BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **18**, No 5
Special Thematic Issue on Optimal Codes and Related Topics
Sofia • 2018 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2018-0019

About an Approach for Constructing Combinatorial Objects

Iliya Bouyukliev[1], *Maya Hristova*[2]

[1]*Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 1113 Sofia, Bulgaria*
[2]*Faculty of Mathematics and Informatics, Veliko Tarnovo University, 5003 Veliko Tarnovo, Bulgaria*
Emails: iliyab@math.bas.bg maqhristova@gmail.com

Abstract: *The classification of combinatorial objects consists of two sub-problems – construction of objects with given properties and rejection of isomorphic objects. In this paper, we consider generation of combinatorial objects that are uniquely defined by a matrix. The method that we present is implemented by backtrack search. The used approach is close to dynamic programming.*

Keywords: *Partitioning of integers, generating combinatorial objects, backstrack search.*

1. Introduction

Many discrete problems are related to the classification of combinatorial objects. This problem consists of two subproblems. The first one is the generation of objects and the other is the rejection of the isomorphic objects among the obtained ones. In this paper, we focus only on the construction task.

There are various methods and algorithms for constructing combinatorial objects. Kaski and Östergärd [7] consider the construction of many combinatorial structures in details: 2-designs (balanced incomplete block designs, BIBDs), linear codes, Hadamard matrices, etc. Brinkmann [3] presents an algorithm for generating regular directed graphs using binary matrices. A very powerful method for classification of graphs is presented in [8]. Dzhumalieva-Stoeva, Bouyukliev and Monev [5] consider the construction of self-orthogonal codes from combinatorial designs and propose an algorithm for generating designs and IMD-matrices. An algorithm for classification of binary self-dual codes is described in [2]. Bulutoglu and Margot [4] present an algorithm for generating orthogonal arrays implementing linear programming. Another algorithm for construction of orthogonal arrays is given by Schoen, Eendebak and Nguyen [9].

In regard to the complexity of the algorithms, the exhaustive generation in many cases is a computationally hard problem [7]. It requires the development of effective algorithms for constructing combinatorial objects. In this paper, we consider only

combinatorial objects that are uniquely defined by a matrix over a given alphabet S. We construct the matrix column by column.

A variant of the method we present is used for construction of linear codes, but the resulting matrix has a predetermined fixed part [1]. This approach has led to many new results, but it is not convenient for use in the construction of other combinatorial objects. Its first implementation was done by emulation of nested "for" loops.

The method we present in this paper is reduced to a specific c partition of integers. The implementation gives a solution at each step. This approach turns out to be more effective, convenient, useful and flexible.

The method that we use has three advantages: the number of isomorphic objects is reduced significantly and early in the construction process, the needed calculations are essentially reduced, the presented method can be combined with canonical augmentation type of isomorph free generation [7].

The paper is organized as follows. In Section 2 we consider some basic definitions that are related to the construction of combinatorial objects. In Section 3 we present the main idea of the algorithm and in Section 4 - the algorithm itself.

2. Basic definitions and notations

There are two main types of generation depending on the properties of the considered objects. In the first type of construction, the structure of the objects of interest is known in advance (for example, the permutations of n elements, all k-element subsets of a given set of n elements, etc.). Such types of objects can be generated quite easily by recursive methods. The second type of generation refers to objects that have an incidence structure (designs, linear codes, etc.). Usually, such combinatorial objects are constructed from a pre-built part of it, which is called a sub-object. The generation of objects of this type is done step by step. At each step, all possibilities for the sub-objects are examined. This technique is implemented by searching on a certain set of sub-objects.

We study the following problem: Construct all combinatorial objects that are defined by an $m_0 \times n_0$ matrix over a given alphabet S with s elements. For the purposes of the proposed method, we need to order the elements of the alphabet, so let $S = \{r_1, r_2, \ldots, r_s\}$, $r_1 < r_2 < \ldots < r_s$.

Definition 1. Search space Ω (related to the considered problem) is the set of all matrices with size $m_0 \times n$ over S, where n is an integer, $1 \leq n \leq n_0$. All $m_0 \times n$ matrices for $n < n_0$ define sub-objects.

If $\sim$ defines an equivalence relation in Ω then $\sim$ splits Ω into equivalence classes. We consider only equivalence relations such that a permutation of the rows of the considered matrix results in a matrix in the same equivalence class. Actually, we are interested in a subset of matrices, $\Gamma \subset \Omega$, for which the required constraints are satisfied (because they represent some specific combinatorial objects). We consider only equivalence relations for which if a matrix G belongs to Γ then all matrices equivalent G are in Γ (Γ is a union of equivalence classes). Then the classification

problem is to find exactly one representative of each equivalence class in Γ. The search process is conveniently modeled through a root tree (search tree).

Definition 2. The search tree is a rooted tree, whose nodes are matrices from Γ. Two nodes are connected with an edge if and only if the matrix, that defines one node can be obtained from the other one by expanding with a column.

Usually, the root of the search tree is the empty matrix (or a single-column matrix in some cases).

Definition 3. A level or depth of a node is the length of the path from the node to the root. This is equal to the number of the columns of the matrix.

The successors of a given node X are sub-objects that can be obtained from X through a single search step. These are all matrices that are obtained from the given matrix by expanding with a column.

Definition 4. Let the matrix $G \in \Gamma$ corresponds to the node X of the search tree. The matrices in Γ, obtained by expanding with a column to G are called direct successors of G (children of X) and are denoted with $C(X)$. The node X is called their parent and is uniquely determined by each of its children.

In fact, the search tree is defined by its search space Γ, with its root and the rule $X \to C(X)$. This rule is generally determined by the property P, which is inherited.

Since we construct the matrix (corresponding to the considered object) column by column, each new column c must satisfy the following two conditions:

(1) The number of coordinates of c that are equal to an element r from the alphabet S must be equal to a given integer w_r.

(2) The column have the property P.

If the columns are too long (the integer m_0 is too big) then we need to use some reduction methods. To clarify the terminology and the algorithms, in this paper we present examples referring to orthogonal arrays.

Let S be a set (alphabet) of s symbols (levels). We will denote these levels by $0, 1, \ldots, s-1$.

Definition 5 [6]. An $N \times k$ array A with entries from S is said to be an orthogonal array with s levels, strength t ($0 \le t \le k$) and index λ if every $N \times t$ subarray of A contains each t-tuple based on S exactly λ times as a row. We will denote such Orthogonal Array by OA(N, t, s^k).

It is easy to see, that for every i-th (new) column of the matrix G of an orthogonal array the following conditions are met:

(i) Each symbol from the alphabet must be found in the column exactly $w = N/s$ times.

(ii) For each set of $t-1$ columns $v_1, v_2, \ldots, v_{(t-1)}$ of the matrix G_{i-1}, taken together with the new i-th column, all t-tuples over S must be contained in these columns exactly λ times.

One of the options to reduce the number of non-equivalent objects is to consider only lexicographically ordered matrices. We use a construction method in which the matrix is generated column by column. As a result, a matrix with lexicographically ordered rows is obtained.

The method that we present uses the algorithmic strategy called back-track search (backtracking). The backtrack search strategy can be presented as depth-first

search of a search tree [1]. With this strategy, the object of interest is constructed sequentially. At each step, all possible objects (or sub-objects, depending on the level) are obtained. Every sub-object is used at each subsequent step to obtain the next objects. In this way, the children at a given step are found through the objects of the previous step. At each step, a set of possible extensions is generated and an attempt is made to extend the current sub-object with the elements of that set. If the current object can be extended, a step forward is made. Otherwise – step back. Depending on the algorithm's goals, when the searched objects are constructed it may stop or step back and continue traversing the search tree.

The backtrack search is a recursive extension of the current object with one step. Let G be a matrix with columns $g_1, g_2, \ldots, g_i$. The backtrack procedure defines a set of extensions of the current object, $G_{i+1} = G_{i+1}(g_1, g_2, \ldots, g_i)$. The procedure recursively invokes itself for every $g_{i+1} \in G_{i+1}$ with input $(g_1, g_2, \ldots, g_i, g_{i+1})$. If all elements of G_{k+1} are considered or the level of the searched objects k is reached, the return stage in the calling procedure is performed. The backtrack algorithmic strategy can be described in general with Algorithm 1.

Algorithm 1. Basic Backtrack Algorithm

```
1. procedure Backtrack((g1, g2, …, gi): object)
2.   if i = k then Print((g1, g2, …, gk));
3.   else
4.     find Gi+1(g1, g2, …, gi);
5.     foreach gi+1 ∈ Gi+1 do Backtrack((g1, g2, …, gi; gi+1));
```

We present a particular variant of Algorithm 1 which combines backtrack with dynamic programming. The specific approach here is the following: If the first $i - 1$ columns of the matrix G are fixed and the set of all possibilities for i-th column is T_i then we use the same set T_i to obtain the possibilities for $(i + 1)$-th column. We give the details in Section 3.

3. The construction strategy

For our purposes we use the strategy described in [1]. The essential is that each vector is not represented by its elements, but by the number of elements within an interval specified by the parent.

Definition 6. Let G be an $m \times n$ matrix with lexicographically ordered rows. We say that the n-tuple $k = (k_1, k_2, \ldots, k_n)$ defines an interval of length l with respect to G if G contains l rows equal to k.

To every lexicographically ordered matrix G we can juxtapose a vector $L(G) = (l_1, l_2, \ldots, l_t)$, whose elements correspond to the lengths of the intervals, defined by the rows of the matrix, $t = s^n$. In particular, each vector-column can be considered as a matrix, and if the elements of the vector are lexicographically arranged, they define intervals in a similar manner.

In this paper, we are interested only in the length of intervals determined by a matrix G and the corresponding vector $L(G)$.

Example 1. Consider the matrix that corresponds to orthogonal array OA(12, 2, 2^4), lexicographically ordered by rows

$$G = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}^{\mathrm{T}}.$$

The first column g_1 consists of 6 zeroes and 6 ones. Its structure defines two intervals for the second vector. Therefore, we can juxtapose the vector $L(g_1) = (6, 6)$ to g_1. The second vector g_2 has to consists of two intervals, each of two cells. If $G_2 = (g_1\ g_2)$ then $L(G_2) = (3, 3, 3, 3)$. To the matrix G_3 consisting of the first three columns, we juxtapose the vector $L(G_3) = (2, 1, 1, 2, 1, 2, 2, 1)$. The vector, corresponding to the matrix G, is $L(G) = (1, 1, 1, 0, 1, 0, 0, 2, 0, 1, 1, 1, 1, 1, 1, 0)$ (including the empty cells).

Let G be an $m \times n$ matrix with lexicographically ordered rows. We denote by G_i the matrix that consists of the first i columns of G, $i = 1, \ldots, n$. Let $L(G_i) = (l_1, l_2, \ldots, l_k)$ be the corresponding vector with the lengths of the intervals. Consider the vector $v = (v_1, v_2, \ldots, v_{l_1}, v_{l_1+1}, \ldots, v_{l_1+l_2}, \ldots, v_m)$.

We call *v possible solution* for $(i + 1)$-th column of G if:

- $v_1 \le v_2 \le \ldots \le v_{l_1}$, $v_{l_1+1} \le \ldots \le v_{l_1+l_2}$, …, $v_{m-lk+1} \le \ldots \le v_m$;
- v satisfies condition (1), given above.

If v satisfies also condition (2), we call it *solution*.

Each solution $v = (v_1, v_2, \ldots, v_m)$ can be written as a vector

$$\hat{v} = \left(v_1^{(1)}, \ldots, v_1^{(s)}, v_2^{(1)}, \ldots, v_2^{(s)}, \ldots, v_k^{(1)}, \ldots, v_k^{(s)}\right),$$

where $v_i^{(j)}$ is the number of the coordinates of v in the i-th interval which are equal to $s_j \in S$. We call $\hat{v}$ the cellular form of v.

Example 2. Let consider the third column of the matrix G, g_3 = (001011011001). Since $L(G_2) = (3, 3, 3, 3)$, the cellular form of g_3 is $\hat{g}_3 = (2, 1, 1, 2, 1, 2, 2, 1)$.

We associate with the solution v for the i-th column two types of intervals – $L_{\text{old}}(\hat{v})$ and $L_{\text{new}}(\hat{v})$. The first type, called *old intervals*, is presented by $L_{\text{old}}(\hat{v}) = L(G_{i-1})$. For the *new intervals* we have $L_{\text{new}}(\hat{v}) = \hat{v} = L(G_i)$.

Example 3. The solution v represented by $\hat{v} = (2, 1, 1, 2, 1, 2, 2, 1)$ corresponds to $L_{\text{old}}(\hat{v}) = (3, 3, 3, 3)$, and defines the new intervals: $L_{\text{new}}(\hat{v}) = (2, 1, 1, 2, 1, 2, 2, 1)$.

We would like to mention that for any two different solutions v' and v'' for the i-th column the vectors of the old intervals coincide, $L_{\text{old}}(\hat{v}') = L_{\text{old}}(\hat{v}'')$, but the vectors of the new intervals $L_{\text{new}}(\hat{v}')$ and $L_{\text{new}}(\hat{v}'')$ are different.

Take two solutions v' and v'' for the i-th column. Let $a = (a_1, a_2, \ldots, a_s)$ and $b = (b_1, b_2, \ldots, b_s)$ be these sub-vectors (parts) of $\hat{v}'$ and $\hat{v}''$ that correspond to the j-th old interval of v' and v'', respectively. We define a partition of a with respect to b which is represented by the vector $\bar{a} = \left(\bar{a}_1^{(1)}, \bar{a}_1^{(2)}, \ldots, \bar{a}_1^{(s)}, \ldots, \bar{a}_s^{(1)}, \bar{a}_s^{(2)}, \ldots, \bar{a}_s^{(s)}\right)$ that satisfies the following two conditions:

(1) $\bar{a}_i^{(1)} + \bar{a}_i^{(2)} + ... + \bar{a}_i^{(s)} = b_i, \quad i = 1,..., s;$

(2) $\bar{a}_1^{(i)} + \bar{a}_2^{(i)} + ... + \bar{a}_s^{(i)} = a_s, \quad i = 1,..., s.$

We denote this partition by a/b. This can be generalized for the vectors $\hat{v}'$ and $\hat{v}''$.

Example 4. Let $\hat{v}'$ = (1, 2, 2, 1, 2, 1, 1, 2) and $\hat{v}''$ = (2, 1, 1, 2, 1, 2, 2, 1). Then (1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1) is one possibility for v'/v''.

Computing the partition of v' with respect to v'' can be reduced to restricted integer partition. This problem in our case can be defined as follows:

Problem 1. Let A, b_1, …, b_s be nonnegative integers. Find all s-tuples $(\bar{a}_1, ..., \bar{a}_s)$ such that $\bar{a}_1 + \bar{a}_2 + \ldots + \bar{a}_s = A$ and $0 \le \bar{a}_i \le b_i$, i = 1, 2, …, s.

This problem is solved by Algorithm 2. We use this integer partition as follows:

Problem 2. Let $a = (a_1, \ldots, a_s)$ and $b = (b_1, \ldots, b_s)$ be vectors of nonnegative integers. Find all s^2-tuples $\left(\bar{a}_1^{(1)}, \bar{a}_1^{(2)}, ..., \bar{a}_1^{(s)}, \bar{a}_s^{(1)}, \bar{a}_s^{(2)}, ..., \bar{a}_s^{(s)}\right)$ such that

(1) $\bar{a}_1^{(1)} + \bar{a}_2^{(1)} + ... + \bar{a}_s^{(1)} = a_1, \quad 0 \le \bar{a}_j^{(1)} \le bh_j^{(1)} = b_j,$

(2) $\bar{a}_1^{(2)} + \bar{a}_2^{(2)} + ... + \bar{a}_s^{(2)} = a_2, \quad 0 \le \bar{a}_j^{(2)} \le bh_j^{(2)} = b_j - a_j^{(1)},$

. . .

(s) $\bar{a}_1^{(s)} + \bar{a}_2^{(s)} + ... + \bar{a}_s^{(s)} = a_s, \quad \bar{a}_s^{(j)} = bh_j^{(s)} = b_j - a_j^{(1)} - \cdots - a_j^{(s-1)},$

where j = 1, 2, …, s.

This problem is solved by Algorithm 3.

4. Algorithm

In this section, we consider two main algorithms. The first one refers to finding the solutions that corresponds to the partitioning of a with respect to the new intervals of b. The second main algorithm is how to use the first algorithm for implementation of backtracking for obtaining a complete solution.

Finding a solution depending on the intervals of another solution is a main step in our algorithm. Also, to find solutions from every level i, we use partitioning of integers. Let A and l_1, l_2, …, l_j are integers. Our goal is to find all partitions of A, such that $a_1 \le l_1$, $a_2 \le l_2$, …, $a_j \le l_j$ and $a_1 + a_2 + \ldots + a_j = A$.

The algorithm (Algorithm 2) that solves this task is reduced to complete restricted integer partition. For its implementation we use the following notation: with s we denote the number of the symbols of given alphabet, with i – the number of the current interval and with l_i we denote the size of this interval (l_i determines the restriction of the partition, the i-th element cannot be greater than l_i). With A we denote the number of a fixed element of the alphabet S and "help" shows whether A can be partitioned in the rest of the cells (this true if $A \le \text{help}_i$). The set of all solutions is denoted with SOL. At each step of the algorithm, a solution is given.

After the initialization of the auxiliary variable "help" (line 21), we invoke the procedure Partition with parameters i = 1, A and s (line 22). If the number of the fixed element of the alphabet is less than or equal to the size of the current cell, then this

element is placed in this cell only and the procedure ends (line 2). Otherwise, if A can be partitioned within an interval with the given size, we invoke recursively the procedure Partition for the next cell of the interval (line 6). If the last cell is reached, the return stage in the calling procedure is performed. In the second part of the algorithm (line 9) the problem set in the beginning is solved, but with new value for A and $s - 1$ steps.

Example 5. Let $A = 3$, $s = 4$ and $L = (1, 2, 1, 1)$. For intervals with these sizes, we have help = (4, 2, 1). The solution at first step of Algorithm 2 will be $a = (1, 2, 0, 0)$. At the next step we have $a = (1, 1, 1, 0)$. The solution at the third step of our algorithm is $a = (1, 1, 0, 1)$. As solution at Step 4, we will have $a = (1, 0, 1, 1)$. The solution at the final step is $a = (0, 1, 1, 1)$.

When we fix all elements of the alphabet, the problem is reduced to solving the following task. Let $A_1, A_2, \ldots, A_s$ are given. They correspond to the number of the elements in the given alphabet. Also $l_1, l_2, \ldots, l_j$ are given. We search for all partitions of $A_1, A_2, \ldots, A_s$ such that $A_1 = a_1^{(1)} + a_2^{(1)} + \ldots + a_j^{(1)}$, $A_i = a_1^{(i)} + a_2^{(i)} + \ldots + a_j^{(i)}$, $a_1^{(1)} + a_1^{(2)} + \ldots + a_1^{(s)} \le l_1$ and $a_i^{(1)} + a_i^{(2)} + \ldots + a_i^{(s)} \le l_i$. For the implementation of Algorithm 3 that solves this task, we use the notation as in Algorithm 2 with the difference that *level* stores the current level.

After the standard initialization of of the required variables, the procedure Partition2 is called with parameters $A = 0$, $i = 1$, s and level = 1 (line 30). If the algorithm is at such a step that the first cell of the interval is current, a takes the number of the next element of the alphabet as value (line 2). To initialize *help* and a_{level} (referring to the solution for the current level), we use the additional procedures init (line 2) and initl (lines 4, 12 and 21). If the number of the current symbol of the alphabet is less than or equal to the size of the current cell, but we have not reached the last level, we call recursively Partition for the next level (for the next symbol of the alphabet, line 5).

Algorithm 2

```
1.  procedure Partition(A, i, s)
2.      if A ≤ l_i then a_i ← A; a ∪ SOL; Print(SOL);
3.      else
4.          a_i ← l_i; A ← A – l_i;
5.          if A ≤ help_i then
6.              if i < s – 1 then Partition(A, i + 1, s);
7.              else
8.                  if A ≤ l_{i+1} then a_{i+1} ← A; a ∪ SOL; Print(SOL);
9.      while a_i > 0 do
10.         A + +; a_i – –;
11.         if A ≤ help_i then
12.             if i < s – 1 then Partition(A, i + 1, s);
13.             else
14.                 if A ≤ l_{i+1} then a_{i+1} ← A; a ∪ SOL; Print(SOL);
```

15. **procedure** Main
16. *Input: A*: the number of a fixed element from *S*, *s*: the number of the symbols of *S*
17. $i \leftarrow 1$; ▷ *i* is the index of the current interval, global variable
18. SOL $\leftarrow \emptyset$; ▷ global variable
19. $a \leftarrow 0$; ▷ the current solution, global variable
20. **for** $j \in 2, 3, \ldots, s$ **do**
21. $\text{help}_j \leftarrow \sum_{i=j}^{s} l_i$;
22. Partition(*A*, *i*, *s*);

If the number of this symbol is larger than the cell size, then a takes the corresponding values (line 7) and if we have not reached the last cell of the interval, recursively call Partition2 for the next cell (line 9). Otherwise, if we have reached the last cell of the interval and its size allows the number of the symbol to store into the cell. The variable *a* takes this number as value (line 12). Again, if we have not reached the last level, we recursively call Partition for the next (line 14).

Let consider all new intervals. With SOL_i we denote the set of all solutions for the current level *i*. Algorithm 4 finds the set $\text{SOL}_{\text{level}}$. Generally, at the first step of the algorithm, the set SOL contains only initial solution *m*, up to equivalence (line 15). This solutions is cell written.

If we have reached *level* – the level of the complete solutions in the procedure PartitionMain, we print the set of solutions $\text{SOL}_{\text{level}}$, for example, in a file (line 2). Otherwise, we find all possible solutions for the next step (line 5). Then we check each of these possible solutions if it is a solution (whether it satisfies the conditions for new column of the matrix, corresponding to the given combinatorial structure) (line 8) and recursively call PartitionMain for the next level with SOL_{i+1} (line 9).

Algorithm 3

1. **procedure** Partition2(*A*, *i*, *s*, level)
2. **if** $i = 1$ **then** $A \leftarrow A_{\text{level}}$; init(level, *s*);
3. **if** $A \le l_{\text{level},i}$ **then**
4. $a_{\text{level},i} \leftarrow A$; initl(level, *s*, *i*); $a \cup$ SOL; Print(SOL);
5. **if** level < $s - 1$ **then** Partition2(*A*, 1, *s*, level + 1);
6. **else**
7. $a_{\text{level},i} \leftarrow l_{\text{level},i}$; $A \leftarrow A - l_{\text{level},i}$;
8. **if** $A \le \text{help}_{\text{level},i}$ **then**
9. **if** $i < s - 1$ **then** Partition2(*A*, $i + 1$, *s*, level));
10. **else**
11. **if** $A \le l_{\text{level},i+1}$ **then**
12. $a_{\text{level},i+1} \leftarrow A$; initl(level, *s*, *i*);
13. $a \cup$ SOL; Print(SOL);
14. **if** level < $s - 1$ **then** Partition2(*A*, 1, *s*, level + 1);
15. **while** $a_{\text{level};i} > 0$ **do**
16. $A + +$; $a_{\text{level},i} - -$;

```
17.       if A ≤ help_level,i then
18.          if i < s – 1 then Partition2(A, i + 1, s, level);
19.          else
20.             if A ≤ l_level,i+1 then
21.                a_level,i+1 ← A; initl(level, s, i);
22.                a ∪ SOL; Print(SOL);
23.                if level < s – 1 then Partition2(a, 1, s, level + 1);
```

```
24.    procedure Main
25.       Input: A1, A2, …, As, s: the number of the symbols of S
26.       i ← 1;          ▷ i is the index of the current interval, global variable
27.       SOL ← 0;                                              ▷ global variable
28.       A ← 0                                                 ▷ global variable
29.       level ← 1              ▷ level is the current level, global variable
30.       Partition2(a, i, s, level);
```

Algorithm 4

```
1.    procedure PartitionMain(SOL_i, i)
2.       if i = level then Print(SOL_level);
3.       else
4.          foreach a, b ∈ SOL_i do
5.             find a/b (POS_SOL_i+1 – all possible solutions for i + 1);
6.             SOL_i+1 = 0;
7.             foreach c ∈ POS_SOL_i+1 do
8.                if c is solution then c ∪ SOL_i+1;
9.             PartitionMain(SOL_i+1, i + 1);
```

```
10.    procedure Main
11.       Input: level: the current level
12.       i ← 1;                                                ▷ global variable
13.       m;    ▷ global variable
14.       SOL_i = 0;                                            ▷ global variable
15.       m ∪ SOL_i;
16.       PartitionMain(SOL_i, i);
```

Acknowledgements: This research is supported by Bulgarian Science Fund under Contract DN-02-2/13.12.2016.

References

1. B o u y u k l i e v, I. "Q-EXTENSION" – Strategy in Algorithms. – In: Proc. of International Workshop ACCT, Bansko, Bulgaria, 2000, pp. 84-89.
2. B o u y u k l i e v a, S., I. B o u y u k l i e v. An Algorithm for Classifcation of Binary Self-Dual Codes. – IEEE Trans. Inform. Theory, Vol. **58**, 2012, pp. 3933-3940.
3. B r i n k m a n n, G. Generating Regular Directed Graphs. – Discrete Mathematics, Vol. **313**, 2013, pp. 1-7.
4. B u l u t o g l u, D. A., F. M a r g o t. Classification of Orthogonal Arrays by Integer Programming. – Journal of Statistical Planning and Inference, Vol. **138**, 2008, pp. 654-666.
5. D z h u m a l i e v a-S t o e v a, M., I. B o u y u l k i e v, V. M o n e v. Construction of Self-Orthogonal Codes from Combinatorial Designs. – Problems of Information Transmissions, Vol. **48**, 2012, No 3, pp. 250-258.
6. H e d a y a t, A. S., N. J. A. S l o a n e, J. S t u f k e n. Orthogonal Arrays: Theory and Applications. Springer, 1999.
7. K a s k i, P., P. R. J. Ö s t e r g a r d. Classification Algorithms for Codes and Designs. – Berlin, Heidelberg, Springer-Verlag, 2006.
8. M c K a y, B. D. Isomorph-Free Exhaustive Generation. – J. Algorithms, Vol. **26**, 1998, pp. 306-324.
9. S c h o e n, E. D., P. T. E e n d e b a k, M. V. M. N g u y e n. Complete Enumeration of Pure-Level and Mixed-Level Orthogonal Arrays. – Journal of Combinatorial Designs, Vol. **18**, 2009, pp. 123-140.

Received 30.09.2017; Second Version 08.12.2017; Accepted 21.12.2017