Hyper-bent Boolean Functions and Evolutionary Algorithms

Luca Mariot[1], Domagoj Jakobovic[2], Alberto Leporati[1], and Stjepan Picek[2]

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

Abstract. Bent Boolean functions play an important role in the design of secure symmetric ciphers, since they achieve the maximum distance from affine functions allowed by Parseval's relation. Hyper-bent functions, in turn, are those bent functions which additionally reach maximum distance from all bijective monomial functions, and provide further security towards approximation attacks. Being characterized by a stricter definition, hyper-bent functions are rarer than bent functions, and much more difficult to construct. In this paper, we employ several evolutionary algorithms in order to evolve hyper-bent Boolean functions of various sizes. Our results show that hyper-bent functions are extremely difficult to evolve, since we manage to find such functions only for the smallest investigated size. Interestingly, we are able to identify this difficulty as not lying in the evolution of hyper-bent functions itself, but rather in evolving some of their components, i.e. bent functions. Finally, we present an additional parameter to evaluate the performance of evolutionary algorithms when evolving Boolean functions: the diversity of the obtained solutions.

Keywords: Bent Functions · Hyper-bent Functions · Genetic Programming · Genetic Algorithms · Evolution Strategies

1 Introduction

Boolean functions are mathematical objects with numerous applications in cryptography, coding theory, and sequences. As such, they received a great deal of attention by the research community in the last decades. *Bent* Boolean functions, which exist only for even numbers of input variables, are those functions that have maximal nonlinearity, that is, they have the highest possible distance from the set of *affine functions*. For their construction, one can employ a number of different *primary constructions* (where bent functions are generated from

scratch) or *secondary constructions* (where one uses already known bent functions to construct larger ones). As an alternative method one can use heuristics, among which *evolutionary algorithms* proved to be especially adept in the last years. In fact, the sheer amount of successful results obtained with evolutionary algorithms makes the evolution of bent functions almost an easy problem [1–3]. Naturally, this is a somewhat oversimplified claim, since we can always aim to evolve Boolean functions of a size large enough that the process will be unfeasible from the computational perspective. There exist a sub-class of bent functions, namely *hyper-bent* functions, that have even stronger properties and are rarer than bent functions. Indeed, Hyper-bent functions are not only as far as possible from all affine functions, but also from all coordinate functions of all *bijective monomials*. Consequently, they can provide a good source of nonlinearity when designing block ciphers [4]. Unfortunately, since hyper-bent Boolean functions are rarer than bent Boolean functions, it could be that such functions are also more difficult to generate than bent functions. Still, the authors of [4] proved that hyper-bent functions exist for every even n, as general bent functions.

In this paper, we examine whether evolutionary algorithms can be a suitable technique for constructing hyper-bent functions, since such techniques proved to be very powerful when considering the generation of bent functions. More precisely, we consider some well-known techniques like genetic algorithms (GA), genetic programming (GP), and evolution strategy (ES) that proved to be able to evolve bent functions for a number of different sizes. We pose the following research questions. Can evolutionary algorithms be used to obtain hyper-bent functions in various sizes? If so, we are additionally interested in what is the richness of the solution set. More precisely, a common argument for using heuristics is that it allows us to obtain a number of different solutions which is sometimes not possible with algebraic constructions. Consequently, we will not only be interested in obtaining hyper-bent functions but also, but also in examining the number of different hyper-bent functions we are able to construct. In order to provide answers to the defined questions, we examine which of the considered evolutionary techniques achieves the best results. To the best of our knowledge, this is the first time evolutionary algorithms are considered for the evolution of hyper-bent functions. The obtained results show us that this problem is much more difficult for evolutionary algorithms than the evolution of bent functions, since the latter has already been investigated in the literature with the same techniques described in this paper. Hence, this problem could also represent a good benchmark for evaluating the performance of heuristics.

This paper is organized as follows. In Section 2, we briefly introduce bent and hyper-bent Boolean functions along with their relevant properties. Section 3 gives an overview of related work. Section 4 discusses the evolutionary algorithms we consider as well as the fitness functions we use in the experiments. In Section 5, we present results from evolutionary algorithms for a number of relevant Boolean function sizes. Additionally, we provide a review of the obtained results and we discuss possible future research directions. Finally, in Section 6, we summarize the main contributions of this work and conclude the paper.

2 Background

We refer the reader to [5] for a thorough introduction to the notions about Boolean functions discussed in this section.

In this work, we consider the set $\{0,1\}$ as the *finite field* with two elements, equipped with the XOR $\oplus$ and AND $\cdot$ as field operations. Given a positive integer $n \in \mathbb{N}$, the set $\{0,1\}^n$ of all binary strings of length n, which is composed of 2^n elements, is denoted as $\mathbb{F}_2^n$ and it is regarded as a vector space over $\mathbb{F}_2$. Additionally, to introduce the definition of hyper-bent functions, we will also consider $\{0,1\}^n$ as the finite field $\mathbb{F}_{2^n}$. The elements of $\mathbb{F}_{2^n}$ can be interpreted as *polynomials* in $\mathbb{F}_2[x]$ modulo an irreducible polynomial $p(x)$ of degree n, i.e. as elements of the quotient ring $\mathbb{F}_2[x]/p(x)$. In particular, the components of an n-bit string identify the coefficients of the associated polynomial in $\mathbb{F}_{2^n}$. Since $\mathbb{F}_{2^n}$ is isomorphic to $\mathbb{F}_2^n$, it can also be considered as a $\mathbb{F}_2$-vector space [6].

A *Boolean function* of n variables is a map $f : \mathbb{F}_2^n \to \mathbb{F}_2$, and it can be uniquely represented by its *truth table* (TT), which is a vector $\Omega_f \in \mathbb{F}_2^n$ of length 2^n that specifies the output value $f(x)$ for each possible input vector $x \in \mathbb{F}_2^n$. Usually, it is assumed that the function values in Ω_f are lexicographically ordered with respect to their inputs.

The *dot product* of two vectors $a, b \in \mathbb{F}_2^n$ is defined as $a \cdot b = a_1 b_1 \oplus \cdots \oplus a_n b_n$, and it satisfies the axioms of *inner product* over the vector space $\mathbb{F}_2^n$. The *absolute trace* of an element x in the finite field $\mathbb{F}_{2^n}$ equals:

$$Tr(x) = x + x^2 + \ldots + x^{2^{n-1}} \ , \tag{1}$$

The trace $Tr(ax)$ of $a, x \in \mathbb{F}_{2^n}$ is also an inner product over $\mathbb{F}_{2^n}$, where ax represents the field multiplication of the elements a and x (that is, polynomial multiplication). A Boolean function L_a is called *linear* if it is defined by an inner product. Thus, in the case of the vector space $\mathbb{F}_2^n$, $a \in \mathbb{F}_2^n$ is an n-bit vector and $L_a : \mathbb{F}_2^n \to \mathbb{F}_2$ is defined as $L_a(x) = a \cdot x$ for all $x \in \mathbb{F}_2^n$. Likewise, for the finite field case $a \in \mathbb{F}_{2^n}$ is a polynomial in $\mathbb{F}_2[x]/p(x)$ and $L_a : \mathbb{F}_{2^n} \to \mathbb{F}_2$ is defined as $L_a(x) = Tr(ax)$. Linear functions which also sum a constant $a_0 \in \mathbb{F}_2$ to the inner product are called *affine*.

The *Walsh-Hadamard transform* is another unique representation for Boolean functions. Formally, given $f : \mathbb{F}_2^n \to \mathbb{F}_2$, its Walsh-Hadamard transform $W_f : \mathbb{F}_2^n \to \mathbb{Z}$ is defined for all $a \in \mathbb{F}_2^n$ as:

$$W_f(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus a \cdot x} \ . \tag{2}$$

In other words, the coefficient $W_f(a)$ measures the correlation between f and the linear function $a \cdot x$. The maximum absolute value $|W_f(a)|$ of W_f over all $a \in \mathbb{F}_2^n$ is also called the *spectral radius* of f. *Parseval's relation* states that the sum of the squared Walsh coefficients is constant for every Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$:

$$\sum_{a \in \mathbb{F}_2^n} W_f(a)^2 = 2^{2n} \ . \tag{3}$$

The *nonlinearity* of a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is defined as the minimum Hamming distance between the truth table of f and the set of affine functions, and it can be expressed as [5]:

$$Nl_f = 2^{n-1} - \frac{1}{2} \max_{a \in \mathbb{F}_2^n} |W_f(a)| \ , \tag{4}$$

from which it follows that the lower the spectral radius of a Boolean function is, the higher its nonlinearity will be. By Parseval's relation, one can see that the the spectral radius is minimum if and only if all Walsh coefficients equal $\pm 2^{\frac{n}{2}}$. Functions satisfying this property are called *bent*, and they achieve the maximum value of nonlinearity $2^{n-1} - 2^{\frac{n}{2}-1}$. Since the Walsh-Hadamard coefficients must be integer numbers, it follows that bent functions exist only when the number of variables n is even.

Remark that the notion of bent function is independent of the underlying inner product used in the Walsh-Hadamard transform [5]. Hence, one could substitute the dot product $a \cdot x$ in Eq. 2 with the trace $Tr(ax)$ and obtain the same set of bent functions. This is important to introduce the notion of hyper-bent functions, the main object we are interested in this paper. In fact, hyper-bent functions are characterized through the so-called *extended Walsh-Hadamard transform*, which is defined over the finite field $\mathbb{F}_{2^n}$ as follows:

$$W_f(a, i) = \sum_{\boldsymbol{x} \in \mathbb{F}_2^n} (-1)^{f(x) \oplus Tr(ax^i)}, \tag{5}$$

where i is coprime with $2^n - 1$, i.e., $gcd(i, 2^n - 1) = 1$. Hence, in the extended transform we are computing several spectra of Walsh coefficients. The reason to consider the linear functions defined by $Tr(ax^i)$ is that x^i represents a *bijective monomial*, since i is coprime with $2^n - 1$. By considering only $i = 1$, one gets the usual Walsh-Hadamard transform.

A function f which is bent with respect to the extended transform for all i coprime with $2^n - 1$ is called *hyper-bent*. Thus, hyper-bent functions have the highest Hamming distance from all affine functions defined by bijective monomials. Notice that the number of indices i coprime with $2^n - 1$ is determined by *Euler's totient function* $\phi(2^n - 1)$, which grows as $\mathrm{O}\big(2^n\big)$. Thus, the number of indices against which the bent property must be checked with the extended transform is exponential in the number of variables n, and it becomes computationally expensive already for small values of n.

Table 1 reports the number of Boolean functions $\mathcal{B}_n$ of n variables, the number of exponents $\mathcal{I}_n$ coprime with $2^n - 1$, the number of bent $\mathcal{A}_n$ and hyper-bent $\mathcal{H}_n$ Boolean functions, and their nonlinearity Nl_f for sizes $n = 4, 6, 8$. The values for $\mathcal{A}_n$ and $\mathcal{H}_n$ have been taken from [7].

Observe that for $n = 4$ variables the resulting space of Boolean functions can be completely enumerated, since it is composed of only $2^{16} = 65\,536$ elements.

Table 1: The number and nonlinearity of Boolean functions for various input sizes n.

n	4	6	8
$\mathcal{B}_n$	2^{16}	2^{64}	2^{256}
$\mathcal{I}_n$	8	36	128
$\mathcal{A}_n$	896	$\approx 2^{32.3}$	$\approx 2^{106.3}$
$\mathcal{H}_n$	56	252	48 620
Nl_f	6	28	120

3 Related Work

Golomb and Gong proposed hyper-bent functions as components of Substitution Boxes to ensure the security of cryptographic algorithms [8]. Charpin and Gong investigate how to classify hyper-bent functions [9]. Carlet and Gaborit show that hyper-bent functions known at the time belong to the $\mathcal{PS}^{\#}_{ap}$ class. Next, they show how such functions can be obtained from certain codewords of extended cyclic code with small dimension and they enumerate hyper-bent functions up to $n = 10$ [10].

As already mentioned, there is a significant corpus of papers dealing with the heuristic generation of bent Boolean functions. At the same time, there are no works, to the best of our knowledge, considering the heuristic generation of hyper-bent functions. Consequently, here we remind the readers on relevant results in the heuristic generation of bent Boolean functions as well as the relevant theoretical results.

As far as we are aware, the first paper that uses evolutionary algorithms with a goal of evolving cryptographic Boolean functions dates back to 1997. There, Millan et al. used genetic algorithms to evolve Boolean functions with high nonlinearity [11]. Millan, Clark, and Dawson experimented with GA to evolve Boolean functions that have high nonlinearity [12]. They used a combination of GAs and hill climbing together with a resetting step in order to find Boolean functions up to 12 inputs with high nonlinearity. Millan, Fuller, and Dawson proposed an adaptive strategy for a local search algorithm for the generation of Boolean functions with high nonlinearity [13]. Additionally, they introduced the notion of the graph of affine equivalence classes of Boolean functions.

Picek, Jakobovic, and Golub used GA and GP to find Boolean functions that have several optimal cryptographic properties [14]. To the best of our knowledge, this is the first application of GP for evolving cryptographic Boolean functions. Hrbacek and Dvorak used CGP in order to evolve bent Boolean functions of sizes up to 16 inputs [1]. The authors experimented with several configurations of algorithms in order to speed up the evolution process where they did not limit the number of generations in their search. With such an approach, they succeeded in finding bent function in each run for sizes between 6 and 16 inputs.

Mariot and Leporati designed a discrete particle swarm optimizer to search for balanced Boolean functions with high nonlinearity [15]. The same authors in [16] used GA where the genotype consists of the Walsh-Hadamard values in order to evolve semi-bent Boolean functions by spectral inversion. An analysis of the efficiency of several evolutionary algorithms when evolving Boolean functions satisfying different cryptographic criteria is given in [17]. Picek and Jakobovic experimented with GP with a goal of evolving algebraic constructions, which are then used to construct bent functions [2]. There, the authors are able to find bent Boolean functions for sizes up to 24 inputs. Picek, Sisejkovic, and Jakobovic experimented with several immunological algorithms to construct bent or balanced, highly nonlinear Boolean functions [3].

4 Experimental Setup

In this section, we describe the solution representations, the search algorithms, and the fitness functions we use in our experiments.

4.1 Truth Table Representation

The most intuitive encoding for a Boolean function is in the form of a truth table. In this case, individuals are represented as strings of bits which present truth tables of Boolean functions. The string length is determined by the number of Boolean variables n and equals 2^n. Using this encoding, we experiment with two search algorithms: a genetic algorithm and an evolution strategy.

For GA, we employ a 3-tournament selection which serves to eliminate the worst individual among three randomly selected ones. After the elimination, a new individual is produced using the crossover operator applied on the remaining two. The new individual immediately undergoes mutation subject to a given individual mutation rate.

The crossover operators are one-point and uniform crossover, performed uniformly at random for each new offspring. The mutation operator is selected uniformly at random between a simple mutation, where a single bit is inverted, and a mixed mutation, which randomly shuffles the bits in a randomly selected subset. The mutation probability is used to select whether an individual would be mutated or not, and the mutation operator is executed only once on a given individual. For example, if the mutation probability is 0.7, then on average 7 out of every 10 new individuals will be mutated and one mutation will be performed on each of those individuals. The selected population size equals 100 individuals, whereas the individual mutation probability is 0.3.

When experimenting with evolution strategy (ES), we use $(\mu + \lambda)$-ES. In this algorithm, in each generation, parents compete with offspring and from their joint set μ fittest individuals are kept. In our experiments, offspring population size λ has a value equal to 5 and parent population size μ has a value of 100. Although it is not standard in the ES literature to have such a big population, we adopted this size since some works (see e.g. [17]) showed that it brings good

results when evolving cryptographic Boolean functions. For further information on ES, we refer interested readers to [18–20].

4.2 Tree Representation

Tree encoding is commonly related to genetic programming (GP) in which the data structures that undergo optimization are executable expressions) [18]. Each individual in a GP population represents a computable expression, whose most common form are symbolic expressions corresponding to parse trees. A tree can represent a mathematical expression, a rule set or a decision tree. The building elements in a tree-based GP are functions (inner nodes) and terminals (leaves, problem variables).

As opposed to truth table encoding, the other option we consider is to use a symbolic representation of a Boolean function. This is performed in a way such that genetic programming can be used to evolve a Boolean function in the form of a syntactic tree. Here, the terminal set consists of the n input Boolean variables, denoted $\{v_0, \ldots, v_n\}$. The function set (i.e., the set of inner nodes of a tree) should consist of appropriate Boolean operators that allow the definition of any function with n inputs.

The function set for genetic programming in all the experiments consists of Boolean functions OR, XOR, AND (taking two arguments), NOT (one argument), and IF. The function IF takes three arguments and returns the second argument if the first one evaluates to 'true', and the third one otherwise.

In our experiments, GP uses the same steady-state tournament selection algorithm of GA. The variation operators are simple tree crossover, uniform crossover, size fair, one-point, and context preserving crossover (selected at random), and subtree mutation [21]. All our experiments suggest that having a maximum tree depth equal to the number of Boolean variables is sufficient (i.e., tree depth equals n). The initial population is created at random and the population size equals 500.

4.3 Boolean Construction Representation

Finally, we experiment with a concept that stems from Boolean algebraic construction methods. In this setting, we try to construct a Boolean function of $n + 2$ variables using previously obtained Boolean functions of n variables. The process can be described as: first, the optimization problem is solved for a Boolean function size which allows the solutions to be found without much effort. In many cases, we can always start with the number of Boolean variables that produces the search space that can be scanned by an exhaustive search. For this problem, we start with $n = 4$, where we can enumerate all possible solutions, e.g. 56 hyper-bent Boolean functions of size 4.

The second step is to use GP to evolve Boolean functions of size $n + 2$; in this case, the terminals are not the $n + 2$ Boolean variables; rather, the terminal set includes *four* predefined Boolean functions with n variables which were previously obtained. These terminals are denoted with f_0, f_1, f_2, and f_3 (input

functions). Additionally, the terminal set includes two independent Boolean variables, v_0 and v_1 (since we are constructing functions in size $n+2$). The choice of having four input functions is inspired by the Rothaus' construction [22], where three bent functions such that their exclusive OR is again bent (consequently, we can consider this as having four functions) are used to construct new bent functions.

Finally, each resulting construction (a GP expression) that includes input functions and additional two variables, represents a new Boolean function with $n+2$ variables. This construction is based on truth table representations of input functions, which are the size of 2^n, and the extension with two Boolean variables which, for every combination of their values, extend the resulting truth table to the size of 2^{n+2}.

With the goal of obtaining hyper-bent functions of size $n+2$, the input functions are presumed to be hyper-bent themselves. Since 56 hyper-bent functions of size 4 are available, we define 14 sets with four input functions in each set, and every construction is evaluated using each of the 14 input sets. As we are interested only in optimizing the objective value (and not in finding a general construction method), we assign the fitness function of a candidate construction as the best objective value among all possible 14 resulting functions.

Note that this process can be self-sustained: if the resulting construction produces the desired solutions, then those solutions can be used as input functions for the next construction stage for size $n+4$. However, in each stage we need to add only two additional Boolean variables; this greatly reduces the search size and computational effort, as compared to the common approach where the solution size is increased with increasing number of Boolean variables. Apart from the terminals set, all the other GP parameters, as well as the function set, remain the same.

4.4 Fitness Functions

The first fitness function aims to find hyper-bent functions by considering the nonlinearity of the bent function for each i (see Eq. (5)). The aim is to minimize the following expression:

$$fitness_1 = \sum_{i=1, gcd(i,2^n-1)=1}^{2^n-1} \left(2^{n-1} - 2^{\frac{n}{2}-1} - Nl_{f_i}\right). \tag{6}$$

Note, we simply subtract the nonlinearity value of the obtained function from the nonlinearity of the bent function (see Section 2). This fitness function we also use when trying to obtain hyper-bent functions via secondary constructions.

In the second fitness function, we consider the whole Walsh-Hadamard spectrum and we penalize proportionally to the number of the Walsh-Hadamard coefficients differing from $2^{\frac{n}{2}}$ (see Eq. (3) and conclusions stemming from it).

Again, the goal is to minimize the following expression:

$$fitness_2 = \sum_{i=1, gcd(i,2^n-1)=1}^{2^n-1} T_i, \tag{7}$$

where T_i equals the number of times the Walsh-Hadamard coefficients differ from $2^{\frac{n}{2}}$ for each i.

Finally, we define the third fitness function which tries to increase the number of cases in the above sum (different values of i) where the obtained function is bent. This factor is multiplied with an arbitrary constant to make it a primary criterion; the secondary criterion is then simply the relative difference from maximum nonlinearity for all the cases where a bent function is not found. The third objective function is defined as a minimization of the following:

$$fitness_3 = 100 \times S_i + \sum_{i=1, gcd(i,2^n-1)=1}^{2^n-1} \left(1 - \frac{Nl_{f_i}}{2^{n-1} - 2^{\frac{n}{2}-1}}\right), \tag{8}$$

where S_i equals the number of different values of i in the above sum where the Nl_{f_i} was not equal to maximum nonlinearity.

5 Experiments

In this section, we first give results for GA, GP, and ES. Afterward, we discuss the difficulty of this problem, the diversity of solutions, and finally, possible future works.

5.1 Results

We present the results of three evolutionary algorithms (GA, ES, and GP) based on the function size and fitness functions defined in the previous section. Regardless of the representation and the search algorithm, every experiment is repeated 30 times and the stopping condition is the number of fitness evaluations, which is set to 500 000.

In the case of Boolean functions of 4 variables, all tested methods easily converge to fitness value of zero, i.e., we find a hyper-bent function in every run. This is expected since the search size is relatively small and this presents no problem regardless of the solution encoding. Consequently, we do not present results for $n = 4$.

For size $n = 6$, the results for all the three algorithms and three variants of fitness functions are shown in Table 2. Since we are using different fitness functions that are not comparable, we divide the results in two groups: the left side of the table shows the statistics of obtained solutions using the objective function for which they were evolved (i.e. a separate set for each one of the three fitness functions). The right half of the table shows the same solutions

evaluated on the basis of the first fitness function, so the efficiency of different fitness functions can be assessed.

In terms of objective functions, we can see that the first and the third fitness function provided solutions of the same quality, while the second fitness was inferior to the other two. At the same time, the tree encoding used with GP was convincingly better than the truth table representation, regardless of the search algorithm. Unfortunately, no single algorithm or objective function was able to provide a hyper-bent Boolean function of 6 variables, which would correspond to the fitness value of zero. When considering the obtained results, we can see that GP performs the best. More precisely, for all three fitness functions, it outperforms GA and ES significantly. At the same time, the behavior for GA and ES is similar but still slight advantage goes to GA.

Table 2: Optimization results for Boolean functions of 6 variables

	Optimized objective				**Evaluated by** $fitness_1$			
$fitness_1$	**min**	**median**	**avg**	**max**				
ES	108	132	129.64	144				
GA	108	132	135.03	156				
GP	72	120	113.73	156				
$fitness_2$	**min**	**median**	**avg**	**max**	**min**	**median**	**avg**	**max**
ES	1 260	1 365	1 361.8	1 422	156	216	217.69	276
GA	1 206	1 380	1 375.9	1 458	144	216	215.51	288
GP	648	864	881.4	1 272	72	204	207.87	432
$fitness_3$	**min**	**median**	**avg**	**max**	**min**	**median**	**avg**	**max**
ES	3 602.7	3 603.8	3 603.6	3 604.3	108	132	128.55	144
GA	3 005.3	3 604.1	3 596.5	3 604.3	108	144	137.77	156
GP	1 802.1	3 003.6	2 570.7	3 604.3	72	120	116	144

As for the Boolean construction method (see Section 4.3), it has also proven unsuccessful; this approach is tested with GP using groups of different previously obtained hyper-bent functions of 4 variables as input functions. In every GP run, the obtained constructed function converged to the objective value of 120 according to the first fitness function. Since the results do not show any deviation, we omit those from the table. Interestingly, our experiments show that the well-known Rothaus construction cannot be used to construct hyper-bent functions despite the fact that it can construct bent functions. To the best of our knowledge, this information is new (albeit, not unexpected). Additionally, since we have been unable to obtain hyper-bent functions for $n = 6$, we were unable to apply this construction method for larger sizes.

Finally, Table 3 presents the results for optimizing Boolean functions of 8 variables, in the same form as Table 2. From the results obtained for $n = 8$, we

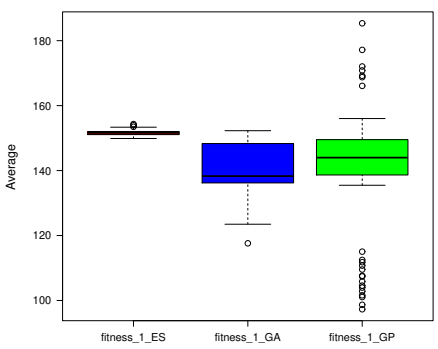
can see that for $fitness_1$, GP is now actually the worst performing algorithm. For the second scenario, $fitness_2$, we observe GP has the best value but as evaluated relative to $fitness_1$, GP is still the worst performing algorithm. Only for $fitness_3$ GP is the best choice. Similar as for $n = 6$, the performance of GA and ES is similar with a small advantage for GA (except for $fitness_2$ as evaluated relative to $fitness_1$).

Table 3: Optimization results for Boolean functions of 8 variables

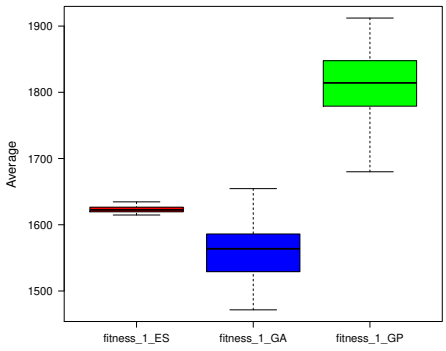
	Optimized objective				**Evaluated by** $fitness_1$			
$fitness_1$	**min**	**median**	**avg**	**max**				
ES	1 472	1 504	1 501.7	1 536				
GA	1 440	1 520	1 527.6	1 616				
GP	1 536	1 632	1 630	1 696				
$fitness_2$	**min**	**median**	**avg**	**max**	**min**	**median**	**avg**	**max**
ES	27 704	27 824	27 818	27 944	1 920	2 096	2 107	2 288
GA	27 568	27 808	27 793	28 016	1 984	2 080	2 095.5	2 272
GP	17 456	17 456	17 456	17 456	3 456	3 456	3 456	3 456
$fitness_3$	**min**	**median**	**avg**	**max**	**min**	**median**	**avg**	**max**
ES	254.04	254.06	254.05	254.06	2 016	2 088	2 149.3	2 336
GA	254.04	254.06	254.05	254.06	1 936	1 984	2 008	2 160
GP	248.06	248.06	248.07	248.1	1 792	1 872	1 938.6	2 176

In Figures 1a until 1e, we display results for $n = 6$, all algorithms and fitness functions 1, 2, and 3, respectively. We display the average values as averaged over 30 runs. When considering $fitness_1$, we see that GP obtains the best results but also the worst average, which indicates that the results are not always stable and more evaluations could be needed. Evolution strategy, on the other hand, performs the worst while having all the values approximately the same. The second fitness allows GP to become more stable and we can actually see that even the worst values for GP are better than median values for GA and ES. Interestingly, we also observe that GA has better general performance than ES but worse outlier solutions. Finally, for $fitness_3$, we see a significant difference in the performance between GP on one side and GA and ES on the other side. More precisely, GA and ES perform similarly and much worse than GP where only the best GA outliers reach the worst performance for GP.

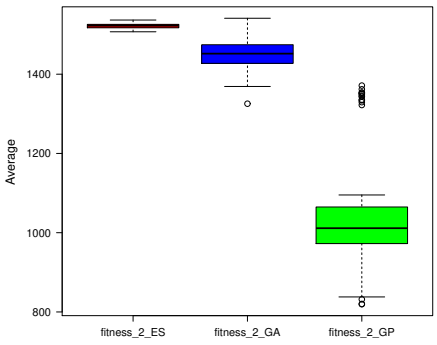
Figures 1b until 1f give results for case when $n = 8$. As already noted, for $fitness_1$ we surprisingly see that GP performs the worst. What is more, both GA and ES exhibit significantly better performance where GA is the best. In the second scenario ($fitness_2$), the situation changes completely and now GP is by far the best. At the same time, the difference between GA and ES is quite



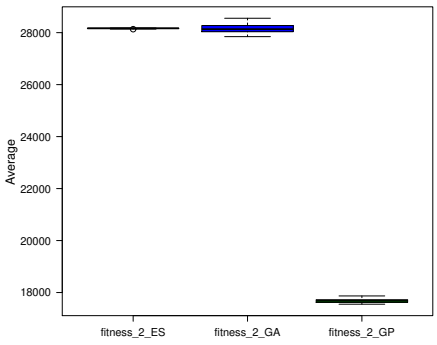
(a) Results for $fitness_1, n = 6$.



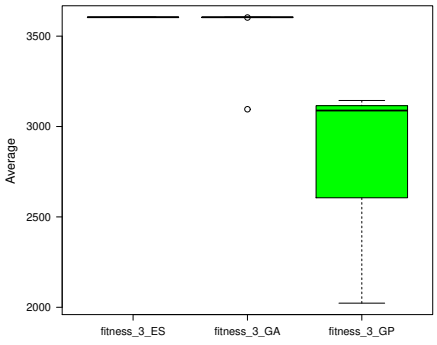
(b) Results for $fitness_1, n = 8$.



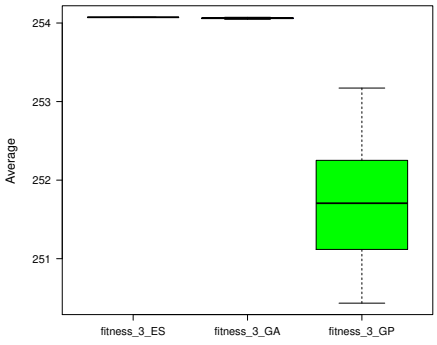
(c) Results for $fitness_2, n = 6$.



(d) Results for $fitness_2, n = 8$.



(e) Results for $fitness_3, n = 6$



(f) Results for $fitness_3, n = 8$.

Fig. 1: Results for all tested algorithms with $n = 6, 8$.

small. Finally, for $fitness_3$, GP is again the best with (almost) no differences between GA and ES.

On the Diversity of Solutions As already mentioned, we propose a new criterion to evaluate the performance of evolutionary algorithms. When working on problems where we know a deterministic method to obtain solutions (as in the case with the hyper-bent functions), the need to use evolutionary algorithms is less obvious. One reason why heuristics still could provide a viable alternative to algebraic constructions is if it can offer a large body of different solutions. Indeed, with algebraic constructions, we are always limited in the number of possible solutions (since different affine transformations always result in solutions from the same class). To that end, we explore how many different solutions are obtained with GP, GA, and ES when $n = 4$, by comparing the truth tables of the resulting hyper-bent functions. For this scenario, all the experiments are run 100 times. For ES and the first fitness function, out of 100 experiments, we obtained 49 different solutions. Next, GA finds 45 solutions and GP only 41 different solutions. For the second fitness function, ES finds 46 unique solutions, GA 52 unique solutions, and, GP 37 unique solutions. When considering fitness function 3, ES obtains 42 unique solutions, GA finds 48 unique solutions, and GP 30 unique solutions.

From these results, we can see that despite the fact that ES and GA performed worse than GP when considering the results obtained from the fitness functions, the situation differs when considering the diversity of solutions. With this criterion, GA and ES actually proved to be more powerful than GP. Still, we can conclude that all three algorithms are able to find a significant number of different solutions since the total number of solutions for $n = 4$ equals 56.

Deceptive Problem or Simply Difficult Problem Since we are unable to find solutions already for $n = 6$, the question is why the problem of finding hyper-bent functions is so difficult. One explanation for the difficulty would be if the problem has some sort of deceptive behavior: since the extended Walsh-Hadamard transform needs to calculate $Tr(ax^i)$ for every i such that $gcd(i, 2^n - 1) = 1$, one scenario would be that solutions for certain values of i conflict with some other values.

To check this, we run the experiments with GP where we checked what happens with every value of i and we noticed that in our experiments we always fail for the same i values. As such, this could indicate a deceptive behavior since it could happen that the solution for a previous value of i got stuck in local optima and it cannot move to another local optimum. Next, we experiment only with those values i where we could not obtain the maximal nonlinearity when considering all allowed i values. Interestingly, we observe that for such values i we cannot obtain maximal nonlinearity even if we consider it as a separate case (for instance, $i = 5$). This is a surprising result since all the experiments up to now indicate that we are able to easily find bent functions. To conclude, there are certain values of i that are difficult for EA even when considered

separately, where by difficult we consider the inability to reach the maximal nonlinearity. Thus, when considering all possible i values, it is impossible to find hyper-bent functions. From these observations, we conclude that the problem is not deceptive but is "simply" difficult in certain components. Currently, we leave it as an open question what does make certain values of i so difficult.

5.2 Future Work

There are several directions along which the present work can be extended. A first idea to improve the performance of GA would be to design specific crossover and mutation operators which reduce the search space of candidate solutions. In fact, as discussed in Section 2, the Walsh-Hadamard coefficients of a bent functions of n variables are all equal to $\pm 2^{\frac{n}{2}}$. Since the Walsh-Hadamard coefficient $W_f(\underline{0})$ of the null vector is the deviation of the function from being balanced, it follows that the truth table of any bent function is composed of $hwt_f = 2^{n-1} \pm 2^{\frac{n}{2}-1}$ ones. This suggests evolving through GA bitstrings with the fixed Hamming weight. A similar strategy was initially proposed by Millan et al. [23] to evolve balanced Boolean functions, where the authors proposed a crossover operator that used counters to keep track of the multiplicities of zeros and ones in the offspring. More recently, the same approach has been used by Mariot and Leporati [16] to evolve plateaued functions by spectral inversion, and by Mariot et al. [24] to evolve binary orthogonal arrays.

In our setting, a possible idea would be to adapt Millan et al.'s counter-based crossover operator to the case of bent functions, with the number of ones in the offspring constrained to be equal to either $2^{n-1} - 2^{\frac{n}{2}-1}$ or $2^{n-1} + 2^{\frac{n}{2}-1}$. Subsequently, it would be interesting to assess if the performances of our GA improve in generating hyper-bent functions.

On a more general level, a computational bottleneck is the calculation of the extended Walsh-Hadamard spectrum. The naive implementation of the Walsh-Hadamard transform has a complexity of $\mathcal{O}(2^{2n})$ where n is the Boolean function size. This can be improved by using the butterfly algorithm where the complexity decreases only to $\mathcal{O}(n2^n)$. Unfortunately, there does not seem to be an easy way to use the butterfly algorithm when calculating the extended Walsh-Hadamard transform. Consequently, the exponential rise in the complexity makes the calculation already extremely difficult for $n \geq 8$, especially coupled with a high number of evaluations occurring in the heuristic approaches. In future work, we plan to explore how to implement more efficiently the extended Walsh-Hadamard spectrum. Alternatively, since it is known that hyper-bent functions have algebraic degree equal to $n/2$ [10], one could try to include this information in the fitness function to speed up the search.

6 Conclusions

In this paper, we consider the evolution of hyper-bent functions, i.e., functions that are bent up to a primitive root change. Hyper-bent functions have real-world

applications and are extremely rare objects, which makes them an interesting target for evolutionary algorithms. Our results indicate this problem to be of extreme difficulty and even out of reach for evolutionary algorithms. Indeed, we are able to find hyper-bent functions only for $n = 4$, which is the dimension where also exhaustive search is easily conducted.

Despite the failure in finding hyper-bent functions for $n > 4$, we can still discuss the performance of tested algorithms where we see that GP behaves the best. This is probably not surprising since GP also showed excellent results when evolving bent functions. When considering the diversity of obtained solutions, we observe that GA is the best but all algorithms show very good results: on average every second solution (or third for GP when considering $fitness_3$) is a new one. Finally, our experiments indicate that the difficulty of evolving hyper-bent functions stems from the fact that our algorithms are not able to find certain bent components of hyper-bent functions. It remains to be explored why is that and how can we overcome this obstacle. We note that our results open the problem of evolving hyper-bent functions as a strong benchmark when evaluating the performance of evolutionary algorithms.

7 Acknowledgments

The authors wish to thank the anonymous referees for their useful comments on improving the presentation quality of the paper. This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-4882. In addition, this work was supported in part by the Research Council KU Leuven (C16/15/058) and IOF project EDA-DSE (HB/13/020).

References

1. Hrbacek, R., Dvorak, V.: Bent Function Synthesis by Means of Cartesian Genetic Programming. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) Parallel Problem Solving from Nature - PPSN XIII. Lecture Notes in Computer Science, vol. 8672, pp. 414–423. Springer International Publishing (2014)
2. Picek, S., Jakobovic, D.: Evolving Algebraic Constructions for Designing Bent Boolean Functions. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016. pp. 781–788 (2016)
3. Picek, S., Sisejkovic, D., Jakobovic, D.: Immunological algorithms paradigm for construction of Boolean functions with good cryptographic properties. Engineering Applications of Artificial Intelligence (2016)
4. Youssef, A.M., Gong, G.: Hyper-bent functions. In: Pfitzmann, B. (ed.) Advances in Cryptology — EUROCRYPT 2001. pp. 406–419. Berlin, Heidelberg (2001)
5. Carlet, C.: Boolean Functions for Cryptography and Error Correcting Codes. In: Crama, Y., Hammer, P.L. (eds.) Boolean Models and Methods in Mathematics, Computer Science, and Engineering, pp. 257–397. Cambridge University Press (2010)
6. Lidl, R., Niederreiter, H.: Introduction to finite fields and their applications. Cambridge university press (1994)

7. Carlet, C., Gaborit, P.: Hyper-bent functions and cyclic codes. Journal of Combinatorial Theory, Series A 113(3), 466 – 482 (2006)
8. Gong, G., Golomb, S.W.: Transform domain analysis of des. IEEE Transactions on Information Theory 45(6), 2065–2073 (1999)
9. Charpin, P., Gong, G.: Hyperbent functions, kloosterman sums, and dickson polynomials. IEEE Transactions on Information Theory 54(9), 4230–4238 (2008)
10. Carlet, C., Gaborit, P.: Hyper-bent functions and cyclic codes. J. Comb. Theory, Ser. A 113(3), 466–482 (2006)
11. Millan, W., Clark, A., Dawson, E.: An Effective Genetic Algorithm for Finding Highly Nonlinear Boolean Functions. In: Proceedings of the First International Conference on Information and Communication Security. pp. 149–158. ICICS '97 (1997)
12. Millan, W., Clark, A., Dawson, E.: Heuristic design of cryptographically strong balanced Boolean functions. In: Advances in Cryptology - EUROCRYPT '98. pp. 489–499 (1998)
13. Millan, W., Fuller, J., Dawson, E.: New concepts in evolutionary search for Boolean functions in cryptology. Computational Intelligence 20(3), 463–474 (2004)
14. Picek, S., Jakobovic, D., Golub, M.: Evolving Cryptographically Sound Boolean Functions. In: Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation. pp. 191–192. GECCO '13 Companion (2013)
15. Mariot, L., Leporati, A.: Heuristic search by particle swarm optimization of boolean functions for cryptographic applications. In: GECCO (Companion). pp. 1425–1426. ACM (2015)
16. Mariot, L., Leporati, A.: A Genetic Algorithm for Evolving Plateaued Cryptographic Boolean Functions. In: Theory and Practice of Natural Computing - Fourth International Conference, TPNC 2015, Mieres, Spain, December 15-16, 2015. Proceedings. pp. 33–45 (2015)
17. Picek, S., Jakobovic, D., Miller, J.F., Batina, L., Cupic, M.: Cryptographic Boolean functions: One output, many design criteria. Applied Soft Computing 40, 635 – 653 (2016)
18. Bäck, T., Fogel, D., Michalewicz, Z. (eds.): Evolutionary Computation 1: Basic Algorithms and Operators. Institute of Physics Publishing, Bristol (2000)
19. Rozenberg, G., Bäck, T., Kok, J.N.: Handbook of Natural Computing. Springer Publishing Company, Incorporated (2011)
20. Beyer, H.G., Schwefel, H.P.: Evolution Strategies A Comprehensive Introduction 1(1), 3–52 (May 2002)
21. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008)
22. Dillon, J.F.: Elementary Hadamard difference sets. Ph.D. thesis, University of Maryland (1974)
23. Millan, W., Clark, A.J., Dawson, E.: Heuristic design of cryptographically strong balanced boolean functions. In: Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding. pp. 489–499 (1998)
24. Mariot, L., Picek, S., Jakobovic, D., Leporati, A.: Evolutionary search of binary orthogonal arrays. In: Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part I. pp. 121–133 (2018)