Journal of AI and Data Mining
Vol 6, No 2, 2018, 355-363
DOI: 10.22044/JADM.2017.1059

A Fast and Self-Repairing Genetic Programming Designer for Logic Circuits

A. M. Mousavi [1*] and M. Khodadadi [2]

1. Department of Electrical Engineering, Lorestan University, Khoramabad, Lorestan, Iran.
2. Department of Electrical Engineering, Azad University, Arak Branch, Arak, Iran.

Received 03 June 2016; Revised 05 November 2016; Accepted 31 May 2017
**Corresponding author: mousavi.m@lu.ac.ir (A..Mousavi).*

Abstract
Usually the important parameters in the design and implementation of combinational logic circuits are the number of gates, transistors, and levels used in the design of a circuit. In this regard, various evolutionary paradigms with different competency have recently been introduced. However, while being advantageous, evolutionary paradigms also have some limitations including a) lack of confidence in reaching the correct answer, b) long convergence time, and c) restriction on the tests performed with a higher number of input variables. In this work, we implement a genetic programming approach that given a Boolean function, outputs an equivalent circuit such that the truth table is covered, and the minimum number of gates (and to some extent, transistors and levels) are used. Furthermore, our implementation improves the aforementioned limitations by incorporating a self-repairing feature (improving limitation a); efficient use of the conceivable coding space of the problem, which virtually brings about a kind of parallelism and improves the convergence time (improving limitation b). Moreover, we apply our method to solve the Boolean functions with a higher number of inputs (improving limitation c). These issues are verified through multiple tests, and the results obtained are reported.

Keywords: *Genetic Programming, Logic Circuits, Design, Optimization.*

1. Introduction

With the emergence of new methods for the optimization problems, the research works in the design of combinatorial logic circuits have also gained a boost. This trend has paved the arena for entering evolutionary paradigms as one of the successful models for solving the optimization problems, in general, and optimization of combinational logic circuits, in particular.

In this paper, we present a method based upon genetic programming that efficiently utilizes the coding space of the logical circuits to accelerate the convergence time of the solutions. As a result, the cases we tested led to less-gate designs (with smaller number of transistors and levels) compared to the earlier works. The distinguishing facets of our approach are the utilization of a new encoding for the logical circuits and self-repairing ability so that the program will be able to recover from incomplete answers. These are utilized along with a proper evaluation function and selection strategy. Applying these features to several test cases shows that the proposed approach is able to achieve satisfactory results in terms of the usual design criteria.

Our work and its achievements will be presented as what follow. In Section 2, we briefly review some earlier works. Section 3 begins with a brief description of genetics programming (GP) and continues with our implementation for the design of combinational logic circuits including the appropriate coding, evolutionary operators, and so on. In Section 4, the results of applying our approach to a series of previously studied circuits as well as the new ones are reported. Finally, Section 5 closes the paper with conclusions and a few suggestions for further research.

2. Related works

One of the classical methods implemented to

simplify digital functions, as taught in textbooks, is the basic Boolean manipulation techniques. These methods, which mainly consist of factoring and removing variables, could lead to a rather straightforward approach of Karnaugh Maps [15]. While the Karnaugh Maps is effective in solving problems with a few variables, problems with more variables require computer-based approaches such as Quin maccLausky [16]. With an increase in the number of design variables and constraints, the complexity of the design process increases. This fact along with a growth in the use of the computational intelligence for the optimization problems has paved the way for applying evolutionary computation paradigms to the design of electronic and logical circuits [3,4]. Earlier works in applying the evolutionary procedures to the design of logical circuits goes back to the application of genetic algorithm (GA) and genetic programming (GP) [1,5], where the emphasis was on the mere generation of circuits rather than the optimization concerns. On the other hand, others performed a comparison between the evolutionary procedures in terms of their ability in convergence. For example, [13] and [14] compare GP and GA, and show that GA may prematurely converge to non-local optima, while GP has a greater chance to find the best solutions.

In [10], a so-called Cartesian Genetic Programming (CGP) method has been proposed for the design of combinatorial logical circuits. In this work, instead of using a tree structure, which is usually used in GP, arrays of strings are used for genotypes, which is more effective in achieving optimal solutions. In a similar manner to CGP, a methodology has been proposed in [12], which is more inclined toward the implementation of Boolean functions rather than focusing on the least gates designs. In [11], GP has been employed for the design of combinatorial logical circuits considering the least number of gates, transistors, and levels. However, there are some limitations. First, the only gates used are the NAND ones. Secondly, there is no concern what so ever regarding the rate of convergence of the program. Thirdly, the results obtained are only compared with the manual designs. Finally, the evaluations reported are limited to functions with four variable inputs.In the current work, our goal was to implement the logical functions with the minimum number of gates (and to some extent, transistors and levels). At the same time, we tried to improve some defects and shortcomings seen in similar works such as low convergence rate (number of generations to get to the answer), not reaching the desired design, high populations for achieving the desired results, and no full coverage of the truth table.

3. Implementation

3.1. Genetic programming

Genetic programming (GP) is one of the several evolutionary paradigms available for solving the optimization problems via computers. In this approach, first of all, an initial population of solutions or computer programs, each of which is a potential solution to the problem, is created. Then each of these candidate solutions is evaluated versus a so-called fitness function in order to measure its fitness. The more fit a solution is, the more chance is given for being selected in the next generation. Then through applying the cross-over and mutation operations, GP would produce a new generation of solutions from a previously elected one. These steps are repeated in GP until a convincing solution is obtained or a certain number of generations are reached.

Since how we encode the conceivable solution space would significantly affect the accuracy as well as the speed of convergence, in what follows we explain our encoding scheme.

3.2. Encoding

As mentioned in the very beginning, GP starts with a population of initial solutions that are usually generated randomly. Each of these candidate solutions is essentially a combinatorial circuit that is made up of a number of logical gates. In practice, each candidate solution circuits needs to be properly defined for the implementation of GP. This process is called encoding of the solution space. One way to do encoding is for each solution to be individually coded and entered into the GP search process. This creates a large number of independent small data chunks, where evolutionary operations such as cross-over, mutation, and evaluation of the candidate solutions would be independently applied to these small items. This approach results in a lengthy GP implementation. On the other hand, we could encode several logic circuits as a single candidate solution for the implementation of GP and aggregately apply the evolutionary operations. This approach speeds up the search cycle by introducing a kind of parallelism in the implementation. For this purpose, several

candidate solutions are gathered in a 2D array, where the number of rows determines the number of solutions and the number of columns determines the levels used in the resulting circuit. This is illustrated in figure 1 for a 5*5 array. In this figure, all the evolutionary operations are applied simultaneously to the 5 solutions. Each element of the array is itself a 1D array, where the numbers in the array indicate the type of gates used together with the inter-connections of their inputs and outputs to the other gates (1D array elements) in the main array. In this configuration, the input of each element is allowed to come only from the previous stage and the last column represents the possible outputs.

[2,0,3,0,2] [1,0,1,0,0] [2,2,4,1,5] [2,3,1,3,3] [5,1,2,2,2]

[2,0,5,0,5] [3,1,2,0,1] [2,2,3,1,5] [3,2,4,1,3] [4,3,2,0,2]

[4,0,1,0,4] [4,1,3,1,3] [5,2,5,2,0] [5,3,2,1,1] [2,3,2,3,0]

[3,0,4,0,2] [1,0,1,0,0] [3,2,4,0,2] [2,1,1,0,3] [2,4,2,4,0]

[4,0,4,0,3] [3,0,2,1,2] [1,2,5,0,2] [5,1,5,1,0] [4,1,5,4,5]

Figure 1. A typical array of 5 candidate solutions.

As it could be seen in figure 1, each element in the array is composed of five numbers with specific meanings. The first and second numbers represent the row and represent the row and column that one of the inputs of the gate should be connected, while the third and fourth numbers represent the row and column that the other input of the gate should be connected (based on two inputs gates). We interpreted the numbers 10, 20, 30, 40, … as the main input variables A, B, C, D, …. The fifth number shows the type of gate according to the equivalences: WIRE = 0, AND = 1, OR = 2, XOR = 3, NAND = 4, NOR = 5.

According to the configuration in figure 1 and depending on from which row of the last column we take the output, 5 combinatorial logic circuits are possible as the candidate solutions. For example, if we take the output from the first row of the last column, the circuit of figure 3a is obtained. In this figure, the element [5 1 2 2 2] in the first row and the fifth column is the output. This element is an OR gate whose first input comes from the output of the gate in the fifth row and the first column, while its second input comes from the output of the gate located in the second row and the second column. The situation is better visualized in figure 2, where the inter-connections for this OR gate is drawn up to the main input variables (A, B, C, …).

When this protocol is applied to all the rows of the last column, 5 circuits in figure 3 are obtained.

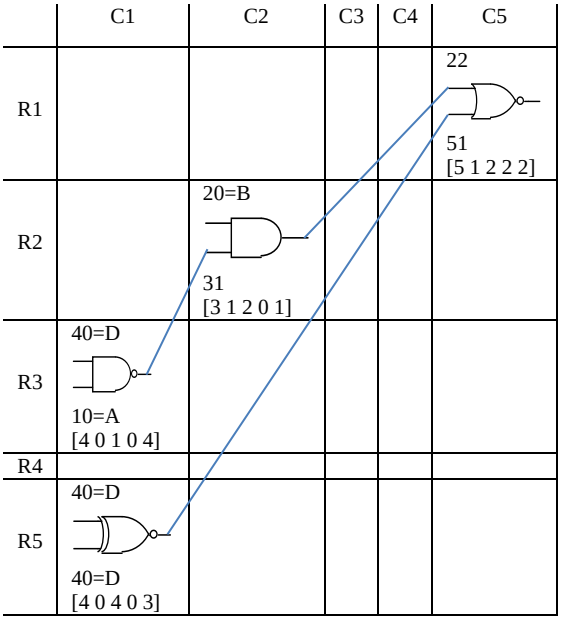

Figure 2. Circuit resulting from first row of figure 1.

3.3. Fitness function

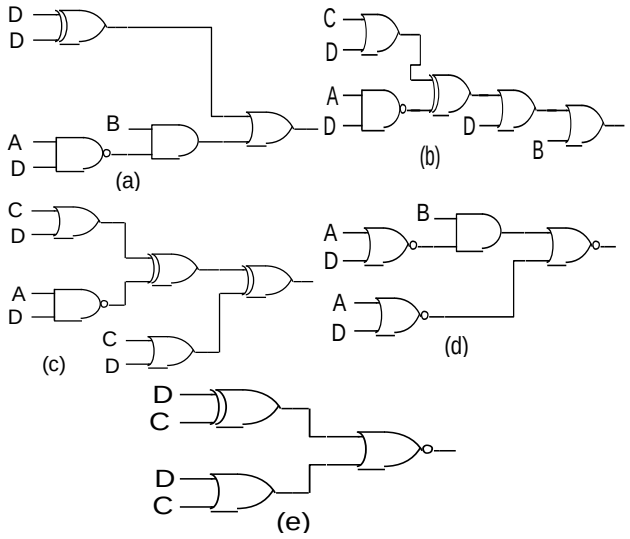

Figure 3. Equivalent circuits for array of figure 1. Last column and a) first row as output; b) second row as output; c) third row as output; d) fourth row as output; e) fifth row as output.

An important factor influencing the success of GP is an appropriate evaluation function that determines the fitness of possible solutions for their respective effectiveness in the next

population. According to the common criteria of optimality for combinational logic circuits including having minimum number of gates and maximum coverage of the truth table, it is natural to consider these criteria in the definition of the evaluation function. Accordingly, the evaluation function would be a weighted sum of the number of false results (compared to the truth table) and the number of gates as in (1).

$$FF = \frac{(10 \times \frac{errors}{2^{in}}) + (\frac{gates}{31})}{11} \quad (1)$$

In this equation, *in* is the number of inputs. E*rrors* is the number of errors of the output of the solution circuit with respect to the truth table. G*ates* is the number of gates used in the proposed circuit. The fitness function is designed so that its values are between 0 and 1. The more its value is close to zero, the more would be the fitness of the proposed solution. Nevertheless, we use the value 1-FF in our drawings such that the more close we get to the value of 1, the proposed solution would be more successful.

While the evaluation function in (1) targets solutions with the lowest number of gates, the number of columns in the array of solutions specifies the maximum number of levels of the solution. Therefore, we can use the number of columns in the array of solutions as a parameter whereby we allow it to vary between 1 and a maximum. For each value of the parameter for which the optimal solution is found such that it covers the truth table completely, we stop and the next values will not be tried. As a result, the maximum number of levels of the designs would be controllable. Note that for a specific choice for the number of columns, our approach still outputs circuits with less level than the number of columns. Thus in the evaluation tests that come in Section 4, we fix the number of columns to 5.

Therefore, with this approach and using the NAND, NOR, and XOR gates for which the number of employed transistors are compared in table 1, our implementation targets the circuits with a minimum number of gates, levels, and transistors.

3.4. Selection and evolution of generations

After the effectiveness of each chromosome in the initial population was calculated, better chromosomes of the population should be selected as the parents of the next generation. In order to maintain the diversity of the next generation, while keeping the elite chromosomes, the roulette wheel and elitism selection approaches are being used together to select the parents for the next population. Then the usual cross-over and mutation operations would be applied to create a new generation.

Table 1. CMOS gate characteristics.

Gate type code	Gate type	Gate symbol	Area (μm)	Number of transistor
0	Wire		0	**0**
1	NOT		1728	**2**
2	AND		2880	**6**
3	OR		2880	**6**
4	NAND		2304	**4**
5	NOR		2304	**4**
6	XOR		4608	**9**
7	XNOR		5184	**9**

4. Evaluation

In this section, we illustrate our implementation of GP and evaluate its performance via Boolean functions. We also compare our results with the related works.

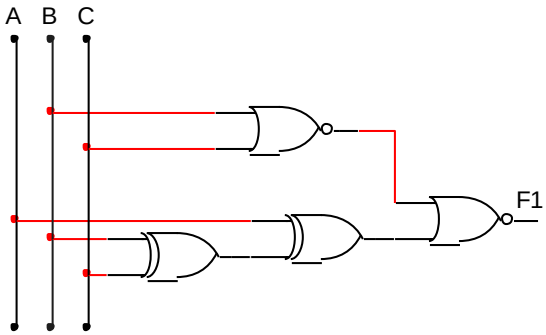

Figure 4. Circuit for F1.

4.1. Test Functions

Test 1: A function with three variables:

$F_1(A,B,C) = \sum (3,5,6)$

The optimized circuit is shown in figure 4. The cnvergence diagram for the circuit is shown in figure 5. The first point to note is a fast convergence process (in generation 10), as could be seen in figure 5. The second point is depicted in table 2. As it could be seen, using our approach, 4 gates and 26 transistors are used for the circuit in 3 levels.

Table 2 compares the results of other works. Note that the Human Designer 1 (HD1) uses the Karnaugh Maps plus Boolean algebra, whereas the Human Designer 2 (HD2) uses the Quine-McCluskey Procedure.

As it can be seen in this table, compared to the classic methods, the proposed method reduced the average number of gates and transistors by 44% and 35%, respectively, and also has some advantages over other approaches such as MGA [8] and NGA [6].

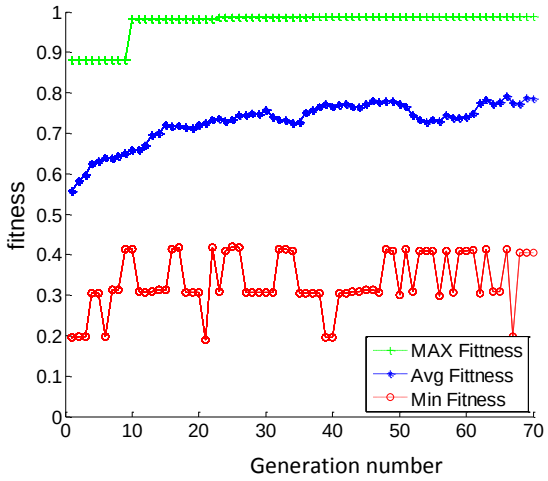

Figure 5. Convergence diagram for F_1.

Table 2. Comparison of proposed method vs. similar approaches in terms of general design parameters for function F_1.

Number of Levels	Number of transistors	Number of gates	Optimized function	Method
3	36	5	F=C.(A⊕B)+B.(A⊕C)	**HD1(KM)[8]**
4	35	6	F=A′.B.C+A.(B⊕C)	**HD2(QM)[8]**
4	32	5	F=(C+B).(B⊕(A⊕C))′	**NGA[6]**
3	27	4	F=(A+B).C⊕(A.B)	**MGA[8]**
3	26	4	F={[(B⊕C)⊕A]+(B+C)′}′	**our approach**

Test 2 : A function with 4 variables

$F_2(A,B,C,D)= \sum (0,1,3,6,7,8,10,13)$.

Figure 6 shows the designed circuit using the proposed approach.

As it could be seen in figure 6, the circuit is

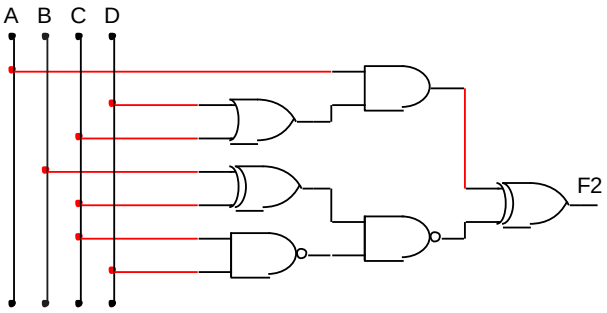

Figure 6. Circuit for F_2.

realized by 6 gates and 38 transistors in three levels. Table 3 (Human Designer 1 uses the Karnaugh Maps plus Boolean algebra to simplify the circuit, whereas Human Designer 2 uses the Sasao method [7]) also compares the results in terms of the optimal circuit parameters. Decreasing the number of gates, transistors, and levels is evident. The convergence diagram for the circuit for function F_2 is shown in figure 7. A fast convergence process is seen in generation 37, though due to the increase in the number of inputs and function's complexity, the process is lengthier compared to F_1.

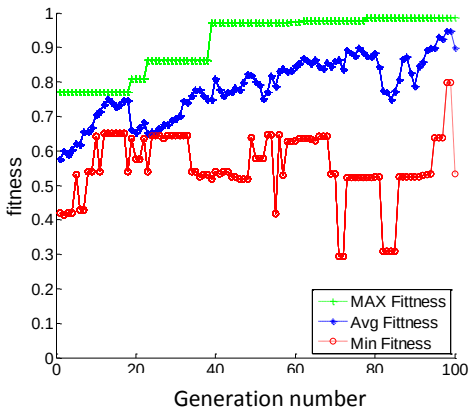

Figure 7. Convergence diagrams for function F_2.

Table 3. Comparison of proposed approach vs. other approaches in terms of general design parameters for function F_2.

Number of levels	Number of transistors	Number of gates	Optimized function	Method
4	56	11	F=((A´.C)⊕(B´.D´))+((C´.D´).(A⊕B´))	**HD1(KM)[8]**
5	65	12	F=C′⊕D′B′⊕CD′A′⊕C′D′B	**HD2(sasao)[8]**
5	61	10	F=(B.C′.D)⊕((B+D)⊕A⊕((C+D)+A)))´	**NGA[6]**
5	47	7	F=((B⊕(B.C))⊕((A+C+D)⊕A)´)	**MGA[8]**
3	38	6	F=((C+D).A)⊕((C.D)´.(C⊕B))´	**our approach**

***Test 3*:** A functions with 4 variables.
$F_3(A,B,C,D)= \sum (0,4,5,6,7,8,9,10,13,15)$.

The designed circuit for this function using the proposed approach is shown in figure 8. The circuit is implemented by 5 gates and 30 transistors in 4 levels. Table 4 shows the comparison results as well.

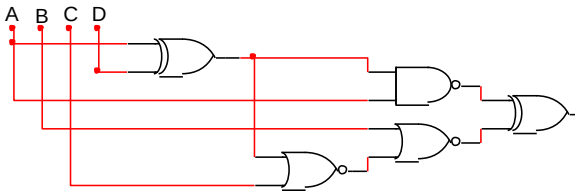

Figure 8. Circuit for F_3.

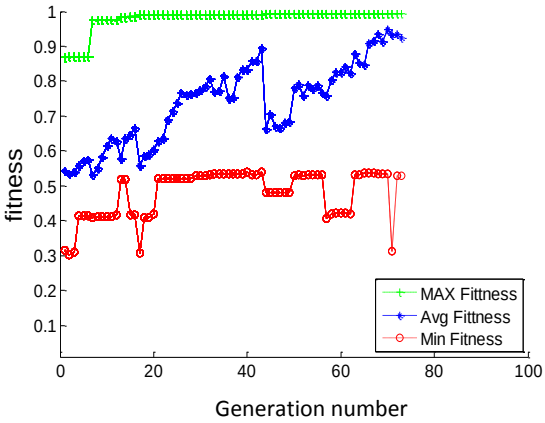

Figure 9. Convergence diagram for F_3.

Table 4. Comparison of proposed method vs. similar approaches in terms of general design parameters for function F_3.

Number of levels	Number of transistors	Number of gates	Optimized function	Method
4	52	9	F=((A⊕B)⊕((A.D).(B+C)))+((A+C)+D)´	**HD1(KM)[8]**
5	44	10	F=A′B+A(B′D′+C′D)	**HD2(QM)[8]**
4	47	7	F=((A⊕B)⊕A.D)+(C+(A⊕D))´	**NGA[6]**
4	47	7	F=((A⊕B)⊕A.D)+(C+(A⊕D))´	**MGA[8]**
4	39	5	F=((A⊕D).C)´⊕{[(A⊕D)+C]´+B}´	**Our approach**

From table 4, one can conclude that compared to the other methods, our approach has achieved a design by an average of 25% less gates, less transistors, and thanks to less levels it produces less propagation delay. In figure 9, the convergence diagram for function F_3 is drawn. It reaches the solution in just 18 generations, which is faster compared to F_2 in spite of having more minterms. This is partially due to the type of minterms in the functions and also the inherent random factors employed in the GP process.

***Test 4*:** A function with 5 variables.

$F_4(A,B,C,D,E)=\sum (0,3,5,6,9,12,15,16,19,21,22,25,28,31)$

A straightforward circuit for the implementation of this function is shown in figure 10. Also the result of applying our GP approach to this function is shown in figure 11. The pace of the convergence is also shown in figure 12. The final solution for F_4 is reached in generation 58, whose 5 variables and number of minterms explain this

lengthier convergence compared to the previous test functions.

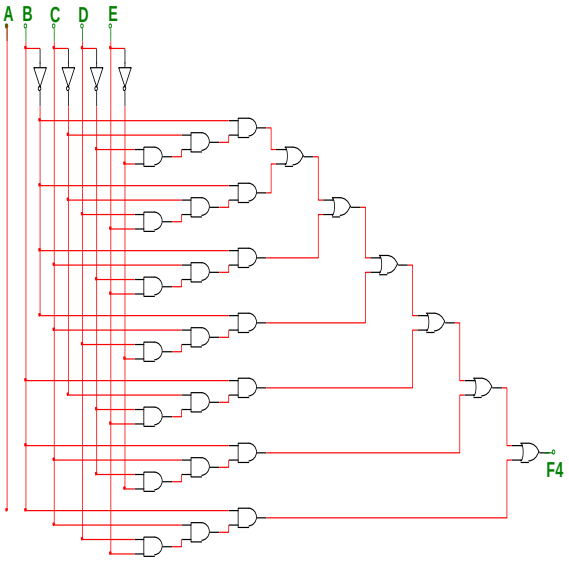

Figure 10. Circuit for F_4.

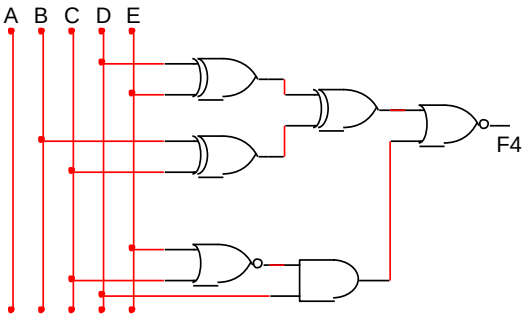

Figure 11. Designed circuit for F_4 using proposed approach.

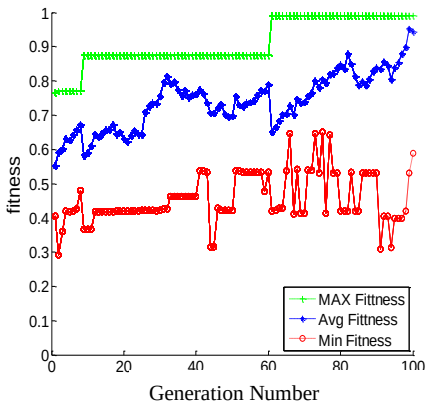

Figure 12. Convergence diagrams for F_4.

5. Self-Repairing

In many cases, evolutionary paradigms might get close to the optimized answers but still have unacceptable errors even with increase in the number of iterations, i.e. the resultant circuit does not fully cover the truth table. In order to overcome this problem (when the rows of the truth table are not properly covered), the following procedure is used to modify the circuit until the truth table is fully covered:

Step 1: The program outputs a circuit (F_{GP}) that (partially) covers the truth table (F_{TT}).

Step 2: A modification function F_{CORR} is obtained as follows:

$$F_{CORR} = F_{GP} \oplus F_{TT} \quad (2)$$

If F_{CORR} equals zero, the output circuit fully covers the truth table, and thus there is no need for a modification circuit; else:

Step 3: F_{CORR} is fed to the program as a new input as F_{TT} (NEW).

$$F_{TT}(NEW) = F_{CORR} \quad (3)$$

Then the program will design a circuit F_{GP}(NEW)

The final output F_{GP}(final) is obtained using:

$$F_{GP}(final) = F_{GP} \oplus F_{GP}(NEW) \quad (4)$$

These are shown in figure 13.

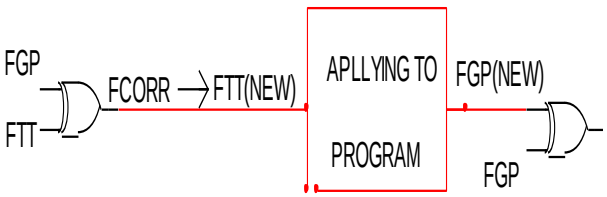

Figure 13. Block diagram for optimized circuit.

Test 5: A function with four variables.

$F_5(A,B,C,D) = \sum(1,2,3,7,9,10,11)$

F_{TT} = [0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0]

Here, F_5 and F_{TT} indicate the same function. The former shows the minterms and the latter shows the truth table. The circuit is shown in figure 14. Here are the outputs for this example:

F_{TT}	0	1	1	1	0	0	0	1	0	1	1	1	0	0	0	0
F_{GP}	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0
F_{CORR}	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

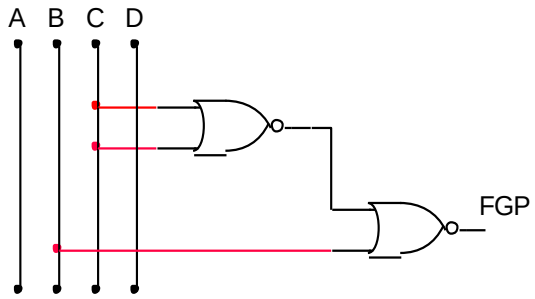

Figure 14. Circuit for F_5 with one error.

The repair process for the circuit starts with F_{CORR} as the new input F_{TT} (NEW) is fed into the optimization process. The program gives the output circuit F_{GP}(NEW) in figure 15.

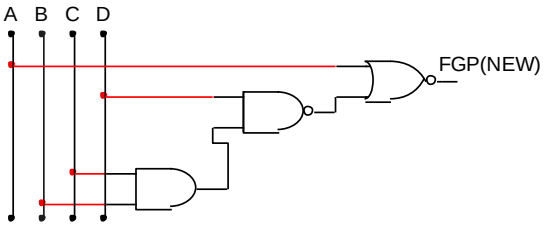

Figure 15. Repairing circuit for F_5 (F_{GP}(NEW)).

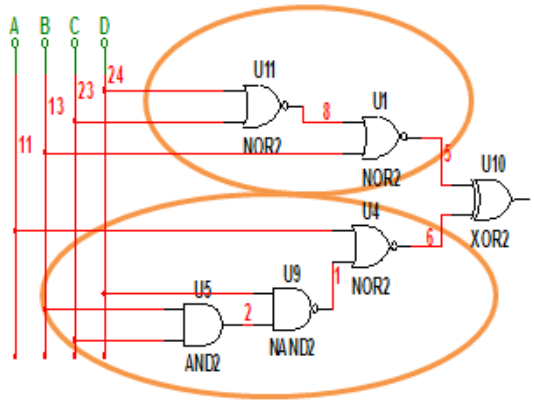

Figure 16. Final circuit for F_5.

Finally, XORing this circuit with the original circuit ($F_{GP} \oplus F_{GP}$(NEW)) gives an error-free circuit depicted in figure 16.

6. Conclusions and Suggestions

In this paper, we implemented a GP approach for optimizing combinational logic circuits. According to a number of assessments, the proposed approach shows its effectiveness with respect to the usual design criteria such as the number of gates and transistors.

Also the effective usage of the coding space of possible answers makes an inherent paralleling in the evaluation steps of the GP, which in turn speeds up the convergence process of our approach. Besides, using this strategy avoids unnecessary computation to be imposed on the program. Since the number of columns in solution arrays indicates the maximum levels of the output circuits, some flexibility comes with the implementation. For example, the program could design a wider circuit (less levels) instead of a lengthier one (more levels). Accordingly, and by using the NAND, NOR, and XOR gates, our implementation will target the designs with a minimum number of gates, transistors, and levels.

In the selection step, using a roulette wheel selection along with an elite one guarantees a sufficient dispersion in the next generation, while transfers a number of best parents to the next generation without any changes. Furthermore, a self-repairing feature produces more truthful circuits in terms of the coverage of the truth tables.

Optimization of the combinational logic circuit with feedback or memory elements could be the next step to test the applicability of our method. Also while some progresses are achieved in programs for the optimization of circuits with higher number of inputs, providing a balance between the number of inputs and an acceptable accuracy and convergence time is still a gorge for evolutionary approaches, and thus could be considered as an opportunity in the future works. In this regard, hybrid paradigms for multi-objective problems such as the one proposed in [17] could be promising.

References

[1] Holland, J. H. (1992). Adaptation in natural and artificial systems: an introductory analysis with applications to biology. A Bradford Book: MIT Press,

[2] Coello, C. A., Christiansen, A. D. & Aguirre, A., H. (2000). Towards Automated evolutionary design of combinational circuits. Computers & Electrical Engineering, vol. 27, no. 1, pp. 1–28.

[3] Kitano, H., & Hendler, J. A. (1994). Massively Parallel Artificial Intelligence. Menlo. Park, California: AAAI/MIT Press.

[4] Louis, S. J. (1993). Genetic algorithms as computational tools for design. PhD thesis, Department of Computer Science, Indiana University.

[5] Koza, J. R. (1992). Genetic Programming. Cambridge, MA: MIT Press.

[6] Coello, C. A., Christiansen A. & Aguirro, A. H. (1996). Use of Evolutionary Techniques to Automate the Design of Combinational Circuits. Department of Computer Science, Tulane University, New Orleans, USA.

[7] Sasao, T. (1993). Logic Synthesis and Optimization. Boston, London, Dordrecht: Kluwer Academic Press.

[8] Coello, C. A., Nacional, L., Avanzada, I., Aguirre , A. H. & Buckles, B. P. (2000). Evolutionary Multiobjective Design of Combinational Logic Circuits. Proc. of the Second NASA/DoD Workshop on Evolvable Hardware, Palo Alto, California, USA, July 13 – 15, pp. 161–170.

[9] Bajer, I. & Jakobović, D. (2012). Automated Design of Combinatorial Logic Circuits. Proc. of the IEEE 35th International Convention, MIPRO. Opatija, Croatia.

[10] Miller, J. F. (1999). An empirical study of the efficiency of learning Boolean functions using a Cartesian Genetic Programming approach. Proc. of the Genetic and Evolutionary Computation Conference, Orlando, Florida, USA, July 13-17, vol. 2, no. 1, pp. 1135–1142.

[11] Rajaei, A., Houshmand, M. & Rouhani, M. (2011). Optimization of Combinational Logic Circuits Using NAND Gates and Genetic Programming. Soft Computing in Industrial Applications, Springer, vol. 96, pp. 405-414.

[12] Karakatic, S., Podgorelec, V. & Hericko, M. (2013). Optimization of Combinational Logic Circuits with Genetic Programming. Electronics & Electrical Engineering, vol. 19, no. 7, pp. 86-89.

[13] Fogel, D. B. (1994). Asymptotic convergence properties of genetic algorithms and evolutionary programming analysis and experiments. Cybernetics and Systems: An International Journal, vol. 25, no. 3, pp. 389-407.

[14] Fogel, D. B. (1995). A Comparison of Evolutionary Programming and Genetic Algorithm on Selected Constrained Optimization Problems. Simulation, vol. 64, no. 6, pp. 397-404.

[15] Karnaugh, M. (1953). The Map Method for Synthesis of Combinational Logic Circuits. Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics, vol. 72, no. 5, pp. 593-599.

[16] McCluskey, E. J. (1956). Minimization of Boolean Functions. Bell Systems Technical Journal, vol. 35, no. 6, pp. 1417-1444.

[17] Lotfi, S. & Karimi, F. (2017). A hybrid MOEA/D-TS for Solving Multi-Objective Problems. Journal of AI & Data Mining, vol.5, no. 2, pp. 183-195.

نشریه هوش مصنوعی و داده کاوی

یک برنامه‌نویسی ژنتیک سریع و خود ترمیم برای طراحی مدارات منطقی

عبدالمجید موسوی[1,*] و محسن خدادادی[2]

[1] گروه برق، دانشکده فنی و مهندسی، دانشگاه لرستان، لرستان، ایران.

[2] گروه برق، دانشکده فنی و مهندسی، دانشگاه آزد اراک، مرکزی، ایران.

ارسال ۲۰۱۶/۰۶/۰۳؛ بازنگری ۲۰۱۶/۱۱/۰۵؛ پذیرش ۲۰۱۷/۰۵/۳۱

چکیده:

معمولا پارامترهای مهم در طراحی و پیاده‌سازی مدارات منطقی ترکیبی، تعداد گیت‌ها و ترانزیستورها و تعداد سطوح طراحی استفاده شده در آنها می‌باشند. در رابطه با این موضوع، اخیرا الگوریتم‌های تکاملی متنوعی با قابلیت‌های متفاوت معرفی شده‌اند. اما، با وجود مزایای غیر قابل انکار الگوهای تکاملی، این روشها دارای محدودیت‌هایی نیز هستند از جمله: الف) عدم اطمینان در رسیدن به جواب درست، ب) صرف زمان زیاد برای همگرایی و رسیدن به جواب، و ج) محدودیت در بهینه‌سازی مدارهایی با تعداد ورودی‌های زیاد. در این مقاله، ما یک پیاده سازی از برنامه‌نویسی ژنتیک ارائه می کنیم که برای یک تابع بولی داده شده، مدار منطقی معادلی طراحی می‌کند که علاوه بر پوشش جدول درستی تابع، از حداقل تعداد گیت (و تا حدی حد اقل ترانزیستور و سطح) برخوردار باشد. بطور خاص، پیاده سازی ما با برخورداری از ویژگیهای ذیل محدودیت‌های فوق الذکر را بهبود می‌بخشد: ویژگی خودترمیمی (برای بهبود محدودیت الف)، استفاده بهینه از فضای کدینگ مسأله که بنوبه خود گونه‌ای از موازی‌سازی را بهمراه داشته و در نتیجه زمان همگرایی الگوریتم بهبود می‌یابد (برای بهبود محدودیت ب). علاوه بر اینها، ما روش خود را برای بهینه‌سازی توابع بولی با ورودی‌های بیشتر بکار خواهیم بست (محدودیت ج). موارد فوق را از طریق آزمون‌های متعدد مورد راستی‌آزمایی قرار داده و نتایج حاصله را در مقاله گزارش می‌کنیم.

کلمات کلیدی: برنامه‌نویسی ژنتیک، مدارات منطقی، طراحی و بهینه‌سازی.