Towards Automated Evolutionary Design of Combinational Circuits

Carlos A. Coello Coello†
Alan D. Christiansen‡
Arturo Hernández Aguirre‡

† (ccoello@xalapa.lania.mx)
Laboratorio Nacional de Informática Avanzada, A.C.
Xalapa, Veracruz 91090, México

‡({adc,hernanda}@eecs.tulane.edu)
211 Stanley Thomas Hall
Department of Computer Science
Tulane University
New Orleans, LA 70118, USA

Abstract

In this paper we propose a methodology based on a genetic algorithm (GA) to automate the design of combinational logic circuits in which we aim to minimize the total number of gates used. Our results are compared against those produced by human designers and by another GA-based approach. We also analyze the importance of using a non-binary representation in this problem despite the commonly accepted notion of universality of the binary representation in all kinds of GA-based applications.

Keywords: circuit design, optimization, genetic algorithms, computer-aided design, artificial intelligence.

1 Introduction

Design is a task that requires knowledge and creativity which are two human attributes normally considered too complex to be automated.

Researchers in Artificial Intelligence (AI) have devoted a lot of work towards automating different aspects of design, but most of the current results

consist of complex and expensive programs that can be easily outperformed by experienced human designers.

The main goal of the research reported in this paper was to develop a low-cost computer-based design tool that could generate combinational logic circuits which are not only fully functional, but also optimum according to some metrics.

Since the definition itself of the term *design* is so elusive, it is convenient to start by stating the definition of design that better fulfills the purposes of this paper:

> Design is the process of deriving, from a specified input/output behavior, a structure (in our case a certain combination of logic gates) that is functional (produces all the outputs desired for all the inputs specified) within a certain set of specified constraints.

Furthermore, we want this design to be optimum in terms of certain structural features (e.g., the number of gates used). It should be added that our current work focuses only on combinational logic circuits, which contain no memory elements and no feedback paths. However, the approach proposed is general enough as to allow its generalization to other (more complex) circuits.

2 Previous Work

A general search technique inspired by natural evolution, called the *genetic algorithm* (GA) [15], has been widely used for optimization tasks [9] and is known to be a very powerful tool in certain domains. In our current work we wish to find a way to use the GA as a design tool, with particular emphasis in the design of combinational circuits.

The design process for combinational logic circuits has evolved from its first notions [36] to a standard element of undergraduate computing curricula [34]. Standard graphical design aids such as Karnaugh Maps [18, 41] are widely used and tools suitable for computer implementation have evolved from the Quine-McCluskey Method [32, 26] to freely available tools such as Espresso [2] and MisII [3] and many commercial products.

Probably the earliest attempt to evolve circuits is Friedman's thesis, that dates back to the mid 1950s [8]. In his thesis, Friedman proposed that a series of control circuits, similar to what we now call neural networks, could be evolved through "selective feedback" in a process analogous to natural

selection. J. W. Atmar [1] was another early researcher to incorporate directly the bit string representing the configuration of a programmable circuit within the genotype of an evolutionary-based technique.

In the contemporary literature, the attempt to use evolutionary-based techniques to design electrical circuits has been called "evolvable hardware" [20, 7]. Within evolvable hardware there are only a few researchers working on the design of circuits at the gate-level.

Louis [25] is one of earliest sources that report the use of GAs to design combinational logic circuits. In his dissertation [24] Louis combines knowledge-based systems with the genetic algorithm, making use of a genetic operator called *masked crossover* that adapts to the encoding, being able to exploit information unused by classical crossover operators. His results, although very encouraging for certain examples, do not seem to have solved the combinational circuit design problem completely. However, his idea of incorporating knowledge about the domain in the genetic operator constitutes a big step toward increasing the power of the GA as a design tool. Unfortunately, the incorporation of knowledge into the GA decreases its usefulness as a *general* search tool. Louis overcomes this problem by defining an operator that he claims to be domain independent, but whose efficiency turns out to depend on the representation used.

Koza [21] has used genetic programming to design combinational circuits. He has designed, for example, a two-bit adder, using a small set of gates (AND, OR, NOT), but his emphasis has been on generating functional circuits rather than on optimizing them. In fact, this is also the case in Louis' research, where the main focus was to provide an easier way to generate functional designs using the GA rather than in optimizing a functional design according to certain metrics. In more recent work, Koza [23, 22] has focused more towards the design of analog circuits in which the goal is to produce their appropriate topology and size so that they are functional given a certain set of components. So far, genetic programming has been considered a more powerful tool in such tasks, because the representation it uses is more powerful for structural design in general [33]. However, genetic programming produces circuits that are highly redundant and difficult to simplify automatically. Furthermore, the computer resources normally required to produce such circuits are very demanding in terms of memory and CPU time [21]. That is why we decided to use instead a matrix representation that is encoded linearly in a chromosome, and turns out to be a compromise between the powerful tree representation used by genetic programming and the relatively limited linear representation used by a conventional genetic algorithm [33]. It must be added that the main limitations

of a linear representation rely on two facts: the fixed length normally associated with it that keeps an expression (a Boolean function in our case) from growing and shrinking during the evolutionary process and the way in which a linear chromosome is decoded (translating each chromosomic position directly into a variable value). By using a variable-length GA and a matrix representation in the decoding stage, we could succesfully deal with these two problems.

Another early effort to codify the basic logic gates (AND, OR, and NOT) along with their possible interconnections was offered by Thompson et al. [38]. Thompson's work focuses on the configuration of a Field Programmable Gate Array (FPGA) using genetic algorithms, and was the basis for the research performed later by most of the other researchers working in evolvable hardware at the gate level.

Miller et al. [28] developed (independently) an approach similar to ours, but using a more compact representation that instead of considering the inputs and gates as completely separate elements in the chromosomic string (as in our case), uses a single gene to encode a complete Boolean expression. Miller's notation does not decrease the total length of the chromosome, but it increases the cardinality of the alphabet needed, having as its main drawback the lack of flexibility of the representation to handle a larger number of inputs (the cardinality of the alphabet in Miller's case grows exponentially with respect to the number of inputs, whereas in our case, it grows linearly). Nevertheless, we will compare the results found by our approach in one example with those previously reported by Miller et al. [28].

The only other work on evolvable hardware at a gate level is the one reported by Higuchi et al. [13] and Iba et al. [16]. In both cases, a variable-length GA with an array representation is used. However, since the focus of these papers is on learning rather than on optimization problems, we will not be able to compare our work with theirs.

It should be mentioned that in the work reported here, we were interested not only in producing functional designs, but also in optimizing them according to certain metrics. This is a quite complicated task for the GA, because designing a fully functional circuit from a random set of invalid circuits is a problem difficult enough as to consume most of the search time of a conventional genetic algorithm. Trying to find the feasible region in this highly constrained search space and then try to locate the optimum within such region is an even more difficult task.

3 Statement of the Problem

The problem of interest to us consists of designing a circuit that performs a desired function (specified by a truth table), given a certain specified set of available logic gates. In circuit design, one can use various criteria to define minimal-cost expressions. For example, from a mathematical perspective, one could minimize the total number of literals or the total number of binary operations or the total number of symbols in an expression. The minimization problem is difficult for all such cost criteria. In gate networks one could minimize the total number of gates subject to such restrictions as fan-in, fan-out, number of levels, or the total number of SSI packages. In general, it is very difficult to find such minimal networks or to prove the minimality of a given network [4]. In spite of this, it is possible to solve a number of minimization problems using systematic techniques, provided that we are satisfied with less general solutions.

The complexity of a logic circuit is a function of the number of gates in the circuit. The complexity of a gate generally is a function of the number of inputs to it. Because a logic circuit is a realization (implementation) of a Boolean function in hardware, reducing the number of literals in the function should reduce the number of inputs to each gate and the number of gates in the circuit—thus reducing the complexity of the circuit.

The algebraic method used to minimize functions is tedious and error prone. Its success depends on our ability to recognize the application of a theorem or a postulate during the minimization process. Such recognition may not be obvious. Furthermore, there is no general set of rules to aid that recognition.

Two popular minimization techniques are the *Karnaugh Map* [18], which is based on a graphical representation of Boolean functions, and the *Quine-McCluskey Procedure* [32, 26], which is a tabular method. Both of these methods are mechanical in nature. Karnaugh Maps are useful in minimizing functions with up to five or six variables. The Quine-McCluskey Procedure is useful for functions of any number of variables and can easily be programmed to run on a digital computer. Generally, several minimum functions can be obtained for a given function using either method, based on the choices made during the minimization process. All minimum functions with the same number of literals yield circuits of the same complexity; hence, any of them can be selected for implementation.

Both the Karnaugh Map and Quine-McCluskey Procedure produce *two-level* circuit forms (e.g., minimum sum of products). This is the best form if the overriding concern is minimizing propagation delay of signals through

the circuit. However, in many cases a greater concern is the minimization of the number of gates present in a circuit, and a small penalty in circuit speed is acceptable. To minimize the total number of gates, it is often necessary to find a multi-level circuit form. In order to find multi-level implementations, the Karnaugh Map and Quine-McCluskey Methods must be combined with other techniques, such as algebraic manipulation of logic expressions.[1]

Additionally, the Quine-McCluskey Procedure is not very efficient: it can be shown that the upper bound on the number of prime implicants is $\frac{3^n}{n}$ [19], where n is the number of inputs in the truth table. This means that the CPU requirements for this procedure grows exponentially with the number of inputs. Furthermore, once the prime implicants have been found, the algorithm needs to find the minimum set cover, which is known to be an NP-complete problem [19]. Also, although some authors have proposed extensions to the basic Quine-McCluskey Procedure that allow to handle XOR gates (see for example [40]), in the original proposal (which we have used here), only the basic gates are allowed (AND, OR, NOT), and a human designer has to perform further refinements in order to introduce XOR gates into the circuit.

Note that the algebraic simplification process depends entirely on one's familiarity with the postulates and theorems and one's ability to recognize their application. Of course, this ability varies from individual to individual. Depending on the sequence in which the theorems and postulates are applied, more than one simplified form of the expression may be obtained. Usually all such simplified forms are valid and acceptable. Thus, there is (in the general case) no single, unique minimized form of a Boolean expression.

In this work, we compare the designs produced by a GA with those generated by a human designer using Karnaugh maps and another one using the Quine-McCluskey Procedure (unless indicated otherwise in the examples). The comparison is in many ways unfair because of differing capabilities of man and machine. For example, a human designer tends to use only the gates NOT, AND, OR and has more difficulties using XOR because the Karnaugh Map and the Quine-McCluskey Procedure do not support the identification of XOR terms as well as they support "seeing" simple product terms. The computer, using our GA approach, and not being restricted by human pattern recognition abilities, uses many XOR gates, often disregarding the NOT gate. Our overall measure of circuit optimality is the total number of gates used, regardless of their kind. This is approximately pro-

[1] A tool like MisII [3] *can* find multi-level forms, but requires human guidance to do so effectively.

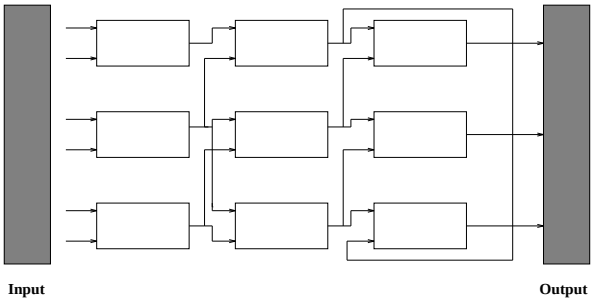

Figure 1: A gate in a two-dimensional template, gets its second input from either one of two gates in the previous column.

portional to the total part cost of the circuit. Obviously, we perform this analysis for only fully functional circuits.

4 Using the Genetic Algorithm

The famous naturalist Charles Darwin defined *Natural Selection* or *Survival of the Fittest* in his book [6] as the *preservation of favorable individual differences and variations, and the destruction of those that are injurious.* In nature, individuals have to adapt to their environment in order to survive in a process called *evolution*, in which those features that make an individual more suited to compete are preserved when it reproduces, and those features that make it weaker are eliminated. Such features are controlled by units called *genes* which form sets called *chromosomes.* Over subsequent generations not only the fittest individuals survive, but also their fittest genes which are transmitted to their descendants during the sexual recombination process which is called *crossover.*

John H. Holland became interested in the application of natural selection to machine learning, and in the late 1960s, while working at the University of Michigan, he developed a technique that allowed computer programs to mimic the process of evolution. Originally, this technique was called *reproductive plans*, but the term *genetic algorithm* became popular after the publication of his book [14] [15].

More information on genetic algorithms may be found in the books by Goldberg [9], Michalewicz [27] and Mitchell [31].

A genetic algorithm for a particular problem must have the following five components [27]:

Input 1	Input 2	Gate Type

Figure 2: Encoding used for each of the matrix elements that represent a circuit.

1. A representation for potential solutions to the problem.
2. A way to create an initial population of potential solutions (this is normally done randomly).
3. An evaluation function that plays the role of the environment, rating solutions in terms of their "fitness".
4. Genetic operators that alter the composition of children.
5. Values for various parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.).

The first interesting aspect of this problem is the encoding (i.e., representation) of solutions as chromosomic strings that the GA can evolve. The representation chosen for our work is a bidimensional matrix as the one suggested by Louis [25] in which each matrix element is a gate (there are 5 types of gates: AND, NOT, OR, XOR and WIRE) that receives its 2 inputs from any gate[2] at the previous column as shown in Figure 1. More formally, we can say that any circuit can be represented as a bidimensional array of gates $S_{i,j}$, where j indicates the *level* of a gate, so that those gates closer to the inputs have lower values of j. (Level values are incremented from left to right in Figure 1). For a fixed j, the index i varies with respect to the gates that are "next" to each other in the circuit, but without being necessarily connected. It is interesting to notice that if a row-order encoding is used, the problem becomes disruptive [25], making it very hard for the GA. The reason is that using such an encoding, any circuit designs that are close in two-dimensional (phenotypic) space may be far apart in one-dimensional (genotypic) space, making it difficult to preserve highly fit schemas (in GA terminology, we say that the problem is deceptive [10]).

A chromosomic string encodes the matrix shown in Figure 1 by using triplets in which the 2 first elements refer to each of the inputs used, and

[2] It is worth mentioning that Louis fixes the position of one of the inputs to reduce the size of the search space [24].

the third is the corresponding gate as shown in Figure 2 (only 2-input gates were used in this work).

Our goal was then to produce a fully functional design (i.e., one that produces all the expected outputs for any combination of inputs according to the truth table given for the problem) which maximizes the number of WIREs[3].

A critical part of getting a GA approach to succeed in this problem has to do with the representation scheme used by the genetic algorithm. Although it has been argued that a binary representation provides the maximum number of schemata [27] it turns out that in some domains such as numerical optimization, alphabets of higher cardinality have proved to provide better results in a shorter period of time than their binary counterparts [5]. With this idea in mind, we decided to experiment with an alphabet of cardinality n, where n can be defined by the user and will be normally taken as the number of rows allowed in our circuit, according to the matrix encoding adopted in this problem. This representation allows the manipulation of shorter strings, it decreases the complexity of the decoding task, and as will be seen in this work, it provides better solutions than its binary counterpart.

Another difficulty is the development of a good fitness function. Again, our initial approach was to use a slight variation of the function suggested by Louis in his dissertation [24], which consists of the number of correct responses obtained by the GA (with respect to the truth table given by the user).

The following formula is used to compute the fitness of an individual $\mathbf{x}$:

$$\text{fitness}(\mathbf{x}) = \begin{cases} \sum_{j=1}^{p} f_j(\mathbf{x}) & \text{if } f(\mathbf{x}) \text{ is not feasible} \\ \sum_{j=1}^{p} f_j(\mathbf{x}) + w(\mathbf{x}) & \text{otherwise} \end{cases} \tag{1}$$

where p is the number of entries of the truth table (normally, $p = 2^n$, being n the number of inputs of the truth table, but p can also be assigned a certain value directly, in case the truth table has don't cares), and the value of $f_j(\mathbf{x})$ depends on the outcomes produced by the circuit $\mathbf{x}$ encoded by the GA (whenever the GA matches the corresponding entry of the truth table at location j, a value of one is assigned to $f_j(\mathbf{x})$; otherwise, a value of zero is assigned). The function $w(\mathbf{x})$ returns an integer equal to the number of WIREs present in the circuit $\mathbf{x}$ encoded by the GA.

In words, we can say that our fitness function works in two stages. At

[3] WIRE basically indicates a null operation, or in other words, the absence of gate, and it is used just to keep regularity in the representation used by the GA that otherwise would have to use variable-length strings.

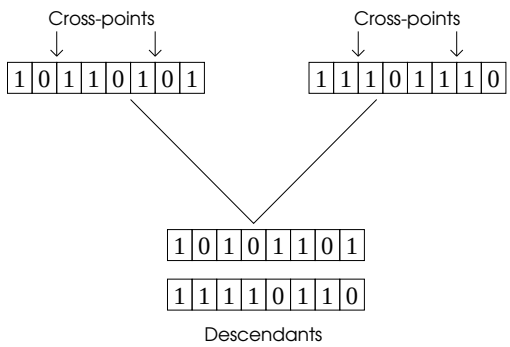

Figure 3: Use of a two-point crossover between two chromosomes. In this case the genes at the extremes are kept, and those in the middle part are exchanged. If one of the two cross-points happens to be at the string boundaries, a single-point crossover will be performed, and if both are at the string boundaries, the parents remain intact for the next generation.

the beginning of the search, only validity of the circuit outputs is taken into account, and the GA is basically exploring the search space. Once a functional solution appears, then the fitness function is modified such that any valid designs produced are rewarded for each WIRE gate that they include, so that the GA tries to find the circuit with the minimum number of gates that performs the function required. It is at this second stage that the GA is actually exploiting the search space, trying to optimize the solutions found (in terms of their number of gates) as much as possible.

It should be mentioned that although at first sight the size of the search space for some instances of this problem may seem too small to even attempt to use a heuristic function, that is not true. For the representation used for this work, if we assume a cardinality n and a chromosomic lenght l, the size of the intrinsic search space is n^l. Both the cardinality and the length of a string depend on the size of the matrix used to solve the circuit. In general: $l = u \times t$, where $t = r \times q$, and r and q are the number of rows and columns of the matrix respectively, and u refers to the number of genes required to represent a triplet. For the case of a GA with n-cardinality (like the one used in this paper), $u = 3$, and for the case of a binary GA, $u = 9$ (we are assuming 3 bits for each of the elements of the triplet).

For the experiments reported here we used a traditional two-point crossover operator (see Figure 3) and a conventional uniform bit mutation operator [9]. In all our experiments we kept the best individual of each generation

(elitism).

5 Comparison of Results

We have used several circuits of different degrees of complexity to test our approach. For the purposes of this paper, 5 examples were chosen to illustrate our approach, and the results produced with the GA were compared with those generated by human designers and, in one case, with another GA-based approach.

In each case, the size of the matrix used to fit the circuit was determined using the following procedure:

1. Start with a square matrix of size 5.
2. If no feasible solution is found using this matrix, then increase the number of columns by one.
3. If no feasible solution is found using this matrix, then increase the number of rows by one.
4. Repeat steps 2 and 3 until a suitable matrix is produced.

As we will see in the following examples, it was normally the case that for small circuits a matrix of 5×5 was sufficient. However, in our last example, it was necessary to reach a matrix size of 6×7. This made necessary to run the GA for more generations, performing, in consequence, more fitness function evaluations. This situation normally arises with circuits having several outputs, although in some cases, such as in the 2-bit multiplier of our fourth example, even a 5×5 matrix may be enough to find the best known circuit.

To distinguish between the (traditional) binary representation, we will refer to that approach as the binary genetic algorithm (or BGA), and our proposed representation that uses an n-cardinality encoding will be called NGA.

The other issue is regarding the crossover and mutation rates. After a series of experiments, we decided to use a crossover rate of 50% and a mutation rate such that each string had a 50% probability of being mutated at a certain position. Since mutation was applied on a single-gene basis, we used as our probability of mutation the result of dividing this 50% by the length of the string. Since the length of the strings used to solve the first four examples using the NGA is 75, the probability of mutation in those cases

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 1: Truth table for the circuit of the first example.

was 0.006667. For the case of our BGA, the probability of mutation was 0.002222 because the length of the strings used for the first four examples was 225. The last example used a longer string (126 for the NGA and 252 for the BGA), which made necessary to use a lower probability of mutation (0.003968 for the NGA and 0.001984 for the BGA).

The maximum number of generations was arbitrarily set to a reasonable large number, and the population size was chosen based on a number of independent runs. For the case of the first four examples, the population size was ranged from 100 to 3000 individuals with increments of 100 (30 runs) and the maximum number of generations was set to 400. In the case of the last example, the population size was ranged from 1000 to 3000 with increments of 100 (20 runs), and the maximum number of generations was set to 2000. In each case, the results shown for each problem (including the population size used) correspond to the best solution at the median of all the runs (either 30 or 20 as indicated before). It is also important to add that whenever we use the term "optimum solution", we mean the best found by our approach, which might be a local optimum, since nobody has been able to determine (analytically) the global optimum for any of the circuits reported in this paper.

5.1 Example 1

Our first example has 4 inputs and one output, as shown in Table 1. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 for the case of the NGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$), and 225 for the case of the BGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 9 \times t = 225$). The cardinality c used for this problem was $max(r, g)$, for the NGA where g refers

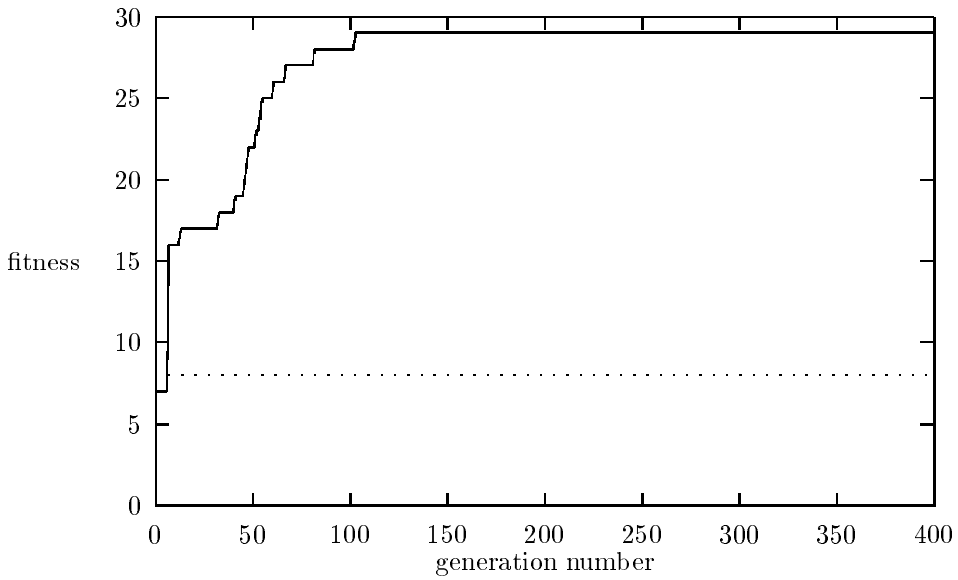

Figure 4: Convergence graph of the NGA used to solve the first example. The feasibility barrier is indicated with a horizontal line (any value above it represents a fully functional circuit).

to the number of allowable gates (since only the inputs from the previous level are considered, the number of columns does not affect the cardinality used by the NGA). Obviously, for the BGA, the cardinality $c = 2$. Since $g = 5$, and $c = 5$, then the size of the intrinsic search space for this problem is $c^l = 5^{75} \approx 2.6 \times 10^{52}$ for the NGA and $2^{225} \approx 5.39 \times 10^{67}$ for the BGA. The graphical representation of the circuit produced by the NGA is shown in Figure 5.

The comparison of the results produced by the NGA, the BGA and two human designers are shown in Tables 2, 3, and 4. In this and all the further examples, designer 1 used Karnaugh Maps plus Boolean algebra identities to simplify the circuit, whereas designer 2 used the Quine-McCluskey Procedure.

The parameters used by the NGA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 700, maximum number of generations = 400. The parameters for the BGA are the same except for the mutation rate that was instead 0.0022. The convergence graph of the NGA is shown in Figure 5.1. Convergence to the optimum was

NGA	Human Designer 1
$F = Z(X+Y) \oplus (XY)$	$F = Z(X \oplus Y) + Y(X \oplus Z)$
4 gates	5 gates
2 ANDs, 1 OR, 1 XOR	2 ANDs, 1 OR, 2 XORs

Table 2: Comparison of results between the n-cardinality GA (NGA) and a human designer for the circuit of the first example

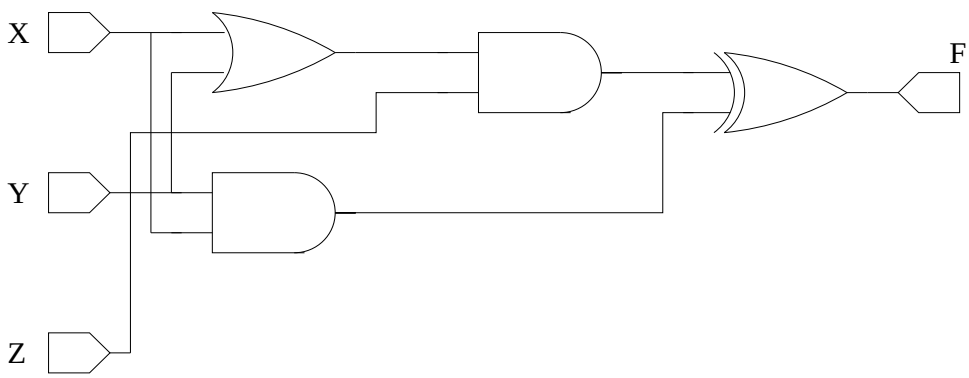

Figure 5: Circuit produced by an n-cardinality GA (NGA) for the first example.

achieved in generation 103 for the case of the NGA. The best solution found by the BGA had a fitness of 27 (the optimum solution had a fitness of 29), and was achieved at generation 188. It should be mentioned, however, that if a larger population size is used (900), the BGA is able to achieve the optimum solution in 197 generations, although at a higher computational expense. The solution found by the BGA has the same number of gates than the one found by the second human designer, although its Boolean expression looks more complex in the first case.

5.2 Example 2

Our second example has 4 inputs and one output, as shown in Table 5. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 for the case of the NGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$), and 225 for the case of the BGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 9 \times t = 225$). Again, the size of the intrinsic search space for this problem is $c^l = 5^{75} \approx 2.6 \times 10^{52}$ for the NGA and $2^{225} \approx 5.39 \times 10^{67}$ for the BGA. The graphical

BGA
$F = ((X \oplus Z) + (X \oplus Y))(Z' \oplus (X \oplus Y))$
6 gates
1 AND, 1 OR, 3 XORs, 1 NOT

Table 3: Results produced by the binary genetic algorithm (BGA) for the circuit of the first example

Human Designer 2
$F = X'YZ + X(Y \oplus Z)$
6 gates
3 ANDs, 1 OR, 1 XOR, 1 NOT

Table 4: Results produced by a second human designer for the circuit of the first example

Z	**W**	**X**	**Y**	**F**
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Table 5: Truth table for the circuit of the second example.

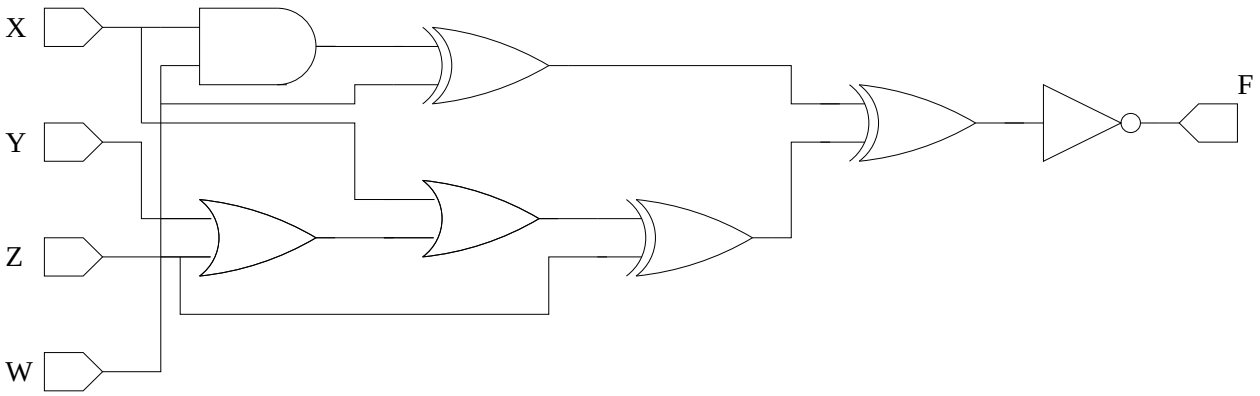

Figure 6: Circuit produced by an n-cardinality GA (NGA) for the second example.

NGA
$F = (((W \oplus WX) \oplus ((Z + X + Y) \oplus Z)))'$
8 gates
1 AND, 3 ORs, 3 XORs, 1 NOT

Table 6: Results produced by the NGA for the second example.

representation of the circuit produced by the NGA is shown in Figure 6.

The comparison of the results produced by the NGA, the BGA, a human designer, and Sasao's approach [35] are shown in Tables 6, 7, 8, and 9. Sasao has used this circuit to illustrate his circuit simplification technique based on the use of ANDs & XORs. His solution uses, however, more gates than the circuit produced by the NGA or the BGA. Note that the solution produced by the NGA is quite atypical, since it uses a negation at the end of the Boolean expression. Savings in this case, with respect to the best known solution were of 20%.

The parameters used by the NGA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 1000, maximum number of generations = 400. The BGA used a mutation rate of 0.0022 and required a larger population size (2000 chromosomes). The convergence graph of the NGA is shown in Figure 5.2. Convergence to the optimum in the case of the NGA was achieved in generation 376, and in the case of the BGA, in generation 328.

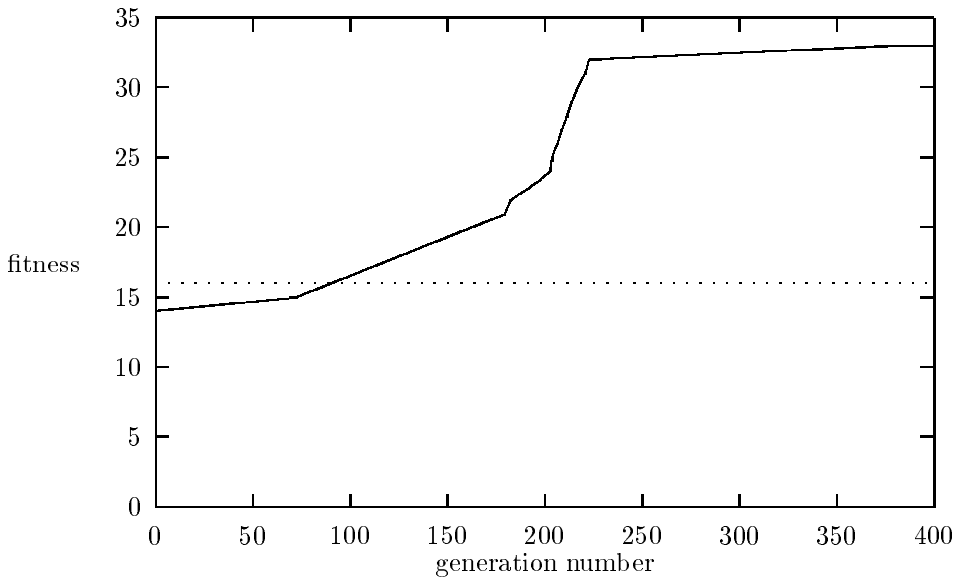

Figure 7: Convergence graph of the NGA used to solve the second example. The feasibility barrier is indicated with a horizontal line (any value above it represents a fully functional circuit).

BGA
$F = (Z \oplus ((W \oplus Y) + XY)) \oplus (Z + (X + Y))'$
8 gates
1 AND, 3 ORs, 3 XORs, 1 NOT

Table 7: Results produced by the BGA for the second example.

Human Designer 1
$F = ((Z'X) \oplus (Y'W')) + ((X'Y)(Z \oplus W'))$
11 gates
4 ANDs, 1 OR, 2 XORs, 4 NOTs

Table 8: Results produced by a human designer for the second example.

Sasao
$F = X' \oplus Y'W' \oplus XY'Z' \oplus X'Y'W$
12 gates
3 XORs, 5 ANDs, 4 NOTs

Table 9: Result produced by Sasao for the second example.

5.3 Example 3

Our third example has 4 inputs and one output, as shown in Table 10. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 for the case of the NGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$), and 225 for the case of the BGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 9 \times t = 225$). Again, the size of the intrinsic search space for this problem is $c^l = 5^{75} \approx 2.6 \times 10^{52}$ for the NGA and $2^{225} \approx 5.39 \times 10^{67}$ for the BGA. The graphical representation of the circuit produced by the NGA is shown in Figure 9.

The parameters used by the NGA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 700, maximum number of generations = 400. The BGA used a mutation rate of 0.0022, and was not able to find the same solution as the NGA (it found a solution with 8 gates instead), requiring a larger population size (900 chromosomes). The convergence graph of the NGA is shown in Figure 5.3. Convergence to the optimum in the case of the NGA was achieved in generation 326, and in the case of the BGA, in generation 227.

The comparison of the results produced by the NGA, the BGA, and two human designers are shown in Tables 11, 12, 13, and 14.

5.4 Example 4

Our fourth example has 4 inputs and 4 outputs, and it is a 2-bit multiplier as shown in Table 15. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 for the case of the NGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$), and 225 for the case of the BGA ($r = 5, q = 5, t = 5 \times 5 = 25, l = 9 \times t = 225$). Again, the size of the intrinsic search space for this problem is $c^l = 5^{75} \approx 2.6 \times 10^{52}$ for the NGA and $2^{225} \approx 5.39 \times 10^{67}$ for the BGA. The graphical representation of the circuit produced by the NGA is shown in Figure 11.

The comparison of the results produced by the NGA, the BGA, a human designer, and Miller et al. [28] are shown in Tables 16, 17 and 18,

W	X	Y	Z	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 10: Truth table for the circuit of the third example.

NGA
$F = ((XY \oplus (X+Z))(Y+(W \oplus Z)))'$
7 gates
2 ANDs, 2 ORs, 2 XORs, 1 NOT

Table 11: Result produced by our NGA for the third example.

BGA
$F = (((W \oplus Z)+WY)((X+Z) \oplus XY))'$
8 gates
3 ANDs, 2 ORs, 2 XORs, 1 NOT

Table 12: Result produced by our BGA for the third example.

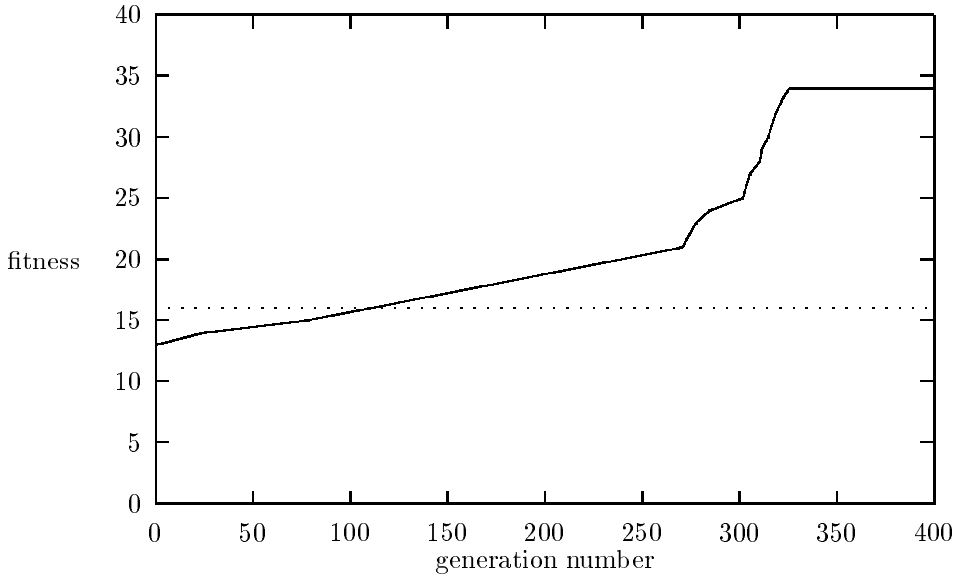

Figure 8: Convergence graph of the NGA used to solve the third example. The feasibility barrier is indicated with a horizontal line (any value above it represents a fully functional circuit).

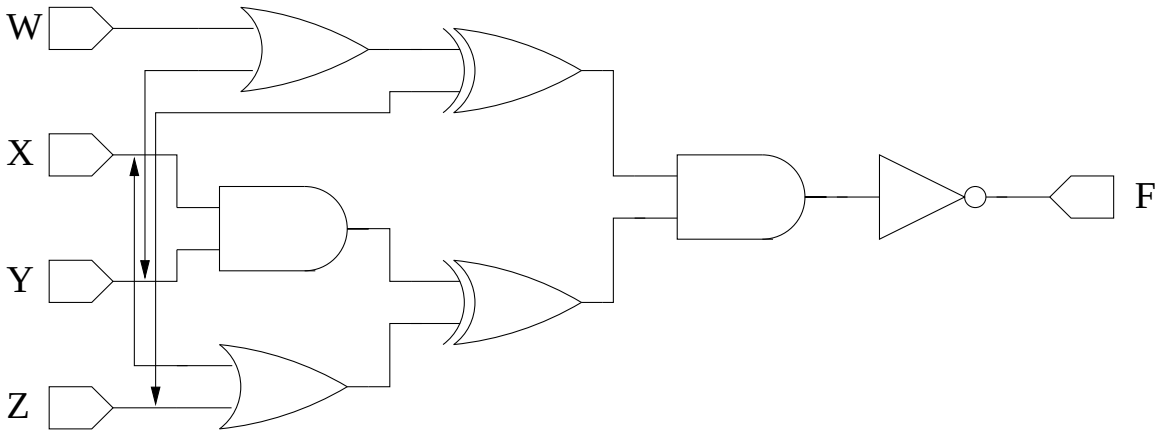

Figure 9: Circuit produced by our NGA for the third example.

Human Designer 1
$F = (Z + WX)' + XY + (WY')Z$
9 gates
4 ANDs, 3 ORs, 2 NOTs

Table 13: Result produced by a human designer for the third example.

Human Designer 2
$F = W'Z' + X'Z' + XY + WY'Z$
12 gates
5 ANDs, 3 ORs, 4 NOTs

Table 14: Results produced by a second human designer for the third example.

$\mathbf{A}_1$	$\mathbf{A}_0$	$\mathbf{B}_1$	$\mathbf{B}_0$	$\mathbf{C}_3$	$\mathbf{C}_2$	$\mathbf{C}_1$	$\mathbf{C}_0$
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Table 15: Truth table for the 2-bit multiplier of the fourth example.

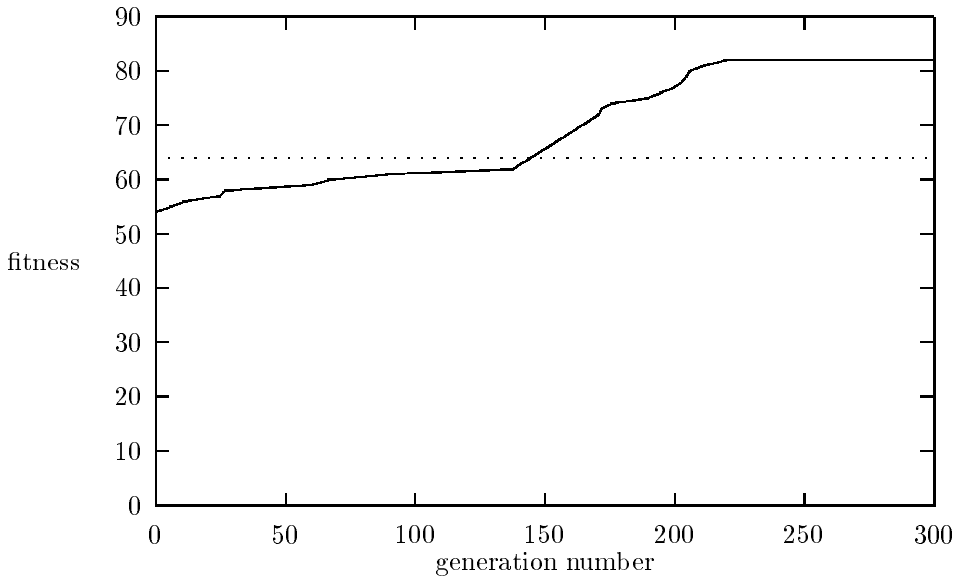

Figure 10: Convergence graph of the NGA used to solve the fourth example. The feasibility barrier is indicated with a horizontal line (any value above it represents a fully functional circuit).

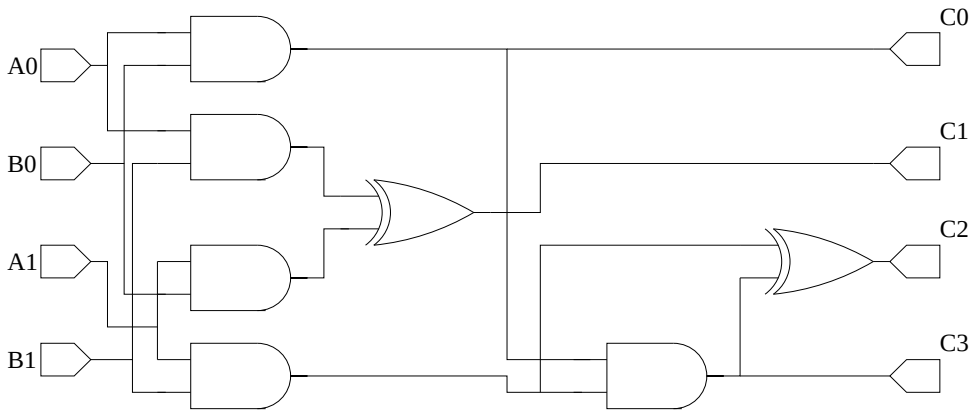

Figure 11: Circuit produced by our NGA for the fourth example.

respectively. It should be mentioned that Miller et al. consider their solution to contain only 7 gates because of the way in which they encoded their Boolean functions (the reason is that they encoded NAND gates, which is a common design practice). However, since we considered each gate as a separate chromosomic element, we count each of them, including NOTs that are associated with AND & OR gates. Regardless of that fact, it is more important to point out that Miller et al. found their solution performing 50 runs of 3,000,000 evaluations each, whereas in our case, we only performed 30 runs of 600,000 evaluations each.

Notice that the only difference between the solution produced by human designer 1 and the NGA is on the output C_2. This is the sort of example in which the solution may seem difficult to achieve by a human designer, because by looking at the solution for C_2 produced by the NGA, one could think that is more inefficient. However, the NGA is actually reusing gates which, in terms of the overall circuit, turns out to be more efficient, because it saves one gate with respect to the best solution produced by a human designer. With respect to the solution of Miller et al. [28], notice that it uses the same value for C_2 as human designer 1, but it has a much more complex expression for C_3. That is the reason why their overall circuit uses two more gates than our solution.

The parameters used by the NGA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 2000, maximum number of generations = 400. The best solution found by the BGA had 8 gates, and was generated with a larger population (2500 chromosomes). The mutation rate used was 0.0022 as in the previous examples. of the NGA is shown in Figure 5.4. Convergence to the optimum in the case of the NGA was achieved in generation 220, and in the case of the BGA, the best solution reported was found in generation 691.

5.5 Example 5

Our fifth example has 4 inputs and 3 outputs, as shown in Table 19. In this case, the matrix used was of size 6×7, and the chromosomic length was 126 for the case of the NGA ($r = 6, q = 7, t = 6 \times 7 = 42, l = 3 \times t = 126$), and 378 for the case of the BGA ($r = 6, q = 7, t = 6 \times 7 = 42, l = 9 \times t = 378$). The cardinality used for the NPGA was $c = max(r, g) = 6$. The size of the intrinsic search space for this problem is $c^l = 6^{126} \approx 1.1 \times 10^{98}$ for the NGA and $2^{378} \approx 6.16 \times 10^{113}$ for the BGA. The graphical representation of the circuit produced by the NGA is shown in Figure 12.

The comparison of the results produced by the NGA, the BGA, and a

NGA	**Human Designer 1**
$C_0 = A_0B_0$	$C_0 = A_0B_0$
$C_1 = A_0B_1 \oplus A_1B_0$	$C_1 = A_0B_1 \oplus A_1B_0$
$C_2 = A_1B_1 \oplus (A_0B_0A_1B_1)$	$C_2 = A_1B_1(A_0B_0)'$
$C_3 = A_0B_0A_1B_1$	$C_3 = A_1A_0B_1B_0$
7 gates	8 gates
5 ANDs, 2 XORs	6 ANDs, 1 XORs, 1 NOT

Table 16: Results produced by our NGA and a human designer for the circuit of the fourth example.

BGA	**Human Designer 2**
$C_0 = A_0B_0$	$C_0 = A_0B_0$
$C_1 = A_0B_1 \oplus A_1B_0$	$C_1 = (B_1 + B_0)(A_1 + A_0)((A_1A_0) \oplus (B_1B_0))$
$C_2 = A_0B_0 \oplus (A_0B_0 + A_1B_1)$	$C_2 = A_1B_1(A_0B_0)'$
$C_3 = A_0B_0A_1B_1$	$C_3 = A_1B_1A_0B_0$
8 gates	12 gates
5 ANDs, 2 XORs, 1 OR	8 ANDs, 1 XOR, 2 ORs, 1 NOT

Table 17: Results produced by the BGA and a second human designer for the circuit of the fourth example.

Miller et al.
$C_0 = A_0B_0$
$C_1 = A_1B_0 \oplus A_0B_1$
$C_2 = (A_0B_0)'(A_1B_1)$
$C_3 = (A_1B_0 \oplus A_0B_1)'(A_1B_0)$
9 gates
6 ANDs, 1 XORs, 2 NOTs

Table 18: Results produced by Miller et al. for the circuit of the fourth example.

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

Table 19: Truth table for the circuit of the fifth example.

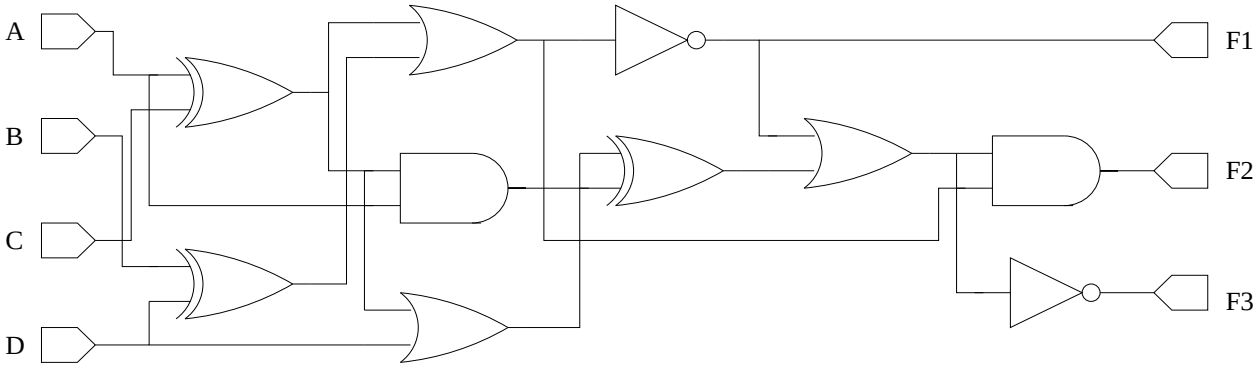

Figure 12: Circuit produced by our NGA for the fifth example.

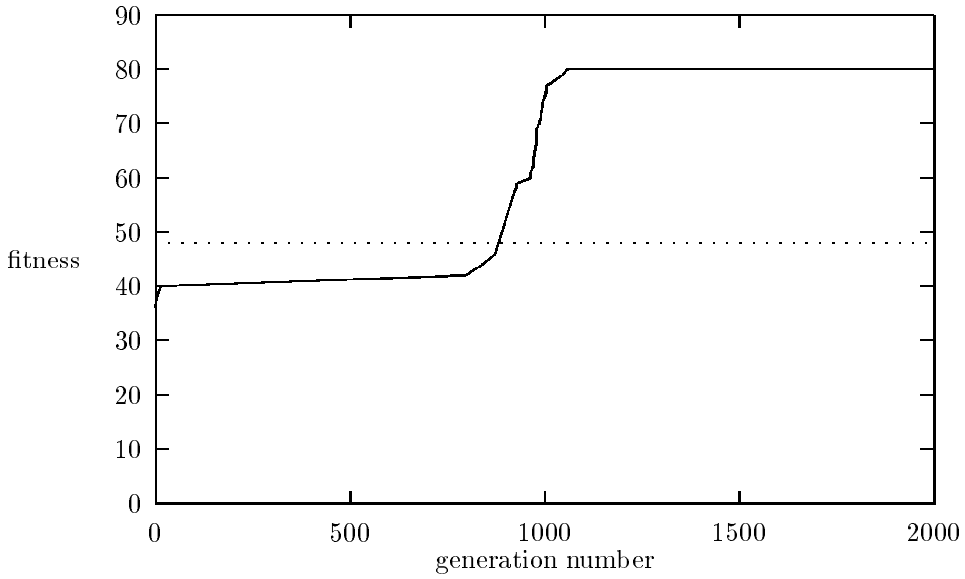

Figure 13: Convergence graph of the NGA used to solve the fifth example. The feasibility barrier is indicated with a horizontal line (any value above it represents a fully functional circuit).

human designer are shown in Tables 20, 21, 22, and 23.

This is an extreme case of how the NGA can reuse blocks of the circuit to optimize the total number of gates. Notice how the functions produced by the NGA for each separated output are more complex, but since they use common blocks, the total number of gates is almost half of what one of the human designers required.

The parameters used by the NGA for this example are the following: crossover rate = 0.5, mutation rate = 0.004, population size = 1000, maximum number of generations = 2000. The best solution found by the BGA had 11 gates and required a larger population (3500 chromosomes), running a larger number of generations. The mutation rate used with the BGA was 0.00132. The convergence graph of the NGA is shown in Figure 5.5. Convergence to the optimum in the case of the NGA was achieved in generation 1059, and the best solution reported for the BGA was found in generation 3755.

NGA
$F1 = ((A \oplus C) + (B \oplus D))'$
$F2 = ((B \oplus D) + (A \oplus C))(((A \oplus C)A \oplus (D + (A \oplus C))) + ((B \oplus D) + (A \oplus C))'$
$F3 = (((A \oplus C)A \oplus ((A \oplus C) + D)) + ((B \oplus D) + (A \oplus C))')'$
11 gates
3 XORs, 3 ORs, 2 ANDs, 3 NOTs

Table 20: Results produced by our NGA for the fifth example.

BGA
$F1 = ((A \oplus C) + (B \oplus D))'$
$F2 = ((A \oplus C) + (B \oplus D)) \oplus (((A \oplus C) + (B \oplus D)) \oplus ((C + A) \oplus A))((C \oplus A) + D')$
$F3 = ((C \oplus A) + D')(((B \oplus D) + (A \oplus C)) \oplus ((C + A) \oplus A)$
10 gates
5 XORs, 3 ORs, 1 ANDs, 1 NOT

Table 21: Results produced by our BGA for the fifth example.

Human Designer 1
$F1 = (A \oplus C)'(B \oplus D)'$
$F2 = B'D(A' + C) + A'C$
$F3 = BD'(A + C') + AC'$
19 gates
2 XORs, 4 ORs, 7 ANDs, 6 NOTs

Table 22: Results produced by a human designer for the fifth example.

Human Designer 2
$F1 = (A \oplus C)'(B \oplus D)'$
$F2 = A'C + (A \oplus C)'(B'D)$
$F3 = (F1 + F2)'$
13 gates
2 XORs, 2 ORs, 4 ANDs, 5 NOTs

Table 23: Results produced by a second human designer for the fifth example.

6 Discussion

It is interesting to notice that both the NGA and the BGA tend to favor the use of XOR gates, since this gate allows to produce in many cases solutions with a shorter symbolic representation. These solutions, however, are not entirely obvious for a human designer who can normally visualize easily only designs with the basic gates (AND, OR, NOT) and with XORs that are not nested. The GA (using either representation), on the other hand, tends to use very often nested XORs to produce the same effect that a human designer would achieve combining the basic gates. There it lies the main reason for which the GA tends to produce circuits that are difficult for a human to design and even to understand.

However, from the two alternative chromosome representations presented in this paper, our results show clearly the superiority of the NGA over the BGA both in terms of speed of convergence and in terms of total number of evaluations performed. The reason seems to be the capability of the NGA to encapsulate a higher-level representation of a circuit, allowing less disruption in the corresponding matrix (i.e., mutations produce more drastic but meaningful changes in the circuit) than when using a binary representation.

Although it may be argued that the relatively high number of evaluations performed by the GA (with either representation) is far beyond the search capabilities of a human designer, it must be said that the GA is in fact exploring a minimum portion of the total search space that is intractable by simple brute force search methods. For instance, for the examples in which the size of the intrinsic search space is 2.6×10^{52}, even assuming that our computer could evaluate 1×10^{12} solutions per second, we would need 8.39×10^{32} years to explore the entire search space using a brute force approach.

Some researchers might question the impact of this work in real-world circuit design, since the methodologies normally adopted by industry impose different specifications from those stated here. There are, however, some uses for the methodology presented in this paper. One possible application can be the evolution of circuits used for learning. Thompson [37], for example, used a GA to evolve the controller of a real robot in which the goal was to "learn" a wall-avoidance behavior. However, another aspect that we consider more important is the possibility of using the approach proposed in this paper to infer design principles. Some researchers have recently suggested the possibility of retrieving the knowledge that emerges during the design process using an evolutionary technique to re-discover and even propose new design rules [29]. This could be a very useful tool to teach

circuit simplification rules and could help us devise the design patterns used by evolutionary algorithms to simplify Boolean functions.

Finally, one issue that deserves attention is the scalability of the approach to real-world circuits. The approach presented in this paper can easily exceed the memory capabilities and processor speed of any workstation when applied to larger circuits unless their outputs are considered separately. This is an intrinsic limitation of the fixed-length representation adopted in this research, but we are exploring the use of more powerful representations (i.e., trees) that can overcome this limitation and allow the solution of larger circuits in a reasonable amount of time [12]. So far, our approach is limited to circuits of up to 6 inputs (i.e., 64 entries) and 6 outputs in their truth table (truth tables with don't cares can also be used). This limitation is, however, due to memory and CPU time constraints of our current computer equipment, and not to the algorithm itself. In any case, evolvable hardware in general is currently limited by the computing power available and all evolvable hardware experiments conducted so far have been on a small scale [39, 30, 11]. The two main problems commonly associated with scalability of evolvable hardware are the following [44]: 1) the scalability of the chromosome representation of electronic circuits and 2) the computational complexity of an evolutionary algorithm. Regarding the first issue, we can reach chromosomic lengths of a couple of thousand bits for circuits with 100 logic gates [38] and, if no constraint is imposed on the connectivity of a circuit, then the length of a chromosome is expected to grow in the order of $O(n^2)$, where n is the number of functional components (e.g., logic gates) [44]. The second issue still remains open, since no one has been able to establish the worst or average case time complexity of an evolutionary algorithm used to solve a particular problem, and current practice in evolvable hardware indicates that is rather common to perform runs that last several days to solve a single circuit with about 100 components (times will vary depending on the computer equipment available) [44].

7 Conclusions

We have shown a technique to design combinational logic circuits using a genetic algorithm, and we have explored the impact of changing from a traditional binary representation to a more compact n-cardinality representation. Our NGA has been able to find circuits that are smaller (in terms of the total number of gates) than those produced by human designers and even other GA-based approaches, performing a relatively small number of

evaluations with respect to the total size of the search space.

By analyzing the solutions produced by the NGA, it can be seen how it reuses components within a circuit as to reduce the total number of gates, even if in the process the Boolean expression for a certain output could become more complex than the one produced by a human designer. This reuse of components seems to be the key to find simplified versions of a circuit, but it becomes harder to understand it as we increase the complexity of the circuit.

Future Work

It is important to realize the difficulties of the GA (with either representation) to even generate a feasible circuit in early generations. If proper matrix dimensions are not provided, the GA may not converge at all regardless of the parameters used.

Although we have provided some basic guidelines to deal with this problem, we want to explore more flexible representations that allow an easier encoding of variable-length Boolean expressions as to minimize the tune up required to design any sort of combinational circuit. We would also like to extend our representation to handle circuits with more than two inputs. Right now, the most promising alternative seems to be the use of a GA with tree-representation (this approach is known as genetic programming [21]), but we still have to define a way of optimizing the Boolean expressions produced which are, generally, quite long, although we have some preliminary results along this research path [12]. Another possibility that we are considering is the use of genetic programming with a grammar representation to generate circuits [42, 43]. The use of a grammar representation seems to have several advantages, mainly because it can constraint the topology of the circuits produced without reducing in a significant way the generative power of the evolutionary technique. Some preliminary work developed by Jones and Joines [17] seems to indicate that this research path is worth exploring.

In the future we would also like to consider other factors in our fitness functions, such as the number of levels, the number of packages of integrated circuits used, the costs of each component, etc. Later on, we would like to move towards more complex circuits (for example sequential), and consider more complex factors such as time delays. Also, it would be desirable to build a graphical interface to our system, which currently has a text-based interface that makes the interpretation of results a little bit difficult for those not familiar with the software.

Acknowledgments

The authors thank the anonymous reviewers for their comments that greatly helped them to improve this paper. The first author acknowledges the support received from CONACyT through project number I-29870 A.

References

[1] J. Wirt Atmar. *Speculation on the Evolution of Intelligence and Its Possible Realization in Machine Form.* PhD thesis, New Mexico State University, Las Cruces, New Mexico, 1976.

[2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1984.

[3] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6 (6):1062–1081, November 1987.

[4] Janusz A. Brzozowski and Michael Yoeli. *Digital Networks.* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[5] Carlos Artemio Coello-Coello. *An Empirical Study of Evolutionary Techniques for Multiobjective Optimization in Engineering Design.* PhD thesis, Department of Computer Science, Tulane University, New Orleans, LA, April 1996.

[6] Charles Darwin. *The Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life.* The Book League of America, New York, 1929. Originally published in 1859.

[7] Hugo de Garis. Evolvable Hardware: Genetic Programming of a Darwin Machine. In Colin Reeves, R. F. Albrecht, and N. C. Steele, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 117–123, Inssbruck, Austria, 1993. Springer-Verlag.

[8] George J. Friedman. Selective Feedback Computers for Engineering Synthesis and Nervous System Analogy. Master's thesis, University of California at Los Angeles, February 1956.

[9] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing, Reading, Massachusetts, 1989.

[10] John J. Grefenstette. Deception Considered Harmful. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 75–91. Morgan Kaufmann, San Mateo, California, 1993.

[11] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development and evolution of hardware behaviors. In E. Sanchez and M. Tomassini, editors, *Toward Evolvable Hardware: The Evolutionary Engineering Approach (Lecture Notes in Computer Science, Vol. 1062)*, pages 250–265, Heidelberg, Germany, 1996. Springer-Verlag.

[12] Arturo Hernández-Aguirre, Carlos A. Coello-Coello, and Bill P. Buckles. A Genetic Programming Approach to Logic Function Synthesis by means of Multiplexers. In Adrian Stoica, Didier Keymeulen, and Jason Lohn, editors, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pages 46–53, Los Alamitos, California, 1999. IEEE Computer Society Press.

[13] Tetsuya Higuchi, Masaya Iwata, Isamu Kaijitani, Masahiro Murakawa, Shuji Yoshizawa, and Tatsumi Furuya. Hardware evolution at gate and function level. In *Proceedings of the International Conference on Biologically Inspired Autonomous Systems: Computation, Cognition and Action*, Durham, North Carolina, March 1996.

[14] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Harbor, Michigan, 1975.

[15] John H. Holland. *Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, Massachusetts, 1992.

[16] Hitoshi Iba, Masaya Iwata, and Tetsuya Higuchi. Gate-Level Evolvable Hardware: Empirical Study and Application. In Dipankar Dasgupta and Zbigniew Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 260–275. Springer-Verlag, Berlin, 1997.

[17] Eric A. Jones and William T. Joines. Genetic Design of Electronic Circuits. In Scott Brave and Annie S. Wu, editors, *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 125–133, Orlando, Florida, August 1999.

[18] M. Karnaugh. A map method for synthesis of combinational logic circuits. *Transactions of the AIEE, Communications and Electronics*, 72 (I):593–599, November 1953.

[19] Randy H. Katz. *Contemporary logic design*. Benjamin/Cummings Publishing Co., Redwood City, California, 1994.

[20] Hiroaki Kitano and James A. Hendler, editors. *Massively Parallel Artificial Intelligence*. MIT Press, Cambridge, Massachusetts, 1994.

[21] John R. Koza. *Genetic Programming. On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.

[22] John R. Koza, David Andre, III Forrest H. Bennett, and Martin A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 132–140, Cambridge, Masachussetts, July 1996. Stanford University, MIT Press.

[23] John R. Koza, III Forrest H. Bennett, David Andre, and Martin A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 123–131, Cambridge, Masachussetts, July 1996. Stanford University, MIT Press.

[24] Sushil J. Louis. *Genetic Algorithms as a Computational Tool for Design*. PhD thesis, Department of Computer Science, Indiana University, August 1993.

[25] Sushil J. Louis and Gregory J. Rawlins. Using Genetic Algorithms to Design Structures. Technical Report 326, Computer Science Department, Indiana University, Bloomington, Indiana, February 1991.

[26] E. J. McCluskey. Minimization of boolean functions. *Bell Systems Technical Journal*, 35 (5):1417–1444, November 1956.

[27] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York, third edition, 1996.

[28] J. F. Miller, P. Thomson, and T. Fogarty. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 105–131. Morgan Kaufmann, Chichester, England, 1998.

[29] Julian F. Miller, Tatiana Kalganova, Natalia Lipnitskaya, and Dominic Job. The Genetic Algorithm as a Discovery Engine: Strange Circuits and New Principles. In *Proceedings of the AISB Symposium on Creative Evolutionary Systems (CES'99)*, Edinburgh, Scotland, April 1999.

[30] Julian F. Miller and Peter Thomson. Evolving Digital Electronic Circuits for Real-Valued Function Generation using a Genetic Algorithm. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming: Proceedings of the Third International Conference*, pages 863–868, San Francisco, California, 1999. Morgan Kaufmann Publishers.

[31] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, Massachusetts, 1996.

[32] W. V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62 (9):627–631, 1955.

[33] Simon Ronald. Robust Encodings in Genetic Algorithms. In Dipankar Dasgupta and Zbigniew Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 29–44. Springer-Verlag, Berlin, 1997.

[34] Charles H. Roth, Jr. *Fundamentals of Logic Design (4th Edition)*. West Publishing Company, Minneapolis, 1992.

[35] Tsutomu Sasao, editor. *Logic Synthesis and Optimization*. Kluwer Academic Press, Dordrecht, The Netherlands, 1993.

[36] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the AIEE*, 57:713–723, 1938.

[37] Adrian Thompson. Evolving electronic robot controllers that exploit hardware resources. In *Proceedings of the 3rd European Conference on Artificial Life (ECAL'95)*, pages 640–656. Springer-Verlag, 1995.

[38] Adrian Thompson, I. Harvey, and Philip Husbands. Unconstrained evolution and hard consequences. In E. Sanchez and M. Tomassini, editors, *Toward Evolvable Hardware: The Evolutionary Engineering Approach (Lecture Notes in Computer Science, Vol. 1062)*, pages 136–165, Heidelberg, Germany, 1996. Springer-Verlag.

[39] Adrian Thompson, Paul Layzell, and Ricardo Salem Zebulum. Explorations in Design Space: Unconventional Design Through Artificial Evolution. *IEEE Transactions on Evolutionary Computation*, 3(3):167–196, September 1999.

[40] Brian C. H. Turton. Extending Quine-McCluskey for Exclusive-Or Logic Synthesis. *IEEE Transactions on Education*, 39(1):81–85, February 1996.

[41] E. W. Veitch. A Chart Method for Simplifying Boolean Functions. *Proceedings of the ACM*, pages 127–133, May 1952.

[42] P. A. Whigham. Grammatically-based Genetic Programming. In J. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41. Morgan Kaufmann Publishers, July 1995.

[43] Peter Wyard. Context Free Grammar Induction Using Genetic Algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 512–517, San Mateo, California, 1991. Morgan Kaufmann.

[44] Xin Yao and Tetsuya Higuchi. Promises and Challenges of Evolvable Hardware. In Tetsuya Higuchi, Masaya Iwata, and W. Liu, editors, *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES'96), Lecture Notes in Computer Science, Vol. 1259*, pages 55–78, Heidelberg, Germany, 1997. Springer-Verlag.

Carlos A. Coello Coello received his Ph.D. in Computer Science from Tulane University in 1996. He has been senior research fellow at the Plymouth Engineering Design Centre (UK), and is currently a researcher at LANIA, in México. His current interest is the use of evolutionary techniques for engineering optimization, manufacturing and design.

Alan D. Christiansen received his Ph.D. in Computer Science from Carnegie Mellon University in 1992, and joined the Tulane University faculty in 1993. He has also been employed at Sandia National Laboratories, Microsoft Corporation, and Science Applications International Corporation. His research interests include artificial intelligence, robotics, computer graphics, modeling and simulation.

Arturo Hernández Aguirre received his Ph.D. in Computer Science from Tulane University in 1999. He worked for several years at the Arturo Rosenblueth Foundation as a Software Consultant in Mexico City and is currently a Visiting Professor at Tulane University. His research interests include computational learning, neural networks, and evolutionary computation.