Journal Homepage: -www.journalijar.com
INTERNATIONAL JOURNAL OF ADVANCED RESEARCH (IJAR)
Article DOI:10.21474/IJAR01/5207
DOI URL: http://dx.doi.org/10.21474/IJAR01/5207

RESEARCH ARTICLE

REEVALUATION OF GENETIC ALGORITHM APPLIED TO MINIMIZE MULTILEVEL BOOLEAN EXPRESSIONS IN DIGITAL SYSTEMS.

Ms. Tiago da Silva Almeida, Mr. Wandro Beckman Maciel, Mr. Pedro Henrique de Castro Lima, Dr. Warley Gramacho da Silva, Dr. Rafael Lima de Carvalho.
Universidade Federal do Tocantins – UFT, Palmas, Tocantins, Brazil.

Manuscript Info

Manuscript History

Received: 18 June 2017
Final Accepted: 20 July 2017
Published: August 2017

Key words:-
Digital systems, Genetic Algorithm, JSON, Boolean expression, Optimization multilevel.

Abstract

Due complexity of digital systems, e.g., cellphones, internet of things, specific applications, this paper revisit the topic of optimization of Boolean expression using Genetic Algorithm, to show this area is not finished and there's much more to explore. For analysis, it was used the modeling approach in Shackleford, et. al. (2000) with some adjusts and was compared with original results and tabular methods, i.e., Quine-McCluskey and Karnaugh map. In results of tests, this proposed approach was inferior at tabular methods and original GA, and the performance was too slow due to the huge search space, because, the search space increases exponentially with quantity of inputs. These facts show the importance of continuous research about this area.

Introduction:-

Digital systems designs are indispensable in any company or university that works with Computer Science. Due the complexity of these designs, they are divided in many level of abstraction and in each level many challenges arise with new parts and models. Some papers approach these tasks and classified them according some methodology, e.g. Riesgo, Torroja, Torre (1999).

Evolutionary or genetic algorithms present great results in many applications fields, accordingly, with optimization of Boolean expression was explored over the years, such as, Shackleford, et. al. (2000), Coello (1996), Sobrinho and Mantovani (2006), Lacerda, Silva and Toledo (2010), Pradhan, Kumarand Chattopadhyay (2008),Hahanov, et. al. (2002), Shieh, Wangand Tsai (1999), Saini (2016), Curtinhas, et. al. (2015). About Genetic Algorithm (GA), the modeling problem is the greater challenge to reach goods results, and that's the specific goal of this paper, revisit and rebuild the modeling problem for GA to optimize Boolean expressions.

Besides, this method could be applied in CAD (Computer Aided-Design) tools, even for educational proposals. CAD tools can best be understood as project information management systems, along with the creation of graphs and simulation of the projects created. These simulations can be used, shared, published, republished and reused in different formats, scales and levels of detail (ERDENER, 2001).

The goal is to couple GA in others softwares, so the methodology is uses a common interface. For this, was chosen JSON (JavaScript Object Notation) language as that interface. JSON is a text format that facilitates the exchange of

CorrespondingAuthor:- Tiago da Silva Almeida.
Address:-109 Norte, Avenida NS 15, ALCNO 14, Bloco Bala 2 - Sala 21, Zipcode 77001-090, Palmas/TO.

data between all programming languages. Its syntax uses only a few symbols that are: braces, brackets, colon, and comma. This simple structure is one of the aspects that make JSON an attractive language in many applications.
JSON also supports value lists, which are created through the use of brackets. Every element included in the bracket range is one element in the list. In this work the JSON notation is used to model an intermediate code for a combinational circuit optimization step. Fig. 1 lists an example of the JSON code pattern that will be used in the paper.

```
{
  "circuitName": "maioria",
  "inputs": [
    {
      "name": "a"
    },
    {
      "name": "b"
    },
    {
      "name": "c"
    }
  ],
  "outputs": [
    {
      "expression": "b c + a c + a b",
      "minimizedExpression": "(a + b) (a + c) (b + c)",
      "name": "result"
    }
  ]
}
```

Figure 1:- Example of JSON file created as input and output for the genetic algorithm.

Within the scope of each circuit are described the outputs and related inputs in outputs and inputs, respectively. Additionally, there are the *expression* and *minimized_expression* objects representing the original Boolean expression and after the minimization process, respectively. The behavior of the circuit is determined by the Boolean function of each existing output for the same circuit.

This paper follows the approach of Shackleford, et. al. (2000) because the authors left questions about delay away and focus in minimize the transistors in the circuit. Thus, the new circuit is a multilevel circuit using only NOR gates, considering universally of gates and less transistor to build a circuit.

The results were compared with the results obtained by the method of Quine-McCluskey and Karnaugh map. Optimizations were tested for three circuit models: Majority function, Odd Parity function and Magnitude Comparator function.

Genetic Algorithm in minimization multilevel Boolean expression:-
Once the input variables of the JSON file of circuit to be optimized and its Boolean expression (see example in Fig. 1), the truth table for model is created for comparison with the true tables of each individual of the population making possible the classification of feasible and not feasible. The number of individuals in population is n_p, the m_p is mutation rate, c_p is crossover rate, are three parameters defined for GA as we can see in Fig. 2.

```
1: function GA(n_p, m_p, c_p)
2:     Generate initial population
3:     for i ← 1 to n_p do
4:         cromossom[n_c] = Random()
5:         population[i] = cromossom
6:     end for
7:     best = population[0]
8:
9:     First evolution
10:    while population[n_p − 1] notfactivel do
11:        crossover(c_p)
12:        orderPopulation()
13:    end while
14:    bestFitness = population[0].fitness
15:
16:    Second evolution
17:    while bestFitness < population[n_p/2].fitness do
18:        crossover(c_p)
19:        orderPopulation()
20:        if population[0].fitness < best.fitness then
21:            best = population[0]
22:        end if
23:    end while
24:    return(best)
25: end function
```

Figure 2:- Pseudo code of genetic algorithm implemented for modeling proposed according to Shackleford, et. al. (2000).

After generating the initial population GA implements a way to always protect the best individual present in each generation. Initially it keeps the chromosome better at zero index of the population (in order, the best fitness occupy the first positions) and at each iteration, a comparison of the individual's fitness of that position is performed with the fitness of the best. If a better cost is found, the variable is updated.

The evolution process is carried out in two stages. The first is responsible for leaving all individuals of the population feasible for later in the second, evolve their costs, evolution is driven by the cost of the circuit. The survival paradigm of fittest (low cost) in which the fittest children randomly replace the less fit individuals ensures the evolutionary advance to an optimal solution (SHACKLEFORD et. al., 2000).

According to Shackleford, et. al. (2000), the number of different logical functions (truth tables) increases exponentially according to the number of inputs input variables, and the number of possible logical functions, n_f, for a binary function of n_i inputs is given by:

$$n_f = 2^{2^{n_i}} \quad (1)$$

For example, for a three-input circuit we have only two hundred and fifty-six possible functions and for a five-input circuit this number jumps to more than four billion possible functions, that is, it grows very fast. Using such a population is unfeasible, GA would become too slow for execution, especially on conventional machines. Due to this problem, the ideal size for the n_p population is given by:

$$n_p = (inputs^2 + 1)^2 \quad (2)$$

The number of inputs squared plus one is the same as the number of logic gates of the circuit squared. For example, the number of individuals in the population for a three-input circuit is one hundred.

The matrix of Fig. 3 is defined so that any input of the function can be connected to any port and any logical port g_n can be connected to any other input g_k where $k > j$. Once complete, the connection matrix is able to describe all

circuits up to n_g ports and n_i inputs of the function. The size of the upper (rectangular) part of the matrix is given by $n_i \times n_g$. Already the size of the lower (triangular) portion is formed by $n_g(n_g - 1)/2n_g$ cells (SHACKLEFORD et. al., 2000).

The rows represent function inputs (rectangular upper part of matrix) or output ports (lower triangular part of matrix). The columns are represented by gates. Inputs are represented by i_n and outputs by f. There are no restrictions on the number of inputs and outputs. The model supports n entries where $n \geq 2$ and n outputs where $n \geq 1$.
Genes are traditionally represented by zeros and ones. The size of a chromosome n_c is defined by,

$$n_c = n_i n_g + \frac{n_g(n_g - 1)}{2} \quad (3)$$

where the number of inputs is represented by n_i and the number of ports per n_g. Therefore, the size of chromosome depends on the number of inputs and gates.

The shape of the binary chromosome vector can be seen from Fig. 3. The genes are distributed in the vector according to the orderly path of the matrix of connections, where the lines relating to the inputs i_1, i_2, ..., i_n are arranged in the first positions of the chromosome vector preceded by the rows of the matrix referring to the logic gates. Note that the output gate (column to the right of the matrix) does not appear on the chromosome because it is the output port of the circuit and can not be attached to any other gate according to the configuration of the matrix of connections.

Gates that have a connection (input) are represented by bit '1'. When there is not connection, then the bit representing this case is '0'. In the matrix of Fig. 3 for example, input i_1 is connected to port g_2 just as port g_1 is connected to port g_3. The Boolean expression is described as: $\overline{\overline{A + \overline{A + B}} + \overline{B + \overline{(A + B)}}}$.

The GA measures the cost of an individual in basic cells (BCs), by summing the individual costs of each logic gate of the circuit. A NOT port (has only one entry) has a cost of two BCs and a three-entry NAND or NOR has a cost of four BCs. If the fan-in of a port is zero, then its cost is also zero. For NAND and NOR ports, the C_B cost of a g port in BCs is given by:

$$C_B(g(fanin)) = \begin{cases} fanin + 1 & if\ fanin \geq 1 \\ 0 & otherwise \end{cases} \quad (4)$$

For example, if a feasible circuit has four NOR ports with two inputs each, then its cost is twelve. However, the GA will produce non-feasible individuals, that is, they do not correctly implement the logical source function and so a cost increment is added as penalty for each instance where the logical network F does not provide the output specified by the function of T source when tested against all possible input combinations (SHACKLEFORD et. al., 2000). These comparisons are made using the truth tables of T and F.

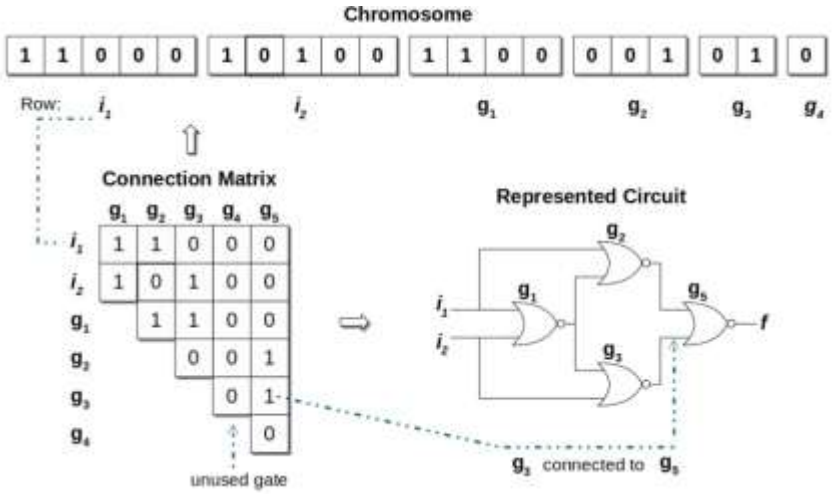

Figure 3:- Representation of chromosome which uses a connection matrix for NOR gate multilevel circuit (SHACKLEFORD et. al. 2000).

When choosing the penalty increment as maximum possible cost (i.e., $n_c + n_g$) for a connection matrix, we can be sure that a network with *n* errors will have a lower cost than a network with *n + 1* errors (SHACKLEFORD et. al., 2000). The cost of the penalty is given by:

$$C_P = (n_c + n_g) \sum_{i=0}^{2^{n_i}-1} \begin{cases} 1 & if\ T(i) \neq F(i) \\ 0 & otherwise \end{cases} \tag{5}$$

Therefore, the cost C for a given chromosome is composed of a penalty cost added to the intrinsic cost of the network:

$$C = C_P + \sum_{i=0}^{n_g} C_B(g_i) \tag{6}$$

The crossover rate defines the number of individuals in the population who donate their genetic material for the generation of offspring. According to the Fig. 5, parents are selected randomly, so GA draws a pair of distinct chromosomes (parents) to generate a single child.

Afterwards, the algorithm chooses the parent that has the highest fitness cost between two and saves the variable called the position of the population. Already the fitness cost of this worst individual is stored in the variable called *fitnessWorst*.

Only one cut-off point is used. This single point is also defined in a random manner, ensuring that the first and / or last position of the chromosome is not chosen. According to Fig. 4, the first half of parent 1 composes the first half of the child generated while the second half of parent 2 composes the second half of child.

After fitness calculation of the child is performed, the value is compared to the value that is stored in the *fitnessWor*st variable. If offspring's fitness is worse than the worst fitness between the two parents, the insertion of child into local population stored in the worst position is not performed, ensuring that the population can always receive better individuals, thus maintaining the evolution in each generation.

```
1: function CROSSOVER(c_p)
2:     for i ← 1 to c_p do
3:         parent1 = population.Random()
4:         parent2 = population.Random()
5:         if parent1.fitness > parent2.fitness then
6:             positionWorst = population[parent1.index]
7:             fitnessWorst = population[parent1.fitness]
8:         else
9:             positionWorst = population[parent2.index]
10:            fitnessWorst = population[parent2.fitness]
11:        end if
12:    end for
13:    cutPosition = Random()
14:    for i ← 0 to cutPosition do
15:        offspring = parent1.genes
16:    end for
17:    for i ← cutPosition to chromossom.size do
18:        offspring = parent2.genes
19:    end for
20:    mutacao(offspring, m_p)
21:    if offspring.fitness > fitnessWorst then
22:        population[positionWorst] = offspring
23:    end if
24: end function
```

Figure 4:- Pseudo code of crossover function implemented for modeling proposed (SHACKLEFORD et. al. 2000).

In the case of the mutation, it happens only in the child every time they are generated. It is important to note that mutation happens before the child's fitness is compared to the worst father's fitness. Once the percentage of mutation m_p is defined, the amount of bits that will be changed in the chromosome vector is the same for all individuals until the end of the GA execution (SHACKLEFORD et. al., 2000).

It is observed in Fig. 2 that stopping criterion is treated during the second GA evolution. When the first evolution is finalized, that is, when all individuals are feasible, the best fitness variable receives the value of the best fitness generated until then. Thus, stop criteria in *while* command (Fig. 2, line 17) is satisfied when the fitness value of the individual occupying the position in the half of the population $n_p / 2$ is greater than the value of better fitness. This is a good approach and ensures that an individual with great aptitude is found, but it can take many more generations to complete than if it did not implement stop criteria.

By conducting the tests, we sought a way of comparing the models of circuits minimized by the GA with the results obtained by the work of Shackleford, et. al. (2000), by the map of Karnaugh and Quine-McCluskey. The basic cells (BCs) presented by Shackleford, et. al. (2000) is used in this paper to evaluate and compare the cost of the results obtained by GA of other methods presented. The evaluation of circuits tested in (BCs) of CMOS technology is relevant, since this is the most used technology in the manufacture of integrated circuits for offering a very low energy consumption.

The GA parameters used for the test cases are presented in Table 1. Column (n_f) shows the number of functions in relation with number of inputs. In the case of majority and parity functions that have three entries, the population was defined as the number of one hundred individuals (n_p), mutation rate (m_p) of twenty percent in the genes of the child chromosome and seventy-five percent of the Crossover (c_p). For the four-variable comparator test, the population (n_p) was two hundred and eighty-nine individuals. The mutation rate (m_p) and the crossover rate (c_p) were the same for that case, twenty percent and seventy five percent respectively.

Experimental tests and results:-

Majority function:-

The behavior of majority function has output '1' if at least two of the three inputs are '1'.Because it is a stochastic algorithm, GA can present good results in one execution and worse results in another. Fig. 5 shows the graph for the five best runs of the function. One observes an abrupt fall of the fitness soon in the first generations. This is due to

the change of steps, from the first evolution to the second when individuals become feasible and free of penalty costs. It is worth emphasizing that each value in the horizontal axis represents the fitness of the best and worst individual in each generation.

The similarity of functions of five executions in the graph shows that the fitness costs for a given circuit have the same levels of values as they evolve. Even for these worst cases, it can be observed that at the end of the run the results are close to the final results of the executions in which the population already had great individuals since its first generation.

Table 2 presents the best and worst fitness values of each run and the number of generations that were required for the stop criteria of the algorithm to be satisfied. The resulting Boolean expression obtained by the tabular methods (Karnaugh map and Quine-McCluskey) and GA for the majority function is given by BC + AC + AB e $\overline{(A+B)} + \overline{(B+C)} + \overline{A+C}$, respectively. Even though the circuits have different types of ports, the amount of fan-in in each of them is the same, totaling a cost of 13 BCs as shown in Table 2.

Odd parity function:-

The odd parity function sets the output to '1' when the number of bits '1' is odd. When an initial population is generated with very good individuals, the tendency is that the first stage of GA evolution takes less time, improves the fitness of the population more quickly, and execution is completed with fewer generations. "run 1" (blue and orange) on the graph of Fig. 6, for example, was the first to start the second evolution stage and the first to be completed, while "run 4" was the opposite.

Note that "run 1" (blue) even being on this list was a test considered as good as some of the best shown. In the two graphs it is possible to observe that the behavior of the evolution of fitness in the second stage, does not undergo practically great changes throughout the generations. It is possible to conclude, in these cases, that the algorithm probably finds, in most runs, the most optimized individual possible for the function.

Another ten tests performed are presented in Table 3. The best individual's fitness obtained in this set of tests was 24, slightly below the average of 26.7. The standard deviation for this case was 1.487, while the average of generations was over thirteen thousand, a high value in relation to the majority function that has the same number of variables. This is due to the fact that there was a considerable variation in the number of generations between the runs.

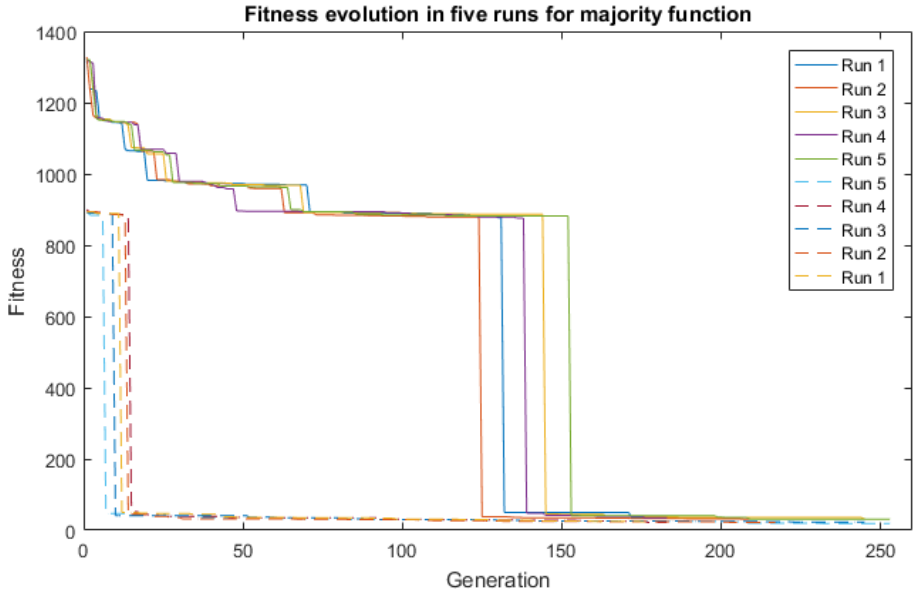

Figure 5:- Fitness evolution in five runs majority function plotting best and worst fitness.

Table 1:- Parameters used in experimental tests.

Inputs	Population (n_g)	(n_f)	Mutation (m_p)	Crossover (c_p)
3	100	256	20%	75%
4	289	65,536	20%	75%

Table 2:- Results in ten runs of majority function divided in best fitness and worst fitness, highlighted in red the best solution had founded.

Run	Best Fitness	Worst Fitness	Generations
1	18	30	4162
2	17	30	4433
3	19	31	3018
4	19	34	2820
5	20	32	3332
6	18	27	9309
7	**13**	**34**	**3266**
8	19	30	5399
9	17	30	4575
10	19	35	2167
Average	17.9	31.3	4248.1
Standard deviation	1.868	2.326	1917.631

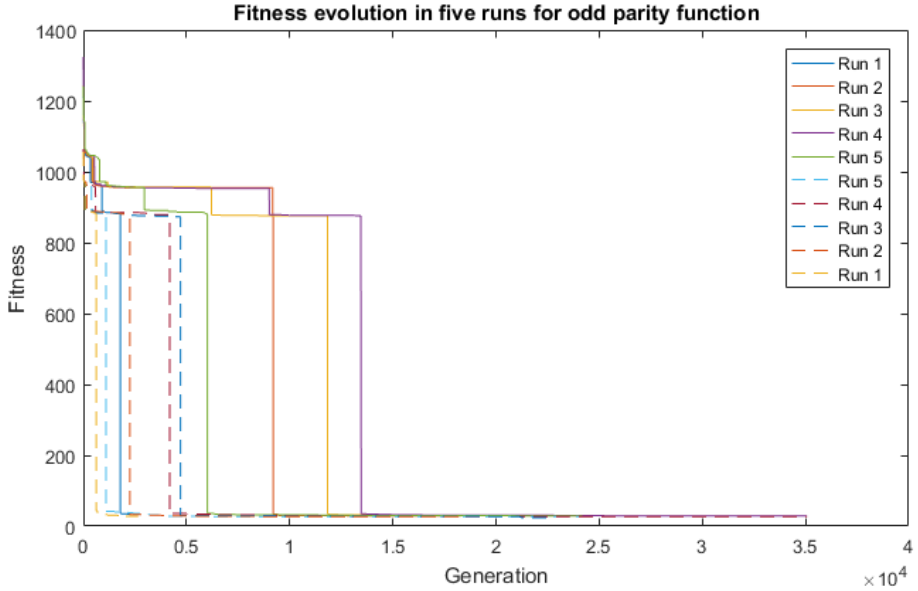

Figure 6:- Fitness evolution in five runs odd parity function plotting best and worst fitness.

With a cost of 33 basic cells (BCs) with eleven logical ports is more expensive than the one generated by GA with only 7 ports and cost of 27 BCs. Among the tests, the odd parity function was the one that GA showed to be more efficient.

It is important to note that the odd parity function is particularly difficult to minimize by the Karnaugh method. The pattern with the same amount of ones and zeros does not have adjacent terms that can be grouped for minimization. Each minterm of odd parity function is therefore an essential prime implicate of the function, thus leaving the larger Boolean expression. GA for the odd parity function is given by $\overline{\left(\overline{(A+B)}+\overline{((A+B)+A+B)}+A\right)}+\overline{\left(\overline{(A+B)}+\overline{((A+B)+B+C)}+B\right)}+\overline{\left(\overline{((A+B)+A+B)}+\overline{((A+B)+B+C)}+C\right)}$

Table 3:- Results in ten runs of odd parity function divided in best fitness and worst fitness, highlighted in red the best solution had founded.

Run	Best Fitness	Worst Fitness	Generations
1	28	33	2657
2	**24**	**31**	**4987**
3	27	29	18295
4	28	30	22670
5	27	32	4366
6	27	29	35065
7	27	32	9128
8	27	29	24184
9	27	29	7528
10	29	35	4946
Average	27.1	30.9	13382.6
Standard deviation	1.286	1.972	10428.624

Comparator function:-

As a result of the number of inputs in this example, individual's chromosome has a number of two hundred and four genes, which is why it contributed to a longer time in the execution of the algorithm in relation to the other two test cases presented.

The average fitness for this function in relation to the Table 4 tests was 59.9, with a standard deviation of 4.989 and an average of more than 2,000 generations

The magnitude comparator is an arithmetic function that compares the magnitudes of two binary integers in the form $<i_1, i_2>$ and $<i_3, i_4>$. The output of the function has resulted in '1' when $(i_1 \times 2) + (i_2 \times 1) > (i_3 \times 2 + i_4 \times 1)$.

In this example of four variables, the initial population generated had high fitness values, and just under twenty-two thousand generations. For example, "run 2" in this test yielded an initial population with values approaching 7,000 fitness. As in the other test cases, the best and worst five runs of Fig. 7, maintains a similar pattern across the tests, when algorithm generates good individuals randomly in the initial population and when it does not generate.

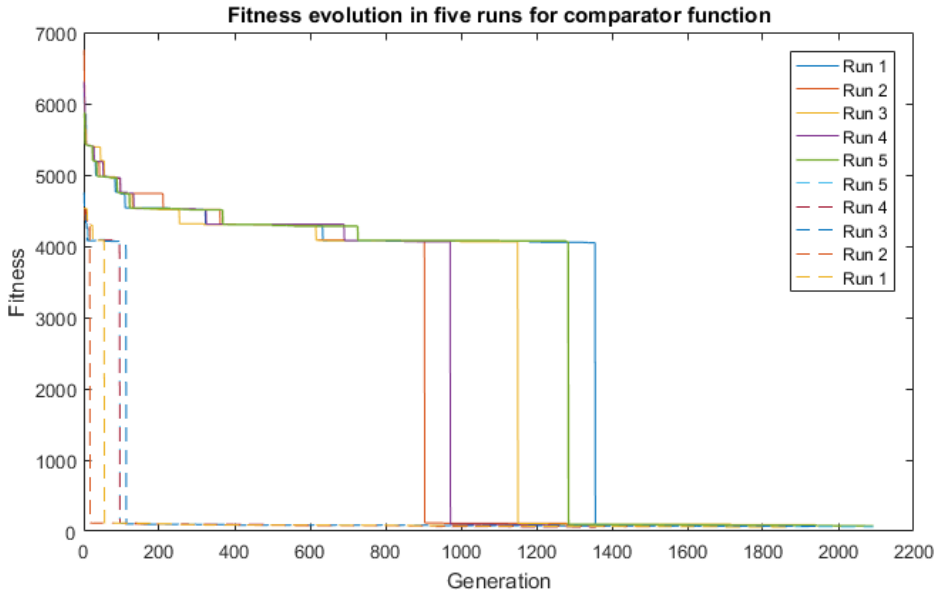

Figure 7:- Fitness evolution in five runs comparator function plotting best and worst fitness.

Table 4:- Results in ten runs of comparator function divided in best fitness and worst fitness, highlighted in red the best solution had founded.

Run	Best Fitness	Worst Fitness	Generations
1	56	67	3223
2	60	76	1869
3	**41**	**75**	**2093**
4	66	86	1513
5	71	93	1939
6	57	69	2009
7	57	75	1357
8	54	90	2190
9	57	69	1442
10	58	71	2604
Average	59.9	77.1	2023.9
Standard deviation	4.989	8.826	536.892

The GA tests for the comparator function were poor in relation to the result obtained by the map of Karnaugh and Quine-McCluskey. The tabular methods obtained a cost of 23 BCs resulting in the circuit of only four logic gates. On the other hand, the result of the GA obtained in its best execution between the realized ones, a cost of 41 BCs. The resulting Boolean expression obtained by the tabular methods (Karnaugh map and Quine-McCluskey) and GA for the majority function is given by $A\bar{C} + AB\bar{D} + B\bar{C}\bar{D}$ e $\overline{\left(B + \left(\overline{B + \left(\overline{A + (\overline{B + \bar{C}})}\right)}\right)\right)} + \overline{\left(C + \left(\overline{B + \left(\left(\overline{A + (\overline{B + \bar{C}})}\right)\right)} + B\right)\right)} + \overline{\left(\left(\overline{\left(\overline{C + (\overline{B + \bar{C}})}\right) + \left(\overline{A + B + (\overline{D + \bar{C}})}\right) + \left((\overline{B + \bar{C}})\right)}\right) + A + (\overline{A + X})\right)}$, respectively.

Table 5 summarizes the general results comparing GA present in this paper, GA in Shackleford, et. al. (2000) and tabular methods.

Table 5:- General results of experimental test and comparing with original paper and a classical method.

		GA	(SHACKLEFORD et. al. 2000)	Quine-McCluskey
Majority function	BC	13	-	13
	Gates	4	-	4
Odd parity function	BC	27	24	33
	Gates	7	8	11
Comparator function	BC	41	20	23
	Gates	12	7	8

The results in this paper were worst of comparison, due a lack of clarity of descriptions parameters used by Shackleford, et. al. (2000). As described in Shackleford, et. al. (2000) the complexity of problem increases too much with addition of inputs, charactering as NP-Problem.

Conclusion and final remarks:-

The goal of this paper was revisit the optimization of multilevel Boolean expression using GA. For this, was used the modeling approach in Shackleford, et. al. (2000). The method, how was implemented, had showed to be weak and with poor results compared with Shackleford, et. al. (2000) and tabular methods, such as Quine-McCluskey and Karnaugh map.

The first tests performed with the GA determined some changes in GA settings throughout development. One of the changes made was to the definition of the stop criteria. The results obtained with these observations allowed to conclude that a configuration not suitable for the problem can compromise the efficiency of the GA and consequently the achievement of good results.

The results obtained during the development of the work including those presented showed that GA can achieve in most cases better results in the task of minimizing combinational circuits in relation to the traditional tabular

methods described throughout the work, even with this disadvantage of owning slow processing. For this reason, the selected test cases needed to be more basic and have fewer variables.

References:-

1. Erdener, E. G. (2001): CAD Standards and Institutions of higher education. Facilities, no. 19(7/8). pp. 287-295, 2001.
2. Riesgo, Y., Torroja T., Torre, E. de la (1999): Design methodologies based on hardware description languages. IEEE Transactions on Industrial Electronics. no. 40(1). pp. 3-12, 1999.
3. Shackleford, B., Okush, E., Yasuda, M., Koizumi, H., Seo, K., Yasuura, H. (2000): Synthesis of minimum-costmultilevel logic networks via genetic algorithm. In: IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences. no. E83-A(12). pp. .2528-2537, 2000.
4. Coello, C. A. (1996): An empirical study of evolutionary techniques for multi objective optimization in engineering design. (Doctoral Dissertation) University New Orleans, LA, USA. 1996.
5. Sobrinho, E. F. G. and Mantovani, S. C. A. (2006): EHW aplicado à síntese de circuitos digitais usando representação por portas lógicas. In: XXXVIII Simpósio Brasileiro de Pesquisa Operacional. pp. 1289-1299, 2006.
6. Lacerda, W. S., Silva, B. de A., Toledo, C. F. M. (2010): Síntese de circuitos digitais utilizando computação evolutiva. In: Congresso Brasileiro de Automática. pp. 2852- 2857, 2010.
7. Pradhan, S. N. and Kumar, M. T. and Chattopadhyay, S. (2008): Three-level AND-OR-XOR network synthesis: A GA based approach. In: APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems, Macao, pp. 574-577, 2008.
8. Hahanov, V. and Babich, A. and Sokolov A. andPudov, V. (2002): Deterministic method of genetic algorithms of test generation for digital systems verification. In: Modern Problems of Radio Engineering, Telecommunications and Computer Science (IEEE Cat. No.02EX542). pp. 257-258, 2002.
9. Shieh, L. S. and Wang, W. and Tsai, J. S. H. (1999): Optimal digital design of hybrid uncertain systems using genetic algorithms. In: IEE Proceedings - Control Theory and Applications, vol. 146, no. 2, 1999, pp. 119-130.
10. Saini, A. (2016): An optimized interconnection network based on genetic algorithm. In: 2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC), Waknaghat, 2016, pp. 468-473.
11. Curtinhas, T. and Cavalcante, T. C. and Oliveira, D. L. and Faria L. A. and Saotome, O. (2015): Minimization and encoding of high performance asynchronous state machines based on genetic algorithm. In: 2015 28th Symposium on Integrated Circuits and Systems Design (SBCCI), Salvador, 2015, pp. 1-6.