)$ is shown in Table 3. Finally, the last $\lceil \frac{(N-v^t)}{2} \rceil$ rows are filled with 1's. It can be easily verified that in a FB the rows and columns are already lexicographically ordered. This FB is constructed in this way in order to preserve the symmetry breaking rules proposed in Sect. 4.1 and is used to initialize our algorithm as shown in Figure 1.

**Table 3.** A CA$(v^t;t,t,v)$ example.

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

[1] The computational complexity of making a full verification of a CA$(N;t,k,v)$ is $N \times \binom{k}{t}$

## 5 Computational Results

In this section, we present a set of experiments accomplished to evaluate the performance of the backtracking algorithm presented in Sect. 3. The algorithms were coded in C and compiled with *gcc* without optimization flags. It was run sequentially into a CPU Intel Core 2 Duo at 1.5 GHz, 2 GB of RAM with Linux operating system.

The test suite used for this experiment is composed of 14 well known binary covering arrays of strength $3 \leq t \leq 5$ taken from the literature [3, 17]. The main criterion used for the comparison is the same as the one commonly used in the literature: the *best size* $N$ found (smaller values are better) given fixed values for $k$, $t$, and $v$.

### 5.1 Comparison Between our Backtracking Algorithm and EXACT

The purpose of this experiment is to carry out a performance comparison of the upper bounds achieved by our backtracking algorithm (B&B) with respect to those produced by the EXACT procedure [17]. For this comparison we have obtained the EXACT algorithm from the authors. Both algorithms were run in the computational platform described in Sect. 5.

Table 4 displays the detailed computational results produced by this experiment. The first two columns in the table indicate the degree $k$, and strength $t$ of the instance. Columns 3 and 5 show the best solution $N$ found by B&B and the EXACT algorithms, while columns 4 and 6 depict the computational time $T$, in seconds, expended to find those solutions.

**Table 4.** Performance comparison between the algorithms B&B and EXACT.

| | | B&B | | EXACT | |
|---|---|---|---|---|---|
| $k$ | $t$ | $N$ | $T$ | $N$ | $T$ |
| 4 | 3 | 8 | **0.005** | 8 | 0.021 |
| 5 | 3 | 10 | **0.005** | 10 | 0.021 |
| 6 | 3 | 12 | **0.008** | 12 | 0.023 |
| 7 | 3 | 12 | **0.018** | 12 | 0.024 |
| 8 | 3 | 12 | 0.033 | 12 | 0.023 |
| 9 | 3 | 12 | 0.973 | 12 | 0.022 |
| 10 | 3 | 12 | 0.999 | 12 | 0.041 |
| 11 | 3 | 12 | 0.985 | 12 | 0.280 |
| 12$^\star$ | 3 | 15 | **1090.800** | 15 | 1100.400 |
| 5 | 4 | 16 | **0.020** | 16 | 0.038 |
| 6 | 4 | 21 | 95.920 | 21 | 0.266 |
| 6 | 5 | 32 | 102.000 | 32 | 0.025 |

From the data presented in Table 4 we can make the following main observations. First, the solution quality attained by the proposed backtracking algorithm

is very competitive with respect to that produced by the state-of-the-art procedure EXACT. In fact, it is able to consistently equal the best-known solutions attained by the EXACT method (see columns 3 and 5).

Second, regarding the computational effort, one observes that in this experiment the EXACT algorithm consumes slightly more computational time than B&B for 6 out 12 benchmark instances (shown in boldface in column 4).

We would like to point out that the instance marked with a star in Table 4 was particularly difficult to obtain using the EXACT algorithm. We have tried many different values for the parameter SCEH (Sub-Combination Equalization Heuristic), and only using a value of 1 the EXACT tool was able to find this instance consuming more CPU time than our B&B algorithm. For the rest of the experiments we have used the default parameter values recommended by the authors.

### 5.2 Comparison Between our Backtracking Algorithm and IPOG-F

In a second experiment we have carried out a performance comparison of the upper bounds achieved by our B&B algorithm with respect to those produced by the state-of-the-art procedure called IPOG-F [18].

Table 5 presents the computational results produced by this comparison. Columns 1 and 2 indicate the degree $k$, and strength $t$ of the instance. The best solution $N$ found by our B&B algorithm and the IPOG-F algorithm are depicted in columns 3 and 5, while columns 4 and 6 depict the computational time $T$, in seconds, expended to find those solutions. Finally, the difference ($\Delta_N$) between the best result produced by our B&B algorithm compared to that achieved by IPOG-F is shown in the last column.

From Table 5 we can clearly observe that in this experiment the IPOG-F procedure [18] consistently returns poorer quality solutions than our B&B algorithm. Indeed, IPOG-F produces covering arrays which are in average 31.26% worst than those constructed with B&B.

## 6 Conclusions

We proposed a new backtracking algorithm that implements some techniques to reduce efficiently the search space. This backtracking algorithm guarantees to discover optimal CAs if they exist or prove their nonexistence, for small instances, given that no computer time restrictions are imposed. Additionally, we have presented a new technique for breaking the symbol symmetry which allow to reduce considerably the size of the search space. Experimental comparisons were performed and show that our backtracking algorithm is able to match some of the best-known solutions for small instances of binary CAs, expending in some cases less computational time compared to another existent backtracking algorithm called EXACT. We have also carried out a comparison of the upper bounds achieved by our backtracking algorithm with respect to those produced by a state-of-the-art procedure called IPOG-F. In this comparison the results

**Table 5.** Performance comparison between the algorithms B&B and IPOG-F.

| | | B&B | | IPOG-F | | |
|---|---|---|---|---|---|---|
| $k$ | $t$ | $N$ | $T$ | $N$ | $T$ | $\Delta_N$ |
| 4 | 3 | 8 | **0.005** | 9 | 0.014 | -1 |
| 5 | 3 | 10 | **0.005** | 11 | 0.016 | -1 |
| 6 | 3 | 12 | **0.008** | 14 | 0.016 | -2 |
| 7 | 3 | 12 | **0.018** | 16 | 0.018 | -4 |
| 8 | 3 | 12 | 0.033 | 17 | 0.019 | -5 |
| 9 | 3 | 12 | 0.973 | 17 | 0.019 | -5 |
| 10 | 3 | 12 | 0.999 | 18 | 0.019 | -6 |
| 11 | 3 | 12 | 0.985 | 18 | 0.034 | -6 |
| 12 | 3 | 15 | 1090.800 | 19 | 0.033 | -4 |
| 13 | 3 | 16 | 1840.320 | 20 | 0.019 | -4 |
| 5 | 4 | 16 | 0.020 | 22 | 0.016 | -6 |
| 6 | 4 | 21 | 95.200 | 26 | 0.017 | -5 |
| 7 | 4 | 24 | 113.400 | 32 | 0.014 | -8 |
| 6 | 5 | 32 | 102.000 | 42 | 0.020 | -10 |
| **Average** | | 15.29 | | 20.07 | | -4.79 |

obtained by our backtracking algorithm, in terms of solution quality, are better than those achieved by IPOG-F for all the studied instances.

Finding optimum solutions for the CA construction problem in order to construct economical sized test-suites for software interaction testing is a very challenging problem. We hope that the work reported in this paper could shed useful light on some important aspects that must be considered when solving this interesting problem. We also expect the results shown in this work incite more research on this topic. For instance, one fruitful possibility for future research is the design of new pruning heuristics in order to have the possibility to generate larger instances of CAs.

**Acknowledgments.** The authors would like to thank Jun Yan and Jian Zhang who have kindly provided us with an executable version of their application EXACT.

## References

1. Tassey, G., RTI: The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD, USA (May 2002)
2. Hartman, A.: 10. In: Graph Theory, Combinatorics and Algorithms. Volume 34 of Operations Research/Computer Science Interfaces Series. Springer (2005) 237–266
3. Colbourn, C.J.: Most recent covering arrays tables. Web Page (Last Time Accesed 4-Nov-2008) http://www.public.asu.edu/ ccolbou/src/tabby/catable.html.

4. Kuhn, D.R., Wallance, D.R., Gallo, A.M.J.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering **30** (2004) 418–421
5. Lei, Y., Tai, K.: In-parameter-order: A test generation strategy for pairwise testing. In: In Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering, Washington, DC, USA, IEEE Computer Society (1998) 254–261
6. Colbourn, C.J., Cohen, M.B., Turban, R.C.: A deterministic density algorithm for pairwise interaction coverage. Proceedings of the IASTED International Conference on Software Engineering (2004)
7. Kuhn, D.R., Okum, V.: Pseudo-exhaustive testing for software. In: SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, Washington, DC, USA, IEEE Computer Society (2006) 153–158
8. Sloane, N.J.A.: Covering arrays and intersecting codes. Journal of Combinatorial Designs **1** (1993) 51–63
9. Bush, K.A.: Orthogonal arrays of index unity. Annals of Mathematical Statistics **13** (1952) 426–434
10. Yan, J., Zhang, J.: Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. Computer Software and Applications Conference(COMPSAC 06) 30th Annual International **1** (2006) 385–394
11. Hartman, A., Raskin, L.: Problems and algorithms for covering arrays. Discrete Mathematics **284** (2004) 149–156
12. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: Orthogonal Arrays: Theory and Applications. Springer-Verlag, New York, US (1999)
13. Cohen, D.M., Fredman, M.L., Patton, G.C.: The aetg system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering **23** (1997) 437–444
14. Cohen, M.B., Colbourn, C.J., H., L.A.C.: Constructing strength three covering arrays with augmented annealing. Discrete Mathematics **308** (2008) 2709–2722
15. Nurmela, K.J.: Upper bounds for covering arrays by tabu search. Discrete Appl. Math. **138**(1-2) (2004) 143–152
16. Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M.: Constraint models for the covering test problem. Constraints **11**(2-3) (2006) 199–219
17. Yan, J., Zhang, J.: A backtracking search tool for constructing combinatorial test suites. The journal of systems and software **81**(10) (2008) 1681–1693
18. Forbes, M., Lawrence, J., Lei, Y., Kacker, R.N., Kuhn, D.R.: Refining the in-parameter-order strategy for constructing covering arrays. Journal of Research of the National Institute of Standards and Technology **113**(5) (2008) 287–297