Solving the even-n-parity problems using Best SubTree Genetic Programming

Oana Muntean, Laura Diosan, Mihai Oltean
Department of Computer Science
Faculty of Mathematics and Computer Science
Babes-Bolyai University, Kogălniceanu 1
Cluj-Napoca, 400084, Romania.
oana.muntean85@gmail.com
{lauras, moltean}@cs.ubbcluj.ro

Abstract

The result of the program encoded into a Genetic Programming (GP) tree is usually returned by the root of that tree. However, this is not a general strategy. In this paper we investigate a new variant where the best subtree is chosen to provide the solution of the problem. The other nodes (not belonging to the best subtree) are deleted. This will reduce the size of the chromosome in cases when its best subtree is different from the entire tree. We have tested this strategy on a wide range of even-n-parity problems. Numerical experiments have shown that the proposed approach performs better compared to standard GP.

1 Introduction

The problem of designing digital circuits by using evolutionary techniques has been intensely analyzed in the recent past [5, 14, 21]. Genetic Programming (GP) and its variants were mainly used for this purpose [13, 7]. Numerical results have shown that GP-based techniques were able to evolve digital circuits better than those designed by human experts.

In this paper we expand a new variant of Genetic Programming, proposed in [16], where the output is not given by the root of the tree as in the case of standard GP. Instead, the best subtree is chosen for providing the result of a chromosome. This is why this variant of GP is called Best SubTree Genetic Programming (BSTGP).

There are two very important features which make BSTGP different from standard GP:

- The best subtree is selected for providing the solution of the problem. This is different from the standard GP where the fitness of a chromosome (tree) is given by its root node.
- Nodes not belonging to the best subtree are deleted. In this way the BSTGP trees may be smaller compared to the GP trees.

All these operations are performed during the fitness computation process. All other elements of a standard GP algorithm [7] remain unchanged.

This approach has been tested against several even-n-parity problems. Numerical results have shown that this variant of GP performs very well compared to the standard Genetic Programming.

The paper is organized as follows: Related work is briefly reviewed in Section 2. The BSTGP approach is described in Section 4. The most important aspect is given in Section 4.1 where the fitness assignment process is detailed. The test problems are briefly introduced in Section 5.1. The results of the numerical experiments are presented in Section 5.3. Section 6 discusses the strengths and weaknesses of the proposed approach. Section 7 indicates a list of possible improvements required for solving higher instances of the problem. Finally, Section 8 concludes our paper and summarizes the further work directions.

2 Related work

Different GP techniques employ different strategies for selecting the sequence of instructions which provides the solution of the problem.

The most prominent example is Cartesian Genetic Programming (CGP) [15] where the output node is

evolved like all other genes. This means that the graph providing the solution might not contain all nodes of the CGP chromosome.

Linear Genetic Programming (LGP) has also been subject to code efficientizations. In [3] the authors have removed the instructions that do not participate to the solution (also called introns). Note that this is different from our approach since, in our case, all nodes of the tree are effective [3], but not all of them belong to the best subtree.

Introns removal is a common operation in GP [1, 11, 17]. Many papers have investigated this issue. Some researchers have argued that introns are beneficial to GP because they protect the code from the destructive effect of crossover.

3 Even-n-parity problem

Our aim is to find a Boolean function that satisfies a set of fitness cases. The particular function that we want to find is the Boolean even-n-parity function. This function has n Boolean arguments and it returns T (*True*) if an even number of its arguments are T. Otherwise the even-n-parity function returns F (*False*) [7, 20]. According to [7] the Boolean even-n-parity functions appear to be the most difficult Boolean functions to detect via a blind random search.

The problem of evolving Boolean functions has been intensively analyzed in the past [4, 6, 7, 8, 20].

In applying a Genetic Programming technique (in particular BSTGP technique) to the even-n-parity function of n arguments, the terminal set T consists of the k Boolean arguments d_0, d_1, d_2, ..., d_{n-1}.

The function set F usually consists of all two-argument primitive Boolean functions (also called gates [14]). Using this complete set we can obtain solutions of the even-n-parity problem by spending less computational resources [20].

Koza [7] has performed a detailed analysis of the even-n-parity problem in his first book about the standard GP paradigm [7]. Later, new results were obtained when the Automatically Defined Functions (ADFs) [8] have been discovered. In both cases only four Boolean functions ($F = \{AND, OR, NAND, NOR\}$) were used.

Koza has established that the solving medium size instances of the parity problems using standard GP without ADFs is computationally expensive. He did not obtain a result for values of $n > 5$. However, better results were obtained when ADFs were used [8] (up to even-11-parity problem).

If we extend this set by including other Boolean functions (such as EQ and XOR) we can obtain solutions for larger instances. For instance, in [20] Genetic Programming using an extended set of 16 function symbols has been used for solving up to even-22-parity problems. Note that in this case a parallel variant of GP and a sub-symbol node representation were used on a network of computers structured in client-server architecture.

Replicating the Koza's studies, Chellapilla [4] has used Evolutionary Programming [24] for solving these problems. He has omitted the crossover operator and has used instead a variety of mutation operators.

Gathercole and Ross [6] have used GP without ADFs and a special fitness function for evaluating a Boolean function. The individual's fitness score is based on how many cases remain uncovered in the ordered training set after the individual quality exceeds an error limit. If this threshold is reached, the remaining, untested, cases are also considered as misclassifications. Their methods has provided better results (in terms of function evaluations) than the standard GP technique, but just for several problems (from *Even* 3 up to *Even* 7).

4 Best Subtree GP

It can be easily seen that each tree has a number of subtrees equal to the number of nodes. For instance, the tree depicted in Figure 1 has 11 subtrees. The distinct ones are depicted in Figure 2.

Usually the result of a GP tree is given by its root node (subtree ST_7 from Figure 2). We have already seen in Section 2 that this is not a general strategy. Various GP techniques choose different subtrees for encoding the solution.

In the approach proposed in [16] it is not chosen a fixed subtree for providing the solution. Instead, it is computed the quality of each subtree and it is selected the best of them for providing the solution of the problem.

More than that, after fitness computation all nodes not belonging to the best subtree are deleted. In this way the size of the entire chromosome might decrease which will lead to a faster search process.

Remark By best subtree we understand the subtree which has the best fitness. For instance, in the case of even-n-parity problem the fitness has to be minimized. Thus the best subtree is the one with the smallest fitness.

4.1 Fitness assignment

We focus our explanation on even-n-parity problems.

The set of fitness cases for an even-n-parity problem consists of the 2^n combinations of the n Boolean arguments. The fitness of an expression encoded into a tree or a subtree is the sum, over these 2^n fitness cases, of the Hamming distance (error) between the returned value by the current expression and the correct value of the Boolean function. Since the standardized fitness ranges between 0 and 2^n, a value closer to zero is better (since the fitness is to be minimized).

Finding the best subtree of a tree is an easy and inexpensive task. We know that when the fitness of an expression (encoded into a GP tree) is computed, the values of all expressions (encoded by the subtrees of the original tree) are also computed. This operation is done in $O(m)$ steps, where m is the number of nodes of the GP subtree. Thus, computing the value of an expression and of all expressions (encoded by subtrees) requires the same number of operations as the computation of the value of the entire expression.

Once we have the values of each sub-expression it is very easy to compute their fitness. We only have to take the difference (in absolute value) between the actual and expected output and then we sum these results for all fitness cases.

For finding the best subtree we employ a bottom-up approach: first of all we compute the fitness of the smallest expressions (subtrees) and then we compute the fitness of increasingly bigger sub-expressions (trees). The last expression whose fitness is computed is the one encoded by the entire tree.

Note that this bottom up strategy is identical with that of computing the value of an expression encoded into a tree.

The lowest fitness indicates which the best subtree of the chromosome is. Therefore, that subtree will become the new chromosome. All other nodes will be deleted.

4.2 Example

Let's consider an even-n-parity problem with two inputs ($n = 2$) and, therefore, with 2^2 fitness cases (see Table 1) and a chromosome with 5 node levels (depicted in Figure 1). We compute the fitness of each subtree (Figure 2) as described in section 4.1 and we cache the results. A possible order for fitness computation is the following: ST_1, ST_2, ST_3, ST_4, ST_5, ST_6, ST_7. Note that there are more than one order in which the fitness can be computed. For instance ST_2, ST_1, ST_4, ST_3, ST_5, ST_6, ST_7 is also a valid order.

Fitness values for each subtree are given in Table 2.

In this example ST_1, ST_2 and ST_5 have the same best quality. Therefore, each of them could become

Table 1. Four fitness cases for even-2-parity problem

Fitness cases	a	b	f(a,b)
i	F	F	T
ii	F	T	F
iii	T	F	F
iv	T	T	T

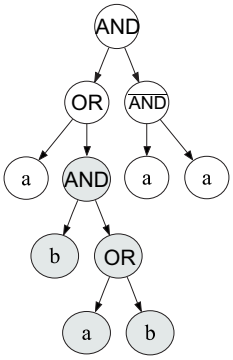

Figure 1. A GP chromosome encoding the Boolean expression $(((a$ OR $b)$ AND $b)$ OR $a)$ AND $(a$ NAND $b)$. The gray-filled nodes form the best fitted expression encoded in this tree.

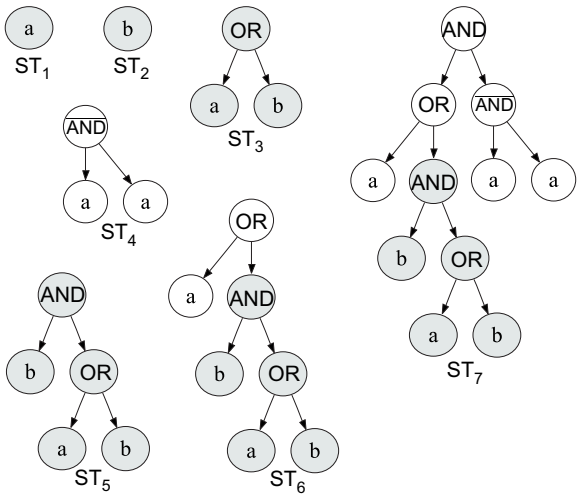

Figure 2. All possible distinct subtrees of the GP chromosome from Figure 1. Gray-filled nodes belong to the best subtree.

Table 2. Fitness of each subtree. First of all the evaluation results for each fitness case are computed. The fitness is given by the Hamming distance between the returned value and he correct value for all fitness cases.

Fitness case	ST_1	ST_2	ST_3	ST_4	ST_5	ST_6	ST_7
i	F	F	F	T	F	F	F
ii	F	T	T	T	T	T	T
iii	T	F	T	T	F	T	T
iv	T	T	T	F	T	T	F
	2	2	3	3	2	3	4

the new chromosome. In this case we pick one of them randomly (for instance ST_5). All other nodes of the original tree are deleted.

5 Experiments

Several numerical experiments with GP and BSTGP are carried out in this section.

5.1 Test problems

Several test problems are chosen for these experiments:

- *Even 3* - The even-3-parity problem has three Boolean inputs and one Boolean output. The number of fitness cases is $2^3 = 8$;
- *Even 4* - The even-4-parity problem has four Boolean inputs and one Boolean output. The number of fitness cases is $2^4 = 16$;
- *Even 5* - The even-5-parity problem has five Boolean inputs and one Boolean output. The number of fitness cases is $2^5 = 32$;
- *Even 6* - The even-6-parity problem has six Boolean inputs and one Boolean output. The number of fitness cases is $2^6 = 64$;
- *Even 7* - The even-7-parity problem has seven Boolean inputs and one Boolean output. The number of fitness cases is $2^7 = 128$;
- *Even 8* - The even-8-parity problem has eight Boolean inputs and one Boolean output. The number of fitness cases is $2^8 = 256$;
- *Even 9* - The even-9-parity problem has nine Boolean inputs and one Boolean output. The number of fitness cases is $2^9 = 512$;
- *Even 10* - The even-10-parity problem has ten Boolean inputs and one Boolean output. The number of fitness cases is $2^{10} = 1024$.

We have limited our experiments to this size because we wanted to explore the power of BSTGP method compared to standard GP. Note that results for higher instances of the even-n-parity problem can be obtained by both GP and BSTGP if we employ the following features: sub-symbolic node representation [20], smooth genetic operators [18], parallel implementation in which GP sub-populations or demes are distributed over a number of workstations [19].

These issues will be discussed in section 7.

5.2 Setup for GP methods

The same function set has been used for both GP methods which contains all 16 Boolean functions with two arguments. Terminal set T consists of the problem inputs.

We used a steady-state evolutionary model as underlying mechanism for our implementations of all GP methods. The algorithm starts by creating a random population of individuals. The following steps are repeated until a termination criterion is met: Two parents are selected using a standard selection procedure. The parents are recombined in order to obtain two offspring which are then considered for mutation. The best offspring O replaces the worst individual W in the current population if O is better than W.

The following standard genetic operations are performed [7]:

- initialization - ramped half and half. No more than 9 levels have been used for initializing each chromosome;
- selection - binary tournament;
- crossover - both models use a one-cutting point crossover with a given probability (p_c). Having two parent trees, we randomly chosen a one-cutting point in the first parent, another cutting-point in the second parent and we exchanged the subtree rooted at the cutting-point in first chromosome with the sub-tree rooted at the cutting-point in the second individual. Two new individuals are obtained by crossover;

- mutation - each node of a chromosome is changed with a given probability (p_m). Terminals are mutated into terminals and functions are mutated into other functions with the same arity. In this way the size of the tree is not changed.

Other parameter settings for both methods are given in Table 3. Note that, in the steady-state model, a generation is considered as being complete when *Number of generations* new individuals are generated.

Table 3. Parameters used by GP and BSTGP for even-n-parity problems

Parameter	Value	
Population size	*Even* 3	50
	Even 4	75
	Even 5	200
	Even 6	100
	Even 7	200
	Even 8	300
	Even 9	200
	Even 10	300
Number of generations	*Even* 3	50
	Even 4	75
	Even 5	100
	Even 6	100
	Even 7	100
	Even 8	200
	Even 9	200
	Even 10	500
Tournament size	2	
Crossover probability	0.8	
Maximal initial depth	9	
Mutation probability	0.1	

5.3 Assessing the performance of the BSTGP algorithm

For assessing the performance of the BSTGP algorithm a very important statistic is of high interest: the computational effort for both algorithms.

Koza [7] suggested the computation of the number of chromosomes, which would have to be processed to give a certain probability of success. To calculate this figure one must first calculate the cumulative probability of success $P(M;i)$, where M represents the population size, and i the generation number. The value $R(z)$ represents the number of independent runs required for a probability of success (given by z) at generation i.

The quantity $I(M;z;i)$ represents the minimum number of chromosomes which must be processed to give a probability of success z, at generation i. The formula are given by the equations (1), (2) and (3). $N_s(j)$ represents the number of successful runs at generation j, and N_{total}, represents the total number of runs:

$$P(M,i) = \frac{\sum_{j=1}^{i} N_s(j)}{N_{total}} \tag{1}$$

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil \tag{2}$$

$$I(M,i,z) = M \times i \times R(z) \tag{3}$$

Note that when $z = 1.0$ the formula (2) and (3) are invalid (all runs successful). In the graphs of Figure 3 z takes the value 0.99.

The curves representing the computational effort needed by BSTGP and GP algorithms to solve all the test problems are depicted in Figure 3.

Figure 3 contains two graphs which together show the relationship between the choice of the number of generations to be run and the total number of individuals that need to be processed, $I(M,i,z)$, in order to yield a solution to every test problem with 99% probability for a population of different sizes (see Table 3). The horizontal axis applies to both of these overlaid graphs and runs between 0 and *Number of generations* from Table 3. The rising curve is the cumulative probability $P(M,i)$ of success and is scaled by the left vertical axis running between 0% and 100%. The falling curve shows, by generation, the total number of individuals $I(M,i,z)$ that must be processed in order to solve the problem with $z = 99\%$ probability, and is scaled by the right vertical axis. Both curves from each graph are based on the average results over 30 runs.

Summarized results at the end of the search process are given in Table 4.

Table 4 shows that a BSTGP technique is able to find the solutions in more runs compared to GP for all test problems. More than that, the average number of generations needed for obtaining a solution is smaller in BSTGP case (in 7 out of 8 cases).

6 Strenghts and weaknesses

The advantages and weaknesses of the proposed method are discussed in this section.

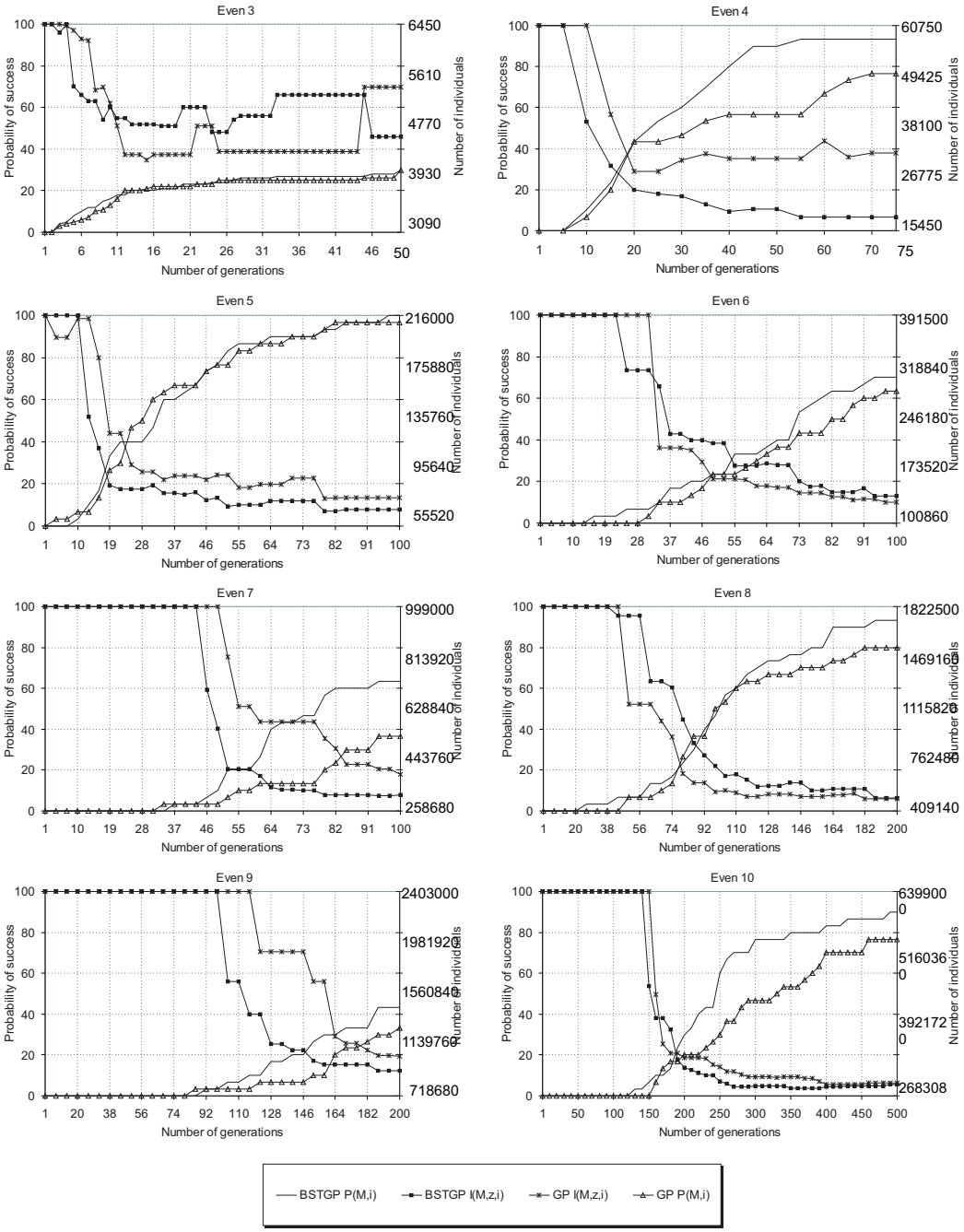

Figure 3. The computational effort and the cumulative probability of success for the test problems.

Table 4. Results obtained by BSTGP and GP for all test problems. For each problem is given the success rate and the average number of generations that are needed for obtaining a solution (only the successful runs are counted here). The results are averaged over 30 runs.

Problem	Success rate		Avg no of generations	
	BSTGP	GP	BSTGP	GP
Even 3	95%	89%	12.93	15.39
Even 4	100%	77%	28.23	40.50
Even 5	100%	97%	36.07	35.93
Even 6	70%	63%	71.03	76.53
Even 7	67%	43%	76.47	89.50
Even 8	93%	80%	108.27	118.70
Even 9	43%	33%	175.17	185.33
Even 10	90%	83%	270.97	335.23

6.1 Strenghts

One of the benefits of BSTGP is the reduced size of trees compared to the standard GP. This leads to a faster search in the solution's space.

Keeping only the best subtree might be an effective way of fighting bloat [2, 10, 12].

6.2 Weaknesses

There are some small difficulties related to the proposed method. A (small) extra amount of memory is required for storing the fitness of each subtree. This space is equal to the number of subtrees of each tree. Than, there is a small overhead when the nodes not belonging to the best subtree are deleted. This overhead is not significant due to the vectorial representation of trees [9].

7 Discussions and further work

Our numerical experiments have been focused on comparing BSTGP with standard GP. This is why we have run only the instances which have a resonable running time on a personal computer.

We have not been interested in obtaining solutions for higher-order instances of the problem. However, the proposed method could be improved and tested for higher instances by implementing some other features: sub-symbolic node representation [20], smooth genetic operators [18].

Poli [20] suggested another set of improvements which can help us to obtain solution for higher-order instances of the even-n-parity problem:

- a parallel implementation in which the sub-populations or demes are distributed over a number of workstations [20],
- involving sub-machine code GP [19] - a technique which allows the parallel evaluation of 32 or 64 fitness cases per program execution.
- variation of one or more of the parameters (as crossover or mutation probability or the initial maximal depth of a tree). This will help us to optimise performance for a specific value of n [20]. However, finding optimal values for these parameters requires multiple trials.

These improvements will be implemented in the BSTGP technique in the near future.

8 Conclusions

A new way of selecting the subtree providing the solution of GP chromosomes was investigated in this paper.

Instead of using the root of the tree as solution provider, any other subtree was seen as a potential solution of the problem. There is neither practical nor theoretical evidence that one of these subtrees is better than the others. Moreover, Wolpert and McReady [23, 22] proved that we cannot use the search algorithm's behaviour so far for a particular test function to predict its future behaviour on that function. Thus, fixing the subtree which provides the solution might not be the best strategy to follow. This is why we have designed a dynamic strategy for selecting the subtree providing the solution of the problem.

Nodes not belonging to the optimal subtree are removed. This has improved the efficiency of the method.

Several numerical experiments have been performed by using eight small instances of the even-n-parity

problems (from *Even* 3 up to *Even* 10). Results have shown the effectiveness of our approach.

In this paper we have analyzed the performance of the BSTGP technique only for even-n-parity problems. In order to have a complete assessment of the method we must test it against other problems (such as regression or classification problems). We also have to embed this strategy into other variants of Genetic Programming in order to see if it is a general one or it works only due to the particularities of the tree-based GP.

References

[1] D. Andre and A. Teller. A study in program response and the negative effects of introns in genetic programming. *Genetic Programming*, pages 12–20, 1996.

[2] T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation*, pages 33–38, 1994.

[3] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE-EC*, 5(1):17–26, 2001.

[4] K. Chellapilla. A preliminary investigation into evolving modular programs without subtree crossover. In J. R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 23–31. Morgan Kaufmann, 1998.

[5] C. A. C. Coello, E. Alba, G. Luque, and A. H. Aguirre. Comparing different serial and parallel heuristics to design combinational logic circuits. In *Evolvable Hardware*, pages 3–12. IEEE Computer Society, 2003.

[6] C. Gathercole and P. Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In J. R. Koza et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127. Morgan Kaufmann, 1997.

[7] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[8] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[9] J. R. Koza, D. Andre, F. H. Bennett III, and M. Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufman, 1999.

[10] W. Langdon. Quadratic bloat in genetic programming. *Proceedings of GECCO-2000*, pages 451–458, 2000.

[11] J. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127, 1991.

[12] N. McPhee and R. Poli. *A Schema Theory Analysis of the Evolution of Size in Genetic Programming with Linear Representations.* Springer, 2000.

[13] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesiangenetic programming approach. In W. Banzhaf et al., editors, *Proceedings of GECCO*, volume 2, pages 1135–1142. Morgan Kaufmann, 1999.

[14] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits—part I. *Genetic Programming and Evolvable Machines*, 1(1–2):7–35, 2000.

[15] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Proceedings of European Conference on Genetic Programming (EuroGP)*, volume 1802 of *LNCS*, pages 121–132. Springer, 2000.

[16] O. Muntean, L. Diosan, and M. Oltean. Best subtree genetic programming. In *GECCO 2007*, 2007.

[17] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. *Advances in Genetic Programming*, 2:111–134, 1996.

[18] J. Page, R. Poli, and W. B. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In R. Poli et al., editors, *Genetic Programming: Second European Workshop EuroGP'99*, pages 39–48. Springer, 1999.

[19] R. Poli and W. B. Langdon. Sub-machine-code genetic programming. In L. Spector et al., editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press, 1999.

[20] R. Poli and J. Page. Solving high-order boolean parity problems with smooth uniform crossover,sub-machine code GP and demes. *Genetic Programming and Evolvable Machines*, 1(1–2):37–56, 2000.

[21] A. Stoica, R. S. Zebulum, X. Guo, D. Keymeulen, M. I. Ferguson, and V. Duong. Silicon validation of evolution-designed circuits. In *Evolvable Hardware*, pages 21–25. IEEE Computer Society, 2003.

[22] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[23] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[24] X. Yao, Y. Liu, and G. Lin. Evolutionary Programming made faster. *IEEE-EC*, 3(2):82, 1999.