ISSN 0361-7688, Programming and Computer Software, 2011, Vol. 37, No. 3, pp. 121–146. © Pleiades Publishing, Ltd., 2011.
Original Russian Text © V.V. Kuliamin, A.A. Petukhov, 2011, published in Programmirovanie, 2011, Vol. 37, No. 3.

A Survey of Methods for Constructing Covering Arrays

V. V. Kuliamin and A. A. Petukhov

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: kuliamin@ispras.ru

Received October 3, 2010

Abstract—The paper presents a survey of methods for constructing covering arrays used in generation of tests for interfaces with a great number of parameters. The application domain of these methods and algorithms used in them are analyzed. Specific characteristics of the methods, including time complexity and estimates of the required memory, are presented. Various—direct, recursive, optimization, genetic, and backtracking—algorithms used for constructing covering arrays are presented. Heuristics are presented that allow one to reduce arrays without loss of completeness, and application domains of these heuristics are outlined.

DOI: 10.1134/S0361768811030029

1. INTRODUCTION

High complexity of modern software systems and importance of tasks solved by them make the problem of verifying correctness of operation of such systems (i.e., checking whether these systems meet the requirements imposed on them in all possible situations) very topical. Such verification relies most often on testing, i.e., analysis of behavior of the system under test on a finite set of specially created test situations.

In order to ensure quality and completeness of the verification, the number of tests should be as large as possible. However, complete testing is impossible, since the number of possible situations in which behavior of real systems is to be checked is practically infinite. Therefore, in practice, one tries to get knowledge of system behavior by observing its behavior in a relatively small number of test situations. Usually, test situations in this case are divided into equivalence classes such that the behavior of the system under test differs insignificantly in situations from one class and change considerably when turning from one class to another. Then, testing completeness is determined as a degree of coverage of special classes of situations by tests. In the course of test execution, system behavior is analyzed, and its conformance to the requirements is checked.

Since testing must be bounded in time, the number of classes is always finite. However, it may be quite large in order to ensure effective testing in a given project. Possible situations and behavior scenarios of the system under test are often classified in terms of some set of factors, characteristics, or parameters, each of which may take a finite number of values. In this case, different classes of situations correspond to different possible combinations of values of the specified factors.

Examples of factors used in the classification of the situations are given below.

• success or failure of execution of a certain operation;

• branching in the code of the tested component, with the value of the corresponding factor being execution of the if or else branches;

• a grammar rule alternative (selection from several possible variants) in grammar-based testing [1, 2]; the values in this case are possible ways to resolve the alternative;

• a category in category-partition testing [3].

In many cases, high-quality testing requires checking relationships between the factors and joint effect of these factors on the system behavior. To perform such checking, it is required to combine factor values in a way that, on the one hand, allows one to carry out accurate testing and, on the other hand, does not increase too much the test suite, since the complete search of all combinations of factors in real systems requires too much expenditures.

In this paper, we consider methods for the creation of test suites based on combinations of factor values, which include all possible combinations of pairs, triples, or greater sets of values of various factors. Such methods were successfully used in testing various systems (see works [2, 4–8]). They allow one to create small, compared to the all-combinations based search, test suites, which, at the same time, permit quality system testing if the following conditions are satisfied:

• There are situations or actions on the system that are described by many parameters or factors.

• Values of each parameter can be divided into several classes such that all significant changes in the system behavior occur when one of the parameters goes from one class to another.

Table 1. Parameter values for testing money transfer from one purse to another by means of the WM Keeper Light interface

Sum transferred	Is currency conversion required?	Type of the purse from which money are transferred	Browser	Authentication method	Operating system
<100 rub.	Not required	WMR rubles	Internet Explorer	Certificate X.509	Windows XP
100–10000 rub.	Required	WMZ US dollars	Mozilla Firefox	Enum-authorization	Windows Vista
>10000 rub.		WME euros	Opera	Login and password	Debian Ubuntu
		WMU grivnas	Google Chrome		Linux SUSE
					Linux RedHat

• It is known that errors in the system behavior occur basically due to combination of a few factors determined by values of the parameters used.

If no additional information on relationship between possible errors and possible parameter values is available, these methods allow one to efficiently construct test suites capable of testing a great variety of situations. Otherwise, this information can be used for constructing more compact targeted tests.

As an example, let us consider testing of the user's interface of the money transfer system WebMoney. This interface can be accessed with the help of any modern browser and various operating systems. The most frequently used operation in this system is money transfer from one purse to another by means of the WM Keeper Light interface. This operation is performed when paying mobile communication or Internet, buying goods in an Internet store, and so on. This operation may be affected by several factors. The most significant among them are the sum of money transferred, necessity of currency conversion, type of the purse from which money are transferred, method of user authentication in the WebMoney system, and the browser and operating system on the user side.

Possible values of these parameters chosen based on the documentation on the WM Keeper Light interface [9] are presented in Table 1.

Now, if we want to check all possible combinations of factor values, we need $3 \cdot 2 \cdot 4 \cdot 4 \cdot 3 \cdot 5 = 1440$ tests. This is not too much, of course; however, manual performing of all of them will require great expenditures.

Results of empirical studies [7, 8, 10] show that the majority of errors (up to 70%) in similar cases are related to certain combinations of values of only two parameters. In [11], it is shown that combinations of pairs of factors in testing yield code coverage up to 80% under more or less reasonable selection of values of individual parameters. In other words, if tests will contain all possible combinations of pairs of parameter values, the major part of errors will be revealed by these tests. In so doing, all pairs of values of various factors can be used in a small test suite. The minimal number of such tests for the given example is equal to 20, since there are 20 possible combinations of purse types and operating systems. A suite consisting of this number of tests can actually be constructed by using techniques described in this work (an example is presented in Table 2).

One can try to compose a test suite containing all possible triples of parameter values, which is also not too large. The above example will require not less than $80 = 5 \cdot 4 \cdot 4$ tests, which is far less than 1440. This suite can also be constructed by using methods described in the paper. Such testing will allow one to detect more errors than the previous one [7, 10].

This work surveys various methods of test construction that use pairs, triples, and greater sets of parameter values. Section 2 contains a definition of covering sets and basic results related to them. The survey itself is presented in Section 3. The concluding section summarizes the discussion.

2. THEORETICAL BACKGROUND

A test suite covering all pairs, triples, or greater sets of possible values of system parameters is associated with the mathematical concept of a covering array.

Suppose that there are k factors affecting system operation, with the first factor taking n_1 different values, the second factor taking n_2 values, and so on. A *covering array of strength t* is a matrix consisting of k columns, whose ith column contains values of the ith factor and any combination of possible values of any t factors is contained in at least one of its rows.

Since factor values themselves are not important, we denote them by numbers from 0 through $n_i - 1$. The set of numbers $(t; k, n_1, \ldots, n_k)$ is called a *covering array configuration*, and the set of covering arrays with this configuration is denoted as $\mathrm{CA}(t; k, n_1, \ldots, n_k)$. The formal definition is as follows: an integer matrix $N \times k$ further denoted as A is a covering array from $\mathrm{CA}(t; n_1, \ldots, n_k)$ if and only if, $\forall i \in [1..N]$, and $j \in [1..k]$ $A_{ij} \in [0..n_j - 1]$ & $\forall j_1, \ldots, j_t \in [1..k]$ $\forall v_1 \in [0..n_{j1} - 1], v_t \in [0..n_{jt} - 1]$ $\exists i \in [1..N]$ $\forall q \in [1..t]$ $A_{ij_q} = v_q$.

Number t in configuration $(t; k, n_1, \ldots, n_k)$ is called the *array strength*, k is the *number of parameters* or factors, and n_i is the *number of values* of the ith parameter. All numbers t, k, n_1, ... n_k are called *characteristics* of the configuration. A covering array of strength t is also

Table 2. Minimal test suite for the WM Keeper Light interface

<100 rub.	Not required	WMR rubles	Internet Explorer	Certificate X.509	Windows XP
100–10000 rub.	Required	WMZ US dollars	Mozilla Firefox	Enum-authorization	Windows XP
>10000 rub.	Not required	WME euros	Opera	Login and password	Windows XP
100–10000 rub.	Required	WMU grivnas	Google Chrome	Certificate X.509	Windows XP
100–10000 rub.	Not required	WMR rubles	Mozilla Firefox	Login and password	Windows Vista
>10000 rub.	Not required	WMZ US dollars	Opera	Certificate X.509	Windows Vista
<100 rub.	Required	WME euros	Google Chrome	Enum-authorization	Windows Vista
>10000 rub.	Not required	WMU grivnas	Internet Explorer	Login and password	Windows Vista
100–10000 rub.	Required	WMR rubles	Opera	Enum-authorization	Debian Ubuntu
>10000 rub.	Not required	WMZ US dollars	Google Chrome	Login and password	Debian Ubuntu
100–10000 rub.	Not required	WME euros	Internet Explorer	Enum-authorization	Debian Ubuntu
<100 rub.	Required	WMU grivnas	Mozilla Firefox	Certificate X.509	Debian Ubuntu
>10000 rub.	Not required	WMR rubles	Google Chrome	Enum-authorization	Linux SUSE
<100 rub.	Required	WMZ US dollars	Internet Explorer	Login and password	Linux SUSE
>10000 rub.	Required	WME euros	Mozilla Firefox	Certificate X.509	Linux SUSE
<100 rub.	Not required	WMU grivnas	Opera	Enum-authorization	Linux SUSE
100–10000 rub.	Required	WMR rubles	Google Chrome	Certificate X.509	Linux RedHat
>10000 rub.	Not required	WMZ US dollars	Internet Explorer	Login and password	Linux RedHat
>10000 rub.	Required	WME euros	Mozilla Firefox	Certificate X.509	Linux RedHat
<100 rub.	Not required	WMU grivnas	Opera	Enum-authorization	Linux RedHat

called sometimes a t-covering array. The number of rows in a covering array is called the array *size*.

A covering array is said to be *minimal* if there does not exist a covering array for the same configuration that contains lesser number of rows. The size of the minimal covering array for configuration $(t; k, n_1, \ldots, n_k)$ is denoted as $\mathrm{CAN}(t; k, n_1, \ldots, n_k)$.

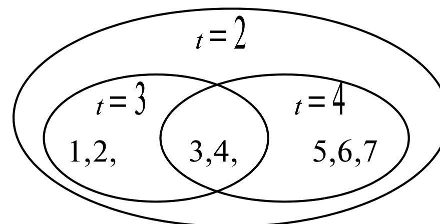
If the number of values of all factors coincide, i.e., $n_1 = n_2 = \ldots = n_k = n$, then the corresponding covering array is said to be *uniform*. Its configuration is denoted as $(t; k, n)$; the set of such arrays is denoted as $\mathrm{CA}(t; k, n)$, and the minimal size of such an array, as $\mathrm{CAN}(t; k, n)$.

If the numbers of values of several factors coincide, i.e., $n_1 = n_2 = \ldots = n_{p1}$, $n_{p1+1} = n_{p1+2} = \ldots = n_{p2}$, and so on, then the configuration of the covering array is written by using the exponential notation $\mathrm{CA}(t;\ k,\ n_{p1}^{p1},\ n_{p2}^{p2} \ldots)$, where $p_1 + p_2 + \ldots = k$.

There exist more complicated covering arrays, namely, *variable strength covering arrays*. These arrays contain all combinations of pairs of factor values $i_1, \ldots, i_{p2}$ $(p_2 \geq 2, p_2 \leq k)$; all combinations of triples of factor values $j_1, \ldots, j_{p3}$ $(p_3 \geq 3, p_3 \leq k)$, ...; and all combinations of t factor values $l_1, \ldots, l_{pt}$ $(p_t \geq t, p_t \leq k, t \leq k)$, where i_p, j_p, ..., l_p are numbers of factors from 1 through k (numbers denoted by different letters may coincide). For example, if there are seven factors, we may be interested in combinations of all pairs of values of seven factors; triples of values of the first, second, third, and fourth factors; fours of values of the third, fourth, fifth, sixth, and seventh factors (see figure).

The formal definition of a variable strength array is cumbersome and is not presented. Configurations of such arrays are denoted as $(t_1, n_1, \ldots, n_{p_1}; t_2, j_1, \ldots, j_{p_2}; \ldots; t_m, l_1, \ldots, l_{p_m})$, or $(t_1; k, n; t_2, j_1 \ldots j_{p_2}; \ldots; t_m, l_1, \ldots, l_{p_m})$ in the uniform case. The figure shows a part of configuration (2; 7, 10; 3, 1, 2, 3, 4; 4, 3, 4, 5, 6, 7).

When describing tests with the help of covering arrays, each row of the covering array corresponds to one test, and the number in the jth column corresponds to the number of class of values of the jth parameter. This correspondence makes it possible to use mathematical theory of covering arrays for constructing test suites. Since it is important not to have too many classes, one should use minimal, or almost minimal, covering arrays.



Graphical representation of a variable strength configuration.

Generally, no effective algorithms exist for constructing minimal covering arrays. The problem of construction of a minimal covering array from CA(t; k, n) is NP-complete, which was proved by reducing it to that of coloring a graph using three colors [12]. The particular problem of finding a minimal array from CA(2; k, n) is also NP-complete, which was proved by Lei and Tai [13] by reducing the problem to that of covering graph nodes.

Size of the minimal covering arrays is known to satisfy the following estimate [14]: CAN(t; k, n) $\leq (t - 1)\log(k)/\log(n^t/(n^t - 1))(1 + o(1))$ as $k \longrightarrow \infty$, which yields CAN(t; k, n) $\leq (t - 1)n^t\log(k)(1 + o(1))$ for $n^t \longrightarrow \infty$. There is also an evident lower bound CAN(t; k, n) $\geq n^t$ based on that the array should contain all possible combinations of t values each of which can be selected in n ways. As can be seen, as the number k of the parameters used grows, the size grows only at logarithmic rate, which explains practical usefulness of the covering arrays for constructing small quality test suites.

Estimates of the covering array size for particular cases, as well as tables that explicitly show sizes of suites for certain configurations, can be found in [15–20]. In this work, these estimates are not presented in order that not to overload the discussion.

Studies of the covering arrays are focused mainly on search of the minimal test suites of various configurations and on the development of efficient (polynomial) algorithms for constructing suites that are close to minimal ones. There are many algorithms of this kind; however, they either work with only special configurations or, for the majority of configurations, produce suites that are much greater than the minimal ones. The goal of this paper is to analyze the existing methods for constructing covering arrays and reveal domains of their possible application, where they construct minimal, or almost minimal, test suites.

3. SURVEY OF ALGORITHMS FOR CONSTRUCTING COVERING ARRAYS

In this survey, we tried to discuss all algorithms for constructing covering arrays presented in works [15–19], as well as a number of additional ones, such as the algorithm based on combining blocks of uniform covering arrays and auxiliary suites [5], "double projection" technique [21], generalization of the IPO algorithm to the case of strength $t > 2$ [22], heuristics for optimization algorithms [8, 23–26], backtracking algorithms [27], algorithm for optimizing a given covering array [28]. In addition, we present estimates of memory required for these algorithms, which are lacking in the above-listed surveys.

The existing algorithms for constructing covering arrays can be classified as follows:

• Combinatorial (direct) algorithms use correspondence between the covering arrays and other combinatorial schemes (sets of Latin squares, group orbits, and the like) in order to construct a covering array when the corresponding scheme is sufficiently simple.

• Recursive algorithms construct covering arrays from the covering arrays for configurations with smaller values of characteristics (for example, with lesser number of factors or strength). Other combinatorial schemes (orthogonal arrays, difference matrices, etc.) can also be used.

• Reduction algorithms construct covering arrays by modifying and reducing covering arrays for configurations with greater values of characteristics.

• Optimization algorithms consider construction of a covering array as a problem of minimization of the number of rows in the corresponding matrix and use methods for finding extrema of this function (greedy matrix construction, simulated annealing, genetic algorithms, etc.).

• Backtracking algorithms successively search all possible values in the cells of the suite for the given configuration and given size of the suite. If the current array is not covering, the algorithm returns one or several steps back and makes another attempt with different values. Such algorithms use certain rules for reducing the number of variants searched.

In what follows, when considering algorithms, we will analyze the following characteristics.

• Class of array configurations that can be obtained by means of the considered algorithm. Situations where the resulting suite turns out minimal are noted.

• Class of the algorithm. The majority of the algorithms discussed fall in one of the above-listed classes. However, some of them can be used both for direct construction of the desired array and for recursive completion of the smaller one.

• Time complexity. In the case of recursive and reduction algorithms, complexity of the algorithms themselves will be estimated without taking into account the way the initial arrays were constructed.

• Amount of the required memory.

• Possibility of constructing some parts of the array independent from one another. Such a possibility is useful for more efficient use of memory, since, in this case, it is not required to store the entire array obtained. Estimates of the required amount of memory in such a saving mode are presented.

• Possibility of taking into account constraints on combinations of parameter values used in the array. Such a possibility is important if not all possible combinations of parameter values of the interface under test have sense or feasible.

In this survey, we do not prove theorems that state that the arrays constructed in one way or another are covering or minimal ones, since the proofs were given in [15–20]. Moreover, we do not discuss algorithms for constructing covering arrays for finite classes of configurations, which can also be found in the above-listed works.

If an algorithm does not rely on random choice of some elements or numbers, the resulting array is determined unambiguously. However, any permutations of columns or rows or permutations of values of any parameter make it possible to obtain covering arrays that are equivalent to the given one.

First, we consider algorithms for constructing uniform arrays and, then, those for constructing nonuniform or variable strength arrays.

3.1. Algorithms for Constructing Uniform Covering Arrays

Direct and recursive methods for constructing uniform covering arrays with equal numbers of values of all parameters are most well studied [16, 18].

3.1.1. "Boolean" algorithm for constructing covering arrays [19, 29, 30]. This combinatorial-type algorithm constructs covering arrays from CA(2; k, 2) for any values of $k \geq 1$.

Brief description of the algorithm.

1. Select the least N satisfying the condition $k \leq C_{N-1}^{\lceil N/2 \rceil}$. Here, $\lceil x \rceil$ is the least integer greater than or equal to x, and C_q^r is the binomial coefficient. This N is equal to the size of the resulting array.

2. Set all elements of the first row of the array equal to zero.

3. The remaining $N - 1$ rows are constructed by columns, which are taken to be all possible sequences of $\lceil N/2 \rceil$ ones and $\lceil N/2 \rceil - 1$ zeros.

It is easy to verify that the array constructed in this way is actually a covering one. The proof of its minimality is given in [19].

The algorithm requires $O(k\log_2 k)$ memory and has time complexity $O(k)$.

3.1.2. "Affine" algorithm for constructing covering arrays [5, 19]. This combinatorial-type algorithm constructs covering arrays from CA(t; n + 1, n) for n equal to a power of a prime number, $n = p^k$, $k \geq 1$, and any values of $t \leq n + 1$. For $t = 3$ and $n = 2k$, an array from CA(t; n + 2, n) is obtained.

Brief description of the algorithm.

1. For each power of a prime number, $n = p^k$, there exists a finite field with this number of elements, which is called Galois field GF(p^k) [31]. A table of elements of field GF(p^k) is constructed as follows.

2. Each row of the table is coded by means of a sequence $a_0a_1...a_{t-1}$, where a_i takes all possible values from field GF(p^k). The total number of rows is n^t.

3. The first column in the row with the code $a_0a_1...a_{t-1}$ contains the value equal to a_0. To the first column, number ∞ is assigned.

4. The second column in the row with the code $a_0a_1...a_{t-1}$ contains the value equal to a_{t-1}. To the second column, number 0 is assigned.

5. The remaining columns, from the third through (n + 1)th with numbers $m = 1...(n - 1)$, are constructed such that the intersection of the row with the code $a_0a_1...a_{t-1}$ and column with number m contains the value calculated by the formula $\Sigma_{i=0}^{t-1} a_i m^i$ in the arithmetic of GF(p^k).

6. For strength $t = 3$ and $p = 2$, one more column consisting of values a_1 is added to the table.

The table constructed in this way is a minimal covering array from CA(t; $p^k + 1$, p^k) or CA(3; $2^k + 2$, 2^k), CAN(t; $p^k + 1$, p^k) = p^{tk}, CAN(3; $2^k + 2$, 2^k) = 2^{3k} (the proof is given in [19]).

Time complexity of the algorithm is estimated as $O(n^t)$, and the required memory, as $O(n)$, since the array can be constructed row by row. However, the implementation will require modeling of arithmetic of polynomials in the Galois field, i.e., creation of the multiplication and addition tables. If irreducible polynomials of the corresponding degree are known, then creation of the tables requires $O(\log_p^4 n)$ operations and $O(\log_p^3 n)$ memory.

3.1.3. Algorithm based on group actions [17, 32–35]. This combinatorial-type algorithm constructs covering arrays for several different configurations described below.

Brief description of the algorithm.

1. An initial array is selected with the number of columns equal to the number of parameters in the configuration. Let the number of columns in it be equal to N_{start}.

2. A group of permutations of a set of possible parameter values is selected, which will act on the initial array as well. Let the order of the group be n.

3. Action of each of n elements of the group on the initial array results in n arrays equal in size to the initial one. One of these arrays—the result of action of the group unity—is the initial array.

4. The arrays obtained are supplemented to one another from the bottom.

5. In a number of cases, an additional array with the number of columns equal to the number of parameters in the configuration is added in the very end. Let the number of rows in it be N_{end}.

6. As a result, the desired array with the number of rows equal to ($nN_{start} + N_{end}$) is obtained.

Selection of the initial array, group, and additional array.

Configurations (3; 2k, q + 1) and their particular case (3; 2k, k) [34], where $k > 2$, $q \geq k - 1$, and q is a power of a prime number. The size of the desired array is $(2k - 1)(q^3 - q) + q + 1$. The algorithm is efficient if $2k$ is a small number, since the array size in this case tends to $O(q^3)$ and the minimal possible array has size $(q + 1)^3$. However, if $2k$ is close to or greater than q, then the array size is $O(q^4)$, which is greater than that of the minimal array by an order of magnitude.

1. For a complete graph with $2k$ vertices, there exists 1-factorization into $2k - 1$ graphs [36].

2. In each factorization, $2k$ vertices are numbered from 0 through $2k - 1$, and k edges are numbered from 0 through $k - 1$.

3. Vertices of the complete graph are made to correspond to columns of the initial array, and graphs in the factorization, to rows. Thus, the initial array will consist of $2k$ columns and $2k - 1$ rows.

4. Turn successively from one factorization graph to another and form rows of the initial array as follows: to the column with the number of vertex i, the number of the edge that is incidental to this vertex in the graph is placed.

5. The initial array A can also be specified as follows:

$$A_{i,j} = 0, \quad 0 \le i \le 2k - 2, \quad j = 0;$$
$$A_{i,j} = |i - j + 1|, \quad |i - j + 1| < k, \quad j \ne 0;$$
$$A_{i,j} = 2k - 1 - |i - j + 1|,$$
$$k \le |i - j + 1| < 2k - 1, \quad j \ne 0.$$

6. Consider the projective group $\mathrm{PGL}(q) = \mathrm{PGL}(2, \mathrm{GF}(q)$ over the finite field $\mathrm{GF}(q)$. This group acts on elements of filed $\mathrm{GF}(q)$ by permutations. Its order is equal to $q^3 - q$. Permutations corresponding to the elements of this group can be calculated by means of $O(q^3)$ operations.

7. The additional array is taken to be array C, where $C_{ij} = i$, $0 \le i \le 2k - 1$, $0 \le j \le q$.

The table obtained in this way is a covering array, which is proved in [34]. However, this array is usually not minimal.

Time complexity of the algorithm, together with finding elements of group $\mathrm{PGL}(q)$ – $O(2k((2k - 1)(q^3 - q) + q + 1) + q^3) = O(k^2q^3)$ and the required memory $O(2k(2k - 1) + q^4 - q^2) = O(k^2 + q^4)$, since the array can be constructed row by row, but the initial array and elements of group $\mathrm{PGL}(q)$ are to be permanently kept in the memory.

Configuration (2; 4, q), where $q \equiv 2 (\mathrm{mod}\ 4)$. The size of the constructed array is $(q + 1)q$ [35].

1. For the initial array, the transposed difference covering array $\mathrm{DCA}(4; q + 1; q)$ is selected, which is defined as follows. Let $(G, *)$ be a group of order q. Then, $\mathrm{DCA}(k; n; q)$ is a matrix A with elements a_{ij}, $0 \le i \le k - 1$, $0 \le j \le n - 1$ belonging to G such that, for any two different rows t and h, $0 \le t < h \le k - 1$, each element of G can be found in the difference vector $\Delta_{th} = \{d_{hj} * d_{tj}^{-1}, 0 \le j \le n - 1\}$ at least once. Methods of finding $\mathrm{DCA}(4; q + 1; q)$, where $q \equiv 2 (\mathrm{mod}\ 4)$, are described in [35].

2. The group is selected to be the ring of residuals Z_q if $q \equiv 6 (\mathrm{mod}\ 12)$ or $Z_2 + \mathrm{GF}(q_1) + \mathrm{GF}(q_2) + \ldots + \mathrm{GF}(q_t)$ if $q \equiv 2$ or $10 (\mathrm{mod}\ 12)$, where $q = 2u$, $u \ge 7$, and $u = q_1q_2\ldots q_t$ is factorization into powers of prime numbers.

3. The additional array is lacking.

The table obtained in this way is a covering array, which is proved in [35]. However, this array is not always minimal.

Time complexity of the algorithm is $O(4(q + 1)q) = O(q^2)$, and the required memory is $O(4(q + 1) + q^2) = O(q^2)$, since the array can be constructed row by row, but the initial array and elements of the group are to be permanently kept in the memory.

Configurations (2; l, g) and (2; l + 1, g) of size $l(g - 1) + 1$ and $(l + 1)(g - 1) + 1$, respectively, where the finite number of pairs (g, l) for configurations of each kind can be found in works [17, 34]. In addition to pair (g, l), vector $(v_0, \ldots, v_{l-1})$, $v_i \in Z_{g-1} \cup \{\infty\}$, $v_0 = \infty$, is required, which is a cover starter for the first configuration and a distinct cover starter for the second configuration.

Let set $D_s = \{(v_j - v_i) (\mathrm{mod}\ (g - 1)) : j - i = s (\mathrm{mod}\ l), v_i \ne \infty, v_j \ne \infty\}$ be given. When $D_s = Z_{g-1}$ for all $1 \le s < l$, then vector $(v_0, \ldots, v_{l-1})$ is called a *(g, l) cover starter.* When $Z_{g-1} \backslash \{0\} \subseteq D_s$ for all $1 \le s < l$ and $\{v_1, \ldots, v_{l-1}\} = Z_{g-1}$, then vector $(v_0, \ldots, v_{l-1})$ is called a *distinct (g, l) cover starter* [17]. These vectors are found by means of complete search [33] or heuristic search [34].

1. The matrix of cyclic shifts of the initial vector is composed. The vector is cyclically shifted once. As a result, a square l-by-l diagonal matrix is obtained. This matrix is taken to be the initial one for configuration $(2; l, g)$. The same matrix augmented by a column of zero elements is taken to the initial matrix for configuration $(2; l + 1, g)$.

2. The group is taken to be the ring of residuals Z_{g-1}. In so doing, the group does not act on symbols ∞.

3. The additional array is the row consisting of symbols ∞.

4. All symbols ∞ are replaced by number g.

The table obtained in this way is a covering array but not a minimal one [17].

If the cover starter is known, time complexity of the algorithm is $O(l(l(g - 1) + 1)) = O(l^2g)$, and the required memory is $O(l + (g - 1)g) = O(l + g^2)$, since the array can be constructed row by row, but the cover starter and elements of the group are to be permanently kept in the memory.

3.1.4. Algorithm reducing array strength [34]. This reduction algorithm constructs an array A belonging to $\mathrm{CA}(t - 1; k - 1, n)$ from an array B belonging to $\mathrm{CA}(t; k, n)$ of size N. The size of array A is equal to N/n. Using results of the previous section, one can construct an array belonging to $\mathrm{CA}(2; 2k - 1, q + 1)$ by means of the initial $\mathrm{CA}(3; 2k, q + 1)$, where $k > 2$, $q \ge k - 1$, and q is a power of a prime number. The size of the resulting array is equal to $((2k - 1)(q^3 - q) + q + 1)/(q + 1) = (2k - 1)q(q - 1) + 1$.

Brief description of the algorithm.

1. Select the jth column of the initial array B.

2. Select one admissible value of parameters.

3. Retain only those rows in the initial array that contain the selected value in the jth column.

4. Remove the jth column.

The table obtained is a covering array, which is proved in [34]. However, this array may not be minimal even if the initial array is minimal. Time complexity of the algorithm is $O(Nk)$, and the required memory has the same estimate, since the array can be constructed row by row, but the initial array is to be permanently kept in the memory.

3.1.5. Multiplicative algorithm [17, 19]. This recursive algorithm constructs a covering array belonging to CA(t; k, n_1n_2) from a covering array belonging to CA(t; k, n_1) and to CA(t; k, n_2). The size of the resulting array is equal to the product of sizes of the initial arrays. If the number of parameters k is great, this algorithm constructs arrays that are far from minimal ones.

Brief description of the algorithm.

1. Suppose that there are two covering arrays: A from CA(t; k, n_1) and B from CA(t; k, n_2).

2. Let us denote elements of these two arrays as a_{ij} and b_{lj}. These arrays have the same number of columns and, possibly, different numbers of rows.

3. Any number from 0 through $(n_1n_2 - 1)$ can unambiguously be represented as $n_2i + m$, where i lies between 0 and $(n_1 - 1)$, and m, between 0 and $(n_2 - 1)$. Thus, there is one-to-one correspondence between indices of rows of the table of the array obtained and pairs of indices of rows of two initial tables (i, m), where i is the index of rows of the first array, and m is the index of rows of the second array.

4. Elements of this table can be constructed by the formula $x_{(i, m)j} = n_2a_{ij} + b_{mj}$.

The table obtained in this way is a covering array with k parameters and n_1n_2 values for strength t; the proof is given in [19]. However, the array constructed in this way may not be minimal even if the initial arrays are minimal. The counterexamples are as follows (see [37]): there exists a covering array for configuration (2; 6, 10) with the number of rows equal to 102, whereas the method described constructs a covering array for configuration (2; 6, 10) with 150 rows using arrays from CA(2; 6, 2) (CAN(2; 6, 2) = 6) and CA(2; 6, 5) (CAN(2; 6, 5) = 25) as the initial minimal arrays.

Time complexity of the algorithm is $O(k(1 + N_1 + N_2))$, where N_1 is the number of rows in the first initial array and N_2 is the number of rows in the second array, and the required memory has the same estimate, since the array can be constructed row by row, but the initial arrays are to be permanently kept in the memory.

3.1.6. Construction of uniform covering arrays of strength two with the use of recursive constructs [17, 19, 33]. Let us introduce several auxiliary definitions [33].

A *partitioned* covering array PCA(N; 2; (k_1, k_2), n) is a covering array of size N from CA(2; $k_1 + k_2$, n) that admits the following partitioning (with regard to the fact that the rows and columns can interchange places):

Here, A_1 is a matrix of size $N - n$ by k_1, A_2 is a matrix of size $N - n$ by k_2, P is a matrix of size n by k_1 each column of which contains all number from 0 through $n - 1$, and X is a matrix of size k_2 by n. Without loss of generality, we may assume that P is matrix D the ith column of which contains value i, $0 \le i \le n - 1$. Below is an example of PCA(15; 2; (14, 6), 3):

Any covering array of size N from CA(2; k, n) can be represented in the form of PCA(N; 2; (1, $k - 1$), n).

SCA(N; 2; (k_1, k_2), n) is a covering array of size N from CA(2; $k_1 + k_2$, n) in which, for all i, $0 \le i \le n - 1$, the first k_1 values in the row with number $N - n + i$ are equal to i and the last k_2 values are equal to zero. For example, an array from CA(2; k, n) can be represented as SCA(n^2; 2; (n, 1), n) if n is a power of a prime number. SCA is a particular case of PCA where all elements X are equal to zero.

This recursive algorithm constructs covering arrays for configurations of several types using recursive schemes. Configurations for which the algorithm works are (everywhere, $r \ge 1$ and n is a power of a prime number if not specified otherwise) SCA (or PCA)($N + r(n^2 - n)$; 2; (k_1n^r, $rk_1n^{r-1} + k_2n^r$), n) (i.e., an array from CA(2; $k_1n^r + rk_1n^{r-1} + k_2n^r$, n) with the number of rows $N + r(n^2 - n)$) using initial SCA (respectively, PCA)(N; 2; (k_1, k_2), n). Using the same initial SCA (or PCA) and having performed additional operations, one can obtain an array for the greater number of parameters: SCA (or PCA) ($N + r(n^2 - n)$; 2; ($n(k_1 + k_2)D_{r,n} + k_1D_{r-1,n}$, $n(k_1 + k_2)D_{r-1,n} + k_1D_{r-2,n}$), n), where $r \ge 2$, $D_{r,t} = \sum_{i=1}^{\lfloor (r+1)/2 \rfloor} C_{i-1}^{r-i} t^{r-i}$, and C_i^r is the number of combinations of i elements taken r at a time. This will be an array from CA(2; $n(k_1 + k_2)D_{r+1,n} + k_1D_{r,n}$, n) with the number of rows $N + r(n^2 - n)$. Numbers N, k_1, k_2, and n take the following values:

(i) $k_1 = 1$, $k_2 = m - 1$, and N is the number of rows in the initial covering array from CA(2; m, n) obtained by another method and represented in the form of PCA(N; 2; (1, $m - 1$), n).

(ii) $k_1 = n$, $k_2 = 1$, $N = n^2$, initial SCA(n^2; 2; (n, 1), n) is an array from CA(2; $n + 1$, n) with the number of rows equal to n^2 obtained by means of the algorithm described in Section 3.1.2.

(iii) $k_1 = 1$, $k_2 = 1$, $N = (l + 1)(n - 1) + 1$, for numbers n and l, there exists a distinct (n, l) cover starter (see Section 3.1.3). The initial SCA is an array from CA(2; $l + 1$, n) with the number of rows equal to $(l + 1)(n - 1) + 1$. This array is obtained by means of the algorithm described in Section 3.1.3 and according to [33] the array is represented in the form of SCA($(l + 1)(n - 1) + 1$; 2; (l, 1), n).

(iv) $n = 3$, ($k_1 = 14$, $k_2 = 1$, $N = 15$), ($k_1 = 60$, $k_2 = 14$, $N = 21$), ($k_1 = 220$, $k_2 = 114$, $N = 27$), ($k_1 = 1092$, $k_2 = 220$, $N = 33$). The initial SCAs are obtained by heuristic methods, which can be found in [33]. Any initial SCA(N; 2; (k_1, k_2), n), where n is a power of a

Table 3. Partitioned covering array PCA

A_1	A_2
P	X

Table 4. PCA(15; 2; (14, 6) [35]

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

Table 5. Construction of the resulting SCA (or PCA)

$A_1 \otimes B_1$	$A_2 \otimes B_1$	$A_1 \otimes B_2$
D	k_1X	O'

prime number, that are close to minimal ones are suitable for the method.

Brief description of the algorithm.

1. Suppose that there is an *initial* SCA(or PCA)(N; 2; (k_1, k_2), n) with partitioning A_1, A_2, D, and X (see Table 3).

2. Suppose that there is an *auxiliary* SCA(or PCA)(M; 2; (l_1, l_2), n) with partitioning B_1, B_2, D', and O'.

3. Joining of matrix A of size N by k and matrix B of size M by l ($A \otimes B$) is matrix C of size $N + M$ by kl the elements of which are defined as follows: $C_{i, (f-1)k+g} = A_{i, g}$, $1 \le i \le N$, $1 \le f \le l$, $1 \le g \le k$, and $C_{N+i, (f-1)k+g} = B_{i, f}$, $1 \le i \le M$, $1 \le f \le l$, $1 \le g \le k$.

4. The resulting SCA(or PCA)($N + M - n$; 2; $(k_1l_1, k_1l_2 + k_2l_1)$, n) is obtained from the initial SCA (or PCA) and auxiliary SCA by joining horizontally matrices $A_1 \otimes B_1$, $A_2 \otimes B_1$, and $A_1 \otimes B_2$. Then, the matrix composed of the horizontal joining of matrices D, k_1X, and O' of appropriate sizes, is joined to the bottom, where k_1X is matrix X repeated k_1 times (see Table 5).

5. Additional actions: if B_1 contains a submatrix $n \times \eta$ ($\eta \times n$) in which all numbers from 0 through $n - 1$ in each row are different, then the rows and columns in the resulting matrix can be permuted such that SCA(or PCA)($N + M - n$; 2; $(\eta(k_1 + k_2), (l_1 - \eta)(k_1 + k_2) + k_1l_2)$, n) is obtained. Altogether, $n\eta(k_1 + k_2)$ permutations will be required.

The table obtained is a covering array, which is proved in [33]. However, this array may not be minimal even if the initial arrays are minimal [33].

Time complexity of the algorithm without additional actions is $O((k_1l_1 + k_1l_2 + k_2l_1)(N + M - n))$ and with additional actions is $O((k_1l_1 + k_1l_2 + k_2l_1)(N + M - n) + n\eta(k_1 + k_2))$. The required memory without additional actions is $O((k_1 + k_2)N + (l_2 + l_1)M)$, since the array can be constructed row by row, but the initial and auxiliary arrays are to be permanently kept in the memory.

To obtain arrays for configurations described above, for the initial SCA (or PCA), the array specified in the corresponding item (i)–(iv) is selected, and for the auxiliary array, an array from CA(2; $n + 1$, n) is taken, where n is a power of a prime number, with n^2 rows, which is SCA(n^2; 2; $(n, 1)$, n). Such an array can be obtained by means of the algorithm described in Section 3.1.2. The construction is repeated r times. For the initial array, the array obtained on the previous step is taken, and for the auxiliary array, the same auxiliary array SCA(n^2; 2; $(n, 1)$, n) is selected. Time complexity for r iterations is $O(\sum_{i=1}^{r}((N + i(n^2 - n))(k_1n^i + ik_1n^{i-1} + k_2n^i)))$ without additional actions and $O(\sum_{i=1}^{r}((N + i(n^2 - n))(n(k_1 + k_2)D_{i+1, n} + k_1D_{i, n})) + \sum_{i=1}^{r-1}(n^2(n(k_1 + k_2)D_{i, n} + k_1D_{i-1, n})))$ with additional actions. The required memory is obtained by adding sizes of the greatest initial array and the auxiliary array: $O((N + (r-1)(n^2 - n))(k_1n^{r-1} + (r-1)k_1n^{r-2} + k_2n^{r-1}) + (n + 1)n^2)$ without additional actions and $O((N + (r-1)(n^2 - n))(n(k_1 + k_2)D_{r, n} + k_1D_{r-1, n}) + (n + 1)n^2)$ with additional actions.

In [33], a method based on binary codes is described, by means of which one can construct SCA($n^2 + r(n^2 - n)$; 2; $(n^{r+1}, (r + 1 + s)n^{r+1-s})$, n), i.e., arrays from CA(2; $n^{r+1} + C_s^{r+1} n^{r+1-s}$, n) with the number of rows equal to $n^2 + r(n^2 - n)$, where $r \ge s \ge 1$ and n is a power of a prime number. However, it is not difficult to check that, in the majority of cases, by means of the methods described, one can obtain SCA of the same size for configurations in which number k_1 and sum $k_1 + k_2$ are greater than or equal to those in the SCA constructed by this method.

3.1.7. Construction of uniform covering arrays of strength three with the use of ordered design [17]. The *ordered design* $OD_\lambda(t, k, v)$ is a matrix of size $\lambda C_t^v t! \times k$ with elements from 0 through v such that (1) each column contains v different values and (2) every t columns contain exactly λ identical rows such that values in different columns are different.

If $\lambda = 1$, then the corresponding ordered designs are denoted simply as OD(t, k, v). If q is a power of a prime number, then there exists OD(3, $q + 1$, $q + 1$) [17].

This recursive algorithm constructs a covering array from CA(3; $q+1$, $q+1$) using OD(3, $q+1$, $q+1$) with the number of rows equal to $q^3 - q$ (for q equal to a power of a prime number) and covering array A from CA(3; $q + 1$, 2) with N rows. The number of rows in the resulting array is equal to $q^3 - q + Nq(q + 1)/2 - (q^2 - 1)$, which is less than that in the array constructed by the algorithm considered in Section 3.1.3 with a wider application domain.

Brief description of the algorithm.

1. From array A, we obtain another $C_2^{q+1} - 1 = q(q + 1)/2 - 1$ versions of this array by replacing values of elements 0 and 1 with values lying between 0 and q having searched all possible combinations of pairs of different values except for already available 0 and 1.

2. The arrays obtained on the previous step (plus array A) are joined to one another from the bottom not including rows of the form $(x, x, \ldots, x)$, where x lies between 0 and q. As a result, an array with $Nq(q + 1)/2 - q(q + 1)$ rows is obtained.

3. The array obtained on the previous step is joined from the bottom to matrix OD(3, $q + 1$, $q + 1$).

4. $q + 1$ rows of the form $(x, x, \ldots, x)$, where x lies between 0 and q, are added.

The table obtained in this way is a covering (not necessarily minimal) array from CA(3, $q+1$, $q+1$) for q equal to a power of a prime number. The proof can be found in [17].

Time complexity of the algorithm is estimated as $O((q^3 - q + Nq(q + 1)/2 - (q^2 - 1))(q + 1)) = O(q^4 + Nq^3)$, and the required memory, as $O((q^3 - q + N)(q + 1)) = O(q^4 + Nq)$, since the array can be constructed row by row, but the initial array and matrix OD(3, $q + 1$, $q + 1$) are to be permanently kept in the memory.

3.1.8. Construction of uniform covering arrays with the use of perfect hash families [17, 32]. Let us introduce several auxiliary definitions.

A *perfect hash family* PHF(N, t; k, m) is a set of $N(k, m)$-hash functions (functions specifying mappings from a set K of cardinality k to a set M of cardinality m, which are denoted as h_j) such that, for any subset X of set K of cardinality t, there exists an injective (perfect) hash function h_j on X. This family can be specified by means of a matrix of size k by N the element $[i, j]$ of which is the value of function h_j on the ith element of set K. Any matrix of size k by N any submatrix of which of size $t \times N$ contains a row of length t all elements of which are different specifies some perfect hash family.

Difference matrix $D(k, n; \lambda)$ is a matrix of size $n \times k\lambda$ with elements from Z_k in which vector of difference of any pair of rows contains each element of field Z_k exactly λ times. For example, if GCD($(n - 1)!$, k) = 1, then matrix $D_{ij} = ij \bmod k$ is the difference matrix $D(k, n; 1)$.

This recursive algorithm constructs a covering array from CA(t; k^{2p}, n) by means of an initial array of size N from CA(t; k, n) if GCD($(t-1)t/2$, k) = 1. The size of the resulting array is $N((t - 1)t/2 + 1)^p$. This algorithm makes it possible also to construct PHF($N((t-1)t/2 + 1)^p$, t; k^{2p}, n) from PHF(N, t; k, n).

Table 6. Construction of the resulting matrix with the use of a difference matrix

$B_{0,0}$	$B_{0,1}$	...	$B_{0,(t-1)t/2}$
$B_{1,0}$	$B_{1,1}$	...	$B_{1,(t-1)t/2}$
...	...	...	...
$B_{k-1,0}$	$B_{k-1,1}$	...	$B_{k-1,(t-1)t/2}$

Brief description of the algorithm.

1. Let A be an initial PHF(N, t; k, n) matrix (or an initial transposed array of size N from CA(t; k, n)).

2. Let D be a difference matrix $D(k, (t-1)t/2 + 1; 1)$, $D_{ij} = ij \bmod k$, $0 \le i \le k - 1$, $0 \le j \le (t - 1)t/2$.

3. Let A^x be a matrix of the same size as A such that $A^x_{ij} = A_{(i+x)j}$, $0 \le x, i \le k - 1$, $0 \le j \le N - 1$.

4. The resulting matrix is composed from matrices $B_{ij} = A^{D_{ij}}$, $0 \le i \le k - 1$, $0 \le j \le (t - 1)t/2$, as shown in Table 6. This will be PHF(N($(t - 1)t/2 + 1$), t; k^2, m) if the initial matrix was a covering array. The proof can be fond in [17]. The constructed array is not necessarily minimal.

5. Applying the algorithm p times, we can obtain a PHF or a covering array for the above-specified configurations.

Time complexity of the algorithm is estimated as $O(k^p((t - 1)t/2 + 1) + Nk^{2p}((t - 1)t/2 + 1)^p) = O(Nk^{2p}t^{2p})$, and the required memory, as $O(k^p((t - 1)t/2 + 1 + N)) = O(k^p(t^2 + N))$, since the array can be constructed row by row, but the initial array and matrix $D(k^p, (t - 1)t/2 + 1; 1)$ are to be permanently kept in the memory.

There is an algorithm [32] that constructs an array from CA(t; k, n) by means of PHF(N_1, t; k, m) and an initial array of size N_2 from CA(t; m, n). The size of the resulting array is equal to N_1N_2.

Brief description of the algorithm.

1. Search all rows of the initial covering array (let us denote this matrix as CA) from 0 through $N_2 - 1$. Let the row selected have number j.

2. Search all columns of the initial PHF from 0 through $N_1 - 1$. Let the column selected have number l.

3. Search all values i from 0 through $k - 1$ and form a row of length k with double index (j, l) as follows: the ith element of the row is element CA_{xj}, where x is element PHF_{il}.

4. The resulting array has N_1N_2 rows and k columns.

The table obtained in this way is a covering array. The proof can be found in [17]. However, the constructed array is not minimal.

Time complexity of the algorithm is $O(N_1N_2k)$, and the required memory is $O(N_1k + N_2m)$, since the array can be constructed row by row, but the initial array and

matrix PHF(N_1, t; k, m) are to be permanently kept in the memory.

Using the first recursive scheme, one can obtain arrays from CA(3; $(2v-1)^{2^j}$, v) with $((2v-1)(q^3-q)+v)4^j$ rows, where $v > 2$, $v \equiv 0, 1 \bmod 3$, $q \geq v-1$ is a power of a prime number, and j is any integer, as well as an array from CA(3; $(2v-3)^{2^j}$, v) with $((2v-1)(q^3-q)+v)4^j$ rows, where $v > 2$, $v \equiv 2 \bmod 3$, and q is subject to the same constraints. To this end, for the initial array, one should take an array from CA(3; $2v$, v) with $(2v-1)(q^3-q)+v$ rows obtained by means of the method described in Section 3.1.3. If $v \equiv 0, 1 \bmod 3$, then, in order that GCD($(t-1)t/2$, k) = GCD(3, k) be equal to 1 (as required in the scheme), it is necessary that $k = 2v-1$; if $v \equiv 2 \bmod 3$, $v > 2$, then $k = 2v-3$. Using this scheme, one can also obtain the required PHFs for the second recursive scheme.

3.1.9. Construction of uniform covering arrays of strength greater than two with the use of recursive constructs (the Roux theorem) [15–17, 32, 38–40]. This recursive algorithm constructs a covering array from CA(3; $2k$, n) using a covering array from CA(3; k, n) and a covering array from CA(2; k, n).

If the numbers of rows in the first and second initial arrays are equal to N_3 and N_2, respectively, then, in the resulting array, it is equal to $N_3 + (n-1)N_2$. For cardinality greater than 3, this algorithm constructs a covering array from CA(t; $2k$, n) using $t-1$ covering arrays form CA(t; k, n), ..., CA(2; k, n).

Brief description of the algorithm for $t = 3$.

1. Suppose that there is an array CA(3; k, n) with N_3 rows. Let us denote its elements as $A^3_{i,j}$ ($1 \leq i \leq N_3$, $1 \leq j \leq k$).

2. Suppose that there is an array CA(2; k, n) with N_2 rows. Let us denote its elements as $A^2_{i,j}$ ($1 \leq i \leq N_2$, $1 \leq j \leq k$).

3. The first N_3 rows of the resulting array (which are denoted as $C_{i,j}$) consisting of $2k$ columns are constructed as follows:

3.1. $C_{i,j} = A^3_{i,j}$, where $1 \leq i \leq N_3$, $1 \leq j \leq k$;

3.2. $C_{i,j} = A^3_{i,j}$, where $1 \leq i \leq N_3$, $k+1 \leq j \leq 2k$.

4. Rows with numbers from $N_3 + 1$ through $N_3 + (n-1)N_2$ are constructed as follows:

4.1. $C_{i,j} = A^2_{(i-N_3-1) \bmod N_2 + 1, j}$, where $N_3 + 1 \leq i \leq N_3 + (n-1)N_2$, $1 \leq j \leq k$;

4.2. $C_{i,j} = (A^2_{(i-N_3-1) \bmod N_2 + 1, j-k} +_n ((i - N_3 - 1) \operatorname{div} N_2 + 1))$, where $N_3 + 1 \leq i \leq N_3 + (n-1)N_2$, $k+1 \leq j \leq 2k$, and $+_n$ is the addition modulo n.

The array obtained in this way is a covering array (see proof in [17]). However, it is not necessarily minimal even if the initial arrays are minimal (see [37]): for configuration (3; 18, 13), there exists an array of size 3912, whereas the method described constructs the covering array of size 4225 using the minimal arrays from CA(3; 9, 13) (CAN(3; 9, 13) = 2197) and CA(2; 9, 13) (CAN(2; 9, 13) = 169) as the initial ones.

Brief description of the algorithm for $t \geq 4$.

1. Suppose that there exist arrays A^t from CA(t; k, n) with N_t rows, ..., A^2 from CA(2; k, n) with N_2 rows.

2. Let B^N_{ij} denote the matrix of size N_iN_j by k obtained from A_i by repeating each row N_j times, where $2 \leq i, j \leq t-2$, and $i + j = t$.

3. Let C^N_{ji} denote the matrix of size N_iN_j by k obtained by repeating matrix A_jN_i times, where $2 \leq i$, $j \leq t-2$, and $i + j = t$.

4. Let E_i denote the matrix of size N_iN_j by $2k$, where the first k columns are matrix B^{Nt-i}_i, and the next k columns are matrix C^{Ni}_{t-i}, $2 \leq i, j \leq t-2$.

5. The first $N_t + (n-1)N_{t-1}$ rows of the resulting array are constructed similar to items 3 and 4 of the algorithm for $t = 3$, with the initial arrays being matrices A^t and A^{t-1}.

6. The remaining rows are obtained by joining matrices E_{t-2}, ..., E_2 to the array obtained on step 5 from the bottom.

The array obtained in this way is a covering array (see proof in [15]). However, it is not necessarily minimal even if the initial arrays are minimal. A counterexample is constructed similar to that in the case of $t = 3$.

Time complexity of the algorithm is $O(2kN_{max}(n + N_{max}(t-3)))$, and the required memory is $O(2kN^2_{max})$ (where N_{max} is the size of the largest initial array (N_t)), since the array can be constructed row by row, but the initial arrays are to be permanently kept in the memory.

There is no opportunity to specify all constraints on all possible combinations of parameter values for all algorithms presented in this section.

3.2. Methods for Constructing Nonuniform Covering Arrays and Variable Strength Covering Arrays

3.2.1. Construction of nonuniform covering arrays of fixed strength by means of uniform covering arrays of similar configuration [16]. This reduction algorithm constructs a nonuniform covering array by means of a uniform covering array of similar configuration. For example, it constructs an array from CA(2; 6, 3, 2, 5, 4, 3, 5) by means of an initial array from CA(2; 6, 5, 5, 5, 5, 5, 5) (i.e., CA(2; 6, 5)). Further study of classes of particular cases where this algorithm yields good results is required. Some thoughts on this subject can be found in [16].

Brief description of the algorithm.

1. Suppose that it is required to construct a nonuniform covering array A from CA(t; k, n_1, ..., n_k).

2. Suppose that there is a uniform covering array B from CA(t; k, n) constructed by another method,

where n is selected such that array B certainly contains all rows of array A (the simplest selection is $n = \max(n_1, ..., n_k)$).

3. Replace values from B that do not belong to the domain of admissible values of A by those that do belong to the domain of admissible values of A.

4. From the array obtained on step 3, delete all redundant rows so that the resulting array is still a covering one. As a result, we obtain a nonuniform covering array from CA(t; k, n_1, ..., n_k).

The resulting covering array depends on the way the redundant values are deleted; i.e., it seems likely that covering arrays constructed for one and the same configuration will be different. The minimality of the array constructed in this way is not guaranteed; however, it can be estimated how close the constructed array to a minimal one is [16]. The algorithm requires detailed analysis of the covering array configuration and needs nontrivial heuristics. Time complexity and the required memory depend on the particular cases. The array construction cannot be optimized by specifying constraints on possible combinations of values, although such constraints can be introduced. The amount of the required memory cannot be optimized either, since, in any case, the entire initial array is to be kept in the memory.

3.2.2. "Double projection" algorithm [21]. *Private pair.* Let us mark by symbol * the elements in the covering array that do not affect coverage of all tuples of size t (i.e., the matrix preserves properties of a covering array upon substitution of any values for these elements).

Let C be a covering array from CA(2; k, n_1, ..., n_k) with N rows. Let i and j be numbers of columns with values σ_i and σ_j, respectively, Then, an ordered pair $\{(i, \sigma_i), (j, \sigma_j)\}$ is called a *private pair* if either both σ_i and σ_j are equal to symbol * or pair (σ_i, σ_j) occurs only once in the submatrix of size $N \times 2$ composed of columns i and j. Let $I = \{i_1, ..., i_s\}$ be a set of columns. A row from CA is an *independent row over set I* if each pair of its values in columns $\{i, j\} \subseteq I$, where $i \neq j$, together with the numbers of these columns, is a private pair.

This reduction algorithm constructs the following nonuniform covering arrays:

• An array from CA(2; $k + 1$, $n_1 - 1$, ..., $n_k - 1$, s) consisting of $N - 1$ rows by means of an array from CA(2; k, n_1, ..., n_k) consisting of N rows, such that the initial array contains an independent row p over set I of cardinality s and value x_i in row p and column $i \in I$ is not symbol * and occurs at least in n_i rows of the initial array.

• An array from CA(2; $q + 1 + t$, $(q - t)^{q+1}$, s^t) of size $q^2 - t$ by means of a uniform array from CA(2; $q + 1$, q)) of size q^2, where q is a power of a prime number, $1 \le t \le q$ and $1 \le s \le q - t$.

Brief description of the algorithm.

1. Suppose that there is an array C of size N from CA(2; k, n_1, ..., n_k) satisfying conditions of item 1.

2. Divide all rows of array C, but p, into $s + 1$ classes C_1, ..., C_s, and D. Class C_1 contains all rows (but p) in which value x_i belongs to column i. D contains the remaining rows. All classes C_1, ..., C_s are different from one another, since p is an independent row over I. For each class C_i, perform steps 3–5 below.

3. For all rows from C_i, place value $i - 1$ to the new column $k + 1$.

4. From the condition that x_i in row p and column $i \in I$ is not symbol * and occurs in at least n_i rows of the initial array, it follows that C_i contains at least $n_i - 1$ rows. Select $n_i - 1$ rows from C_i and replace value x_{li} in the l_ith column by any admissible value such that all values in the l_ith column of the selected rows are different.

5. In all remaining rows from C_i, replace values in columns from I by symbol *.

6. In all columns not belonging to I (with index j), replace all x_{pj} by symbol *.

7. Delete row p.

8. In column $k + 1$ of rows from D, place *.

9. Rename values in each column, but column $k + 1$, such that they contain values from 0 through $n_i - 1$. This yields an array for configurations of the first type.

10. For configurations of the second type, for the initial array, select an array from CA(2; $q + 1$, q) with q^2 rows (constructed by means of the method described in Section 3.1.2), where q is a power of a prime number. In this array, any row is an independent row over any set of columns.

11. Select t rows p_1, ..., p_t to be deleted such that their first q elements with index j varying from 0 through q are equal to j and the last elements are equal to zero. The number of such rows is not greater than q.

12. The first s columns are included in set I. There is no sense to take more than $q - t$ columns, since the number of possible values in the column added cannot be greater than $q - t$.

13. Perform steps 1–9 for selected p_1, ..., p_t rows with the selected initial array and set I. This can be done by one pass through the array.

The arrays obtained are covering, but not necessarily minimal, ones (the proof can be found in [21]).

Time complexity of the algorithm is $O(N(k + 1))$, and the required memory is $O(Nk)$, since the initial arrays are to be permanently kept in the memory.

3.2.3. Construction of nonuniform arrays of strength $t = 2$ by combining blocks from uniform covering arrays and auxiliary arrays [5]. This recursive algorithm constructs a covering array from CA(2; k, n_1 ..., n_k), where $k > \max(n_1, ..., n_k)$. Further studies are required in order to extend this method to the case of CA(t; k, n_1, ..., n_k), $t > 2$, and the case of variable strength arrays.

Brief description of the algorithm.

1. Suppose that it is required to construct a nonuniform covering array from CA(2; k, n_1, ..., n_k). Let $n \ge \max(n_1, ..., n_k)$ be the least integer that is a power of a

prime number, and let $k > n$ (i.e., the number of the parameters is sufficiently large).

2. Construct a uniform array A' from CA(2; $n + 1$, n). The number of rows in the array is equal to n^2.

3. An auxiliary array $B(n^2 - 1, n + 1, n, d) = A'$ of the first type is A' without the first row and with the columns successively repeated d times. The number of rows in such an array is equal to $n^2 - n$, and the number of columns is $(n + 1)d$.

4. An auxiliary array $R(n^2 - nn, n, d) = A'$ of the second type is A' without the first n rows and the first column, with the remaining columns being successively repeated d times. The number of rows in such an array is equal to $n^2 - n$, and the number of columns is nd.

5. An auxiliary array $I(c, d)$ of the third type is the array of ones consisting of c rows and d columns.

6. An auxiliary array $N(n^2 - n, n, d)$ of the fourth type is the array of pairs of size $n \times d$ with the appended from the bottom array of triples of size $n \times d$, and so on, array of n's of size $n \times d$. Array $N(cn, d)$ has c rows and nd columns.

7. Construct the resulting array from the auxiliary ones for $s = \log_{n+1}k$ cycles. If $s = l$, from the array constructed on step 2, we take k columns. The array obtained is the resulting one.

8. The algorithm of selection of auxiliary arrays and the numbers of their repetitions on each step is described in [5]. It is too cumbersome to be fully presented here.

The array obtained in this way is a covering (the proof can be found in [5]), but not necessarily minimal, array. Examples presented in [5] show that the arrays constructed by this method are sufficiently small. The size of the array obtained is $\lceil \log_{n+1}(k) \rceil (n^2 - 1) + 1$. Time complexity of the algorithm is $O(n^2 + k\log^2 k)$, and the required memory is $O((n^2(n + 1) + k(\lceil \log_{n+1}(k) \rceil (n^2 - 1) + 1)) = O(n^3 + n^2 k \log_{n+1}k)$, where $n \geq \max(n_1, ..., n_k)$ is the least integer equal to a power of a prime number. The construction cannot be optimized in terms of memory used, since the entire array is to be constructed at once rather than row by row.

This algorithm does not allow one to extend the already existing set of rows.

There is also no way to specify constraints on combinations of parameter values that reduce the array size.

All methods presented below do not depend on the construction of uniform covering arrays and can be applied for the construction of both nonuniform and uniform arrays.

3.2.4. Recursive construction of uniform arrays of strength two [17, 33, 41, 42].

Array profile. By symbol *, we mark elements in the covering array that do not affect coverage of all tuples of size t (i.e., upon substitution of any values for these elements, the matrix preserves properties of the covering array). The profile of a matrix of size $N \times k$ is a row of length k in which value d_i is equal to the number of symbols * in the ith column of the matrix.

This recursive algorithm constructs covering arrays of strength two if there exist A from CA(2; k, v_1, ..., v_k) of size N with profile (d_1, ..., d_k) and, for all $i \in [1, k]$, B_i from CA(2; l_i, $m_{i,1}$, ..., $m_{i,li}$) of size M_i with profile ($f_{i,1}$, ..., $f_{i,li}$) such that $m_{ij} \leq v_i$ for all $j \in [1, l_i]$. By means of this algorithm, one can construct an array from CA(2; $l_1 + ... + l_k$, $m_{l,1}$, ..., $m_{1,l1}$, ..., $m_{k,1}$, ..., $m_{k,lk}$) consisting of $T = N + \max(M_i - d_i)$ rows.

Brief description of the algorithm.

1. Suppose that there is a nonuniform covering array A from CA(2; k, n_1, ..., n_k) of size N with profile (d_1, ..., d_k).

2. Suppose that, for all $i \in [1, k]$, there are nonuniform covering arrays B_i from CA(2; l_i, $m_{i,1}$, ..., $m_{i,li}$) of size M_i with profile ($f_{i,1}$, ..., $f_{i,li}$) such that $m_{ij} \leq n_i$ for all $j \in [1, l_i]$.

3. The first N rows of the resulting array consisting of $T = N + \max_{i=1}^{k} (M_i - d_i)$ rows and $l_1 + ... + l_k$ columns (we denote its columns as $C_{(i,j)}$, where $1 \leq i \leq k$, $1 \leq j \leq l_i$) are constructed as follows:

3.1 for all $1 \leq i \leq k$, fix i; now, l_i is a constant and A_i is the fixed i th column of matrix A;

3.2 further, for all $1 \leq j \leq l_i$, make column $C_{(i,j)}$ of the resulting array equal to A_i, i.e., repeat $A_i l_i$ times to obtain the ith block of rows with identical elements; d_i rows in the ith block are rows consisting of *.

4. Rows with numbers from $N + 1$ through T in each block are also filled with *.

5. In each ith block, starting from the top, select M_i rows that entirely consist of *. Such a row contains l_i elements.

6. The jth element in the kth row selected on the previous step is assigned the value of the jth element of the kth row of the B_ith array.

7. If there remain rows that entirely consist of * or are identical to the already constructed ones, they are deleted. The remaining * are replaced by arbitrary values that are admissible for the given column.

The array obtained in this way is a covering (the proof can be found in [41]), but not necessarily minimal, array. How to reduce the array size is described in [33].

Time complexity of the algorithm is estimated as $O((N + M)L + \sum_{i=1}^{k} (M_i l_i))$, and the required memory, as $O((N + M)L + \max_{i=1}^{k} (M_i l_i))$, where N is the size of the initial array, $M = \max_{i=1}^{k} (M_i - d_i)$, and $L = \sum_{i=1}^{k} l_i$. The construction cannot be optimized in terms of memory used, since the entire array is to be constructed at once rather than row by row.

There is also no way to specify semantic constraints that reduce the array size.

3.2.5. Optimization algorithms. Further, we consider a family of optimization, or the so-called greedy, algorithms. Algorithms of this kind are often applied to solving problems for which no efficient solution techniques exist. All greedy algorithms rely on the following general principle: every time when a choice is to be made, the variant that appears to be the best one at the moment is selected; i.e., a locally optimal variant is selected in the hope that it leads to the optimal solution of the global problem [43]. As applied to the algorithms for construction of covering arrays, this means that, on the current step, the algorithm selects the row that covers the greatest number of the uncovered pairs, triples, or other sets of values.

Generally, the greedy algorithms not always lead to an optimal solution [43]. As for construction of covering arrays, the greedy algorithms also often yield arrays that are not minimal.

Advantages of the greedy algorithms are as follows:

• capability of constructing uniform and mixed covering arrays for any parameter configurations; some algorithms can easily be modified for constructing variable strength covering arrays;

• sound time asymptotics;

• in some cases, arrays can be constructed row by row rather than as an entire matrix;

• the greedy algorithms can be used both independently and jointly with other techniques for minimizing the arrays obtained;

• it is possible to extend an algorithm by specifying constraints on possible combinations of parameter values reducing the array.

Disadvantages of the greedy algorithms are as follows:

• the constructed array is not always minimal and, quite often, is far from minimal one; for the majority of the greedy algorithms, the better the time estimate, the greater the size of the constructed array;

• in the general case, no methods exist for estimating proximity of the constructed array to the minimal array;

• great amount of memory is sometimes required, since all uncovered combinations (pairs, triples, etc.) of the parameter values are to be kept in the memory.

Arrays of one and the same configuration constructed by different greedy algorithms may differ, since values of some parameters are selected randomly in the course of construction [18].

The search of classes of configurations of covering arrays for which a particular greedy algorithm finds minimal, or close to minimal, arrays is a separate problem in the study of greedy algorithms. In this paper, this problem is not considered, and we assume that the application domain of all greedy algorithms includes any configurations of uniform and nonuniform covering arrays of fixed strength.

Now, let us consider some known greedy algorithms in more detail.

3.2.5.1. Algorithm based on the addition of a new parameter (IPO) [13, 18, 22].

Brief description of the algorithm.

1. Construct covering array of strength t for the first t parameters (complete search of combinations of the parameter values).

2. Add the next parameter (new column).

3. Expansion along the horizontal. Add values to the new column to cover as many tuples of cardinality t that include values of the new parameter as possible.

4. Expansion along the vertical. Add rows to the initial array to cover the remaining tuples of cardinality t.

5. Add the next parameter and return to step 2.

After slight modification, this algorithm can be used for the extension of the already existing covering array from $CA(t; k, n_1, \ldots, n_k)$ to an array from $CA(t; k + p, m_1, \ldots, m_k, \ldots, m_{k+p})$, where $n_i \le m_i$, $1 \le i \le k$, and also $\exists j\ 1 \le j \le k$, $n_j < m_j$, and/or $p > 0$.

Algorithms of expansion along the horizontal and vertical for $t = 2$ are described in [13]. Algorithms of expansion along the vertical always add a minimal number of tests for covering tuples of cardinality two. The time estimate $O(d^4k + d^2k^3)$ of the expansion along the vertical for $t = 2$, where k is the number of the parameters and $d = \max(n_1, \ldots, n_k)$, presented in [13] is not correct; the correct estimate is $O(d^4k^2 + d^2k^3)$. In that paper, two algorithms of expansion along the horizontal are described: one has exponential operation time and guarantees minimality of the array obtained, and the other has polynomial time estimate $O(d^5k^3)$, but does not guarantee minimality of the array. In [13], the latter estimate is mistakenly given as $O(d^5k^2)$.

A modification of the algorithm combining expansions along the vertical and horizontal is presented in [44].

1. Suppose that an initial array from $CA(2; k, n_1, \ldots, n_k)$ is to be supplemented by one parameter, which takes n_{k+1} values, where $n_{k+1} \le n_k$.

2. Copy columns k and $k + 1$ row by row in accordance with the following rule: if the value in the ith row of the kth column is less than n_{k+1}, then this value is placed to the ith row of the $(k + 1)$th column; otherwise, symbol * is placed to this position. In the course of copying, if a pair of values from the last and added columns is met for the first time, this row is marked as "mandatory." In this way, we cover all pairs from the added column and all columns, but the last one, as well as all pairs of form (x, x) from the added column and the last column. After this, it remains to cover $n_{k+1}n_k - n_{k+1}$ pairs.

3. For each uncovered pair from columns $(j, k + 1)$ with values (a, b), we do the following:

3.1. Select row p, the jth column of which contains *, or row p that is not marked as "mandatory," which contains a in the jth column. If there is no such a row, then add a new row that contains a in the jth column

and b in the $(k + 1)$th column. To the other columns, * is placed. Then take the next pair (step 3).

3.2. If row p has been found and contains * in the jth column, a is placed to this position. Row p is marked as "mandatory." Value b is placed to the $(k + 1)$th column of row p. Suppose that the previous value was b'. If this is *, take the next pair (step 3). Otherwise, for all pairs that have been covered after the replacement of b with b' (pairs from columns $(h, k + 1)$ with values (x, b'), where $1 \le h \le k - 1$ and x is the value in column h of row p), perform step 3.

In [44], an incorrect estimate of the covering array growth was used, due to which time complexity was calculated incorrectly. The correct values of time complexity and the required memory are $O(d^6k^3\log k)$ and $O(kd^2\log k + d^2)$, respectively, since it is required to keep in the memory the entire array and uncovered pairs for last two parameters.

For $t > 2$, the algorithm of expansion along the vertical from [13] was modified in [22].

1. Uncovered tuples of cardinality t are denoted as $((p_{k1}, w_1), ..., (p_{k(t-1)}, w_{t-1}), (p_i, u))$, where i the ordinal number of the added parameter, u is the value of this parameter, p_{kj} is the ordinal number of the already added parameter, and w_j is the value of this parameter, $k_j < i$, $1 \le j \le t - 1$.

2. The set of the initial rows of the array is denoted as T, and the set of new rows, as T'. Set T' is initially empty.

3. Search all uncovered tuples and select the next tuple $((p_{k1}, w_1), ..., (p_{k(t-1)}, w_{t-1}), (p_i, u))$.

4. If T includes a row that contains special symbol *, which denotes an arbitrary value, on the place with number p_{kj0} and either appropriate values w_j or symbols * in any combination on all other places, then all symbols * in such a row are replaced by values w_j corresponding to numbers of these places.

5. Otherwise, T' is supplemented with a row that contains values w_j on places p_{kj} and symbols * on all other places.

6. If not all uncovered tuples have been considered, go to step 3.

This algorithm adds the required number of tests to cover tuples of cardinality t; however, it cannot guarantee minimality of the array. The size of the array obtained will depend on the order the uncovered tuples of cardinality r are considered (step 3).

The modified polynomial (not guaranteeing minimality) algorithm of expansion along the horizontal for $t > 2$ looks as follows [22].

1. Suppose that it is required to supplement an initial array of size N with a column p_i with feasible values $v_1, ..., v_{ni}$.

2. If N is less than the possible number of values for the new column n_i, then add value v_j to the jth row of the initial array, $1 \le j \le N$. The construction is completed.

3. Otherwise, search rows to which the column has not been added yet.

4. For each row, search all values of column p_i.

5. For each pair (extendable row, value) calculate how many tuples of cardinality t have been covered as a result of extension of this row by the new value.

6. Select the pair (extendable row, value) that maximizes the number of the covered tuples. The selected value is placed to the selected row.

7. From the uncovered tuples of cardinality t, remove those that become covered.

8. If not all rows have been searched, return to step 5.

In work [22], steps 3–7 look more complicated. In that work, it is assumed that a uniform array is constructed, which makes it possible to use heuristics, which, in turn, considerably speed up construction of the array and reduce the amount of memory used.

Suppose that it is required to extend a uniform covering array from CA(t; $k - 1$, n) with the number of rows equal to r to a covering array from CA(t; k, n). The extension of row i (of length $k - 1$) by value v is the ith row of the initial array to which value v is added from the left. The length of the extended row is k. Since the initial array is covering, all tuples of cardinality t in it have already been covered, and it remains to cover only tuples in which the added column takes part. In [22], two auxiliary matrices are used to store information about covered tuples:

• In matrix $T_{i, v}$, the number of tuples covered as a result of extension of row i by value v and the number of tuples covered by the previous extensions are written. The number of the new tuples covered as a result of extension of row i by value v is equal to $C_{t-1}^{k-1} - T_{i, v}$ (since the array is uniform). The size of matrix T is r by n.

• $\mathrm{Cov}_{\Lambda, v}$ is a Boolean matrix in which element Λ, v is "true" if tuple Λ is already covered. Symbol Λ denotes a tuple of cardinality $t - 1$ where the last column does not take part (all such tuples can be numbered and the number of such tuples is equal to $n^{t-1}C_{t-1}^{k-1}$), and v is one of feasible values between 0 and $n - 1$. The size of matrix Cov is equal to $n^tC_{t-1}^{k-1}$.

The algorithm described in [22] effectively manipulates these data structures, and time complexity of the extension of a covering array from CA(t; $k - 1$, n) with r rows to a covering array from CA(t; k, n) is estimated as $O(r^2/n^{t-1}C_{t-1}^{k-1} + rn^tC_{t-1}^{k-1})$. The estimate of the required memory is $O(rn + n^tC_{t-1}^{k-1})$.

The total time required for the construction of an array from the very beginning is estimated as $O(N^2/n^{t-1}C_t^k + Nn^tC_t^k)$, where N is the number of rows of the resulting array. These estimates were obtained in [22] based on empirical data with regard to some assumptions imposed by the authors. If we add one more assump-

tion that the algorithm constructs almost minimal arrays and take into account that the number of rows in the almost minimal array is $N \leq n^t\log(n^tC_t^k)$, time complexity of the algorithm can be estimated as $O(n^{2t}\log^2(n^tC_t^k)/n^{t-1}C_t^k + n^{2t}\log(n^tC_t^k)C_t^k)$. The estimate of the required memory is $O(n^{t+1}\log(n^tC_t^k) + n^tC_{t-1}^{k-1})$.

In [22], a heuristics is presented that considerably reduces steps of the algorithm that support matrix $n < 10$ in the actual state. Instead of traversing matrix $T_{i,v}$ and placing correct values in $\mathrm{Cov}_{\Lambda,v}$, it is filled with an "average" value. If this heuristics is used, time complexity of the extension of a covering array from CA(t; $k-1$, n) with r rows to a covering array from CA(t; k, n) is estimated as $O(r^2n + rC_{t-1}^{k-1} + rn^tC_{t-1}^{k-1})$. The estimate of the required memory is $O(rn)$.The total time required for the construction of an array from the very beginning by the algorithm with the heuristics is estimated as $O(N^2nk + NC_t^k + Nn^tC_t^k)$, where N is the number of rows of the resulting array.

In a more general case of a nonuniform array, where the above-mentioned heuristics cannot be applied, time complexity estimate of modified algorithms for adding one parameter with n_{k+1} possible values to an array from CA(t; k, n_1, ..., n_k) will differ by an order of magnitude:

• for the expansion along the vertical, $O(t(m_{k+1}^{2t}(k + 1) + m_{k+1}^t(k+1)^2))$;

• for the expansion along the horizontal, $O(m_{k+1}^{2t}(k+1)^2n_{k+1})$ where $m_{k+1} = \max(n_1, ..., n_{k+1})$.

The total time complexity estimate will be

• $O(t(d^{2t}k^2 + d^tk^3))$for the expansion along the vertical and

• $O(d^{2t+1}k^3)$, where $d = \max(n_1, ..., n_k)$ for the expansion along the horizontal.

For the entire IPO algorithm, the estimate is $O(tk^6d^t(d^t + k + d^tk))$.

The amount of the required memory depends on the array size, the number of the uncovered tuples of cardinality t, and the number of tuples covered on each step. The greatest amount of memory is required when the last column is added; at this stage, the array size and the numbers of the uncovered and covered tuples of cardinality t will take maximum values: the array size will be $O(kd^t\log k)$ [18], the number of uncovered and covered tuples will be $O(d^tC_{t-1}^{k-1})$, and the amount of memory required for them will be $O(td^tC_{t-1}^{k-1})$. The total required memory is equal to $O(d^t(k\log k + tC_{t-1}^{k-1}))$, where $d = \max(n_1, ..., n_k)$.

The construction cannot be optimized in terms of memory used, since the entire array is to be constructed at once rather than row by row.

The original algorithm does not allow one to specify constraints on combinations of parameter values; however, it can easily be modified to take into account semantic constraints, which reduce the array size.

3.2.5.2. Algorithms based on selection of the best rows among candidates [4, 17, 45–47]. This family of optimization algorithms is efficient to use for extending an existing covering array to a bigger one (in height or width) that includes the original array.

Only algorithms from this family are capable of constructing variable strength arrays from the very beginning or by extending the already existing arrays of smaller size. To implement construction of variable strength arrays, it is required to invent appropriate heuristics (see below).

For particular configurations, the application domain of one or another heuristic is determined experimentally. For array construction from the very beginning, see works [4, 17, 23, 45–49]. As for extending an already existing array, we have not found any publications.

Brief description of the family of algorithms [17].

1. Suppose that it is required to construct a covering array from CA(t; k, n_1, ..., n_k).

2. Repeat steps 3–13 N_1 times.

3. While there tuples of cardinality t uncovered by the constructed array, perform steps 4–11.

4. Repeat steps 5–11 N_2 times to construct the next row.

5. Successively select parameters until no one is left performing steps 6–9.

6. Using rule PR_1 (a heuristic criterion of the best selection), among all k parameters, select a set of parameters P to which values are assigned in the first turn.

7. Using rule PR_2, among all parameters from set P, select one parameter p_i to which values are assigned in the first turn. Possible values of parameter p_i are numbers from 0 through n_i.

8. Using rule VR_1, among all values of parameter p_i, select a set of values V that are considered to be most appropriate.

9. Using rule VR_2, from set V, select one value v and assign value v to parameter p_i.

10. After successive repetition of steps 5–9 for all parameters, a row is obtained, which is test T.

11. If this row covers more tuples of cardinality t than that constructed the previous time, we keep it. Return to step 4.

12. After steps 3–11, a covering array CA is obtained.

13. If the number of rows in this array is less than that constructed the previous time, we keep the array. Return to step 2.

14. After N_1 repetitions of the construction, we obtain an array for configuration (t; k, n_1, ..., n_k). Since selection of the row (test T) covering the greatest num-

ber of tuples of cardinality t (steps 5–10) is an NP-complete problem [12], heuristics PR_1, PR_2, VR_1, and VR_2 are required in order to solve this problem for polynomial time. In [17], it is proved that the size of the covering array grows logarithmically as the number of parameters k grows. Based on this proof, a hypothesis was put forward that, for efficient construction of a covering array, on steps 5–10, it is not necessary to construct the row that covers the greatest number of tuples of cardinality t; instead, it is sufficient to construct a row that covers an *average number* of tuples of cardinality t. We denote this number as δ. In [17], it is calculated as $\delta = \Sigma_{1 \le p1 < \ldots < pt \le k}\delta_{p1, \ldots, pt}$, where $\delta_{p1, \ldots, pt} = r_{p1, \ldots, pt}/(n_{p1} \ldots n_{pt})$ and $r_{p1, \ldots, pt}$ is the number of the uncovered tuples of cardinality t that include values of parameters $p_1, \ldots, p_t$. Number $\delta_{p1, \ldots, pt}$ is called local density, and δ, global density.

Possible variants of the heuristic selection criterion PR_1 are as follows:

1. The best parameter is that with the greatest number of possible values. We denote this criterion as MAX_N.

2. The best parameter is that that is included in the greatest number of tuples of cardinality t uncovered at the given stage. For example, suppose that there are uncovered pairs $(v_{2,1}, v_{3,2})$, $(v_{1,2}, v_{2,2})$, and $(v_{2,2}, v_{3,1})$, where $v_{i,j}$ is the jth value of the ith parameter. Then, in accordance with this criterion, the best parameter is the second parameter ($i = 2$), since it is met three times. This criterion takes into account only uncovered tuples remaining after the previous parameters have been fixed. We denote this criterion as P_IN_MAX_UC.

3. The best parameter is that that is included in a certain number, equal to δ (not necessarily maximal), of uncovered tuples of cardinality t. This criterion is more flexible and makes it possible to take into account (under appropriate choice of δ) not only fixed parameters but also those that have not been fixed yet (see [17]). This is criterion δ.

4. The best parameter is selected in a random way. We denote this criterion as RAND.

Since several parameters may meet the PR_1 criterion of the best parameter, criterion PR_2 is needed to select one parameter among several ones. Possible variants of the heuristic selection criterion PR_2 are as follows:

1. Take the parameter with the first ordinal number. This is the BY_ORD criterion.

2. Select the parameter in a random way.

3. To resolve collisions of the first and third variants of PR_1, for PR_2, we can take the second variant of PR_1 (i.e., to be greedy once more) and, then, one of the first two criteria PR_2 in order to select the only one variant.

Possible variants of the heuristic selection criterion VR_1 are as follows:

1. The best value is that that covers the greatest number of tuples of cardinality t that have not been covered at the given stage for the parameters that have already been fixed. We denote this criterion as V_IN_MAX_UC.

2. The best value is that that covers a certain number, equal to δ (not necessarily maximal), of uncovered tuples of cardinality t. This criterion is denoted as δ.

3. The best value is selected in a random way.

Criterion VR_2 is selected similar to PR_2.

The majority of the currently existing commercial and free tools generating combinatorial test suites use just greedy algorithms for constructing covering arrays. Table 7 shows heuristics used by these algorithms.

Some variants of the greedy algorithms (ATGT [8], PICT [26]), to construct each row, search uncovered tuples of cardinality t–t parameters and t values–rather than parameters and values. Several selected tuples form a row of the test. These algorithms use the following heuristics for searching tuples:

1. Random search. In this case, the algorithm is launched several times, and the best solution is selected.

2. Search using lexicographical order. This variant does not yield good results.

3. Tuple t_1 is considered to be better than (is selected before) tuple t_2 if pairs parameter–value available in tuple t_1 occur more rarely in the already covered tuples than pairs parameter–value available in tuple t_2.

4. Tuple t_1 is considered to be better than tuple t_2 if the pair parameter–value from tuple t_1 that was least often used in the previous tests occurs more rarely in the already generated tests than the pair parameter–value from tuple t_2 that was least often used in the previous tests.

5. Tuple t_1 is considered to be better than tuple t_2 if the pair parameter–value from tuple t_1 that was most frequently used in the previous tests occurs more rarely in the already generated tests than the pair parameter–value from tuple t_2 that was most frequently used in the previous tests. Although this criterion seems to be opposite to the previous one, it generally does not result in the reverse order of tuples.

6. Tuple t_1 is considered to be better than tuple t_2 if its pairs parameter–value are distributed more uniformly in the previous tests. For each pair, the mean-square deviation is calculated. This criterion is based on the same considerations as those used in algorithm DDA [46]: each new row–test should cover an average number of tuples (δ).

The last four criteria can be used in two ways: (1) tuples can be selected independent from one another or (2) pairs parameter–value that have already been used in the constructed part of the test row are taken into account when the next tuple is selected. The latter way demonstrates better results but involves greater time expenditures.

Table 7. Heuristics of the covering array generation tools

	PR_1	PR_2	VR_1	VR_2	N_1	N_2
AETG [4, 45]	RAND*	–	V_IN_MAX_UC	RAND	≤50	≤50
TCG [47]	MAX_N	BY_ORD	V_IN_MAX_UC	BY_ORD	1	$\max(n_1 \dots n_k)$
DDA [46]	Δ	BY_ORD	Δ	BY_ORD	1	1
CTS [16, 19]	P_IN_MAX_UC**	RAND	V_IN_MAX_UC**	RAND	1	1
Jenny [48]	RAND	–	–***	–	1	1

Notes: * At the first passage of the parameter selection loop, a parameter is selected whose value occurs in the greatest number of tuples of cardinality t that have not been covered at the given stage.

** For t parameters and t values of these parameters, rules P_INMAX_UC and V_IN_MAX_UC, respectively, are applied: those t parameters and their values are selected that contain parameters and values included in the greatest number of the uncovered tuples. For all other pairs (parameter, value), vector $(p_0, \dots, p_{t-1})$ is constructed, where p_i is the number of uncovered tuples (not taking into account $t - 1 - i$ values that have already been fixed) of cardinality t that contain the given pair. The best pair is selected as a pair for which the corresponding vector takes the greatest value. Vectors are compared lexicographically. If there are several pairs with equal vectors, the best pair is selected randomly.

*** No heuristic is used. A row is selected that covers the greatest number of tuples of cardinality t. In the general case, this requires exponential time.

We have not found any information about heuristics used in the greedy algorithms employed by the following tools [49]: CATS, TestCover.com, CaseMarker, and Pro-test. Analysis of heuristics used in industrial tools was carried out in work [23]. They decided to use P_IN_MAX_UC as PR_1, RAND as PR_2, V_IN_MAX_UC as VR_1, δ as VR_2, $N_1 = 5$, and $N_2 = 1$. In addition, instead of repetition of construction of a current test, they employ a heuristic algorithm of increasing the number of tuples covered by each newly constructed test. In other words, instead of step 11 of the above-discussed algorithm, heuristic search of a local extremum in the row is performed. In a number of particular cases, such a technique makes it possible to construct smaller covering arrays in less time compared to the industrial tools listed in Table 3. By means of this method, the least covering arrays for configurations where the number of parameters values is equal to 2 and the strength is 5–8 were constructed in 2007 [37].

This family of algorithms is characterized by polynomial operation time, which depends on the heuristics used. In the general case, the operation time is estimated as $O(k(f(\mathrm{PR}_1) + f(\mathrm{PR}_2))d(f(\mathrm{VR}_1) + f(\mathrm{VR}_2)))$, where $d = \max(n_1, \dots, n_k)$, and $f(\mathrm{PR}_1), f(\mathrm{PR}_2), f(\mathrm{VR}_1)$, and $f(\mathrm{VR}_2)$ are operation times of the corresponding heuristics. The memory estimate is calculated similar to that for the IPO algorithm: $O(d^t(k\log k + tC_{t-1}^{k-1}))$. The array can be constructed row by row; however, all already constructed rows are to be stored in the memory. The algorithm permits constraints on possible combinations of parameter values. In [8, 24–26], it is shown how to modify almost any greedy algorithm from this family such that it will take into account such constraints. Moreover, it is shown how they can be specified in the form of logical predicates. Comparison of efficiency of using various logical solvers SAT is also provided [8, 24, 25].

3.2.6. Solution of optimization problem. Further, we consider algorithms based on solving the following optimization problem. Suppose that we have a set of feasible solutions and an estimation function $c(S)$ defined for all $S \in \Sigma$. The optimal solution is a feasible solution with the minimal estimation function. Transformation T is an action applied to S to obtain a new feasible solution.

The problem of covering array construction can be reduced to the above problem as follows: Σ is a set of all arrays (possibly, with uncovered combinations), S is one of the arrays, $c(S)$ is the number of the uncovered combinations (if $c(S) = 0$, then S is a covering array), and T is a variation of values of elements of array S.

3.2.6.1. Hill climbing [17]. This optimization algorithm is efficient for improving (i.e., reducing) the existing test suites for any configurations. For the initial solution, an available test suite is selected.

Brief description of the algorithm.

1. Repeat array construction not less than N_1 times or until a covering array is constructed (steps 2–7).

2. Select an initial solution in a random way. It becomes the current solution S.

3. Apply a random transformation of the current solution to obtain solution S'.

4. If $c(S') \le c(S)$, then S' is taken to be the current solution.

5. If the stopping heuristic condition is fulfilled, go to step 6; otherwise, return to step 3.

6. After steps 2–5, we have solution S, which is not necessarily a covering array.

7. If the constructed array is covering and its size is less than the size of the previously constructed array, keep it and return to step 2.

8. Having repeated construction N_1 times (or more), we obtain the desired array.

In the general case, the algorithm has exponential time complexity and memory expenditures. The construction cannot be optimized in terms of memory used, since the entire array is to be constructed at once rather than row by row. The original algorithm does not allow one to specify constraints that reduce size of the array or speed up its construction.

In [2], the algorithm was modified for work with variable strength covering arrays. It is successfully used for small values of parameters of covering array configurations.

3.2.6.2. Tabu search [17, 50, 51]. This optimization algorithm is efficient in the same cases where the hill climbing algorithm is efficient. However, owing to the possibility of specifying constraints, this algorithm can work much faster in real applications [50].

The idea of the algorithm is the same as that of the hill climbing algorithm, but there are certain distinctions.

1. Repeat array construction N_1 times for a configuration Q with k parameters for which the upper and lower bounds of the array size (N_{min}, N_{max}) are known.

2. Select an initial solution—an array of a size belonging to the range (N_{min}, N_{max})—in a random way. It becomes the current solution S.

3. Apply a random transformation of the current solution as described below.

3.1. Select in a random way an uncovered tuple of cardinality t from the set of uncovered tuples and denote it as UT.

3.2. Select rows from solution S such that variations of only one element of such a row cover UT. If there are no such rows, select one row in a random way.

3.3. For each selected row, replace its element (or elements) such that UT became covered and calculate the cost of the new solution (with replaced element), i.e., the number of the uncovered tuples of cardinality t.

3.4. Among all solutions, select solutions with the lowest cost that do not contradict the constraints used.

3.5. If such a solution is not unique, select solution S' in a random way. S' is taken to be the current solution.

4. If S' is a covering array, store S' and, then, select a new random initial solution the number of rows in which is less than that in S' and return to step 3. If the number of loops in the covering array construction exceeded N_2, go to step 5; otherwise, return to step 3.

5. If there is a covering array constructed on the previous steps, keep it; otherwise, select a new random initial solution in which is greater than that in S' and return to step 3.

6. After steps 2–5, we have solution S that is a covering array.

7. If the size of the constructed array is less than the size of the previously constructed array, keep it and return to step 2.

8. Having repeated construction N_1 times, we obtain the desired covering array.

The following constraint (see step 3.4) is described in [50]: if solution S' became current in the previous T loops (steps 3–4 ($1 \leq T \leq 10$)), it cannot become current anymore. This constraint does not allow the algorithm to get caught in an endless loop.

It is possible to specify other constraints as well. They can be used to assign greater weight to modifications that result in some desirable features.

In the general case, the algorithm has exponential time estimate. If an upper and lower bounds of the array size N_{min} and N_{max} are known, time complexity is estimated as $O(N_1 k N_{max} d(k + \log T) + dkN_{max}(T + k^2))$, where d is the maximum number of admissible parameter values, and the required memory, as $O(kN_{max}T)$, where T is the number of the stored arrays to implement constraints. The memory use cannot be optimized, since the entire array is to be constructed at once rather than row by row.

These estimates considerably exceed estimates for optimization algorithms from other families (for example, because N_1 should be sufficiently large); therefore, it is advisable to use this algorithm only for research purposes.

3.2.6.3. Simulated annealing [17, 39, 51–53]. This optimization algorithm is efficient in the same cases as the hill climbing algorithm. The algorithm is protected from getting into a cycle in a local minimum, which, theoretically, makes it possible to find a better solution compared to simple hill climbing. Since this algorithm relies on probabilistic foundation, it needs further study to collect statistics and make clear for what configurations it works better.

In [39], the simulated annealing is efficiently used for construction of initial covering arrays and arrays of special form for their subsequent use in recursive constructions. These recursive constructions are described in Sections 3.1.6–3.1.9 of this paper. Such a combination makes it possible to obtain relatively small (compared to other methods) covering arrays in reasonable time.

Brief description of the algorithm. This is a generalization of the hill climbing algorithm. The algorithm selects in a probabilistic way a current solution with the worst estimation function. The worst solution is selected with the probability $e^{-(c(S') - c(S))/KT}$, where $c()$ is an estimation function, K is a constant, and T is a parameter called a *temperature*, which can be modified in the course of operation. This allows the algorithm to avoid infinite looping in a local minimum and continue search of the global extremum. The annealing degree reduces by small steps, making it possible to reach an equilibrium state by applying a sequence of transformations of the current solution. Usually, the annealing degree reduces in accordance with the equation $T = \alpha T$, where α is a number that is slightly less than 1. As soon as an appropriate stopping condition is fulfilled, the current value is taken to be an approximate solution of the problem.

In the general case, the algorithm has exponential time complexity. If an upper and lower bounds of the array size N_{min} and N_{max} are known, time complexity is estimated as $O(N_1k + k^3N_{max})$, and the required memory, as $O(kN_{max})$. The memory use cannot be optimized, since the entire array is to be constructed at once rather than row by row.

These estimates considerably exceed estimates for optimization algorithms from other families; therefore, it is advisable to use this algorithm only for research purposes.

No provision is made in the original algorithm for specification of constraints that reduce the array size or speed up its construction.

3.2.6.4. Great deluge [17, 54, 55]. This algorithm is classified among the so-called threshold accepting algorithms. The method is similar to that used in the simulated annealing algorithm. It differs from the latter in that, instead of using probability when accepting a solution with the worst value of the estimation function, the solution is accepted when its cost does not exceed a certain current value, which is called a water level. In the course of the algorithm operation, the water level gradually falls, reducing the number of possible transformations. Quite often, this method demonstrates fast convergence to a solution [54, 55].

The algorithms described in Section 3.2.6 have poor time estimates and, therefore, almost are not used in industry tools. However, they often construct arrays that are closer to optimal ones than arrays constructed by the greedy algorithms. For this reason, these algorithms are often used in research. They can also be used for constructing variable strength covering arrays under appropriate selection of the estimation function.

3.2.7. Genetic algorithms [17, 51]. The basic idea of the genetic algorithms is control of the solution population and its development with the help of two operations—mutation and crossing. Mutations introduce local variations in a solution belonging to a population, and crossing combines a part of one solution with a part of another solution. Solution survival in the new generation depends on whether this solution is useful compared to other solutions in the generation.

In the case of the covering arrays, the solution population is a set of arrays (not necessarily covering ones) of given size N, and N is the parameter of a current run of the algorithm. The population size is supported fixed and equal to M. On each step of the algorithm, crossing and mutation occur. Crossing is a random selection of two arrays and recombination of their rows to obtain two new arrays possessing features of both parents. Mutation is a modification of the new arrays, for example, by the rules of step 3 of the tabu search algorithm or in some other way, After sorting, only M best arrays, which have the best usefulness function (for example, the least number of the uncovered tuples of cardinality t), survive. The use of the genetic algorithms for searching covering arrays has been poorly studied. The first results on the use of the genetic algorithms for searching covering arrays were published in [51]. Time complexity of this algorithm is $O(N_1Mk^2(N_{max} + d^2))$, where N_1 is the number of runs of the algorithm, $d = \max(n_1, ..., n_k)$, and N_{max} is an upper estimate of the array size. The required memory is $O(kMN_{max})$.

In [56], a memetic genetic algorithm for configurations of the form (3, k, 2) was described. For the initial population, M arrays are selected (M is divisible by 4) in which the numbers of 0's and 1's in each column are equal to one another (or differ by one if the number of rows is odd); 0 and 1 are selected randomly. The M arrays are divided into four sets with the same number of arrays in each set. Two pairs of sets are crossed by the following rule: a row with number i is selected in a random way; then, $C_{mn} = A_{mn}$ ($D_{mn} = B_{mn}$) if $m \le i$ or $C_{mn} = B_{mn}$ ($D_{mn} = A_{mn}$) if $m > i$, where A_{mn} and B_{mn} are parents and C_{mn} and D_{mn} are descendants. For mutation, an algorithm of searching a local minimum of uncovered triples in the array, which is based on the simulated annealing, is used. As a result of mutation, several arrays are obtained. For the terminal mutation, an array that has the least number of uncovered triples and is most distant from the initial one is selected, where the distance is measured by the number of different elements in the array (similar to the Hamming distance). After the crossings and mutations have been performed, there remain M arrays with the least number of the uncovered triples. For several configurations, this algorithm has constructed arrays that turned smaller than the arrays constructed earlier by other algorithms. However, no results of comparison of the algorithm with other techniques that are potentially capable of constructing quality arrays (for example simulated annealing and tabu search) are presented in [56]. The question of whether it is possible to obtain similar results by other methods for less time and with lesser memory expenditures remains open.

The published results count not in favor of the genetic algorithms: for the majority of configurations, the time of generation of a covering array by a genetic algorithm is greater than that by the simulated annealing or tabu search algorithms for the same configuration.

3.2.8. Backtracking search [27]. Algorithms of this type use certain rules of successive search of possible values in the array cells for a given configuration (t; k, n_1, ..., n_k), $n_1 \ge ... \ge n_k$, and array size N. Since algorithms of this type require much time and memory for the construction of an array from the very beginning, they are used only for research purposes, for small values of parameters of the configurations, or for augmenting an already existing array, under the condition that the upper bound of its size N is known.

Brief description of the algorithm.

1. The array construction begins with some initial array. In the general case, this may be complete search

of the first t parameters (which have the greatest number of values).

2. The other cells of matrix $k \times N$ are searched. The current value in a current cell of the array is selected by using the CH heuristics.

3. Step 2 is repeated until the array is completely constructed or the stopping heuristics SH fires.

4. If the array obtained is covering, the algorithm stops; otherwise, the array backtracks to the previous state, and the algorithm returns to step 2.

The heuristics imposing constraints on the selection of a cell and value were proposed in [27] and are as follows:

• All vectors representing rows and columns of an array should be arranged in lexicographical order. In other words, if i_1 and i_2 are numbers of rows and $i_1 < i_2$, then, for the vectors composed of the cell values, we have $(v_{i1,1}, \ldots, v_{i1,k}) \leq (v_{i2,1}, \ldots, v_{i2,k})$. The same is true for the columns.

• Let all possible values for each column j be ordered from 0 through n_j, and let M_j be the greatest ordinal number among the values that have already been placed to column j. Then, the number of the next value in column j may vary from 0 through $M_j + 1$.

A number s, $1 \leq s \leq t$, is selected. For each tuple of cardinality s, the frequency of its occurrence in some set of columns is calculated. Frequencies for one and the same set of columns must not differ by more than 1.

• The selected values must meet user-specified constraints on possible combinations of parameter values.

• If several values for a current cell do not meet the above-listed constraints, there are several ways to select one of them:

• to select the value that is lexicographically first;

• to select a random value;

• to select the value that covers the greatest number of the uncovered tuples of cardinality t. It is noted in [27] that this criterion is efficient only for some classes of configurations; however, it requires more computational resources compared to the previous two criteria.

The stopping heuristics proposed in [27] is as follows. Let X be the set of uncovered tuples of cardinality t containing column c, and let Y be the set of tuples of cardinality t containing column c in which values in column c have not been placed yet. Calculate set $N(X)$ of tuples adjacent to X, i.e., tuples from Y that differ from X only by a value in column c. If cardinality of set $N(X)$ is less than cardinality of set X, then all tuples from X cannot be covered, whatever the way the values in cells of tuples from Y have been placed. It is required to return to the previous step.

All these heuristics can be used simultaneously, and the array obtained will be covering. The proof can be found in [27].

The required time is difficult to estimate, since there is no information on how much the proposed heuristics reduce the search. In the general case, the time of the algorithm operation can be estimated as $O(MNkf(\mathrm{CH})f(\mathrm{SH}))$, where $M \geq k(N - N_{\mathrm{init}})$ is the number of the arrays searched, $f(\mathrm{CH})$ and $f(\mathrm{SH})$ are operation times of the corresponding heuristics, and N_{init} is the size of the initial array. The required memory is $O(Nk^2(N - N_{\mathrm{init}}) + d^t C_k^t)$, $d = \max(n_1, \ldots, n_k)$, since all arrays in the entire selection branch, as well as information about all tuples of cardinality t, are to be kept in the memory.

The algorithm provides an opportunity of specifying constraints on possible combinations and using them in the heuristics, which reduce the operation time of the algorithm.

3.2.9. Optimization of a given covering array [28]. This hybrid algorithm, which can simultaneously be classified among the reduction and optimization algorithms, is efficiently used [28] for improving (reducing) the already existing test suites for any configurations. The idea of the algorithm consists in searching elements of the initial array from $CA(t; k, n_1, \ldots, n_k)$ that do not affect coverage of tuples of cardinality t. These elements can be replaced by symbol *, which means that any feasible value can be substituted for it such that the array remains covering. Let N be the number of rows in the initial array.

Brief description of the algorithm.

1. All elements of the initial array are considered to be candidates for the replacement by symbol *.

2. Out of k columns of the initial array, all possible ordered combinations of t columns $(c_1, \ldots, c_t)$ are selected. The number of such combinations is C_t^k. For each combination of columns, perform steps 3–7.

3. Look through all rows of the initial array. For each row r, the values in columns $(c_1, \ldots, c_t)$ of row r form vector $(v_1, \ldots, v_t)$. If this vector was met for the first time, then elements of the array in row r and columns $(c_1, \ldots, c_t)$ are removed from the candidates for the replacement by *.

4. After completion of the iterations of steps 2–3, all elements remaining in the list of candidates for the replacement by * are replaced by *.

5. If, after step 4, the array contains rows entirely consisting of *, these rows are deleted.

6. The row containing the greatest number of * is moved to the last place in the array. This number is stored.

7. For all remaining rows containing *, look through all elements of row r where * occurs (let this element occur in column c). If the value in the last row of column c is *, replace element of row r in column c by a random value. Otherwise, replace it by the value in the last row of column c.

8. Permute all rows, except for the last one, in a random way to obtain a new initial array. Return to step 2.

9. If after M_1 repetitions of steps 2–8, no rows were deleted or the number of symbols * in the last row did not increase, the array obtained is saved.

10. For the new array, the array obtained on the previous step is selected. In this array, the last row becomes the first one, and a random row containing symbol * becomes the second row. In the first and second rows, all * are replaced by random feasible values. Return to step 2.

11. Steps 2–10 are repeated M_2 times. After each iteration, out of the saved and the newly obtained arrays, the lesser one is saved.

Time complexity of the algorithm is $O(M_2(M_1(C_t^k tN + Nk + N - 1) + 6k))$, and the required memory is $O(Nk + d^t)$, $d = \max(n_1, ..., n_k)$. The construction cannot be optimized in terms of memory, since the array is to be constructed at once rather than row by row. No provision is made for specifying constraints on possible values of parameters.

3.3. Survey Summary

Information about all above-discussed methods for constructing covering arrays is collected in Tables 8–10 with indication of the configuration for which each method is advisable to use.

The following notation is used in the table headings:

• **M** shows whether the constructed array is minimal;

• **E** shows whether the algorithm can be used for extending initial arrays;

• **S** shows whether the algorithm permits addition of semantic constraints.

The following notation is used in the table bodies:

• $d = \max(n_1, ..., n_k)$ for configurations of the form $(t; k, n_1, ..., n_k, ...)$;

• N_e is the number of the algorithm runs;

• $N_{\max}$ is the upper estimate of the array size.

For the recursive algorithm of construction of uniform arrays of strength $t = 2$,

• $k_1 = 1$, $k_2 = m - 1$, N is the number of rows in the initial covering array from CA(2; m, n) obtained by another method;

• $k_1 = n$, $k_2 = 1$, $N = n^2$;

• $k_1 = 1$, $k_2 = 1$, $N = (l + 1)(n - 1) + 1$; for numbers n and l, there exists an (n, l) cover starter;

$n = 3$, $(k_1 = 14, k_2 = 1, N = 15)$, $(k_1 = 60, k_2 = 14, N = 21)$, $(k_1 = 220, k_2 = 114, N = 27)$, $(k_1 = 1092, k_2 = 220, N = 33)$.

The direct algorithms are advisable to use for constructing covering arrays with a small number of parameters, as well as in some other particular cases. Their basic advantage is that they construct almost minimal (or even minimal) arrays in a reasonable time. The basic disadvantage of these algorithms is a very narrow application domain and impossibility of discarding rows of the array that do not meet constraints on feasible combinations.

The recursive algorithms are good complements of the direct algorithms for some special classes of initial array configurations. For example, the Roux method constructs an almost minimal array from CA(3;20,9) consisting of 1377 rows using an array from CA(3;10,9) consisting of 729 rows and an array from CA(2;10,9) having 81 rows. Both initial configurations belong to the class $(t; p^k + 1, p^k)$. However, for other classes of configurations or if the initial arrays are not minimal, the resulting array may occur considerably larger than the one constructed, e.g., by a greedy algorithm [37].

The recursive algorithms should be used with caution and only in special particular cases. The advantage of the recursive algorithms is that they construct almost minimal arrays for particular cases in a reasonable time. Their basic disadvantage is a narrow application domain and impossibility of specifying constraints.

The greedy algorithms are universal and, therefore, are used in the majority of known tools for constructing covering arrays [49]. They possess the following important from the practical standpoint features:

• the possibility of discarding array rows that do not meet constraints on feasible combinations;

• the possibility of constructing variable strength arrays; and

• the possibility of extending the already existing test suites.

The basic disadvantage of the greedy algorithms is that the proximity of the resulting array to the minimal one is inversely proportional to the operation time of the algorithm. The use of the greedy algorithms for optimization needs further studies.

The application domains of the algorithm of removing redundant values and block combining algorithms also need additional studies. These methods show pretty good results; however, the implementation of the block combining method in the TConfig tool [5] has a narrow application domain, and the method of removing redundant values cannot be implemented as a tool without preliminary formalization and algorithmization. Some studies of this problem are discussed in [16, 19].

The class of covering array configurations for which a good solution exists can considerably be extended by combining different methods. In order to find an efficient combination of methods, it is required to analyze the initial configuration of the array. A solution based on combining direct, recursive, and optimization algorithms—the CTS package—was suggested in [16]. An extension of the technique of combining algorithms used in CTS looks like a promising direction of future studies.

Combination of methods based on analysis of initial combinations may rely on the following principles:

Table 8. Algorithms for constructing uniform covering arrays

Algorithm	Application domain	Class of the algorithm	Time complexity and memory expenditures	M	E	S
Boolean	$(2; k, 2)$	Direct	$O(k)$, Memory: $k\log_2 k$	+	–	–
Affine	$(t; n+1, n)$, n is a degree of a prime number; $(3; 2k+2, 2k)$	Direct	$n^t + (n+1)\log_2(n+1) + O(\log_p^4 n)$, Memory: $O(n) + O(\log_p^3 n)$	+	–	–
Group actions	$(3; 2k, n+1)$, n is a degree of a prime number	Direct	$O(2k((2k-1)(n^3-n)+n+1)+n^3)$, Memory: $O(2k(2k-1)+n^4-n^2)$	–	–	–
	$(2; 4, n)$, $n \equiv 2 \pmod 4$		$O(4(n+1)n)$, Memory: $O(4(n+1)+n^2)$			
	$(2; l, g)$, (g, l) cover starter $(2; l+1, g)$, distinct (g, l) cover starter		$O(l(l(g-1)+l))$, Memory: $O(l+(g-1)g)$			
Strength reduction	$(t-1; k-1, n)$ from $(t; k, n)$	Reduction	$O(Nk)$, Memory: $O(Nk)$	–	–	–
Multiplication	$(t; k, n_1n_2)$ from $(t; k, n_1)$ and $(t; k, n_2)$, N_1, N_2 is the number of rows in the initial arrays	Recursive	$O(k(1+N_1+N_2))$, Memory: $O(k(1+N_1+N_2))$	–	–	–
Uniform, recursive, $t = 2$	$(2; k_1n^r + rk_1n^{r-1} + k_2n^r, n)$, $r \geq 1$, n is a degree of a prime number	Recursive	$O(\sum_{i=1}^{r}((N+i(n^2-n))(k_1n^i + ik_1n^{i-1} + k_2n^i)))$, Memory: $O((N+(r-1)(n^2-n))(k_1n^{r-1} + (r-1)k_1n^{r-2} + k_2n^{r-1}) + (n+1)n^2)$	–	–	–
	$(2; n(k_1+k_2)D_{r+1,n} + k_1D_{r,n}, n)$, $r \geq 1$, n is a degree of a prime number, $D_{r,t} = \sum_{i=1}^{\lfloor (r+1)/2 \rfloor} C_{i-1}^{r-i} t^{r-i}$		$O(\sum_{i=1}^{r}((N+i(n^2-n))(n(k_1+k_2)D_{i+1,n} + k_1D_{i,n})) + \sum_{i=1}^{r-1}(n^2(n(k_1+k_2)D_{i,n} + k_1D_{i-1,n})))$, Memory: $O((N+(r-1)(n^2-n))(n(k_1+k_2)D_{r,n} + k_1D_{r-1,n}) + (n+1)n^2)$			
Ordered design	$(3; n+1, n+1)$ from $OD(3; n+1, n+1)$ and $(3; n+1, 2)$ of size N, n is a degree of a prime number	Recursive	$O((n^3-n+Nn(n+1)/2-(n^2-1))(n+1))$, Memory: $O((n^3-n+N)(n+1))$	–	–	–
Perfect hash family	$(t; k^{2p}, n)$ from $(t; k, n)$ of size N, $GCD((t-1)t/2, k) = 1$ $(3; (2v-1)^{2\hat{\ }j}, v)$, $v \equiv 0, 1 \bmod 3$, $(3; (2v-3)^{2\hat{\ }j}, v)$, $v \equiv 2 \bmod 3$, $v > 2$, $q \geq v-1$ is a degree of a prime number, j is an arbitrary integer	Recursive	$O(k^p((t-1)t/2+1) + k^{2p}((t-1)t/2+1)^pN)$, Memory: $O(k^p((t-1)t/2+1+N))$	–	–	–
	$(t; k, n)$ from $PHF(N_1, t; k, m)$ and $(t; m, n)$ of size N_2		$O(N_1N_2k)$, Memory: $O(N_1k+N_2m)$			
Theorem Roux	$(3; 2k, n)$ from $(3; k, n)$ and $(2; k, n)$ $(t; 2k, n)$ from $(t; k, n)$, ..., $(t-2; k, n)$, $t \geq 4$	Recursive	$O(2N_{t\max}k(n + N_{t\max}(t-3)))$, Memory: $O(2kN_{t\max}^2)$, $N_{t\max}$ is the number of rows in the largest initial array	–	–	–
IPO for uniform arrays	Any configurations for which there are no more efficient methods. Complement of $(t; k, n)$ to $(t; k+p, n)$, where $p > 0$	Greedy	$O(n^{2t}\log^2(n^tC_t^k)/n^{t-1}C_t^k + n^{2t}\log(n^tC_t^k)C_t^k)$, Memory: $O(n^{t+1}\log(n^tC_t^k) + n^tC_{t-1}^{k-1})$	–	+	+

Table 9. Algorithms for constructing nonuniform covering arrays

Algorithm	Application domain	Class of the algorithm	Time complexity and memory expenditures	M	E	S
Removing redundant values	Further studies are needed	Reduction	Depends on the size of the initial array	–	+	+
Double projection	$(2; n + 1 + t, (n - t)^{n+1}, s^t)$, n is a degree of a prime number, $1 \le t \le n$ and $1 \le s \le n - t$	Reduction	$O(N(k + 1))$, Memory: $O(Nk)$	–	+	–
Combining blocks	$(2; k, n_1 \dots n_k)$ and $k > \max(n_1 \dots n_k)$. Further studies are needed	Recursive	$O(n^2 + k\log_2 k)$, Memory: $O(n^2(n + 1) + k(\lceil \log_{n+1} k \rceil (n^2 - 1) + 1))$, where $n \ge d$ and $n = p^m$, p is a prime number, $k \ge 1$	–	–	–
Nonuniform, recursive, $t = 2$	$(2; l_1 + \dots + l_k, m_{1,1}, \dots, m_{1,1l}, \dots, m_{k,1}, \dots, m_{k,1k})$, from $(2; k, v_1, \dots, v_k)$ with profile $(d_1, \dots, d_k)$, and $(2; l_i, m_{i,1}, \dots, m_{i,1i})$ with profile $(f_{i,1}, \dots, f_{i,1i})$, $i \in [1, k]$ and $m_{ij} \le v_i$, $j \in [1, l_i]$	Recursive	$(N + M)L + \sum_{i=1}^{k}(M_i l_i)$, Memory: $(N + M)L + \max_{i=1}^{k}(M_i l_i)$, $M = \max_{i=1}^{k}(M_i - d_i)$, $L = \sum_{i=1}^{k} l_i$, where M_i is the number of rows in the initial arrays	–	–	–
IPO	Complement of $(t; k, n_1 \dots n_k)$ to $(t; k + p, m_1, \dots, m_{k+p})$, where $n_i \le m_i$, $i \in [1, k]$, $\exists j: j \in [1, k]$, $n_j < m_j$ and/or $p > 0$	Greedy	$O(tk^6 d^t(d^t + k + d^t k))$, Memory: $O(d^t(k\log k + tC_{t-1}^{k-1}))$	–	+	+
Initial array optimization	Row reduction in the initial array $(t; k, n_1 \dots n_k)$ of size N	Reduction, optimization	$O(N_e(M(C_t^k tN + Nk + N - 1) + 6k))$, M is the number of repeated runs for searching improvements, Memory: $O(Nk + d^t)$	–	+	–

• Use of direct algorithms in combination with recursive algorithms whenever possible, since the resulting arrays will be constructed in a small time and will occur close to the minimal ones. An example is configuration $(2; k, n)$, where n is represented as a product of powers of prime numbers $n = p_1^{w1} p_2^{w2} \dots p_r^{wr}$ such that $k \le (p_i^{wi} + 1)p_i^{wi} + 1$, where p is a prime number. Construct r arrays A_i from CA$(2; k, p_i^{wi})$ and multiply them by the recursive multiplication algorithm. If $k \le p_i^{wi} + 1$, then the minimal A_i is obtained by means of the Galois field. If $p_i^{wi} + 1 < k \le (p_i^{wi} + 1)p_i^{wi} + 1$, construct array B_i from CA$(2; \lceil k/p^w \rceil - 1, p_i^{wi})$ by means of the Galois field, and, then, construct A_i from B_i by the recursive method for uniform arrays of cardinality 2 (see Section 3.1.4).

• The recursive and greedy algorithms are better not to combine, since, most likely, the resulting array for such a configuration will be easier to obtain by other methods. In other words, the example of configuration $(2; k, n)$ is not always generalized, because, although any n can be represented as a product of powers of prime numbers $n = p_1^{w1} p_2^{w2} \dots p_r^{wr}$ for $k > (p_i^{wi} + 1)p_i^{wi} + 1$, as well as for many other $k \le (p_i^{wi} + 1)p_i^{wi} + 1$, in the general case, array A_i from CA$(t; k, p_i^{wi})$ can be constructed in less time by a greedy algorithm, which will give us a nonminimal array. The multiplication with at least one nonminimal array may yield an array that is considerably larger than the array from CA(2; k, n) constructed from the very beginning by, for example, the greedy algorithm [37].

• Completion of minimal (or almost minimal) arrays constructed by the direct methods (together with recursive ones) for configurations with the lesser number of parameters and parameter values. For example, it is efficient to take a minimal array from CA$(2; k, n)$ and complete it by means of the IPO algorithm [13] up to an array from CA$(2; k + m, 2n)$, where m is a small number of added parameters. Starting from some m, construction by means of the greedy algorithm of row-by-row addition with the use of, for example, the DDA heuristics will be more efficient [46].

• Use of the algorithm of removing redundant values from the minimal (or almost minimal) arrays constructed by the direct methods (together with recursive ones) for configurations with the greater number of parameters and parameter values. Suppose that we have a minimal array from CA$(2; k, n)$. From this

Table 10. Methods that can easily be modified for constructing variable strength covering arrayss

Algorithm	Application domain	Class of the algorithm	Time complexity and memory expenditures	M	E	S
Greedy, row-by-row	Any configurations for which there are no more efficient methods. Extension of the initial array	Greedy	$O(k(f(PR_1)+f(PR_2))d(f(VR_1)+f(VR_2)))$, $f(PR_1), f(PR_2), f(VR_1), f(VR_2)$ are functions of the heuristic operation time, Memory: $O(d^t(k\log k + tC_{t-1}^{k-1}))$	–	+	+
Hill climbing	Theoretical studies, minimal array search	Greedy, optimization problem	Exponential time and memory	–	–	–
Tabu search	Theoretical studies, minimal array search	Greedy, optimization problem	$O(N_e k N_{\max} d(k + \log T + O(dkN_{\max}(T + k^2))))$, Memory: $O(kN_{\max}T)$, T is the number of stored arrays in the tabu search implementation	–	–	+
Simulated annealing	Theoretical studies, minimal array search	Greedy, optimization problem	$O(N_e k + O(k^3 N_{\max}))$, Memory: $O(kN_{\max})$	–	–	–
Great deluge	Theoretical studies, minimal array search	Greedy, optimization problem	$O(N_e k + O(k^3 N_{\max}))$, Memory: $O(kN_{\max})$	–	–	–
Genetic algorithms	Theoretical studies, minimal array search	Genetic	$O(N_e M k^2 (N_{\max} + d^2))$, Memory: $O(MkN_{\max})$, M is the population size	–	–	–
Backtracking search	Theoretical studies, minimal array search, extension of the initial array	Backtracking search	$O(MNkf(\mathrm{CH})f(\mathrm{SH}))$, $M \geq k(N - N_{\mathrm{init}})$ is the number of arrays searched through, $f(\mathrm{CH})$, $f(\mathrm{SH})$ are functions of the heuristic operation time, Memory: $O(Nk^2(N - N_{\mathrm{init}}) + d^tC_t^k)$, N_{init} is the size of the initial array	–	+	+

array, we can efficiently obtain an array from CA(2; k, n_1 ,..., n_k), where $n_i \leq n - m$ and $m \geq 0$ is a small number, by applying the method of removing redundant values.

• If the size of the array constructed by one of the fast methods—multiple recursion or a greedy algorithm with rapidly operating heuristics—is too large, one can try to reduce it by means of the method of initial array optimization [28] or other optimization methods (for example, simulated annealing). In certain cases, it may occur advantageous to generate a large array and, then, spend some time for its reduction.

Our survey showed that none of the tools for constructing covering arrays uses in full measure all known methods for optimal construction of almost minimal arrays. As a result, in many particular cases, the well-known tools construct arrays that are much larger than the minimal ones, or do this for longer time, or require more memory compared to algorithms that use heuristics suitable for the given particular cases. The CTS package uses a combined approach; however, this solution can substantially be improved.

There is a need in the development of a new tool that would analyze the initial configuration of a covering array and choose an optimal algorithm for this configuration. To achieve the best results, this tool should combine algorithms.

4. CONCLUSIONS

In this paper, we analyzed application domains of various techniques and methods for constructing covering arrays. Advantages and disadvantages of the methods discussed have been noted, and time complexity and the required memory were estimated. We considered direct, recursive, optimization, genetic, and backtracking algorithms. We also discussed heuristics that make it possible to reduce covering arrays and their applicability domains.

Having analyzed methods for constructing covering arrays, we arrived at the conclusion that analysis of the array configuration and combination of the construction methods make it possible to extend classes of the covering array configurations for which efficient solutions exist. None of the existing tools (except for CTS [16]) analyzes the array configuration and uses advantages of known algorithms and heuristics for efficient array construction. The solution suggested in the CTS package uses only few efficient combinations of algorithms.

One of the promising trends of future development is analysis of the initial combination of the covering

array and selection of an optimal algorithm or combination of algorithms for this configuration. Principles of combining various algorithms for constructing the least covering array with the least time and memory expenditures have been formulated in the paper.

REFERENCES

1. Zelenov, S.V. and Zelenova, S.A., Automatic Generation of Positive and Negative Tests for Testing Syntactic Analysis Phase, *Trudy Instituta Sistemnogo Programmirovaniya RAN* (Proceedings of the Institute of System Programming, RAS), 2004, vol. 8, no. 1, pp. 41–58.
2. Hoffman, D., Sobotkiewicz, L., Wang Hong-Yi, Strooper, P., Bazdell, G., and Stevens, B., Test Generation with Context Free Grammars and Covering Arrays, in *Testing: Academic and Industrial Conf. - Practice and Research Techniques*, Windsor, UK, 2009, pp. 83–87.
3. Ammann, P. and Offutt, J., Using Formal Methods to Derive Test Frames in Category-Partition Testing, *Safety, Reliability, Fault Tolerance, Concurrency, and Real Time Security: Proc. of the 9-th Ann. Conf. on Computer Assurance (COMPASS'94),* 1994, pp. 69–79.
4. Cohen, D.M., Dalal, S.R., Fredman, M.L., and Patton, G.C., The AETG System: An Approach to Testing Based on Combinatorial Design, *IEEE Trans. Software Engineering*, 1996, vol. 23, no. 7, pp. 437–444.
5. Williams, A.W., Determination of Test Configurations for Pairwise Interaction Coverage, *Proc. of 13-th Int. Conf. on Testing Communicating Systems (TestCom)*, 2000, pp. 59–74.
6. Williams, W. and Probert, R.L., A Measure for Component Interaction Test Coverage, *Proc. ACS/IEEE Int. Conf. on Computer Systems and Applications*, 2001, pp. 301–311.
7. Bryce, R. and Colbourn, C.J., One-Test-at-a-Time Heuristic Search for Interaction Test Suites, *Proc. of Genetic and Evolutionary Computation Conf. (GECCO), Search-based Software Engineering track (SBSE)*, London, 2007, pp. 1082–1089.
8. Calvagna, A. and Gargantini, A., Combining Satisfiability Solving and Heuristics to Constrained Combinatorial Interaction Testing, *Proc. of the 3-rd Int. Conf. on Tests and Proofs*, Zurich, 2009, pp. 27–42.
9. Documentation on Service WM Keeper Light. http://webmoney.ru/rus/about/demo/light/index.shtml
10. Patton, G.C., DAT (Defect Analysis Team) 1986–1990 Overview, *Internal Bellcore Technical Memo*, 1991.
11. Cohen, D.M., Dalal, S.R., Parelius, J., and Patton, G.C., The Combinatorial Design Approach to Automatic Test Generation, *IEEE Software,* 1996, vol. 13, no. 5, pp. 83–87.
12. Seroussi, G. and Bshouty, N.H., Vector Sets for Exhaustive Testing of Logic Circuits, *IEEE Trans. Information Theory*, 1988, vol. 34, no. 3, pp. 513–522.
13. Lei, Y. and Tai, K.C., In-parameter Order: A Test Generation Strategy for Pairwise Testing, *Proc. of the 3rd IEEE High Assurance System Engineering Symp.,* 1998, pp. 254–161.
14. Godbole, A.P., Skipper, D.E., and Sunley, R.A., T-Covering Arrays: Upper Bounds and Poisson Approximations, *Combinatorics, Probability Computing*, 1996, vol. 5, pp.105–118.
15. Martirosyan, S. and Van Trung, T., On T-covering Arrays, *Designs, Codes and Cryptography*, 2004, vol. 32, nos. 1–3, pp. 323–339.
16. Hartman, A. and Raskin, L., Problems and Algorithms for Covering Arrays, *Discrete Math.,* 2004, vol. 284, pp. 149–156.
17. Colbourn, C.J., Combinatorial Aspects of Covering Arrays, *Le Matematiche (Catania)*, 2004, vol. 58, pp. 121–167.
18. Grindal, M., Offutt, A.J., and Andler, S.F., Combination Testing Strategies: A Survey, *Software Testing, Verification, Reliability*, 2005, vol. 15, no. 3, pp. 167–199.
19. Hartman, A., Software and Hardware Testing Using Combinatorial Covering Suites, *Proc. of Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, 2005, pp. 266–327.
20. Colbourn, C.J., Keri, G., Rivas Soriano, P.P., Schlage-Puchta, J.-C., Covering and Radius-Covering Arrays: Constructions and Classification, *Discrete Applied Math.*, 2010, vol. 158, no. 11, pp. 1158–1180.
21. Colbourn, C.J., Strength Two Covering Arrays: Existence Tables and Projection, *Discrete Math.*, 2008, vol. 308, nos. 5–6, pp. 772–786.
22. Forbes, M., Lawrence, J., Lei, Y., Kacker, R.N., and Kuhn, D.R., Refining the In-parameter-order Strategy for Constructing Covering Arrays, *J. Res. Nat. Inst. Stand. Tech.*, 2008, vol. 113, no. 5, pp. 287–297.
23. Konstantinov, A.V., Automation of Test Construction with the Use of Combinatorial Methods, *MS Thesis*, Moscow State University, 2007.
24. Cohen, M.B., Dwyer, M.B., and Shi, J., Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach, *IEEE Trans. Software Eng.*, 2008, vol. 34, no, 5, pp. 633–650.
25. Cohen, M.B., Dwyer, M.B., and Shi, J., Exploiting Constraint Solving History to Construct Interaction Test Suites, *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION,* 2007, pp. 121–132.
26. Czerwonka, J., Pairwise Testing in Real World, *24-th Pacific Northwest Software Quality Conference*, 2006.
27. Yan, J. and Zhang, J., A Backtracking Search Tool for Constructing Combinatorial Test Suites, *J. System Software*, 2008, vol. 81, no. 10, pp. 1681–1693.
28. Nayeri, P., Colbourn, C.J., and Konjevod, G., Randomized Postoptimization of Covering Arrays, *Lecture Notes in Computer Science*, Springer, 2009, vol. 5874, pp. 408–419.
29. Edelman, A., The Mathematics of the Pentium Division Bug, *SIAM Review*, 1997, vol. 39, no. 1, pp. 54–67.
30. Greene, C., Sperner Families and Partitions of a Partially Ordered Set, *Combinatorics,* Hall, Jr., M. and van Lint, J., Eds., Dordrecht, Holland, 1975, pp. 277–290.
31. Lidl, R. and Niederreiter, H., *Finite Fields*, Reading, Mass.: Addison-Wesley, 1983. Translated under the title *Konechnye polya*, Moscow: Mir, 1988.

32. Chateauneuf, M. and Kreher, D., On the State of Strength-three Covering Arrays, *J. Combinatorial Designs*, 2002, vol. 10, no. 4, pp. 217–238.
33. Colbourn, J., Martirosyan, S.S., Mullen, G.L., Shasha, D., Yucas, J.L., and Sherwood, G.B., Products of Mixed Covering Arrays of Strength Two, *J. Combinatorial Design*, 2006, vol. 14, no. 2, pp. 124–138.
34. Meagher, K. and Stevens, B., Group Construction of Covering Arrays, *J. Combinatorial Design*, 2005, vol. 13, no. 1, pp. 70–77.
35. Yin, J., Constructions of Difference Covering Arrays, *J. Combinatorial Theory (A),* 2003, vol. 104, no. 2, pp. 327–339.
36. Harari, F., *Graph Theory*, Reading, Mass.: Addison-Wesley, 1969. Translated under the title *Teoriya grafov*, Moscow: Mir, 1973.
37. Tables of the Known Least Covering Arrays. http://www.public.asu.edu/~ccolbou/src/tabby/catable.html.
38. Sloane, N., Covering Arrays and Intersecting Codes, *J. Combinatorial Designs*, 1993, vol. 1, no. 1, pp. 51–63.
39. Cohen, M.B., Colbourn, C.J., and Ling, A.C.H., Constructing Strength Three Covering Arrays with Augmented Annealing, *Discrete Math.*, 2008, vol. 308, no. 13, pp. 2709–2722.
40. Colbourn, C.J., Martirosyan, S.S., Van Trung, T., and Walker II, R.A., Roux-type Constructions for Covering Arrays of Strengths Three and Four, *Designs, Codes Cryptography*, 2006, vol. 41, no. 1, pp. 33–57.
41. Stevens, B. and Mendelsohn, E., New Recursive Methods for Transversal Covers, *J. Combinatorial Designs*, 1999, vol. 7, no. 3, pp. 185–203.
42. Stevens, B., Ling, A., and Mendelsohn, E., A Direct Construction of Transversal Covers Using Group Divisible Designs, *Ars Combin.*, 2002, vol. 63, pp. 145–159.
43. Cormen, T., Leiserson, C., Rivest, R., and Stein, K., *Introduction to Algorithms,* The MIT Press, 2001, 2nd ed. Chapter 16: Greedy Algorithms.
44. Calvaga, A. and Gargantini, A., IPO-s: Incremental Generation of Combinatorial Interaction Test Data Based on Symmetries of Covering Arrays, *Proc. of the IEEE Int. Conf. on Software Testing Verification and Validation Workshops*, Denver, Colorado, 2009.
45. Cohen, M., Dalal, S.R., Fredman, M.L., and Patton, G.C., The Automatic Efficient Test Generator (AETG) System, *Proc. of the 5-th Int. Symp. on Software Reliability Engineering,* 1994, pp. 303–309.
46. Colbourn, C.J., Cohen, M.B., and Turban, R.C., A Deterministic Density Algorithm for Pairwise Interaction Coverage, *Proc. of the IASTED Int. Conf. on Software Engineering*, 2004, pp. 242–252.
47. Tung, Y.-W. and Aldiwan, W.S., Automating Test Case Generation for the New Generation Mission Software System, *Proc. IEEE Aerospace Conf.*, 2000, pp. 431–437.
48. Jenkins, B., Tool for Pairwise Testing. http://burtleburtle.net/bob/math/jenny.html, 2005.
49. Pairwise Testing, Combinatorial Test Case Generation. http://www.pairwise.org/tools.asp.
50. Nurmela, K., Upper Bunds for Covering Arrays by Tabu Search, *Discrete Applied Math.,* 2004, vol. 138, nos. 1–2, pp. 143–152.
51. Stardom, J., Metaheuristic and the Search for Covering and Packing Arrays, *MS Thesis*, Simon Fraser University, 2001.
52. Stevens, B., Transversal Covers and Packings, *Ph. D. Thesis, Mathematics*, Univ. of Toronto, 1998.
53. Cohen, M.B., Colbourn, C.J., and Ling, A.C.H., Augmenting Simulated Annealing to Build Interaction Test Suites, *Proc. of Int. Symp. Software Requirements Engineering, (ISSRE 2003)*, 2003, pp. 394–405.
54. Dueck, G., New Optimization Heuristic – The Great Deluge Algorithm and the Record-To-Record Travel, *J. Computational Phys.*, 1993, vol. 104, pp. 86-92.
55. Dueck, G. and Scheuer, T., Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulating Annealing, *J. Computational Phys.*, 1990, vol. 90, pp. 161–175.
56. Rodriguez-Tello, E. and Torres-Jimenez, J., Memetic Algorithms for Constructing Binary Covering Arrays of Strength Three, *Proc. of Artificial Evolution 2009*, pp. 86–97.