Extracting and Re-Using Design Patterns from Genetic Algorithms using Case-Based Reasoning

Eduardo Islas Pérez
Instituto de Investigaciones Eléctricas
Av. Reforma # 113
Col. Palmira 62490
Temixco, Morelos, MEXICO
eislas@axp18.iie.org.mx

Carlos A. Coello Coello*
CINVESTAV-IPN
Depto. de Ingeniería Eléctrica
Sección de Computación
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco
México, D. F. 07300, MEXICO
ccoello@cs.cinvestav.mx

Arturo Hernández Aguirre
Centro de Investigación en Matemáticas
Área Computación
Callejón Jalisco s/n
Mineral de Valenciana
Guanajuato, Guanajuato 36240, MEXICO
artha@cimat.mx

June 17, 2003

Abstract

This paper proposes a scheme in which case-based reasoning techniques are employed to extract design patterns from a genetic algorithm used to optimize

*Corresponding author

combinational circuits at the gate level. The approach seems to be able to (implicitly) rediscover several of the traditional Boolean rules used for circuit simplification and it also (implicitly) finds new simplification rules. Also, we illustrate how the approach can be used to reduce convergence times of a genetic algorithm using previously previously found solutions as cases to solve similar problems.

Keywords: circuit design, genetic algorithms, case-based reasoning, computer-aided design, artificial intelligence.

1 Introduction

There is a great contrast between the way in which physical systems have been designed by the bottom-up method (blind evolution) and the top-down method employed by human designers. In the former case entire systems are constructed and tested *in situ* without a conscious application of design principles. In the latter, systems are "evolved" by a process of human ingenuity, which employs a set of theorems, rules, concepts and principles. It is indeed curious that the most intrincate designs found in Nature (from which man may be its most complex creation) are really a byproduct of a set of blind forces of physics guided by a survival mechanism [5, 4].

Although it is difficult for an evolutionary algorithm to suggest directly new design principles (because new "laws" must be implicitly induced from samples instead of explicitly deduced from the whole domain), it is feasible to infer such principles through a careful study and analysis of its behavior on a set of examples.

That is precisely the focus of this paper. We propose that by employing a blind evolutionary approach new design concepts may emerge, and that these concepts can be re-used for solving new problems. The emergence of design patterns has already been observed by human designers who have captured such patterns in the form of design principles, theorems, and laws. In our approach, though, humans play a minor role. Concepts being formed during the evolutionary process are analyzed and new emergent design patterns are identified and stored for further use, challenging traditional assumptions and principles.

We believe that a well-suited domain to test our hypothesis (any other well defined domain could also be used), is precisely the field of combinational circuit design, since in this area human designers have well-defined design principles and simplification rules. The objective is also well-defined: to produce an electronic or algebraic machine that carries out a definite function (e.g., addition) on a number of input variables. Additionally, we want this design to be optimum in a sense. For the purposes of this paper, optimality will be defined in terms of the number of gates employed by a function circuit (i.e., we wish to produce a circuit that matches all the outputs of the truth table and, at the same time, we want such a circuit to use as few gates as possible). In this paper, we will show how an evolutionary algorithm can produce (through an emergent process that implicitly takes place within the search engine of the genetic algorithm) simplification rules that human designers can use. In fact, we will see how some of these rules are really the same traditionally used by human designers. However, others are entirely new simplification rules which, in some cases, may not even be intuitive to

a human designer.

Another interesting aspect of this work is that we show how case-based reasoning can be used to perform modular design of circuits. The idea is to use small components as building blocks to produce more complex circuits. This idea, although intuitive, is not completely straightforward in practice, since the selection pressure of an evolutionary algorithm may destroy partial solutions to a problem. Our approach is therefore, to use a database of solutions previously found that have some (potentially) useful information. Then, using techniques from case-based reasoning, we retrieve this information when designing similar circuits (similarity has to be defined according to certain criteria in this context) and incorporate it in the population of another evolutionary algorithm, as to reduce convergence times and to encourage modular design. The system will be illustrated with the design of a full adder.

2 Related Work

This paper extends our previous work in combinational circuit design using genetic algorithms (GAs) [1, 2], and it attempts to show the potential of incorporating domain-specific knowledge generated by the GA itself into other GAs used to solve similar problems.

Apparently, the first attempt to combine case-based reasoning (CBR) and GAs was done by Louis et al. [17]. In this paper, the authors use CBR-principles to explain solutions found by a GA. This same idea was also discussed in Louis' dissertation [15], where he proposed a system that combined CBR with GAs to improve performance of the GA. These ideas were further developed by Louis & Johnson [16] and by Liu [14]. Although Louis [15] and Louis & Johnson [16] used a few examples from circuit design (mainly parity checkers) to illustrate their principles, they did not focus their work specifically on the design of combinational circuits as in our case. Nevertheless, our current proposal has been influenced by this prior work. Note, however, that Louis et al. [17] extract knowledge only after finishing a GA run, whereas in the work presented in this paper, knowledge can also be extracted *during* a GA run.

Several other researchers have proposed approaches that combine CBR and GAs. See for example [26, 21, 22]. However, the emphasis of these papers has been to illustrate the benefits of this sort of hybrid scheme rather than emphasizing a certain application domain like in our case.

Also, some researchers in evolvable hardware have pointed out the potential benefits of using GAs as a discovery engine capable of producing novel and even inspirational designs. Miller et al. [20], for example, showed that through the evolution of a hierarchical series of examples, it was possible to rediscover the well-known ripple-carry principle for building adder circuits of any size. However, no CBR is used in this work. The possibility of seeing the extraction of design rules from an evolutionary algorithm as a form of data mining is also suggested by [18]. Finally, in [19], the techniques for landscape analysis developed in [24] are studied. Also, the authors discuss the use of case-based reasoning techniques to extract and reuse rules implicitly used by an evolutionary algorithm [19]. In this case, a nearest neighbor matching function is used to rank cases in the case-base.

Vassilev et al. [25] proposed a navigation strategy to evolve "conventional" circuit designs taking advantage of the so-called *neutrality bridge* (this is the term used by the authors to denote the gap between the designs produced by a human and the designs produced by an evolutionary algorithm). This can also be seen as a form of reuse of previous domain knowledge (the designs produced by humans in this case) to improve the performance of an evolutionary algorithm.

Thomson [23] explored the potential of evolving larger systems more quickly via a method of visualizing the subcomponents of the final solution when they appear. Taking these partially evolved solutions from short runs and feeding them to another GA, the convergence time of the GA can be improved. This work is closer to our own, but unlike our proposal, Thomson does not use CBR in his system.

Recently, Haddow et al. [7] proposed to use L-systems in evolvable hardware applications based on FPGAs (Field Programmable Gate Array). Since the universal building blocks available in programmable logic have been found hard to evolve [10, 12], they propose to adapt L-systems to evolving digital circuits within the constraints of the technology.

The problem the evolvable hardware community faces is to find building blocks suitable for evolution. Gero & Kazakov [6] have also studied this problem but in the architectural design domain. Their method works in two stages: first, the building blocks that produce designs with desired characteristics are evolved; then these building blocks are used to seed the initial population for evolving the final design.

Our approach does not need to evolve suitable building blocks since the evolving set of logic gates is known in advance. The set is sound and complete in Boolean logic (a design issue that Gero & Kazakov cannot prove for their problem domain), thus our goal is to assists the evolutionary process by providing it with simplification rules previously used in the evolution of related problems.

Our work aims then to explore the potential of CBR combined with GAs to design combinational circuits which can be optimized according to a certain metric (number of gates, in our case).

3 Case-Based Reasoning

Case-Based Reasoning (CBR) is a problem-solving paradigm that in many respects is fundamentally different from other major AI approaches [13]. Instead of relying solely on general knowledge of a problem domain, or making associations along generalized relationships between problem descriptors and conclusions, CBR is able to utilize the specific knowledge of previously experienced, concrete problem situations (cases). Finding a similar past case, and reusing it in the new problem situation helps to solve a new problem. A second important difference is that CBR is also an approach to incremental, iterative learning, since a new experience is retained each time a problem has been solved, making it immediately available for future problems.

Much of human knowledge is based on how a previous problem was solved instead of applying abstract and specific rules about a possible solution to that problem. In CBR if the same situation is presented many times, the solution always has to be found by returning to the beginning.

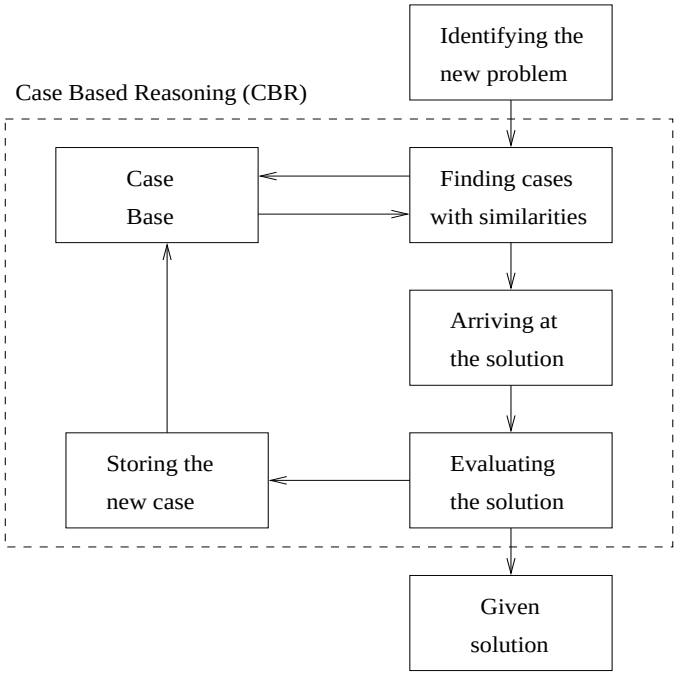

Figure 1: General structure of a CBR system.

A CBR system can be divided in the following main stages (see Figure 1):

1. **Identifying the new problem:** The system receives the input case (new problem) and analyzes its most important attributes and characteristics in order to search amongst the cases that are most similar to the cases in the case base. The attributes used to measure the similarity between the cases are called indexes.

2. **Finding cases with similarities to the new case:** The following step is to find the cases that have most attributes in common with the attributes of the new case using the indexes found in the previous step. Sometimes it is necessary to reduce the subset in order to find the most relevant cases. The algorithm should be fast and efficient and the design is a critical and important aspect when the case base is sufficiently large. The selection of cases from the case base could be considered as analogous to natural selection due to the fact that it is based only on the distance measure (similarity rather than fitness) between the new case and each case in the case base.

3. **Arriving at the Solution:** Once we have the most similar cases, the system starts the adaptation process, which consists of the combination and modification of the most similar cases to form a new solution, and additionally an interpretation or

an explanation depending on the application of the system. In most applications it is better if the system explains how it finds the new case.

4. **Evaluating the solution:** The solution obtained in the previous stage is a tentative or potential solution. It is necessary to do an evaluation of the proposed solution before giving it to the final user. This evaluation should show the qualities and weaknesses of the solution for the evaluation of its usefulness.

5. **Assignment and storing of the new case:** Once the solution has been created and evaluated, it is given to the user and then it is possible to create a new case. This new case is formed from the solution found and the original case (problem). Indexes are assigned to the new case and it is stored in the case base.

6. **Explaining, repairing and testing:** If the solution fails, it is important that the system obtains and analyzes the information in order to avoid making the same mistakes. If something unusual happens, the system should try to explain it. Subsequently, the system repairs the solution based on the explanation and returns to the evaluation stage.

4 Statement of the Problem

We propose an approach to extract design patterns from a genetic algorithm used to design combinational circuits. We will extract knowledge at two stages of the evolutionary process: at the end of a run and during a run. In the first case, the knowledge to be extracted will be the laws of Boolean algebra used by the evolutionary algorithm to design a circuit. These laws will be obtained after comparing the circuits produced from two or more runs of the GA (with different parameters) with the solution produced by a human expert.

In the second case (extraction during a run), the knowledge extracted will be the building blocks that the circuit structurally maintains during its evolutionary process. When some individuals arrive at a certain (predefined) threshold in their fitness value during the evolutionary process, it means that these circuits have evolved long enough to contain good building blocks and we can then extract the knowledge that they contain and store it in a case base for further use.

We are interested in showing the potential of combining GAs with case-based reasoning to improve performance of the GA used to solve similar problems. The idea is to store solutions that were previously generated by the same GA and use them as a memory of "past experiences". Then, we can use a mechanism to detect cases similar to the one being solved and retrieve from this "memory" some solutions (or past experiences) that can be useful to solve the problem at hand.

For the experiments described next, we use the genetic algorithm with integer representation and matrix representation (encoded as fixed-length linear chromosomes) that we have adopted in previous work [1, 2] (see Figure 2). Our GA uses a fitness

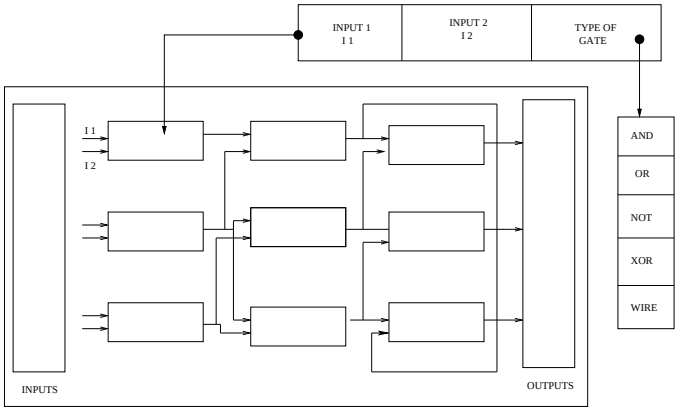

Figure 2: Matrix used to represent a circuit. Each gate gets its inputs from any gate in the previous column. Note the encoding adopted for each element of the matrix as well as the set of available gates used.

function that works in two stages: first, it tries to reach the feasible region (i.e., it tries to produce circuits which match all the outputs of the truth table) and then, once feasible circuits are available, it tries to maximize the number of WIRE gate. WIRE represents the absecence of gate. WIRE can be seen as a special type of gate that does not perform any task and it passes one of its inputs directly to the gate to which it is connected. In our actual implementation, we use WIRE1 and WIRE2, where the number indicates the input to transfer. This type of gate is used to allow variable-length Boolean expressions within our fixed-length representation. By maximizing the number of WIREs of a feasible circuit, we are actually trying to minimize the number of gates that a (feasible) circuit contains.

5 Proposed System

The proposed system that combines a GA with CBR is depicted in Figure 3. Note that in this figure we used the term "scheme". A schema is a pattern of bits which (according to the schema theorem [9]) the genetic algorithm uses to approach the optimum of a problem over time. A schema is therefore a string composed by three symbols: one, zero and "don't cares" (which are represented by an asterisk in this figure). As their name indicates, "don't cares" have a undefined value which, at the moment is irrelevant (it could be either one or zero). Over time, all "don't cares" will become either one or zero and all the asterisks will then be eliminated.

To understand better the way in which our system works, we will describe in more detail the process of extracting knowledge in the two situations previously mentioned:

1. **At the end of the evolutionary process**: In this case, we perform complete runs of a GA solving a certain circuit. Once a solution is found, a new case is formed with such a solution and the original problem. The original problem will be con-

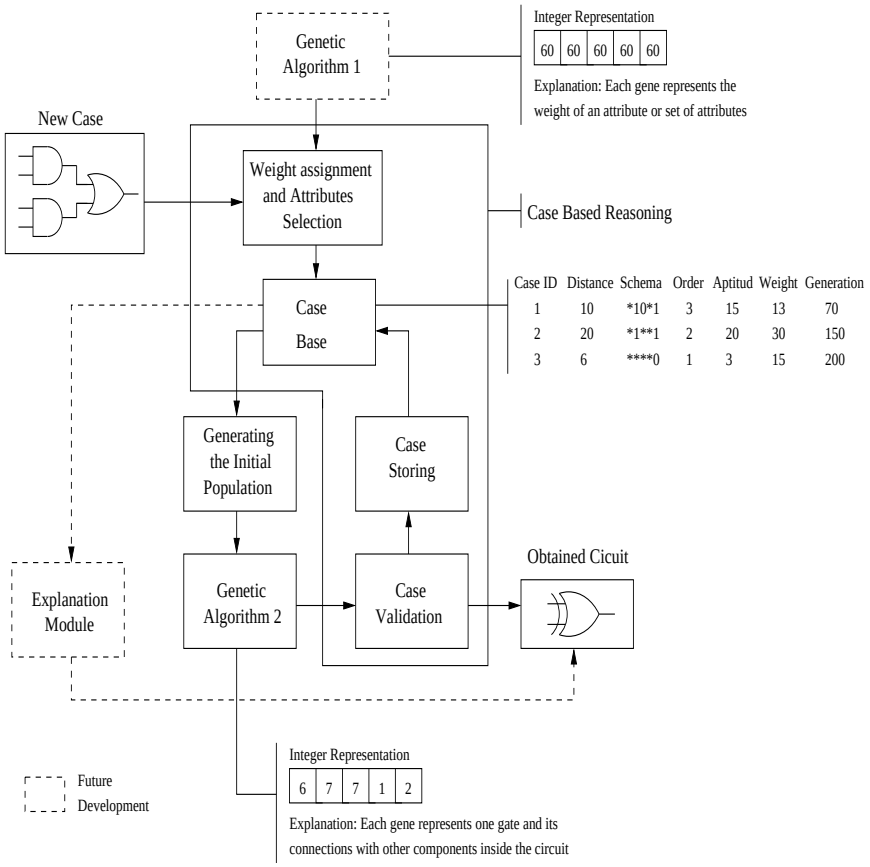

Figure 3: Proposed system to optimize combinational logic circuits using GAs and CBR.

sidered as the attributes in the case base and the solution will be the output of the case. The system will assign other attributes, in order to have indexes that help retrieving the most similar cases in a more efficient way.

2. **During the evolutionary process**: In this case, our work is inspired on the research of Louis [15]. The GA records data for each individual in the population as it is created and evaluated. Such data includes a fitness measure, the genotype and chronological data, as well as some information on the individual's parents. This collection of data is the initial case data. Though normally discarded by the time an individual is replaced, all of the case data collected is usually contained in the genetic algorithm's population at some point and it is easy to extract. When a sufficient number of individuals have been created over a number of generations, the initial case data is sent to a clustering program. A hierarchical clustering program clusters the individuals according to both, the fitness and the alleles of the genotype. This clustering constructs a binary tree in which each leaf includes the data of a specific individual. The binary tree structure provides an index for the initial case base. The numbers at the leaves of the tree correspond to the case number (an identification number) of an individual created by the GA. An abstract case is computed for each internal node based on the information contained in the leaves and nodes beneath. The final case base includes: 1) cases corresponding directly to GA individuals (at the leaves) and 2) more abstract cases made up of information generalized from the leaves.

The new knowledge gathered during these situations is entered as cases in the CBR system from where it is retrieved to seed the initial population of a new problem.

5.1 Representing circuits as strings

Figure 5 shows an example of the three different representations of a logic circuit that we normally adopt in the system proposed in this paper: (1) a graphical representation using two-input gates (used to illustrate the final solutions produced by our system), (2) a symbolic two-dimensional matrix (used by our system to represent the solutions found during the evolutionary process) and (3) a string of integers (the genotypes manipulated by our evolutionary algorithm to perform the search). The integer representation adopted for the genotypes is composed of triplets representing the two inputs and the gate type. All the gates and possible inputs available are encoded by an integer. For example, the triplet **(0 2 0)** represents that the gate AND (encoded in the string by number zero) receives its first input from the element zero (assuming the matrix representation described in Figure 2) and its second input from the element two. In our implementation, the values encoded for the inputs are added one when decoded. Therefore, this is interpreted as AND(1,3), where 1 indicates the row of the first output and 3 indicates the row of the second output (the column is the previous to the location of the gate).

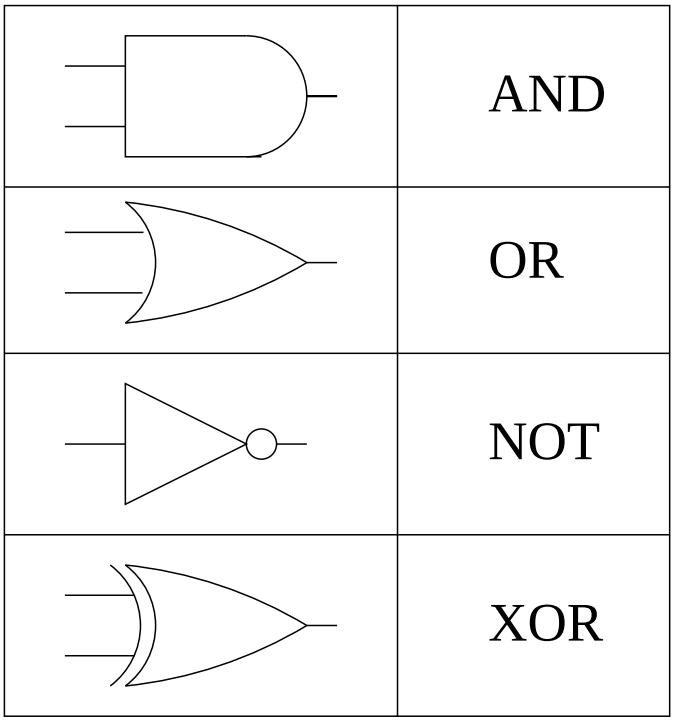

Figure 4: Graphical representation of the gates used to build the circuits in in this paper.

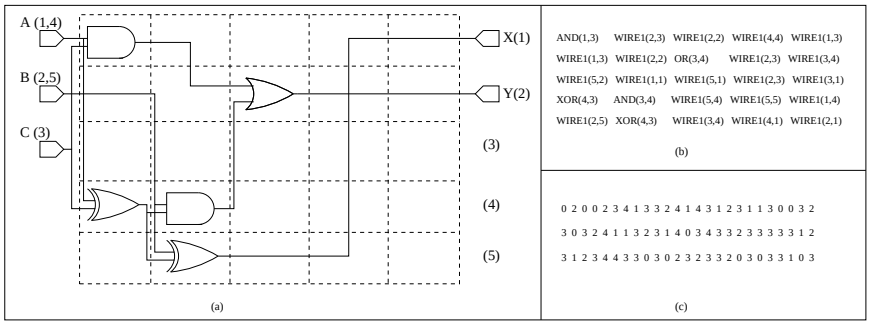

Figure 5: Three different ways of representing the same circuit: (a) a graphical representation using two-input gates, (b) a symbolic matrix, and (c) a string of integers. The graphical representation adopted for the gates is explained in Figure 4.

Table 1: Cases for knowledge extraction at the end of the evolutionary process.

Case ID	Num. Inputs	Num. Outputs	Output Values	Fitness	Genotype
1	3	2	0110100100010111	39	3230132431232134103231
2	3	2	0110100100010111	38	323013243123214413 3204
3	3	2	0110100100010111	39	0200234133241431231130
4	2	2	0110100100010111	31	0100142134131433134130
5	3	2	0110100100010111	36	0100142134131433134130
6	2	2	0110100100010111	31	0200234133241431231130

5.2 Representing circuits in the case base

Depending on the stage at which knowledge is extracted, the representation adopted to store it in the case base can vary:

1. **At the end of the evolutionary process**: The cases will be stored from problems that have been solved previously and they will be used for seeding the initial population of a GA. The attributes contained in this part of the case base are the following[1]:

   - Case ID
   - Number of Inputs
   - Number of Outputs
   - Output Values
   - Fitness
   - Genotype

   Some examples of these sort of cases stored in the case base are shown in Table 1. The attribute **output values** is used to verify that the outputs indicated in the truth table are satisfied. All those individuals matching the desired output values are selected as "cases" to be stored in the case base. Upon their storage, these cases are sorted according to their fitness (from largest to smallest fitness value). The idea is that the cases with the highest fitness can be used in the future to seed the initial population of another genetic algorithm.

2. **During the evolutionary process**: The best individuals are recognized during early generations of the evolutionary process. Afterwards they are stored as cases in the case base and retrieved in later generations. Some of the attributes that are contained in this part of the case base are the following:

[1] This scheme presents certain resemblance with the one proposed by Louis [15].

Table 2: Cases for knowledge extraction during the evolutionary process. See Section 5 for an explanation of the term “schema”.

Case ID	Distance	Schema	Order	Fitness	Weight	Generation
1	5	710*13*2*	6	30	6	50
2	2	**4*50*2*	4	60	8	30
3	8	163*14*41	7	15	4	67
4	8	350610*7*	7	65	4	32
5	7	214*16169	8	30	6	50
6	4	**3*10*2*	4	60	8	26

- Case ID
- Distance from the root of the tree to the level of the case
- Schema for the case
- Schema order
- Average fitness
- Weight: Number of leaves (individuals) below
- Generation information: the earliest and latest leaf occurrence as well as the average in the subtree.

Some examples of this sort of cases stored in the case base are shown in Table 2.

Additionally, we also perform some analysis (by hand) to try to understand the way in which the GA performs the simplification of a circuit. As we will show in the examples presented next, the GA is able to rediscover several of the simplification rules commonly used in Boolean algebra and, furthermore, was able to discover “new” simplification laws that are stored in the case base and can also be used by human designers.

6 Examples

Next, we provide two examples of how the knowledge is extracted both at the end and during the evolutionary process of a GA with integer representation used to design combinational logic circuits at the gate-level. In the first case, at the end of the evolution, knowledge is derived from the Boolean rules that represent sound transformations performed by the GA over the circuits. In the second case, during the evolution process, knowledge is derived from stationary building blocks (without change during the evolution) utilized by the GA to construct a circuit.

Table 3: Truth table for the circuit of the first example (a parity checker).

A	B	C	D	X
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

6.1 Example 1

In this case, the aim is to find the Boolean expression that corresponds to the circuit whose truth table is provided in Table 3. We will start by providing the steps followed to extract knowledge at the end of the evolutionary process. First, we performed 10 runs using integer representation and the following parameters[2]: population size=600, maximum number of generations = 200, crossover rate = 0.6, mutation rate = 0.001. The best solution found from these runs has 9 gates and its corresponding Boolean expression is shown (under "GA Setup 1") in Table 4. This Boolean expression is not better than the best solution found by a Human Designer using Karnaugh maps (this solution has 6 gates). However, we additionally performed 10 more runs using a population size of 3000 and a maximum number of generations of 120. The best solution found from these runs has 4 gates (i.e., it is better than the solution produced by a human expert) and its corresponding Boolean expression is shown (under "GA Setup 2") in Table 4.

6.1.1 Analysis

The next step was to analyze (by hand) the solutions produced by our GA with respect to those generated by the human designer. This analysis intends to show how the selection mechanism used by our GA is implicitly "discovering" simplification rules similar to those adopted by human designers. Therefore, such rules do not need to be

[2]The parameters indicated were empirically derived after performing a set of experiments.

Table 4: Comparison of results between a human designer and two setups of our GA for the first example. Note that $\oplus$ is used to represent XOR, $+$ is used to represent OR, and $'$ is used to represent NOT. The absence of operator indicates AND.

Human Designer
$X = ((A \oplus B)' \oplus (C \oplus D)')'$
6 gates
3 XORs, 3 NOTs
GA Setup 1
$X = ((A \oplus C) \oplus B)' \oplus ((B'B) + D')'$
9 gates
1 AND, 1 OR, 3 XORs, 4 NOTs
GA Setup 2
$X = ((A \oplus B) \oplus (C \oplus D))'$
4 gates
1 NOT, 3 XORs

provided explicitly to the GA in order to produce useful circuits and are instead derived by the "blind" search engine used by a GA.

Let us analyze the solution found for the circuit described before:

$$X = ((A \oplus B)' \oplus (C \oplus D)')' = ((A \oplus B) \oplus (C \oplus D)') \tag{1}$$

We have discovered a "new" DeMorgan's theorem[3] for XOR gates of the type:

$$(S' \oplus T')' = (S \oplus T)' \tag{2}$$

A case stored in the case base as a product of the analysis at the end of the evolutionary process is shown in Table 5.

Then, we performed an analysis during the evolutionary process, trying to detect the basic building blocks used by the evolutionary algorithm to generate the best solutions produced. Figures 6, 7 and 8 show several snapshots of the solutions produced by our GA with the second set of parameters previously described (population size = 3000, maximum number of generations = 120). From these pictures, we can see that the circuit has a fitness value of 17 at generation 9 and we were able to recognize the building blocks used by the GA (such building blocks are indicated with a thicker box). At generation 67, the maximum fitness has increased, reaching 27, and we can observe that the building blocks previously mentioned have moved to a different position. Finally, when reaching generation 101, we have a fitness of 37 (i.e., a feasible circuit with only 4 gates). Although the building blocks are in a different position, the circuit has the same behavior as in earlier stages of the design. For this reason, we proceed to store it in our case base.

[3]By "new" we mean that this DeMorgan theorem is not part of the basic set of Boolean algebra simplification rules normally adopted for circuit design.

Original case		Solution	Description	Number of gates eliminated
Case 1	$(S' \oplus T')'$	$(S \oplus T)'$	DeMorgan's theorem applied to XOR obtained from the comparison between the solution by the second run of the GA and the solution obtained by a human designer	4 - 2 = 2

Table 5: Case stored in the case base at the end of the evolutionary process for the circuit whose truth table is shown in Table 3.

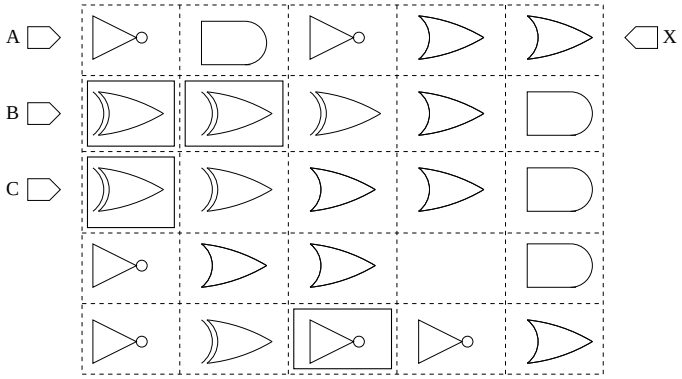

Figure 6: Solution obtained at generation 9 for the circuit whose truth table is shown in Table 3.

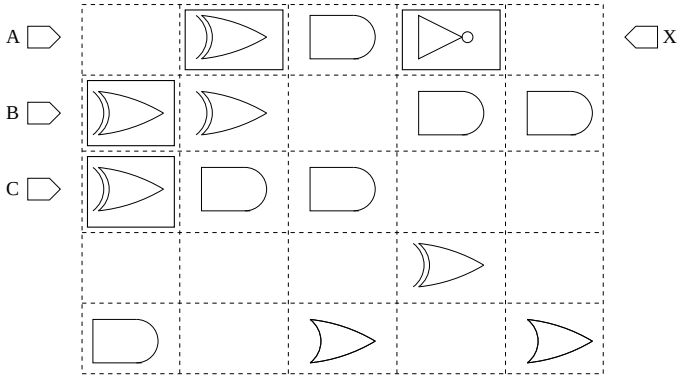

Figure 7: Solution obtained at generation 67 for the circuit whose truth table is shown in Table 3.

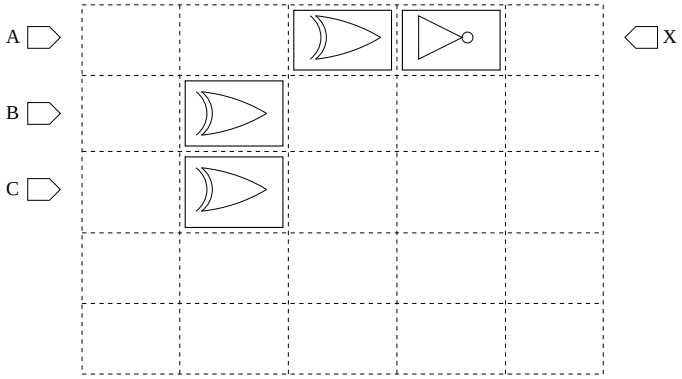

Figure 8: Solution obtained at generation 101 for the circuit whose truth table is shown in Table 3.

Table 6: Truth table for the circuit of the second example (a subtractor).

A	**B**	**C**	**X**	**Y**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

6.2 Example 2

In this case, we want to find the Boolean expression that corresponds to the circuit whose truth table is provided in Table 6.

First, we performed 10 runs using integer representation and the following parameters[4]: population size=100, maximum number of generations = 2000, crossover rate = 0.5, mutation rate = 0.006. The best solution found from these runs has 7 gates and its corresponding Boolean expression is shown (under "AG Setup 1") in Table 7. This Boolean expression can be contrasted with the best solution found by a Human Designer using Karnaugh maps (this solution has 16 gates). Additionally, we performed 10 more runs using a population size of 700 and a maximum number of generations of 400. The best solution found from these runs has 6 gates and its corresponding Boolean expression is shown (under "AG Setup 2") in Table 7.

[4]The parameters indicated were empirically derived after performing a set of experiments.

Table 7: Comparison of results between a human designer and two setups of our GA for the second example.

Human Designer
$X = A'B + A'C + BC$
$Y = A'B'C + A'BC' + A'B'C' + ABC$
16 gates
8 ANDs, 5 ORs, 3 NOTs
AG Setup 1
$X = (B \oplus (A \oplus C)) \oplus (A \oplus AC)$
$Y = (B \oplus (A \oplus C)) \oplus (B(A \oplus C)$
7 gates
2 ANDs, 1 OR, 4 XORs
AG Setup 2
$X = (A'(B \oplus (A \oplus C)) + BC$
$Y = B \oplus (A \oplus C)$
6 gates
2 XORs, 2 ANDs, 1 OR, 1 NOT

6.2.1 Analysis

The next step was to analyze (by hand) the solutions produced by our GA with respect to those generated by the human designer:

If we take the solution found by the human designer and we factorize C and C' in Y, we have that:

$$Y = A'B'C + A'BC' + AB'C' + ABC = C(A'B' + AB) + C'(A'B + AB') \quad (3)$$

To transform this equation in terms of an XOR gate:

If $S = C$ then it is necessary that $S' = C'$

and if $T = A'B' + AB$ then it is necessary that $T' = A'B + AB'$

If we apply the DeMorgan's theorem to T, we have that:

$$T' = (A'B' + AB)' = (A'B')'(AB)' = (A + B)(A' + B') \quad (4)$$

Applying the distributive law and some basic theorems:

$$T' = (A + B)(A' + B') = AA' + AB' + A'B + BB' = A'B + AB' \quad (5)$$

Verifying that $T' = A'B + AB'$ is the negation of $T = A'B' + AB$.

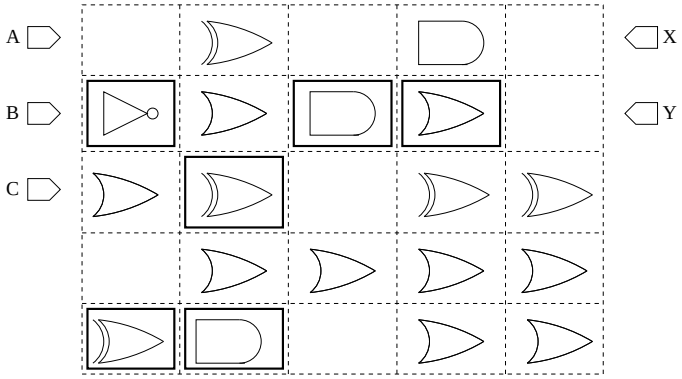

Figure 9: Solution obtained at generation 325 for the circuit whose truth table is shown in Table 6.

Then we can rewrite the eq. (3) as follows:

$$Y = C(A'B' + AB) + C'(A'B + AB') = C \oplus (A'B + AB') \tag{6}$$

If we apply the operation of a XOR gate to the operand between parentheses, we have:

$$Y = C \oplus (A'B + AB') = C \oplus (A \oplus B) \tag{7}$$

Applying the commutative law, we have:

$$Y = C \oplus (A \oplus B) = (A \oplus B) \oplus C \tag{8}$$

We have the same result obtained from the solution found by the second set of runs of the GA for Y, so we can store this equality in the case base and we will have a reduction in the number of gates.

The following can be easily seen from the previous equations:

- In eq. (3) we rediscovered the distributive law
- In eq. (4) we rediscovered the DeMorgan's theorem
- In eq. (5) we rediscovered the distributive law and some basic theorems
- In eq. (6) we rediscovered the operation of an XOR gate
- In eq. (7) we rediscovered the operation of an XOR gate
- In eq. (8) we rediscovered the commutative law

The cases stored in the case base as a product of the analysis at the end of the evolutionary process are summarized in Table 8.

Then, we performed an analysis during the evolutionary process, trying to detect the basic building blocks used by the evolutionary algorithm to generate the best solutions

Original case		Solution	Description	Number of gates eliminated
Case 1	$A'B'C + A'BC'$ $+AB'C' + ABC$	$(A \oplus B) \oplus C$	Comparison between a human designer and the best solution found by the GA	13 - 2 = 11
Case 2	$(A'B' + AB)'$	$A'B + AB'$	Case obtained with the comparison between a human designer and the best solution found by the GA	6 - 5 = 1

Table 8: Cases stored in the case base at the end of the evolutionary process for the circuit whose truth table is shown in Table 6.

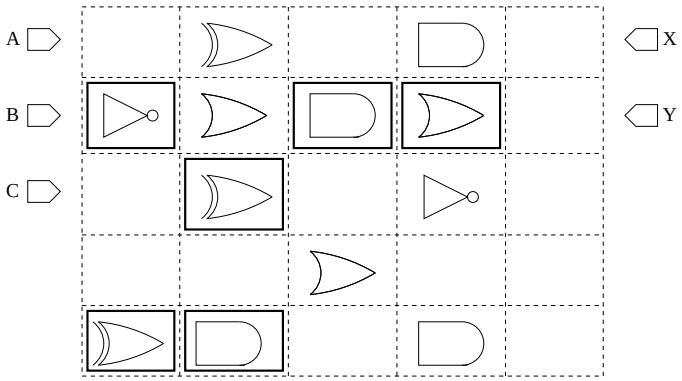

Figure 10: Solution obtained at generation 366 for the circuit whose truth table is shown in Table 6.

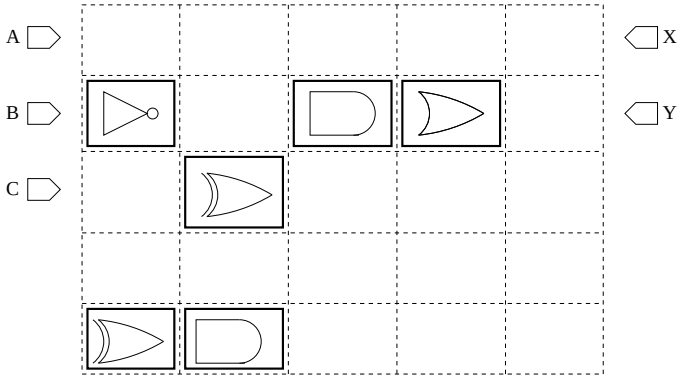

Figure 11: Solution obtained at generation 398 for the circuit whose truth table is shown in Table 6.

produced. Figures 9, 10, and 11 show several snapshots of the solutions produced by our GA with the second set of parameters previously described (population size = 700, maximum number of generations = 400). From these pictures, we can see that the circuit has a fitness value of 23 at generation 325 and we were able to recognize the building blocks used by the GA (such building blocks are indicated with a thicker box). At generation 366, the maximum fitness has increased, reaching 29, and we can observe that the building blocks previously mentioned remain in their same position. Finally, when reaching generation 398, we have a fitness of 35 (i.e., a feasible circuit with only 6 gates). Since the building blocks previously mentioned remain in the same position, we proceed to store them in our case base. The building block found will be stored using integers (since our GA used an integer representation), using asterisks (i.e., 'don't care' symbol) for those positions in the circuit different from the building block.

This same process was applied to several other circuits, including a 2-bit magnitude comparator, a half-adder and a full adder. The details of these experiments are available at [11].

6.3 A Case Study: Use of CBR to Design a 2×2 bit Adder

To provide an insight into some of the possible applications of our work, we chose a second example in which we want to illustrate how can we use previously acquired knowledge (derived from the design of a half 2×2 adder) to produce a full 2×2 adder.

We were interested in analyzing different possibilities regarding the use of CBR to improve the performance of the GA. Therefore, we decided to perform three experiments:

- **First Experiment**: Only previous solutions to the full adder circuit with different fitness values were stored in a case base and some of these individuals were retrieved to seed a percentage of the initial population of a GA before running it. The individuals were taken from different generations with different fitness

values in a previous run for the full adder circuit. The initial population was a mixture of previous solutions (10%) and random solutions (90%). This mixture is necessary to avoid an excessive selection pressure that would cause premature convergence. However, the issue of finding the proper number of cases to be injected in the population of a GA is still an open research area [16]. There is, however, previous empirical evidence that indicates that the use of the best previously found solutions are not necessarily good cases and that injecting a large number of cases does not always lead to a better performance of the GA [14]. The best known solution to this circuit has a fitness of 36 (i.e., a feasible circuit with five gates), and we stored solutions with a fitness value of up to 22.

- **Second Experiment**: Some solutions to different logic circuits including the full adder, the half-adder, the comparator and other circuits were stored in a case base. The most similar cases would then be used to seed a portion of the initial population of a GA before running it. The same mixture of individuals as before was adopted in this case.

- **Third Experiment**: Some solutions to different logic circuits including all the circuits as in step 2, but without including the full adder circuit were stored in a case base. The most similar cases would then seed a part of the initial population of a GA were retrieved before running it. The same mixture of individuals as before was adopted in this case.

The results produced from the three experiments are shown in Figures 12, 13 and 14. As we expected, when previous knowledge is used, the GA arrives more rapidly to the best known solution to this circuit. In the first experiment, our GA converges, on average, at generation 87, whereas the GA without knowledge required almost 100 generations to converge (on average). In the second experiment, the GA arrived to the best known solution to this circuit slightly faster when introducing feasible solutions previously found (as compared to the GA without knowledge). We observed that the GA retrieved from the case base the previous solution to the full adder (with fitness of 22), instead of the solution to the half adder. This is explained by the fact that the full adder (being the same circuit to be solved) presents a greater resemblance with the circuit being designed. The experiment showed us the capability of our system to discriminate among several circuits until it finds one the presents the greatest resemblance with the circuit to be designed. In our third experiment, we can observe that the GA begins to evolve from a fitness value of 14 in generation one, analogously to the GA with its initial population randomly generated. However, the circuit evolves in a completely different way due to the fact that the system retrieves as the most similar case the previous solution found for the half adder circuit. Note how the GA that uses the case base finds a valid circuit at generation 34, whereas the conventional GA finds a valid circuit at generation 45. This illustrates how the use of case base reasoning can actually help the GA to explore the search space in a more efficient way.

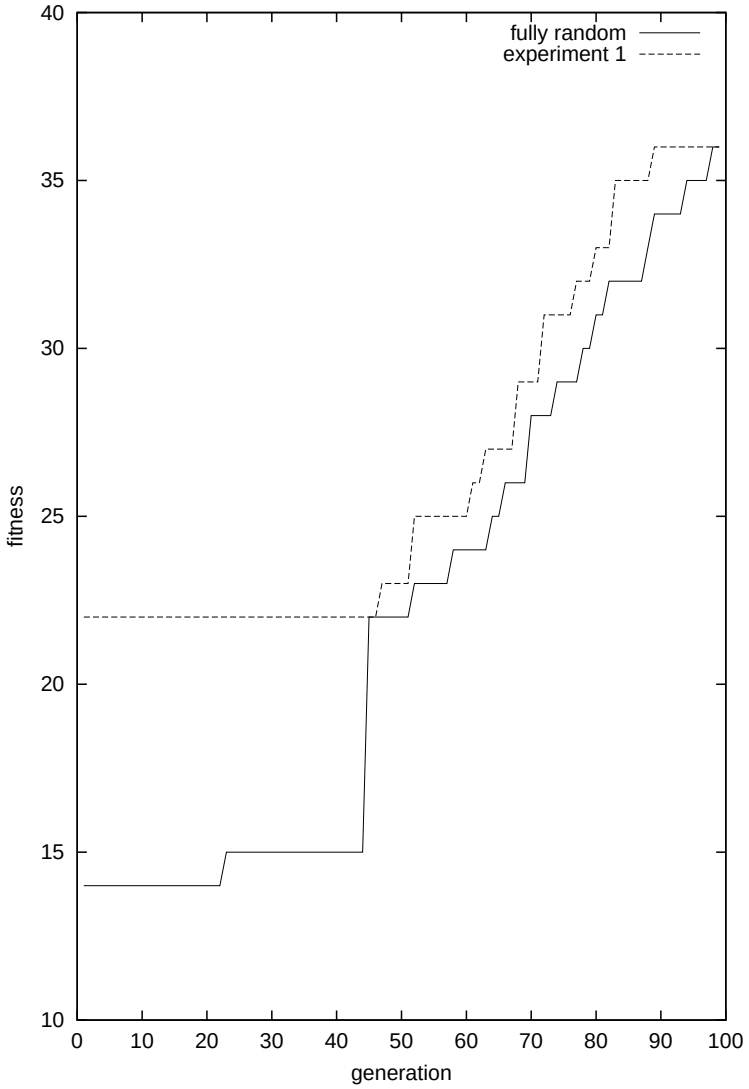

Figure 12: Convergence graph corresponding to the first experiment performed. The label “experiment 1” indicates the runs in which we used cases previously generated by other runs of our GA (i.e., use of the case base).

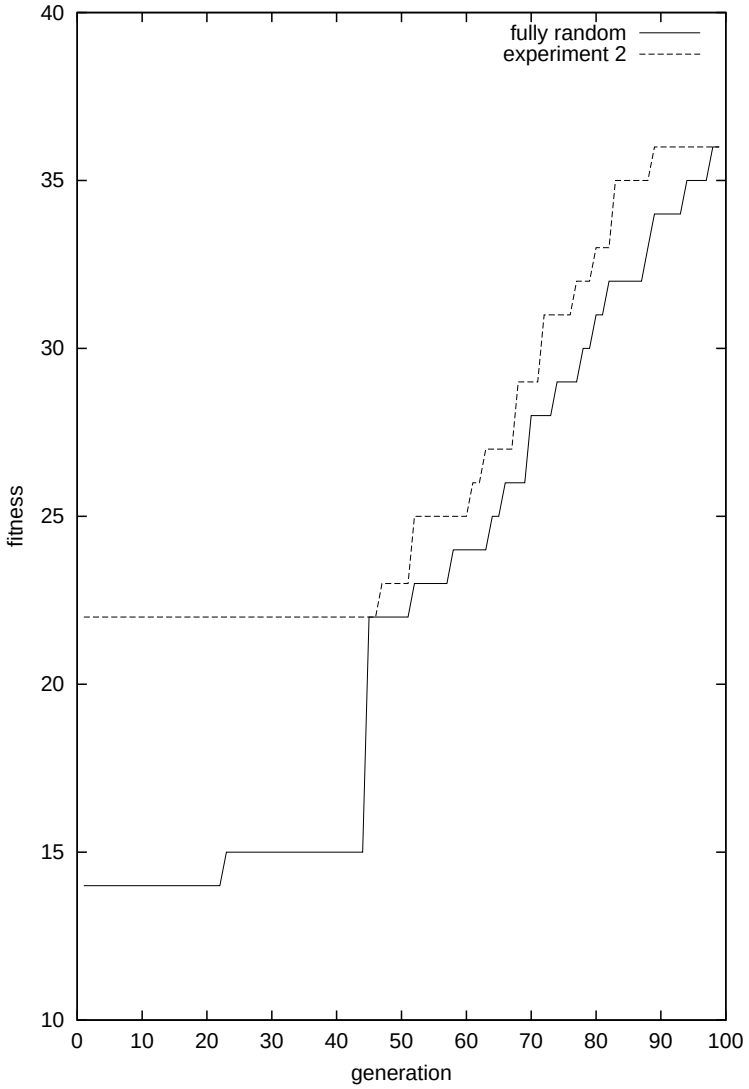

Figure 13: Convergence graph corresponding to the second experiment performed. The label “experiment 2” indicates the runs in which we used cases previously generated by other runs of our GA (i.e., use of the case base).

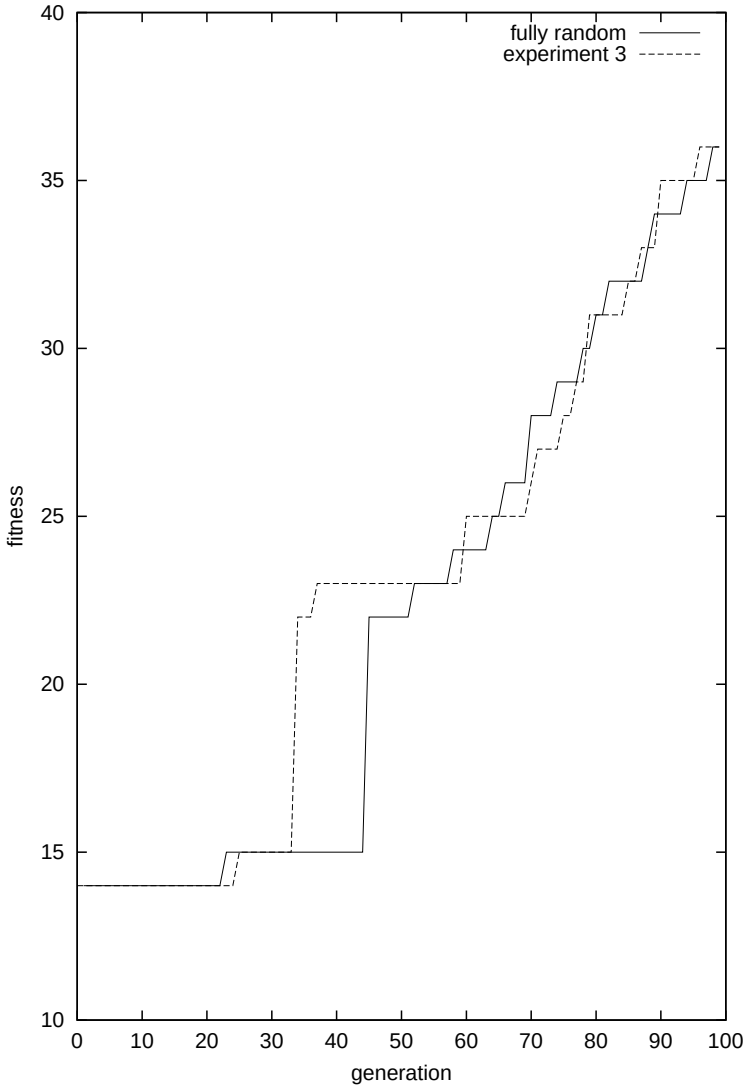

Figure 14: Convergence graph corresponding to the third experiment performed. The label “experiment 3” indicates the runs in which we used cases previously generated by other runs of our GA (i.e., use of the case base).

7 Conclusions and Future Work

We have illustrated the potential of combining CBR with a GA to improve performance. The introduction of domain-specific knowledge within a GA is not straightforward, and care must be taken of not biasing the search too strongly as to produce premature convergence. The mixture of individuals proposed in this work (10% of the population were taken from the case base and 90% were randomly generated) seems to be a good choice, at least for the small and medium size circuits used in our experiments [11]. However, more experimentation in this direction is still necessary. We are also currently extending our system to use it with genetic programming [8] and with the any colony system [3], as well as with more complex circuits.

Our approach extends some of the previous efforts to extract design patterns from a GA used to design circuits [20, 23], since we show not only how these patterns can be extracted, but also how can they be reused by a GA to design other circuits.

The use of previous experiences can improve the convergence of a GA used to solve similar problems, as we illustrated with the full adder problem. More important yet, is the fact that this sort of system can be applied to other domains, and that is precisely one of the future research paths that we would like to explore.

We are also interested in analyzing the schema processing performed by the GA when trying to solve a circuit, as to identify potentially difficult problems. This could provide us with some important information regarding the limitations of GAs in this domain and it is certainly a future research path that is worth exploring.

8 Acknowledgements

The authors thank the reviewers for their comments that greatly helped them to improve the contents of this paper. The second author acknowledges support from CONACyT through project No. 32999-A. The third author ackowledges partial support for this work from the Consejo de Ciencia y Tecnología del Estado de Guanajuato Project No. 01-02-202-111.

References

[1] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Automated Design of Combinational Logic Circuits using Genetic Algorithms. In D. G. Smith, N. C. Steele, and R. F. Albrecht, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 335–338. Springer-Verlag, University of East Anglia, England, April 1997.

[2] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Use of Evolutionary Techniques to Automate the Design of Combinational Circuits. *International Journal of Smart Engineering System Design*, 2(4):299–314, June 2000.

[3] Carlos A. Coello Coello, Rosa Laura Zavala Gutiérrez, Benito Mendoza García, and Arturo Hernández Aguirre. Ant Colony System for the Design of Combinational Logic Circuits. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pages 21–30. Springer-Verlag. Lecture Notes in Computer Science No. 1801, Edinburgh, Scotland, April 2000.

[4] 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, 1929. (Originally published in 1859).

[5] Richard Dawkins. *The Blind Watchmaker*. W.W. Norton & Company, UK, 1996.

[6] John S. Gero and Vladimir A. Kazakov. Evolving Building Blocks for Design using Genetic Engineering: A Formal Approach. In J. S. Gero and F. Sudweeks, editors, *Advances in Formal Design Methods for CAD*, pages 29–48. University of Sidney, Sidney, Australia, 1995.

[7] Pauline C. Haddow, Gunnar Tufte, and Piet van Remorte. Shrinking the Genotype: L-systems for EHW. In Yong Liu, Kiyoshi Tanaka, Masaya Iwata, Tetsuya Higuchi, and Moritoshi Yasunaga, editors, *Evolvable Systems: From Biology to Hardware*, pages 128–139. Springer-Verlag. Lecture Notes in Computer Science No. 2210, Tokyo, Japan, October 2001.

[8] Arturo Hernández Aguirre, Carlos A. Coello Coello, and Bill P. Buckles. A Genetic Programming Approach to Logic Function Synthesis by means of Multiplexers. In D. Keymeulen A. Stoica and J. Lohn, editors, *The First NASA/DoD Workshop on Evolvable Hardware*, pages 46–53. IEEE Computer Society, 1999.

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

[10] Hitochi Iba, Masawa Iwata, and Tetsuya Higuchi. Gate-Level Evolvable Hardware: Empirical Study and Applications. In D. Dasgupta and Z. Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 259–276. Springer-Verlag, Berlin, Germany, 1997.

[11] Eduardo Islas Pérez. Development of a Learning Platform using Case Based Reasoning and Genetic Algorithms. Case Study: Optimization of Combinational Logic Circuits. Master's thesis, Maestría en Inteligencia Artificial, Facultad de Física e Inteligencia Artificial, Universidad Veracruzana, November 2000. (Available at: `http://delta.cs.cinvestav.mx/~ccoello/`).

[12] Masaya Iwata, Isamu Kajitani, Yong Liu, Nobuki Kajihara, and Tetsuya Higuchi. Implementation of a Gate-Level Evolvable Hardware Chip. In Yong Liu, Kiyoshi Tanaka, Masaya Iwata, Tetsuya Higuchi, and Moritoshi Yasunaga, editors, *Evolvable Systems: From Biology to Hardware*, pages 38–49. Springer-Verlag. Lecture Notes in Computer Science No. 2210, Tokyo, Japan, October 2001.

[13] Janet Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[14] Xiaohua Liu. Combining Genetic Algorithms and Case-based Reasoning for Structure Design. Master's thesis, Department of Computer Science, University of Nevada, 1996.

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

[16] Sushil J. Louis and Judy Johnson. Solving Similar Problems using Genetic Algorithms Case-Based Memory. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 283–290, San Francisco, California, 1997. Morgan Kaufmann Publishers.

[17] Sushil J. Louis, Gary McGraw, and Richard Wyckoff. Case-based reasoning assisted explanation of genetic algorithm results. *Journal of Experimental and Theoretical Artificial Intelligence*, 5:21–37, 1993.

[18] Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the Evolutionary Design of Digital Circuits—Part I. *Genetic Programming and Evolvable Machines*, 1(1/2):7–35, April 2000.

[19] Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the Evolutionary Design of Digital Circuits—Part II. *Genetic Programming and Evolvable Machines*, 1(3):259–288, July 2000.

[20] 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)*, pages 65–74, Edinburgh, UK, 1999.

[21] Connie Loggia Ramsey and John J. Grefenstette. Case-Based Initialization of Genetic Algorithms. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 84–91, San Mateo, California, 1993. Morgan Kauffman Publishers.

[22] John W. Sheppard and Steven L. Salzberg. Combining Genetic Algorithms with Memory Based Reasoning. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 452–459, San Francisco, California, July 1995. Morgan Kaufmann.

[23] Peter Thomson. Circuit Evolution and Visualisation. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pages 229–240. Springer-Verlag, Edinburgh, Scotland, April 2000.

[24] Vesselin K. Vassilev, Terence C. Fogarty, and Julian F. Miller. Information Characteristics and the Structure of Landscapes. *Evolutionary Computation*, 8(1):31–60, Spring 2000.

[25] Vesselin K. Vassilev, Dominic Job, and Julian F. Miller. Towards the *automatic* Design of More Efficient Digital Circuits. In Jason Lohn, Adrian Stoica, Didier Keymeulen, and Silvano Colombano, editors, *The Second NASA/DoD Workshop on Evolvable Hardware*, pages 151–160, Los Alamitos, California, July 2000. IEEE Computer Society.

[26] Zhiming Zhang and T. Warren Liao. Combining Case-Based Reasoning with Genetic Algorithms. In Scott Brave and Annie S. Wu, editors, *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 305–310, Orlando, Florida, 1999.