Evolutionary Search of Binary Orthogonal Arrays

Luca Mariot[1(✉)], Stjepan Picek[2], Domagoj Jakobovic[3], and Alberto Leporati[1]

[1] DISCo, Università degli Studi di Milano-Bicocca,
Viale Sarca 336/14, 20126 Milano, Italy
{luca.mariot,alberto.leporati}@unimib.it
[2] Cyber Security Research Group, Delft University of Technology,
Mekelweg 2, Delft, The Netherlands
S.Picek@tudelft.nl
[3] Faculty of Electrical Engineering and Computing,
University of Zagreb, Unska 3, Zagreb, Croatia
domagoj.jakobovic@fer.hr

Abstract. Orthogonal Arrays (OA) represent an interesting breed of combinatorial designs that finds applications in several domains such as statistics, coding theory, and cryptography. In this work, we address the problem of constructing binary OA through evolutionary algorithms, an approach which received little attention in the combinatorial designs literature. We focus on the representation of a feasible solution, which we encode as a set of Boolean functions whose truth tables are used as the columns of a binary matrix, and on the design of an appropriate fitness function and variation operators for this problem. We finally present experimental results obtained with genetic algorithms (GA) and genetic programming (GP) on optimizing such fitness function, and compare the performances of these two metaheuristics with respect to the size of the considered problem instances. The experimental results show that GP outperforms GA at handling this type of problem, as it converges to an optimal solution in all considered problem instances but one.

Keywords: Orthogonal arrays · Genetic algorithms
Genetic programming · Boolean functions

1 Introduction

The field of *combinatorial designs* provides an interesting source of problems for heuristic optimization techniques. Depending on the size of the support set and the particular nature of the balancedness constraints, the two main research questions addressed in combinatorial design theory are the following:

1. *Existence:* Does a design with a particular set of parameters (i.e., support set, balancedness constraints) exist?

A. Auger et al. (Eds.): PPSN 2018, LNCS 11101, pp. 121–133, 2018.
https://doi.org/10.1007/978-3-319-99253-2_10

2. *Construction:* Once the existence question for a specified kind of design is positively answered, is there an efficient method to generate its instances?

Since the existence question of a design can always be cast as a combinatorial optimization problem [2], it follows that the use of heuristic techniques can contribute to the above research questions in a twofold way: first, by providing concrete examples of designs with specific parameters, hence answering the existence question in positive; second, once the existence question has been settled, by providing another method for efficiently constructing designs.

Despite this, the amount of literature devoted to the use of heuristic optimization techniques for constructing combinatorial designs is rather limited (see Chapter 6 of [2] for a concise survey). This is especially true for the case of *orthogonal arrays* (OA), which represent one of the most interesting breeds of combinatorial designs, due to their numerous applications in other research domains such as the *design of experiments*, *error-correcting codes* and *cryptography* [11]. Indeed, one can find only the papers by Safadi et al. [9] and Wang et al. [12] that deal with the construction of *mixed-level orthogonal arrays* (MOA), respectively through *genetic algorithms* (GA) and *simulated annealing* (SA). Nonetheless, MOA represent a very specific kind of OA, and to the best of our knowledge there are no works in the literature addressing the heuristic design of classic OA through evolutionary algorithms.

The aim of this paper is to begin filling this gap by considering the construction of orthogonal arrays through *evolutionary algorithms* (EAs), in particular *genetic algorithms* and *genetic programming* (GP). Beside its potential impact in other domains mentioned above, this research is also interesting from the evolutionary computing point of view. As a matter of fact, evolving OA through evolutionary heuristics requires to define suitable encodings and variation operators, which could find applications also in other optimization problems. Additionally, depending on the difficulty of converging to an optimal solution, designing OA could also represent an interesting benchmark problem for new evolutionary algorithms and optimization heuristics, as well as for more established ones.

Since the present work is the first one in this line of research, we consider in particular the modeling aspects of the optimization problem, focusing on the encodings for the feasible solutions and the design of variation operators to evolve them. For this reason, we begin by tackling the case of *binary* orthogonal arrays, since this allows us to represent the candidate solutions of our problem as sets of *Boolean functions*. More specifically, we take the *truth tables* of such Boolean functions as the columns of a binary matrix, which actually corresponds to the phenotype of a candidate solution. On the other hand, the genotype is either a set of binary strings for GA or a set of Boolean trees for GP.

In order to evaluate the candidate solutions evolved by GA and GP, we design a fitness function based on the *Minkowski distance* that measures the deviation of a binary matrix from being an orthogonal array having specified parameters, with the goal of minimizing it. In the case of GA, we also exploit a basic property of orthogonal arrays to design ad-hoc crossover and mutation operators, which ensure that the Boolean functions composing an individual are balanced, thus

reducing the resulting search space. For GP, we incorporate this property as an additional penalty factor in the fitness function, since there is no straightforward way to design GP variation operators that enforce the balancedness constraint at the tree level.

We compute the size of the search spaces respectively explored by GA and GP in terms of the number of variables of the Boolean functions and the columns of the binary matrices involved, showing that the resulting search spaces cannot be exhaustively enumerated already for Boolean functions of $n = 4$ variables and $k = 8$ columns.

The experimental results show that GP largely outperforms GA at evolving binary OA, even though the latter actually explores a smaller search space. As a matter of fact, GA is able to find orthogonal arrays defined by up to 8 Boolean functions of 4 variables, while GP arrives one step further by obtaining also orthogonal arrays composed of 16 functions of 5 variables. This performance difference is analogous to the findings reported in [5], where the authors observed that GP outperforms GA in the generation of cellular automata defining *orthogonal Latin squares*, which are a type of combinatorial designs closely connected with orthogonal arrays. Consequently, the present work brings additional empirical evidence that GP is a better metaheuristic at handling optimization problems related to combinatorial designs.

2 Basic Definitions

We begin by giving the basic definition of orthogonal arrays, following the notation used by Hedayat et al. [3]:

Definition 1. *Let S be a finite set of s symbols (called the* support set*) and let $N, k, t, \lambda \in \mathbb{N}$ with $0 \leq t \leq k$. An $N \times k$ matrix A with entries from S is an* orthogonal array *with s levels, k columns, strength t, and index λ (for short, an $OA(N, k, s, t)$) if in each submatrix of N rows and t columns each t-uple over S occurs exactly λ times.*

Clearly, if A is an $OA(N, k, s, t)$, then it follows that $\lambda = N/s^t$.

This is the reason why the parameter λ is usually omitted from the specification of an OA.

A basic property of orthogonal arrays of strength t is that they satisfy the balancedness constraint also for smaller strengths, as shown in [3]:

Theorem 1. *Let A be an $OA(N, k, s, t)$ with $\lambda = N/s^t$. Then, A is also an $OA(N, k, s, t - i)$ with $\lambda = N/s^{t-i}$ for all $1 \leq i < t$.*

An OA without repeated rows is called *simple*. If $S = \{0, 1\}$ (i.e., the symbol set is the Boolean alphabet), then the OA is called *binary*.

Simple binary OA have an important application in defining the support of *correlation-immune Boolean functions*, which play an important role in the design of countermeasures for *side-channel attacks* [1].

Finally, we give a basic definition of Boolean functions and their truth tables:

Definition 2. *A* Boolean function *of n variables is a mapping $f : \mathbb{F}_2^n \to \mathbb{F}_2$. Assuming that the vector of $\mathbb{F}_2^n$ are lexicographically ordered, the* truth table *associated to f is the 2^n-bit vector $\Omega(f)$ defined as follows:*

$$\Omega(f) = (f(0,0,\cdots,0), f(0,0,\cdots,1), \cdots f(1,1,\cdots,1)). \tag{1}$$

In particular, a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is called *balanced* if the number of zeros in its truth table (and thus also the number of ones) equals 2^{n-1}.

We can now formulate the combinatorial optimization problem which we will investigate in the rest of this work. We represent the columns of binary orthogonal arrays with the truth tables of a set of Boolean functions. This can be formally stated as follows:

Problem 1. Let $n, k, t \in \mathbb{N}$. Find k Boolean functions of n variables $f_1, \cdots f_k : \mathbb{F}_2^n \to \mathbb{F}_2$ such that the matrix

$$A = [\Omega(f_1)^\top, \Omega(f_2)^\top, \cdots, \Omega(f_k)^\top] \tag{2}$$

is an $OA(2^n, k, 2, t)$, with $\lambda = 2^{n-t}$.

In other words, solving Problem 1 requires finding a set of k Boolean functions of n variables whose truth tables, when put one next to the other, form the columns of an orthogonal array with $N = 2^n$ rows, k columns, 2 levels, and strength t.

3 Specification of GA and GP

3.1 Solutions Encoding

Since Problem 1 requires finding a set of k Boolean functions whose truth tables form an $OA(2^n, k, 2, t)$, the encoding of the feasible solutions can be reduced to an appropriate representation of sets of Boolean functions which can be easily handled by evolutionary algorithms. Depending on the underlying heuristic (GA or GP), we adopted the following approaches:

1. *GA encoding*: The chromosome c of an individual is defined as follows:

   $$c = (b_1, \cdots, b_k),$$

   where, for all $i \in \{1, \cdots, k\}$, $b_i \in \mathbb{F}_2^{2^n}$ is a bitstring of length 2^n that represents the truth table of the i-th Boolean function $f_i : \mathbb{F}_2^n \to \mathbb{F}_2$ composing a feasible solution. The GA crossover and mutation operators are applied component-wise on each bitstring b_i.
2. *GP encoding*: The chromosome c in this case is defined as:

   $$c = (T_1, \cdots T_k),$$

   where, for all $i \in \{1, \cdots, k\}$, T_i is a *Boolean tree* which encodes a Boolean function of n variables, using a given set of Boolean operators. In particular,

the 2^n-bit string representing the i-th column of the array is determined by evaluating T_i for all possible input combinations on the leaf nodes, and taking the corresponding outputs of the function as the values computed at the root node. Similar to the GA encoding case, the GP variation operators are applied component-wise for each tree in the chromosome of an individual (or in a pair of individuals, in the case of tree crossover).

3.2 Fitness Function

Once a suitable chromosome encoding has been designed, one needs to define a *fitness function* to determine how good the candidate solutions produced by an evolutionary algorithm are with respect to the optimal ones. In our case, an optimal solution is defined as a set of k Boolean functions whose truth tables form the columns of a binary orthogonal array. Hence, a preliminary idea could be to determine, for each possible subset of t columns of a candidate solution, how many t-uples are repeated more than λ times, and then minimize this deviation over all possible subsets of t columns.

Let us formalize the discussion above. Given a set of k Boolean functions $f_1, \cdots, f_k : \mathbb{F}_2^n \to \mathbb{F}_2$, let A be the $2^n \times k$ matrix formed by placing side by side the transpose of the truth tables $\Omega(f_1), \cdots, \Omega(f_k) \in \mathbb{F}_2^{2^n}$. Additionally, let $I = \{i_1, \cdots, i_t\}$ be a subset of t indices, with $1 \leq i_j \leq k$ for all $j \in \{1, \cdots, t\}$, and let A_I denote the $2^n \times t$ submatrix obtained by considering only the columns of A specified by the indices of I. For all binary t-uples $x \in \mathbb{F}_2^t$, let $A_I[x]$ denote the number of occurrences of x in A_I, and define the λ-*deviation* of x as:

$$\delta(A_I, x) = |\lambda - A_I[x]| \,. \tag{3}$$

Then, given $p \in \mathbb{N}$, we define the p-*deviation* of A_I as:

$$\Delta(A_I)_p = \left(\sum_{x \in \mathbb{F}_2^t} \delta(A_I, x)^p \right)^{\frac{1}{p}} \,. \tag{4}$$

In particular, one may notice that Eq. (4) corresponds to the *Minkowski distance* (or L^p distance) between the vector $\Lambda = (\lambda, \cdots, \lambda)$ and the vector $(A_I[(0, \cdots, 0)], \cdots A_I[(1, \cdots, 1)])$.

We can now define the fitness function for our optimization problem, which amounts to the sum of the deviations of all possible $N \times t$ submatrices of A:

$$fit_p(A) = \sum_{I \subseteq [k] : |I| = t} \Delta(A_I)_p. \tag{5}$$

Clearly, if A is an orthogonal array with the required parameters, then $fit_p(A) = 0$. As a consequence, the optimization objective is to *minimize* fit_p.

3.3 Variation Operators

Recall from Theorem 1 that any OA of strength t is also an OA for all strengths $i < t$. Considering the extreme case where $i = 1$, this means that for each column of the array we must see every symbol of the support set equally often. Since in our problem we are considering binary OA where the number of rows equals $N = 2^n$, it follows that each column of an optimal solution must be composed of 2^{n-1} zeros and 2^{n-1} ones or, equivalently, that the corresponding Boolean function of n variables must be balanced.

We can exploit this fact to reduce the size of the search space of feasible solutions explored by our GA. In fact, since we are interested only in sets of k balanced Boolean functions, we can adopt variation operators that preserve their balancedness. To this end, we employ a slightly modified version of the crossover operator originally proposed by Millan et al. [6]. In particular, let p_1 and p_2 be two balanced bitstrings. Then, we generate a balanced offspring chromosome c using the following procedure:

Balanced-Crossover(p_1, p_2))
Initialization: Set two counters cnt_0 and cnt_1 to zero.
Loop: Until all positions in the offspring chromosome c have been filled:

1. Sample a random position $i \in \{1, \cdots, 2^n\}$ (without replacement)
2. If one of the two counters is equal to 2^{n-1}, then set $c[i]$ to the opposite value (i.e., 1 if $cnt_0 = 2^{n-1}$ or 0 if $cnt_1 = 2^{n-1}$)
3. Otherwise, randomly choose between $p_1[i]$ and $p_2[i]$ and copy the corresponding value in $c[i]$, increasing the relevant counter.

Output: Return c

As one can observe, our crossover operator uses two counters to keep track of the number of zeros and ones in the child chromosome during its generation. Until these two counters are less than half of the chromosome length, a random position is sampled and the gene to be copied is randomly selected from one of the two parents. Then, when one of the two counters reaches the 2^{n-1} threshold, all remaining positions in the child are filled with the opposite value. This ensures that the child chromosome is also balanced.

Regarding the mutation operator, we opted for a simple swap-based operator. More precisely, each column composing an individual is mutated with a small probability by swapping two bits in it, so that the balancedness of the corresponding Boolean function is preserved. In particular, the swap is performed between two random positions holding different values, in order to produce a mutated individual which differs from the original one.

On the contrary, for GP there is no straightforward way to design crossover and mutation operators which ensure that the resulting trees map to the balanced Boolean functions. Hence, in this case we chose to employ classic GP variation operators, specifically simple tree crossover, uniform crossover, size fair, one-point, and context preserving crossover [8] (selected at random) and subtree mutation. Additionally, we considered the balancedness constraint at the fitness

function level, using a penalty factor. In particular, let $\delta_{0,1}(i) = |\#0 - \#1|$ be the absolute value of the difference between the number of ones and the number of zeros in the i-th column of a binary array A. Then, the new fitness function minimized by GP equals:

$$fit_p(A) = \sum_{I \subseteq [k]: |I| = t} \Delta(A_I)_p + \sum_{i=1}^{k} \delta_{0,1}(i). \tag{6}$$

4 Analysis of the Search Space

We now give some basic combinatorial remarks that allow us to compute the sizes of the solution spaces. By taking into account the bare statement of Problem 1, one can see that the number of feasible solutions depends only on the number of columns k composing the array and on the number of variables n of the Boolean functions whose truth tables represent those columns. The number of Boolean functions of n variables is 2^{2^n}, since it equals the number of bitstrings of length 2^n, which are in one-to-one correspondence with the truth tables of such functions. Hence, it follows that the number of ways one can choose a set of k Boolean functions of n variables is given by

$$\mathcal{F}_{n,k} = \binom{2^{2^n}}{k}, \tag{7}$$

which corresponds to the size of the search space $\mathcal{F}_{n,k}$ induced by Problem 1. Indeed $\mathcal{F}_{n,k}$ is actually a *subset* of the search space explored by our GP algorithm. This is due to the fact that different Boolean trees evolved by GP can be semantically equivalent (i.e. evaluate to the same truth table, such as x and $NOT(NOT(x))$).

On the contrary, the search space explored by our GA coincides the set of binary $2^n \times k$ matrices whose columns are balanced, or equivalently to the space of all subsets of k balanced Boolean functions of n variables. The number of balanced Boolean functions of n variables is

$$\mathcal{BAL}_n = \binom{2^n}{2^{n-1}}, \tag{8}$$

since it is equal to the number of bitstrings of length 2^n that include 2^{n-1} ones. Thus, the number of combinations of k balanced n-variable Boolean functions is

$$\mathcal{G}_{n,k} = \binom{\mathcal{BAL}_n}{k} = \binom{\binom{2^n}{2^{n-1}}}{k}, \tag{9}$$

which gives the size of the search space explored by GA.

A natural question that arises is up to which values of the parameters n and k the two sets $\mathcal{F}_{n,k}$ and $\mathcal{G}_{n,k}$ are amenable to exhaustive search. Table 1 reports the corresponding sizes for increasing values of n, along with the dimensions of the spaces of all Boolean functions and balanced functions of n variables.

Table 1. Search space sizes with respect to n and k.

n	N	k	$\mathcal{B}_n$	$\mathcal{BAL}_n$	$\mathcal{F}_{n,k}$	$\mathcal{G}_{n,k}$
2	4	2	16	6	120	15
3	8	4	256	70	$1.7 \cdot 10^8$	916 895
4	16	8	65 536	12 870	$8.4 \cdot 10^{33}$	$1.8 \cdot 10^{28}$
5	32	16	$4.2 \cdot 10^9$	$6.0 \cdot 10^8$	$6.4 \cdot 10^{140}$	$1.3 \cdot 10^{127}$

From Table 1, one can see that the sizes of the two search spaces grow very quickly with respect to the number of variables of the Boolean functions involved, and that exhaustive enumeration is already unfeasible for $n \geq 4$ variables.

5 Experiments

5.1 Problem Instances

Table 2 reports the problem instances on which we run our GA and GP heuristics. In particular, each row of the table reports the number of variables n of the involved Boolean functions, the number of rows $N = 2^n$ of the OA, the number of columns k, the strength t, and the index λ. In the rest of this section, we refer to a problem instance by (N, k, t, λ). We selected these instances from the orthogonal array library published by Sloane [10]. We chose these particular parameters combinations since they contain both instances that can be exhaustively enumerated (those with $n = 3$, which we used for tuning our algorithms) and the smallest instances that are not amenable to exhaustive search.

Table 2. OA parameters/problem instances

	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
n	3	3	3	3	3	4	4	5	5	6
N	8	8	8	8	16	16	16	32	32	64
k	4	4	5	7	8	8	15	16	31	32
t	2	3	2	2	2	3	2	3	2	3
λ	2	1	2	2	4	2	4	4	8	8

5.2 Evolutionary Algorithms Parameters

As mentioned in Sect. 3.1, the GP encoding uses elementary Boolean operators to build one or more trees, each representing an independent Boolean function, whereas the corresponding Boolean variables are used as terminals. The function set in our experiments comprise the binary operators AND, OR, XOR, $XNOR$,

and the unary operator NOT. Additionally, we include the function IF, which takes three arguments and returns the second one if the first one evaluates to true, and the third one otherwise. The maximum tree depth is varied depending on the number of Boolean variables, which determines the number of rows of the target orthogonal array.

Regarding the population size, we set it to 500 individuals for GP and 50 for GA. The reason for this difference is that after performing some preliminary experiments, we observed that using larger population size in GA did not improve its performance. For the selection process, we employed a steady-state selection with a 3-tournament operator for both GA and GP, that in each iteration randomly selects three individuals for the tournament and eliminates the worst one. A new individual is created immediately by crossing over the remaining two from the tournament, which then undergoes mutation respectively with probability 0.5 in GP and 0.2 in GA.

Concerning the fitness function, after some preliminary tuning tests we observed that using the Minkowski distance with $p = 2$ yielded the best results, hence we adopted fit_2 for all subsequent experiments. Likewise, we set the termination condition for both GA and GP to 500 000 fitness evaluations after observing from a preliminary round of experiments that optimal solutions are mostly found before reaching this number of evaluations. Finally, each experiment is repeated 30 times.

5.3 Results

Table 3 presents the results for genetic algorithms and genetic programming in the form of success rate (in percentages) of finding an optimal solution, i.e., an orthogonal array with given properties. We denote by GP_d a GP experiment where the maximum tree depth is d. It can be observed that GP outperforms by far GA at converging to an optimal solution. As a matter of fact, GA is able to generate OA only up to 16 rows and 8 columns, with the $(16, 8, 3, 2)$ problem instance having a very low success rate. On the contrary, GP was able to find an optimal solution at least once in all instances but one (the last row with orthogonal array of 64 rows and 32 columns). Similar to GA, one can see greatly differing success rates depending on the size of the problem instance. We varied the maximum tree depth parameter to determine the conditions under which GP is able to produce an optimal solution. It can be seen that having the maximum tree depth equal to the number of variables n is enough to obtain an orthogonal array. Reasonably, the problem becomes much harder to solve also for GP when the number of variables and the number of trees (i.e., array columns) grow.

Table 4 shows the basic statistical indicators for the fitness of the best individuals found by GA and GP for every considered problem instance, as well as the average time needed to either obtain an optimal solution, or terminate the run after 500 000 evaluations. In the GA case, we did not experiment with the $(64, 32, 3, 8)$ combination, since as remarked above GA could not even converge on the smaller instances with 32 rows. These results are based on GP experiments with the largest maximum tree depth in every configuration.

Table 3. GP and GA success rates for different problem sizes. Success rates are rounded to the nearest integer.

Exp.	Heuristic				
	GA	GP_2	GP_3	GP_4	GP_5
(8, 4, 2, 2)	100	100	100	-	-
(8, 4, 3, 1)	100	100	100	-	-
(8, 5, 2, 2)	100	100	100	-	-
(8, 7, 2, 2)	87	0	100	-	-
(16, 8, 2, 4)	27	100	100	100	-
(16, 8, 3, 2)	3	0	100	97	-
(16, 15, 2, 4)	0	0	90	93	-
(32, 16, 3, 4)	0	-	6	10	-
(32, 31, 2, 8)	0	-	0	2	-
(64, 32, 3, 8)	-	-	0	0	0

Table 4. Statistical indicators for GA and GP (largest max tree depth for GP).

Exp.	GA					GP				
	min	avg	std	max	time (s)	min	avg	std	max	time (s)
(8, 4, 2, 2)	0	0	0	0	<1	0	0	0	0	<1
(8, 4, 3, 1)	0	0	0	0	<1	0	0	0	0	<1
(8, 5, 2, 2)	0	0	0	0	<1	0	0	0	0	<1
(8, 7, 2, 2)	0	0.533	1.38	4	7	0	0	0	0	1
(16, 8, 2, 4)	0	2.333	1.75	6	38	0	0	0	0	1
(16, 8, 3, 2)	0	39.96	10.9	57.41	110	0	0.565	3.09	16.97	13
(16, 15, 2, 4)	52	65.4	6.41	80	147	0	0.533	2.03	8	48
(32, 16, 3, 4)	1 174	1 266	43.4	1 349	1 995	0	83.72	41.5	135.8	1 212
(32, 31, 2, 8)	654	684	14.5	714	1 125	0	32	13.9	64	692
(64, 32, 3, 8)	-	-	-	-	-	18 812	19 159	116	19 355	15 308

The data are consistent with those reported in Table 3, indicating that GP has far better performances than GA on all problem instances, under the same number of fitness evaluations.

6 Conclusions and Perspectives

In this paper, we considered how evolutionary algorithms can be used to evolve binary orthogonal arrays. To that end, we formulated the combinatorial optimization problem as the search of a set of k Boolean functions of n variables

whose truth tables must be the columns of a binary OA with specified parameters. We chose GA and GP as heuristics to solve this problem, each working on a specific solution encoding for the candidate solutions of the problem.

The results show genetic programming greatly outperforming genetic algorithms. Interestingly, for all instances but one (the largest), GP is able to find at least one successful solution. On the other hand, GA managed to solve at least once only 6 instances out of the 10 considered, with lower success rates than GP. This experimental finding is interesting, as it contrasts with the fact that GA actually explored a smaller search space than that of GP, since the former evolved only sets of balanced Boolean functions. This observation is somewhat analogous to what has been reported by Mariot et al. in [5], where GP also outperformed GA at evolving orthogonal Latin squares based on cellular automata, even though also in that case GA was exploring the smaller search space of *pairwise-balanced* Boolean functions. Considering also our findings, this seems to indicate that GP is a better optimization heuristic at handling problems related to combinatorial designs. Moreover, since there is no other work in the literature concerning the heuristic construction of binary OA, the results that we obtained in our experiments could represent a first baseline of comparison for future research in this domain, for example by investigating the performances of other evolutionary optimization methods like discrete PSO [4] or Cartesian GP [7], which already proved useful to evolve balanced Boolean functions.

More generally, an interesting question pertains the comparison between our evolutionary approach and other non-heuristic methods to construct OA already known in the relevant literature. In particular, one can observe that most of the existing constructions of OA are based on *algebraic methods*, which usually leverages on finite fields and coding theory (see for example [3]). However, it is necessary to remark that a straightforward comparison between our heuristics and these algebraic methods is not possible, due to the great differences that the two approaches adopt to generate OA. Indeed, our heuristic approach casts the problem in terms of optimization: starting from a population of candidate solutions which most likely do not contain an optimal solution, evolve them until an OA is obtained. On the contrary, algebraic methods usually work by constructing new OA starting from previously existing ones, which have already been obtained through other constructions and/or exhaustive search. This the case, for example, of the *juxtaposition construction* or the *X4 construction* surveyed in [3]. As a consequence, barely looking at the size of the OA produced by our GA and GP would bring to the conclusion that our approach is no match for the more established algebraic constructions, since the latter manage to create significantly bigger OA (as one can see for example in Sloane [10]. However, an important aspect to remark is that most of the known algebraic constructions arise from *linear error-correcting codes*, thus yielding *linear orthogonal arrays*. This basically means that each row in the OA is a linear combination of the remaining rows. Hence, these methods do not provide for a great diversity, and the OA produced by them are actually a subset of all possible OA. On the contrary, our optimization approach does not assume any linearity constraint on the optimal solutions, hence it can generate a wider variety of OA.

Considering the above remarks, it appears natural that the main direction for future research is to improve the performances of GA and GP over larger problem instances, in order to obtain binary OA of higher dimensions. To this end, one could adopt a different fitness function that yields a smoother fitness landscape over the search space of candidate solutions. A possible idea to accomplish this would be to define fitness functions based on Theorem 3.30 in [3], which shows that a binary matrix is a binary OA of strength t if and only if its *Walsh-Hadamard transform* vanishes for all subsets of rows having at most $i \leq t$ nonzero entries.

A second direction would be to start from an OA with a certain number k of columns, and then incrementally add columns which still satisfy the balancedness constraints with the previous ones, thus yielding a larger OA. In this case, GA and GP would work on a single Boolean function at a time, possibly making convergence easier on larger problem instances.

Acknowledgments. This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-4882.

References

1. Carlet, C., Guilley, S.: Correlation-immune boolean functions for easing counter measures to side-channel attacks. Algebraic Curves Finite Fields: Cryptograph. Other Appl. **16**, 41–70 (2014)
2. Colbourn, C.J., Dinitz, J.H.: Handbook of Combinatorial Designs. CRC Press, Boca Raton (2006)
3. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: Orthogonal Arrays: Theory and Applications. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4612-1478-6
4. Mariot, L., Leporati, A.: Heuristic search by particle swarm optimization of boolean functions for cryptographic applications. In: Genetic and Evolutionary Computation Conference, Companion Material Proceedings , GECCO 2015, Madrid, Spain, 11–15 July 2015, pp. 1425–1426 (2015)
5. Mariot, L., Picek, S., Jakobovic, D., Leporati, A.: Evolutionary algorithms for the design of orthogonal latin squares based on cellular automata. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, 15–19 July 2017, pp. 306–313 (2017)
6. Millan, W., Clark, A., Dawson, E.: Heuristic design of cryptographically strong balanced boolean functions. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 489–499. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054148
7. Picek, S., Jakobovic, D., Miller, J.F., Batina, L., Cupic, M.: Cryptographic boolean functions: one output, many design criteria. Appl. Soft Comput. **40**, 635–653 (2016)
8. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming (2008). http://lulu.com and freely available at http://www.gp-field-guide.org.uk. (With contributions by J.R. Koza)
9. Safadi, R., Wang, R.: The use of genetic algorithms in the construction of mixed multilevel orthogonal arrays. Technical report, Olin Corp Cheshire CT Olin Research Center (1992)

10. Sloane, N.J.: A library of orthogonal arrays. Fixed-level arrays with more than three levels: OA 16(4.2) (2007)
11. Stinson, D.R.: Combinatorial Designs: Constructions and Analysis. Springer, Heidelberg (2007). https://doi.org/10.1007/b97564
12. Wang, R., Safadi, R.: Generating mixed multilevel orthogonal arrays by simulated annealing. In: Page, C., LePage, R. (eds.) Computing Science and Statistics, pp. 557–560. Springer, New York (1992). https://doi.org/10.1007/978-1-4612-2856-1_100