A Novel Efficient Mutation for Evolutionary Design of Combinational Logic Circuits

Francisco A.L. Manfrini[1,2], Heder S. Bernardino[1], and Helio J.C. Barbosa[1,3(✉)]

[1] Universidade Federal de Juiz de Fora (UFJF), Juiz de Fora, MG, Brazil
heder@ice.ufjf.br
[2] Instituto Federal do Sudeste de Minas Gerais, Juiz de Fora, MG, Brazil
francisco.manfrini@ifsudestemg.edu.br
[3] Laboratório Nacional de Computação Científica (LNCC), Petrópolis, RJ, Brazil
hcbm@lncc.br
http://www.hedersb.uk.to
http://www.lncc.br/hcbm

Abstract. In this paper we investigate evolutionary mechanisms and propose a new mutation operator for the evolutionary design of Combinational Logic Circuits (CLCs). Understanding the root causes of evolutionary success is critical to improving existing techniques. Our focus is two-fold: to analyze beneficial mutations in Cartesian Genetic Programming, and to create an efficient mutation operator for digital CLC design. In the experiments performed the mutation proposed is better than or equivalent to traditional mutation.

Keywords: Cartesian genetic programming · Point mutation operator · Circuit design · Combinational circuits

1 Introduction

The design of circuits is an important research field and the corresponding optimization problems are complex and computationally expensive. The design of a Combinational Logic Circuits (CLC) is based on the data from a truth table that lists all possible combinations of input logic levels with the corresponding output logic level. Given a certain truth table, it is possible to identify a CLC that meets the conditions prescribed by the truth table using traditional techniques and/or metaheuristics [5,9,14,16].

Several strategies for the design of combinational circuits have been reported [2,5,6,9,14,16]. The aim of these approaches is to find a functional solution, and to minimize the number of gates. Nowadays, CGP (Cartesian Genetic Programming) [15] is one of the most efficient methods for evolutionary design and optimization of digital combinational circuits [16,21]. CGP is a genetic programming technique in which the programs are modeled as directed acyclic graphs (DAG) and, thus, a large number of computational structures can be easily represented, such as CLCs [17]. That graph is represented by a matrix of potentially connected elements.

J. Handl et al. (Eds.): PPSN XIV 2016, LNCS 9921, pp. 665–674, 2016.
DOI: 10.1007/978-3-319-45823-6_62

The literature shows that different function sets are used in the evolutionary design. Koza [12] designed circuits using a small set of gates $\Gamma = \{and, or, not\}$. Miller *et al.* [15,17] and, recently, Goldman and Punch [10,11] used 4 types of gates $\Gamma = \{and, or, nand, nor\}$. Coello *et al.* [2–4,9] used 5 types of gates $\Gamma = \{and, not, or, xor, wire\}$. In [8], Gajda expanded the set of functions and used 9 types of gates $\Gamma = \{and, or, not, nand, nor, xor, wire, c_0, c_1\}$ where *not* and *wire* are unary functions (taking the first input of the gate) and c_k is a constant generator with the value k.

Understanding how search operators interact with solution representation is a critical step in order to design new techniques for improved search. There have been a number of previous studies into various aspects of GP evolution. For instance, [10,11] created methods to prevent wasted CGP evaluations and methods to overcome CGP's search limitations imposed by genome ordering [13].

The remainder of this paper is organized as follows: Sect. 2 summarises Cartesian Genetic Programming while Sect. 3 describes the proposed ideas. The computational experiments are presented in Sect. 4, where the obtained results are compared to those from the literature. Section 5 presents some discussions and, finally, Sect. 6 concludes the paper.

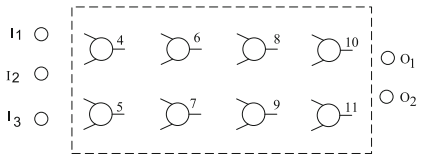
2 Cartesian Genetic Programming

In 1999, Miller [15] proposed a new form of Genetic Programming, called Cartesian Genetic Programming, in which the programs are modeled as directed acyclic graphs (DAG). Recently, [19] presented a CGP method that encodes programs via cyclic graphs. CGP provides a great generality enabling the representation of neural networks, circuits, and other computational structures [17]. Some features can be highlighted:

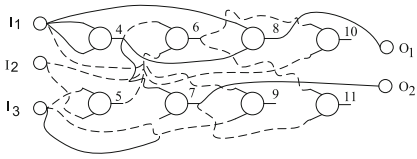
- CGP represents an individual using a matrix of processing nodes.
- Nodes contain genes describing what function they perform and how they are connected to other nodes.
- DAGs are represented by a collection of nodes connected by directed edges.
- CGP has three parameters associated to the representation and mapping process: the number of columns, the number of rows, and *levels-back*. *levels-back* controls the connectivity of the graph by constraining which columns a node can get its inputs from.
- Offspring are created by means of mutation.
- Offspring replace parents when they are better or have the same fitness value.
- The most common form of CGP uses a $(\mu + \lambda)$ reproduction strategy, where μ parents generate λ offspring and then, from the $(\mu + \lambda)$ individuals, the top μ are taken to be parents in the next generation.

Figure 1a shows an example of the matrix representation adopted by CGP, where I_1, I_2, I_3 are the primary inputs, O_1, O_2 are outputs, and each node represents an operation or its function (*or, if, switch*, ...). Figure 1b shows an example where *number of columns* = 4, *number of rows* = 2, and *levels-back* = *number*

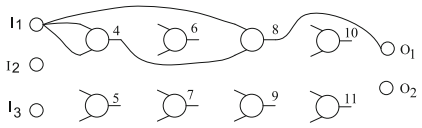
of columns. The nodes can have their inputs connected to the outputs of any nodes in the columns to the left of the current one or to a primary input. In this case, nodes 5, 6, 9, 10, and 11 are neutral, having no influence on the phenotype. These nodes are referred to as *inactive*. When a node is connected to an output (directly or indirectly), it is called *active*, as nodes 4, 7, and 8. Inactive nodes allow for genetic drift, as individuals can be mutated without changing their fitness. Figures 1c and d show phenotypes associated with the outputs O_1, and O_2, respectively.



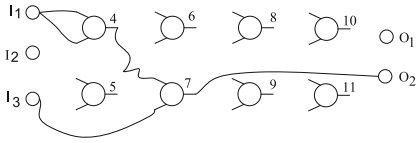
(a) 2x4 CGP representation matrix with 3 inputs and 2 outputs.



(b) CGP representation of an individual.



(c) Phenotype output O_1.



(d) Phenotype output O_2.

Fig. 1. CGP representation

Modern CGP practice has mostly done away with *rows* and *levels-back* in favor of *rows* = 1 and *levels-back* as the sum of the number of columns with the number of inputs.

2.1 Single Active Mutation (SAM)

CGP's usual variation operator is a point mutation. However, different implementations can be found in the literature, making it hard to define a standard version of the algorithm. For instance, in some papers [15] this operator chooses a set number of genes at random to be mutated, while in other papers [20] each gene can be mutated with a certain probability, allowing any number of genes to be mutated at once.

When mutations occur in non-coding sections of the genotype, no modification will appear in the phenotype and, consequently, both individuals (mutated and non-mutated) have the same fitness. In order to avoid this situation Goldman and Punch [10] proposed a method in which a single active gene is modified every time an offspring is generated. This alternative will be referred to as "SAM" here. SAM's iterative process generates an offspring by mutating randomly selected genes until an active gene is changed. When this mutation operator is used, one

can see that: (i) one active gene is mutated, (ii) inactive genes may be changed, and (iii) no mutation rate needs to be specified by the user. As SAM achieved the best results in the hardest test-problem from [10], here we adopted this mutation operator as a baseline for all the computational experiments.

3 Description of the Biased Single Active Mutation

Traditionally, the mutation operator used in CGP is a point mutation operator, in which a randomly chosen position in the matrix representation is replaced by another randomly selected value. As the elements of the matrix are composed by a function/operation and its inputs, two different modifications can occur. When a function is chosen for gene mutation, then a valid value is the address of any function in the function set (here called mutation type gate), whereas if an input gene is selected to be modified, then a valid value is the address of the output of any previous node in the genotype, or of any program input.

The proposed approach is based on the idea of analyzing the behavior of the genotype during the evolutionary process for a given set of problems. Based on this analysis, we create a bias to help direct gene mutation when applied to other problems. For each run, every time the child has a fitness value better than that of its parent (the child proceeds to the next generation), we say a beneficial mutation occurred. Every time such improvement occurs, we check whether this mutation occurred on the function executed by the parent. In this case, we store the new and beneficial transition, from the previous (in the parent) to the new function (in the child).

At the end of the evolutionary process, we create the frequency table of all transitions, giving rise to a probability distribution. The creation of the probabilities transition matrix is illustrated in Fig. 2. This probability distribution is utilized to guide the evolutionary process. Every time a function is to be changed, this probability distribution will be used. This new mutation operator proposed here is referred to as *biased* SAM.

Figure 3 shows an example of the biased mutation, where a gate *and* is selected to be mutated. The new value of that node is chosen according to the probabilities present in the transition probabilities matrix. In this example, the new gene in the child is a *nor* gate.

4 Case Study

4.1 Analysis of the Evolution

Initially four benchmark problems, taken from [1], were chosen where the success of the mutations applied during the evolutionary process is studied as explained in Sect. 3. We used the expanded set of functions as in [8]: $\Gamma = \{and, or, not, nand, nor, xor, wire, c_0, c_1\}$, We also used $\mu = 1$, $\lambda = 4$, *number of rows*=1, *number of columns*=100, and *levels-back* = *number of columns*, as in [16].

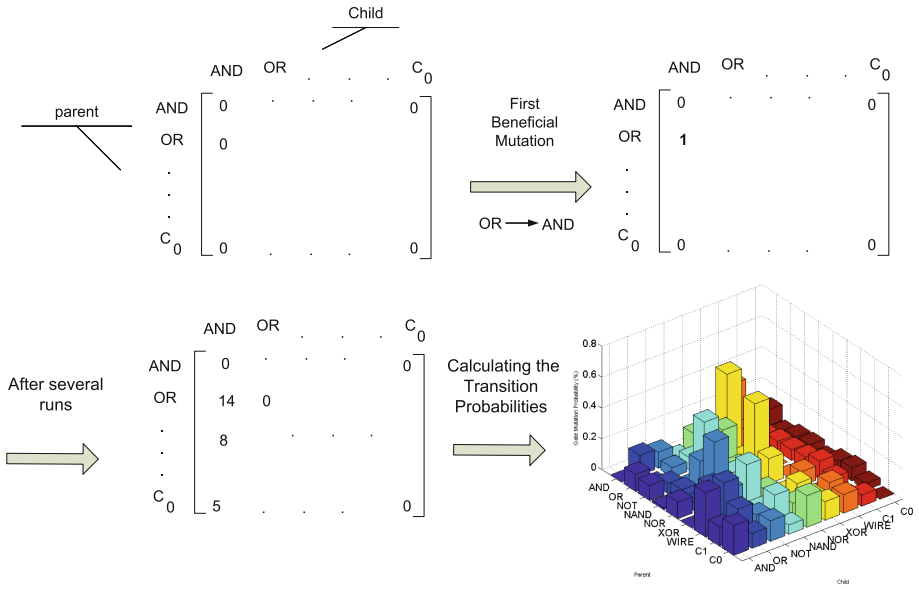


Fig. 2. Illustration of the generation of the transition probabilities matrix.

The test-problems are defined as:

Circuit 1: The first problem has four inputs and one output. The set F indicates the rows of the truth table in which the outputs are equal to one: $F = \{0, 1, 3, 6, 7, 8, 10, 13\}$.

Circuit 2: The second problem has five inputs, one output, and $F = \{2, 3, 6, 7, 10, 11, 13, 15, 18, 19, 21, 23, 25, 27, 29, 31\}$.

Circuit 3: The third problem has four inputs, three outputs, and $F_1 = \{0, 5, 10, 15\}$; $F_2 = \{1, 2, 3, 6, 7, 11\}$; $F_3 = \{4, 8, 9, 12, 13, 14\}$.

Circuit 4: The fourth problem has five inputs, three outputs, and $F_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 28, 29, 30, 31\}$; $F_2 = \{0, 2, 4, 6, 7, 8, 10, 12, 14, 15, 16, 18, 20, 22, 23\ 24, 26, 28, 30, 31\}$; $F_3 = \{4, 5, 12, 13, 20, 21, 28, 29\}$.

A hundred independent runs were performed and the algorithm is terminated when a correct circuit is found or the maximum number of evaluations is reached; here, 100000 evaluations are allowed.

For each beneficial mutation, the exchanges are stored and frequency of occurrence of each exchange will be used in order to build a matrix of transition probabilities. Figure 4 presents a bar plot of the values stored in that matrix at the end of the analysis. Notice that Fig. 3 shows one particular case: the probabilities for the *and* gate of a parent. The matrix of transition probabilities will be used to guide mutation (biased mutation) in other problems as will be seen in Sect. 4.2.

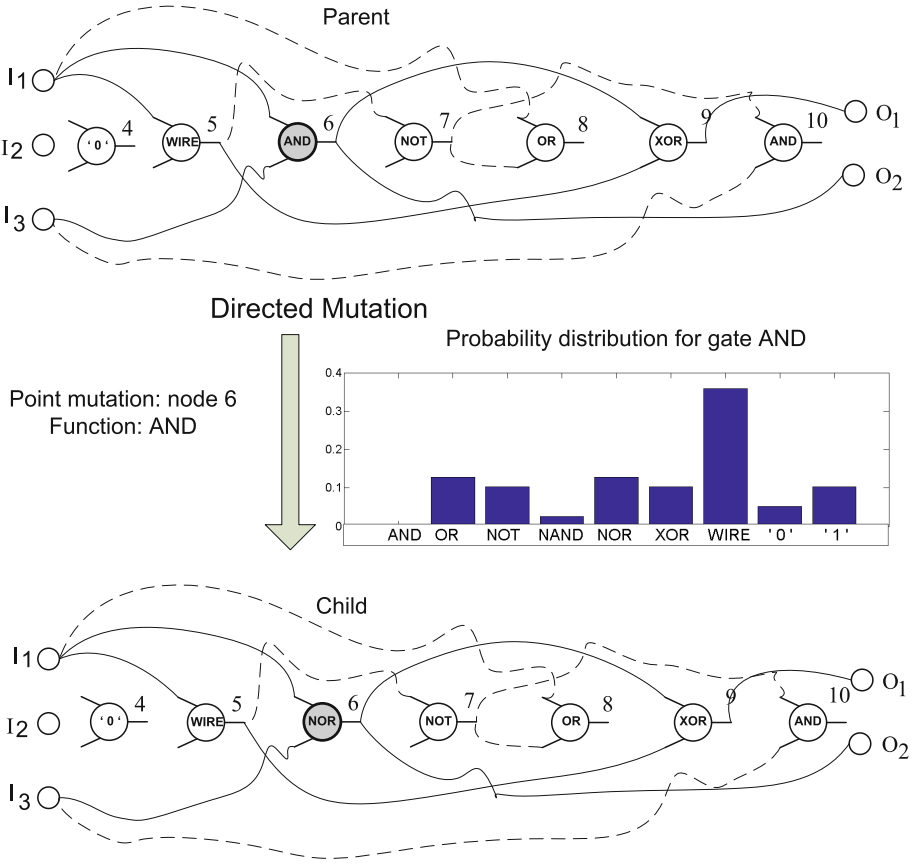


Fig. 3. Example of the biased mutation operator. When a gate (function) is selected to be mutated, then it is replaced by another one using a roulette wheel defined by the transition probabilities matrix obtained by counting the beneficial mutations.

4.2 Designing a Combinational Logic Circuit

For a comparative study four benchmark problems studied by Goldman and Punch [11] and widely used in the electronics literature [7,18] were chosen to verify the effectiveness of our approach. All experiments were implemented in MATLAB and for statistical analysis we used SPSS. The following values were calculated and used in the comparisons: the number of times a feasible solution is found (we call it a "hit"), the median of the number of objective function evaluations required to obtain a feasible solution (called here "MES"), and the number of beneficial mutations per thousand evaluations performed. Notice that larger values of this ratio indicate a smaller number of objective function evaluations unnecessarily wasted and, consequently, an increase in the performance of the method.

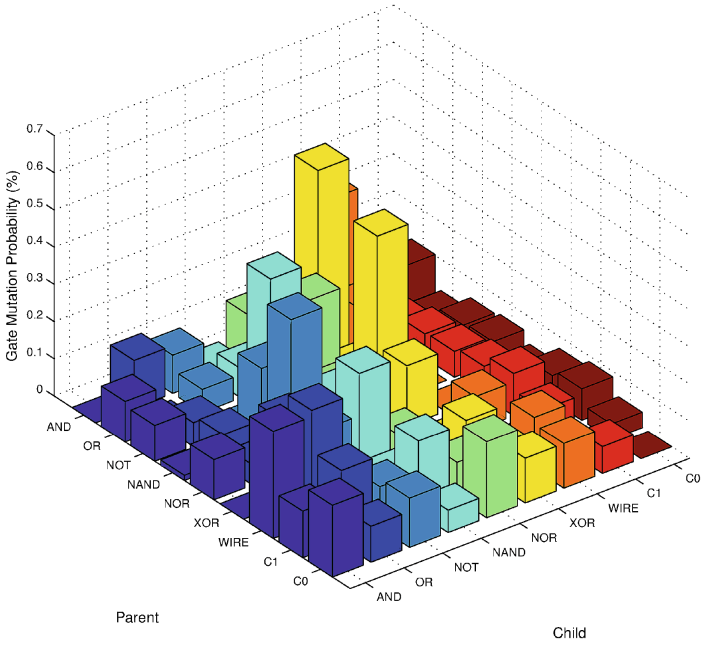


Fig. 4. Values in the transition probabilities matrix obtained using the four benchmark problems shown in Sect. 4.1, considering the beneficial mutations from parent to child gates.

For all test-problems, we used $\mu = 1$, $\lambda = 4$, performed 51 independent runs, and adopted the function set $\Gamma = \{and, or, not, nand, nor, xor, wire, c_0, c_1\}$. Also, for each problem, we employed the same number of nodes used by Goldman and Punch [11]. To ensure that a feasible solution is always found at the end of a run, a sufficiently large number of function evaluations (5000, 500000, 500000, 1000000, respectively for problems 1, 2, 3, and 4) is pre-defined for each problem. Goldman and Punch [11] were able to solve those problems in 95 % of the runs with 1487, 42278, 74939, and 611034, evaluations respectively. The maximum number of evaluations allowed here are at least 60 % higher than those required in [11] to solve the problem.

The results of each type of mutation are shown in Table 1. The first line for each problem shows the control configuration corresponding to SAM, as described in Sect. 2.1.

Problem 1: The first problem, 3-Bit Parity, is considered very simple, but is the most common test-problem in the CGP literature [11,17,22,23] and may help understand how the mutation changes affect the results. The topology configuration was *number of rows* = 1, *number of columns* = 500, as chosen in [11]. It can be seen that the biased mutation converges to a feasible solution with a smaller number of evaluations. The proposed technique obtained MES = 269 while the control configuration has MES = 413.

Table 1. Comparison between standard and biased SAM for the four test-problems used here. "Hits" represents the number of times that a given approach found a feasible solution. The number of times that a beneficial mutation occurs every 1000 evaluations is denoted by "Beneficial Mutation/1000 *evals*". "MES" is Median Evaluations to Success. The *p*-values calculated with the Mann-Whitney U test over the MESs are also presented.

Circuit	Single active mutation	Hits (%)	Beneficial mutation/1000 *evals*	MES MES	Confidence interval	*p*-value
Bit	Standard	98	15.7	413	285 .. 469	–
Parity	Biased	100	15.9	269	189 .. 393	0.049
16 to 4 bit	Standard	100	1.6	19473	16673 .. 22953	–
encoder	Biased	100	1.9	16153	14525 .. 20837	0.103
4 to 16 bit	Standard	90	1.0	332813	293501 .. 360637	–
decoder	Biased	100	2.1	165665	145681 .. 184161	0
3-Bit	Standard	76	1.3	559385	464745 .. 744909	–
Multiplier	Biased	88	1.6	435781	382125 .. 550645	0

Problem 2: The second problem is the 16-4 bit encoder, which can be found in [10,11]. The topology configuration was *number of rows* = 1, *number of columns* = 2000, as chosen in [11]. The proposed technique obtained MES = 15951 while the control configuration has MES = 19469.

Problem 3: The third problem, is the 16-4 bit decoder proposed in [10,11]. The topology configuration was *number of rows* = 1, *number of columns* = 1000, as chosen in [11]. The proposed technique obtained MES = 161889 while the control configuration has MES = 325193.

Problem 4: The fourth problem is the 3-bit multiplier, which can be found in [10,11]. The topology configuration was *number of rows* = 1, *number of columns* = 5000, as in [11]. This problem is very difficult by comparison to the other ones. The proposed technique obtained MES = 430487 while the control configuration has MES = 559385.

5 Discussion of the Results

Analyzing the transition probability matrix extracted from the four benchmark problems in Sect. 4.1, one can see that the most important modification during mutations is to change a *nand* gate in the parent to a *xor* gate. On the other hand, relatively fewer cases were observed in which a beneficial mutation arises from changing a *not* gate into an *or* gate. Thus, the reinforcement of the occurrence of the first exchange, and the avoidance of the second one can potentially improve the performance of the algorithm.

It is interesting to note in Fig. 4 that there is no gate to be preferable in the mutation for all cases. The probability of the transition varies with the gate to be modified. For instance, the *xor* gate has a higher probability value when *nand* or

or gates are being modified while it presents a lower probability of improvement when replacing *wire* or *not* gates.

When the entire transition probabilities matrix is considered in the search, the results (presented in Sect. 4.2) are better than those obtained by the baseline (SAM) version, for all test-problems used here. Thus, one can see that the extraction of knowledge is possible and useful in improving the performance of CGP.

Finally, notice that beyond the decrease in the number of objective function evaluations to reach a feasible solution, the ratio between the average number of beneficial mutations and the average number of evaluations to success increased, showing that the efficiency of the search is also improved for all problems tested.

6 Conclusions

This paper proposed a new mutation for automatic design of combinational logic circuits via Cartesian Genetic Programming. Through the analysis of the evolutionary process in a given set of problems it was possible to gain knowledge and then use it to guide the search. The incorporated knowledge about the performance of the mutation operator constitutes an important step towards increasing the power of CGP as a design tool.

Experimental results confirmed the superiority of the new biased mutation operator over a standard mutation in reducing the number of fitness evaluations in the design of combinational logic circuits.

Nevertheless, a more in-depth study of the evolutionary mechanisms and beneficial mutations remains as a promising research area. The rationale behind the design of the biased Single Active Mutation applied here to circuit design, is not restricted to this type of application; other design domains can be investigated.

Acknowledgment. The authors would like to thank the support provided by CNPq (grant 310778/2013-1), FAPEMIG (grants APQ-03414-15 and PEE-00726-16), and PPGMC/UFJF.

References

1. Alba, E., Luque, G., Coello Coello, C.A., Hernández Luna, E.: Comparative study of serial and parallel heuristics used to design combinational logic circuits. Optim. Methods Softw. **22**(3), 485–509 (2007)
2. Coello, C.A.C., Aguirre, A.H., Buckles, B.P.: Evolutionary multiobjective design of combinational logic circuits. In: Proceedings of 2nd NASA/DoD Workshop on Evolvable Hardware, pp. 161–170. IEEE (2000)
3. Coello, C.A.C., Alba, E., Luque, G.: Comparing different serial and parallel heuristics to design combinational logic circuits. In: Proceedings of NASA/DoD Conference on Evolvable Hardware, pp. 3–12. IEEE (2003)
4. Coello, C.A.C., Christiansen, A.D., Aguirre, A.H.: Use of evolutionary techniques to automate the design of combinational circuits. Int. J. Smart Eng. Syst. Des. **2**, 299–314 (2000)

5. Coello, C.A.C., Luna, E.H., Hernández-Aguirre, A.: Use of particle swarm optimization to design combinational logic circuits. In: Tyrrell, A.M., Haddow, P.C., Torresen, J. (eds.) ICES 2003. LNCS, vol. 2606, pp. 398–409. Springer, Heidelberg (2003)
6. Coello, C.A.C., Zavala, R.L., García, B.M., Hernández-Aguirre, A.: Ant colony system for the design of combinational logic circuits. In: Miller, J.F., Thompson, A., Thompson, P., Fogarty, T.C. (eds.) ICES 2000. LNCS, vol. 1801, pp. 21–30. Springer, Heidelberg (2000)
7. Ercegovac, M.D., Moreno, J.H., Lang, T.: Introduction to Digital Systems. Wiley, Hoboken (1998)
8. Gajda, Z., Sekanina, L.: An efficient selection strategy for digital circuit evolution. In: Tempesti, G., Tyrrell, A.M., Miller, J.F. (eds.) ICES 2010. LNCS, vol. 6274, pp. 13–24. Springer, Heidelberg (2010)
9. García, B.M., Coello, C.A.C.: An approach based on the use of the ant system to design combinational logic circuits. Mathw. Soft Comput. **9**(3), 235–250 (2002)
10. Goldman, B.W., Punch, W.F.: Reducing wasted evaluations in cartesian genetic programming. In: Krawiec, K., Moraglio, A., Hu, T., Etaner-Uyar, A.Ş., Hu, B. (eds.) EuroGP 2013. LNCS, vol. 7831, pp. 61–72. Springer, Heidelberg (2013)
11. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programming's evolutionary mechanisms. IEEE Trans. Evol. Comput. **19**(3), 359–373 (2015)
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection, vol. 1. MIT Press, Cambridge (1992)
13. Luke, S., Panait, L.: A comparison of bloat control methods for genetic programming. Evol. Comput. **14**(3), 309–344 (2006)
14. Manfrini, F., Barbosa, H.J.C., Bernardino, H.S.: Optimization of combinational logic circuits through decomposition of truth table and evolution of sub-circuits. In: IEEE Congress on Evolutionary Computation (CEC), pp. 945–950 (2014)
15. Miller, J.F.: An empirical study of the efficiency of learning Boolean functions using a cartesian genetic programming approach. In: Proceedings of Genetic and Evolutionary Computation Conference, vol. 2, pp. 1135–1142 (1999)
16. Miller, J.F.: Cartesian genetic programming. Springer, Berlin (2011)
17. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Trans. Evol. Comput. **10**(2), 167–174 (2006)
18. Tocci, R.J., Widmer, N.S., Moss, G.L.: Digital Systems. Pearson, Upper Saddle River (2011)
19. Turner, A.J., Miller, J.F.: Recurrent cartesian genetic programming. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 476–486. Springer, Heidelberg (2014)
20. Turner, A.J., Miller, J.F.: Neutral genetic drift: an investigation using cartesian genetic programming. Genet. Program. Evol. Mach. **16**(4), 531–558 (2015)
21. Vasicek, Z.: Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates. In: Machado, P., et al. (eds.) EuroGP 2015. LNCS, vol. 9025, pp. 139–150. Springer, Berlin (2015)
22. Walker, J.A., Miller, J.F.: The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. IEEE Trans. Evol. Comput. **12**(4), 397–417 (2008)
23. Yu, T., Miller, J.F.: Neutrality and the evolvability of boolean function landscape. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, p. 204. Springer, Heidelberg (2001)