Focus

Soft Computing 9 (2005) 44–53 © Springer-Verlag 2003
DOI 10.1007/s00500-003-0333-8

Extraction and reuse of design patterns from genetic algorithms using case-based reasoning

E. I. Pérez, C. A. C. Coello, A. H. Aguirre

Abstract. In this paper, we propose a case-based reasoning scheme in which we extract domain knowledge (in the form of design patterns) from a genetic algorithm used to optimize combinational logic circuits at the gate level. Such information is used in two ways: first, we show how the selection pressure of the genetic algorithm is biased by Boolean simplification rules that are normally adopted by human designers, including some which are not completely straightforward. Secondly, we reuse some of these design patterns extracted from the evolutionary process to reduce convergence times of a genetic algorithm using previously found solutions as cases to solve similar problems.

Keywords Genetic algorithms, Case-based-reasoning, Circuit design

1
Introduction

The use of evolutionary algorithms for design task has been subject of a considerable amount of research in the last few years. However, the attempt to extract design patterns from an evolutionary algorithm used for a design task is less common in the literature, mainly because of the difficulty of this problem. In general, it is considerably difficult for an evolutionary algorithm to suggest directly new design principles or to be able to extract such principles from the search performed by an evolutionary algorithm, Nevertheless, in this paper we suggest that the careful study of the behavior of an evolutionary algorithm when solving a design problem could help us to infer the design principles that are actually guiding the evolutionary process.

We believe that a well-suited domain to test our hypothesis 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) simplification rules that human designers can use.

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. [12]. In this paper, the authors use CBR-principles to explain solutions found by a GA. This same idea was also

Published online: 13 October 2003

E. I. Pérez (✉)
Instituto de Investigaciones Eléctricas Av. Reforma #113 Col. Palmira 62490 Temixco Morelos, MEXICO e-mail: eislas@iie.org.mx

C. A. C. Coello
CINVESTAV-IPN Departamento de Ingeniería Eléctrica Sección de Computación Av. Instituto Politécnico Nacional No. 2508 Col. San Pedro Zacatenco México, D.F. 07300, MEXICO

A. H. Aguirre
Centro de Investigación en Matemáticas Área Computación Callejón Jalisco s/n Mineral de Valenciana Guanajuato, Guanajuato 36240, MEXICO

The second author acknowledges support from CONACyT through project No. 32999-A. The third author acknowledges partial support for this work through CONACyT Project No. I-39324-A.

discussed in Louis' dissertation [10], where he proposed a system that combined CBR with GAs to improve performance of the GA. These ideas were further developed by Louis and Johnson [11] and by Liu [9]. Although Louis [10] and Louis and Johnson [11] 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.

Gómez de Silva Garza and Maher [4, 5] proposed a case adaptation method based on genetic algorithms. Their approach, which was implemented in a computational system called GENCAD, uses domain knowledge to help the genetic algorithm decide which solutions generated through the evolutionary process can be useful in the future. The main idea is to establish a set of cases previously identified that will be used to feed the initial population of a genetic algorithm. Then, they use the search engine of the genetic algorithm to adapt such cases until a suitable solution is found.

Several other researchers have proposed approaches that combine CBR and GAs. See for example [20, 16, 17]. 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. [15], 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 [13]. Finally, in [14], the techniques for landscape analysis developed in [19] are studied. Also, the authors discuss the use of case-based reasoning techniques to extract and reuse rules implicitly used by an evolutionary algorithm [14]. In this case, a nearest neighbor matching function is used to rank cases in the case-base.

Recently, Thomson [18] 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.

The problem the evolvable hardware community faces is to find building blocks suitable for evolution. Gero and Kazakov [3] 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 and Kazakov cannot prove for their problem domain), thus our goal is to assist 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 [8]. 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, sustained learning, since a new experience is retained each time a problem has been solved, making it immediately available for future problems.

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 does not always has to be found by returning to the beginning.

A CBR system can be divided in the following main stages (see Fig. 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.

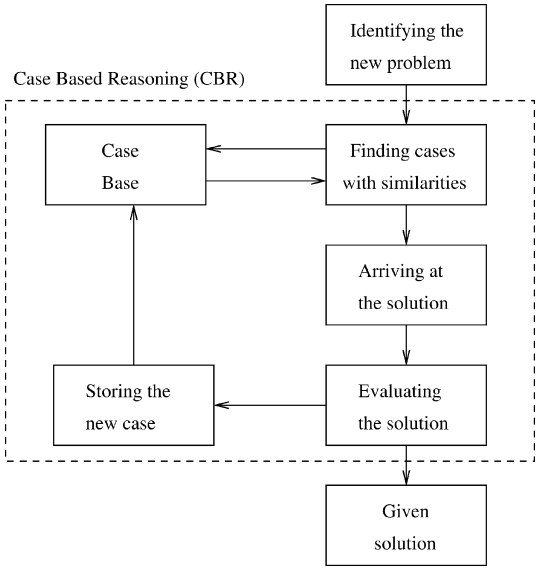

Fig. 1. General structure of a CBR system

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 more 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 Boolean laws used by the evolutionary algorithm to design a circuit. These laws will be obtained after comparing the results 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 Fig 2).

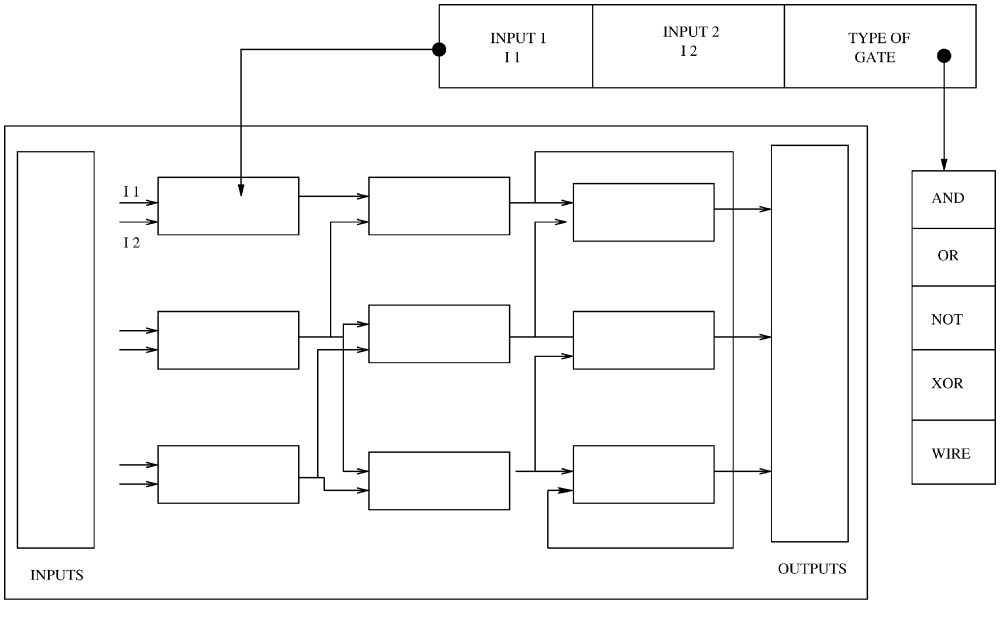

Fig. 2. Matrix used to represent a circuit. Each gate gets its inputs from either of the gates in the previous column. Note the encoding adopted for each element of the matrix as well as the set of available gates used

Our GA uses a fitness 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 gates (WIRE means no gate; this type of gate is used to allow variable-length Boolean expressions within our fixed-length representation); this aims to produce feasible circuits that have as few gates as possible.

5 Proposed system

The proposed system that combines a GA with CBR is depicted in Fig 5.

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 considered 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 in 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 [10]. 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.

5.1 Representing circuits as strings

Figure 4 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

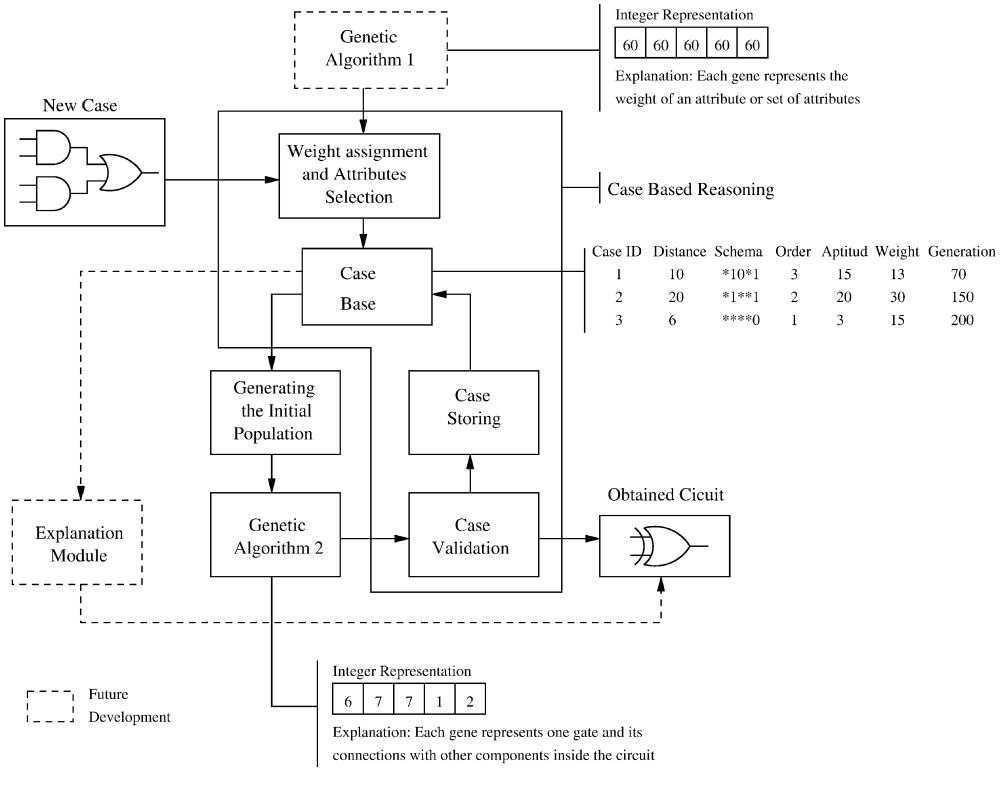

Fig. 3. Proposed system to optimize combinational logic circuits using GAs and CBR

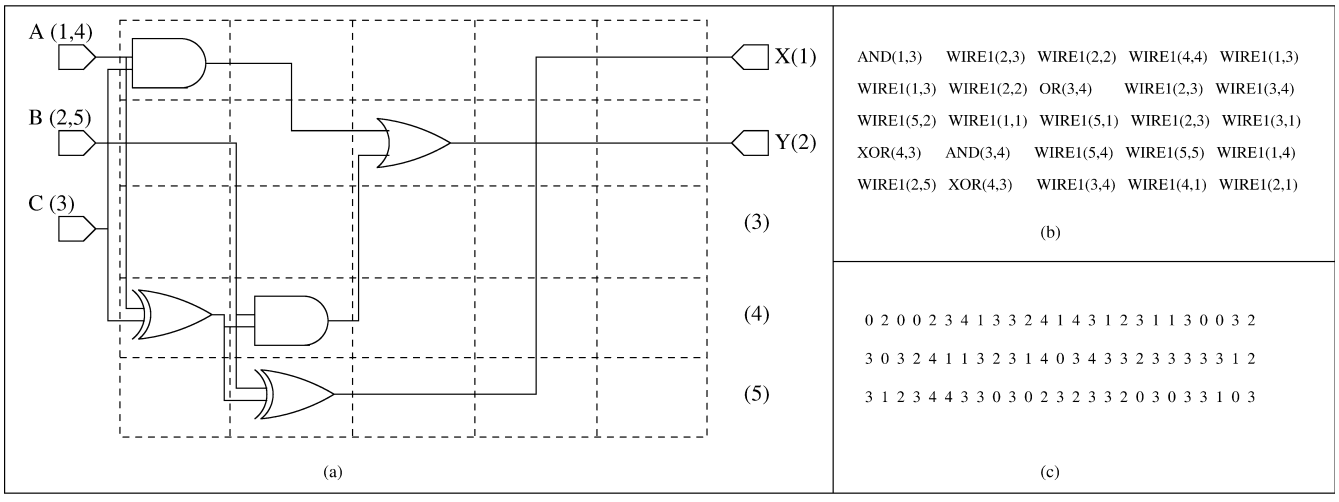

Fig. 4. Three different ways of representing the same circuit: **a** a graphical representation using two-input gates, **b** a symbolic matrix, and **c** a string of integers

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 element zero (assuming the matrix representation described in Fig. 4) and its second input from element two.

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 this 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.

[1] This scheme presents certain resemblance with the one proposed by Louis [10].

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:

- 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 an example of how is the knowledge 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.

6.1 An example

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

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	00000000000100111	39	3230132431232134103231
2	3	2	01101001000010100	38	3230132431232144133204
3	3	2	00000000000100111	39	0200234133241431231130
4	2	2	00000001000011001	31	0100142134131433134130
5	3	2	00000000000000011	36	0100142134131433134130
6	2	2	00000001000011001	31	0200234133241431231130

Table 2. Cases for knowledge extraction during the evolutionary process

CaseID	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

Table 3. Truth table for the circuit of the example

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

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:

Table 4. Comparison of results between a human designer and two setups of our GA for the example

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

$$X = ((A \oplus B)' \oplus (C \oplus D)')' = ((A \oplus B) \oplus (C \oplus D)') \quad (1)$$

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

$$(S' \oplus T')' = (S \oplus T)' \quad (2)$$

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 5, 6 and 7 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

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

[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.

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

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	$4 - 2 = 2$

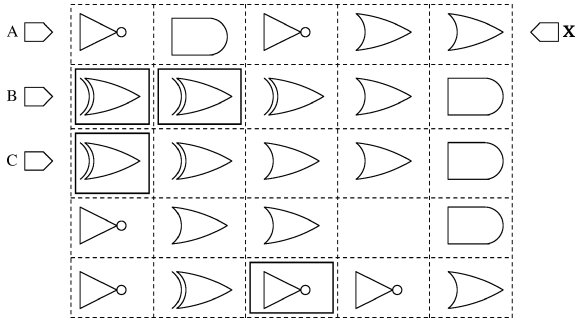

Fig. 5. Solution obtained at generation 9 for the circuit whose truth table is shown in Table 3

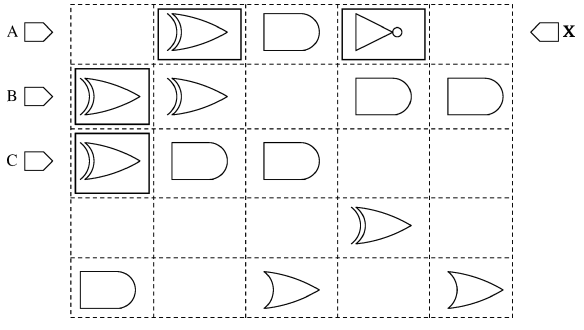

Fig. 6. Solution obtained at generation 67 for the circuit whose truth table is shown in Table 3

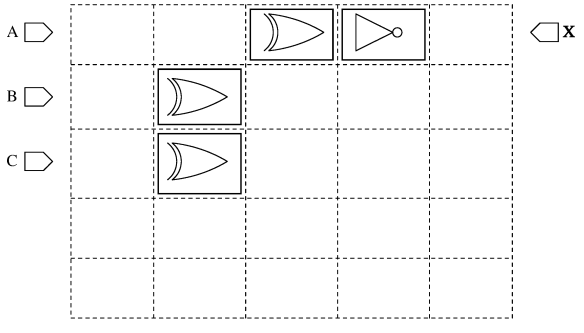

Fig. 7. Solution obtained at generation 101 for the circuit whose truth table is shown in Table 3

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.

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 [7].

6.2 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 [11]. 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 [9]. 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 Figs 8, 9 and 10. 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

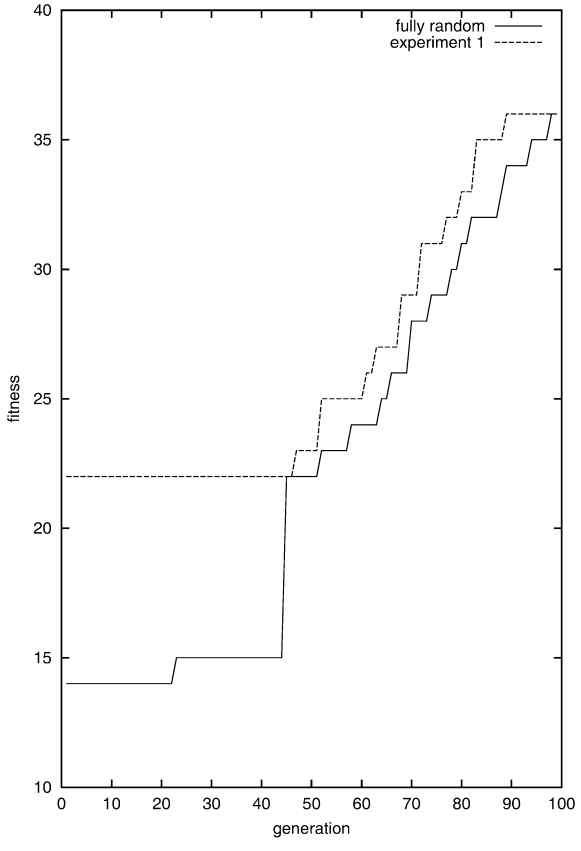

Fig. 8. Convergence graph corresponding to the first experiment performed. The label "experiment 1" indicates the runs in which we used cases previously generated by other runs of our GA (i.e., use of the case base)

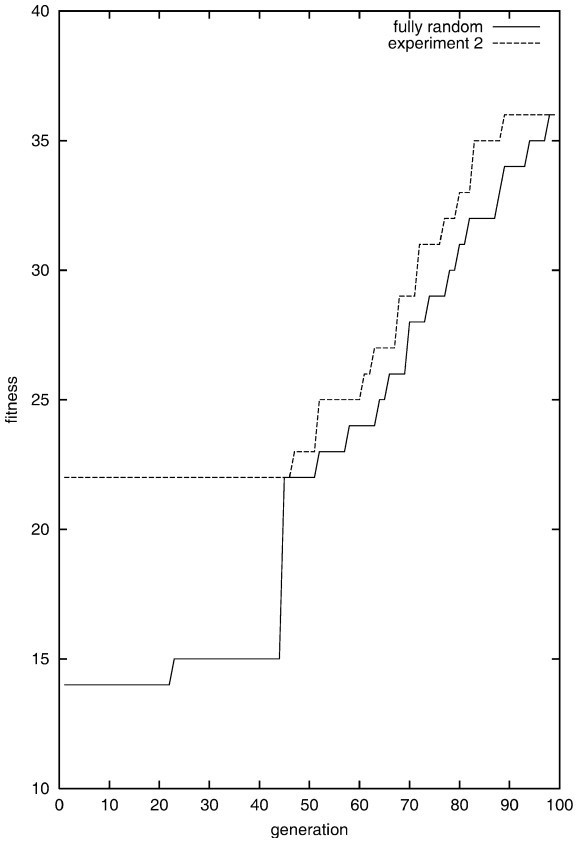

Fig. 9. Convergence graph corresponding to the second experiment performed. The label "experiment 2" indicates the runs in which we used cases previously generated by other runs of our GA (i.e., use of the case base)

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.

7 Conclusions and future work

We have presented an approach in which previous "experience" (in the form of examples) is exploited by a genetic algorithm in order to improve its 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 [7]. However, more experimentation in this direction is still necessary. We are also currently extending our system to use it with genetic programming [6], 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

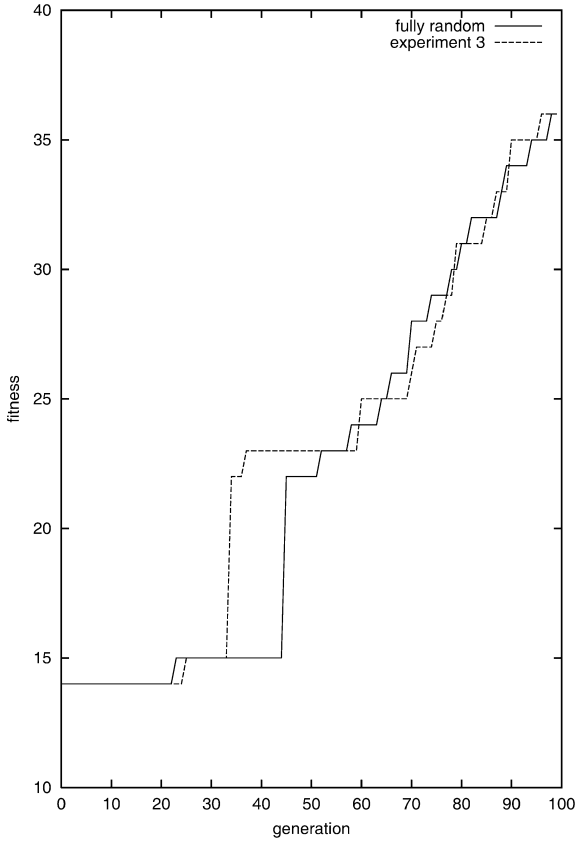

Fig. 10. Convergence graph corresponding to the third experiment performed. The label "experiment 3" indicates the runs in which we used cases previously generated by other runs of our GA (i.e., use of the case base)

[15, 18], 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.

References

1. **Carlos A. Coello C, Christiansen AD, Aguirre AH** (1997) Automated Design of Combinational Logic Circuits using Genetic Algorithms. In: DG Smith, NC Steele, RF Albrecht (eds) Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms, pp. 335–338. Springer-Verlag, University of East Anglia England
2. **Carlos A. Coello C, Alan D Christiansen, A. H. Aguirre** (2000) Use of Evolutionary Techniques to Automate the Design of Combinational Circuits International Journal of Smart Engineering System Design **2**(4): 299–314
3. **Gero JS, Kazakov VA** (1995) Evolving Building Blocks for Design using Genetic Engineering: A Formal Approach. In: JS Gero, F Sudweeks (eds) Advances in Formal Design Methods for CAD, pp. 29–48. University of Sydney, Sydney Australia
4. **de Silva Garza AG, Maher ML** (1999) An Evolutionary Approach to Case Adaptation. In: Case-Based Reasoning Research and Applications: Proceedings of the Third International Conference on Case-Based Reasoning ICCBR-99, pp. 162–172, Monastery Seeon, Munich Germany, Springer-Verlag
5. **de Silva Garza AG, Maher ML** (2000) A Process Model for Evolutionary Design Case Adaptation. In: John Gero (ed.) Proceedings of the Sixth International Conference on Artificial Intelligence in Design, pp. 393–412, Worcester, Massachusetts, Kluwer Academic Publishers
6. **Aguirre AH, Coello CAC, Buckles BP** (1999) A Genetic Programming Approach to Logic Function Synthesis by means of Multiplexers. In: D Keymeulen A. Stoica, J. Lohn (eds) The First NASA/DoD Workshop on Evolutionable Hardware, pp. 46–53. IEEE Computer Society
7. **Pérez EI** (2000) 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, (Available at: http://delta.cs.cinvestav.mx/∼ccoello/)
8. **Kolodner J** (1993) Case-Based Reasoning. Morgan Kaufmann Publishers, San Mateo California
9. **Liu X** (1996) Combining Genetic Algorithms and Case-based Reasoning for Structure Design. Master's thesis, Department of Computer Science, University of Nevada
10. **Sushil J. Louis** (1993) Genetic Algorithms as a Computational Tool for Design. PhD thesis, Department of Computer Science, Indiana University
11. **Louis SJ, Johnson J** (1997) Solving Similar Problems using Genètic Algorithms Case-Based Memory. In: Thomas Bäck (ed) Proceedings of the Seventh International Conference on Genetic Algorithms, pp. 283–290, San Francisco, California, Morgan Kaufmann Publishers
12. **Louis SJ, McGraw G, Wyckoff R** (1993) Case-based reasoning assisted explanation of genetic algorithm results. J. Exper. Theor. Artifi. Intel. **5**: 21–37
13. **Miller JF, Job D, Vassilev VK** (2000) Principles in the Evolutionary Design of Digital Circuits—Part I. Genetic Programming and Evolvable Machines **1**(1/2): 7–35
14. **Miller JF, Job D, Vassilev VK** (2000) Principles in the Evolutionary Design of Digital Circuits—Part II Genetic Programming and Evolvable Machines **1**(3): 259–288
15. **Miller JF, Kalganova T, Lipnitskaya N, Job D** (1999) The Genetic Algorithm as a Discovery Engine: Strange Circuits and New Principles. In: Proceedings of the AISB Symposium on Creative Evolutionary Systems (CES'99). pp. 65–74, Edinburgh UK
16. **Ramsey CL, Grefenstette JJ** (1993) Case-Based Initialization of Genetic Algorithms. In: Stephanie Forrest (ed) Proceedings of the Fifth International Conference on Genetic Algorithms. pp. 84–91, San Mateo, California, Morgan Kauffman Publishers
17. **Sheppard JW, Salzberg SL** (1995) Combining Genetic Algorithms with Memory Based Reasoning. In: Larry Eshelman (ed) Proceedings of the Sixth International Conference on Genetic Algorithms. pp. 452–459, San Francisco, California, Morgan Kaufmann
18. **Thomson P** (2000) Circuit Evolution and Visualisation. In: Julian Miller, Adrian Thompson, Peter Thomson, Terence C. Fogarty (eds) Evolvable System: From Biology to Hardware. pp. 229–240 Springer-Verlag, Edinburg, Scotland

19. **Vassilev VK, Fogarty TC, Miller JF** (2000) Information Characteristics and the Structure of Landscapes. Evolutionary Computation **8**(1): 31–60
20. **Zhang Z, Liao TW** (1999) Combining Case-Based Reasoning with Genetic Algorithms. In: Scott Brave, Anies S. Wu, (eds) Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference pp. 305–310, Orlando, Florida