ARTIFICIAL INTELLIGENCE DEPARTMENT


THESIS

To obtain the degree in:

Master of Science in Artificial Intelligence

Eduardo Islas Pérez

Xalapa de Enríquez, Ver. November 2000
Universidad Veracruzana

Artificial Intelligence Department

M. Sc. Thesis


<table>
<thead>
<tr>
<th>Name</th>
<th>Organization</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed by</td>
<td>Eduardo Islas Pérez</td>
<td>13 Nov, 2000</td>
</tr>
<tr>
<td></td>
<td>Universidad Veracruzana and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Laboratorio Nacional de Informática</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avanzada (LANIA) – México</td>
<td></td>
</tr>
<tr>
<td>Directed by</td>
<td>Carlos Coello Coello</td>
<td>21 Sep, 2000</td>
</tr>
<tr>
<td></td>
<td>Laboratorio Nacional de Informática</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avanzada (LANIA) - México</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Richard Chapman</td>
<td>28 Jul, 2000</td>
</tr>
<tr>
<td>Reviewed by</td>
<td>José Negrete</td>
<td>5 Oct, 2000</td>
</tr>
<tr>
<td></td>
<td>Universidad Veracruzana - México</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Katya Rodríguez</td>
<td>30 Oct, 2000</td>
</tr>
</tbody>
</table>
# CONTENTS

1 INTRODUCTION  

2 CONCEPTS  

2.1 Combinational Logic Circuits Optimization  
2.1.1 Justification for Logic Minimization  
2.1.2 Laws and Theorems of Boolean Algebra  
2.1.3 Karnaugh maps.  
2.1.4 Quine-McCluskey Method.  

2.2 Evolutionary Algorithms.  
2.2.1 Genetic Algorithms.  
2.2.2 Evolutionary Programming.  
2.2.3 Genetic Programming.  
2.2.4 Evolution Strategies.  

2.3 Case Based Reasoning.  

2.4 Previous Works in Combinational Logic Circuits Optimization using EAs and/or CBR.  

3 PROPOSED SYSTEM  

4 PROBLEM DESCRIPTION  

4.1 The Knowledge that will be extracted from the evolutionary process of the circuit  

4.2 Small Scale Integrated Circuit (SSI), Circuit A [Coello 2000].  
4.2.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.  
4.2.2 Knowledge extraction during the evolutionary process of the logic circuit.  

4.3 SSI, Circuit B [Lam 1988].  
4.3.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.  
4.3.2 Knowledge extraction during the evolutionary process of the logic circuit.  

4.4 Medium Scale Integrated Circuit (MSI), subtraction circuit [Morris 1982].  
4.4.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.  
4.4.2 Knowledge extraction during the evolutionary process of the logic circuit.  

4.5 MSI, Parity bit checking [Morris 1982].  
4.5.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.  
4.5.2 Knowledge extraction during the evolutionary process of the logic circuit.  

4.6 MSI, 2-bit magnitude comparator circuit [Lam 1988]  
4.6.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.  
4.6.2 Knowledge extraction during the evolutionary process of the logic circuit.  

4.7 MSI, half-adder [Lam 1988].  
4.7.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.  
4.7.2 Knowledge extraction during the evolutionary process of the logic circuit.  

5 DEVELOPMENT OF SYSTEM  

5.1 SIMULATION ENVIRONMENT  

5.2 Development of Genetic Algorithm  

5.3 Development of Evolutionary Programming (EP)  

5.4 Development of Genetic Programming (GP)  

6 TEST NATURE  

6.1 Discussion  

6.2 Results  

6.3 Analysis  

6.4 Conclusions  

7 future work  

APPENDIX  

Aumbled offline data collection  

B Simulation of Circuit A  

C Simulation of Circuit B  

D Simulation of MSI, Parity bit checking  

E Simulation of MSI, 2-bit magnitude comparator circuit  

F Simulation of MSI, half-adder  

G Genetic Algorithm  

H Genetic Programming  

I Evolutionary Programming  

J Comparative Study  

K Results  

L Analysis  

M Conclusions  

N Future work  

INDEX
4.8  MSI, full-adder [Lam 1988].  
    4.8.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.  
    4.8.2 Knowledge extraction during the evolutionary process of the logic circuit.

4.9  How the knowledge will be automatically extracted  

4.10 How the extracted knowledge will be stored in the knowledge base  
    4.10.1 Representation of combinational logic circuits as strings  
    4.10.2 Representation of combinational logic circuits as cases in the case base.

5  APPLYING CBR AND GAS TO FIND THE SOLUTION FOR THE FULL ADDER CIRCUIT  
    5.1  First step.  
    5.2  Second step.  
    5.3  Third step.

6  CONCLUSIONS

7  FUTURE DEVELOPMENTS
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 principles. In the latter, systems are “evolved” by a process of human ingenuity, which employs a set of rules, concepts and principles. It is indeed curious that organisms such as human beings which are capable of imagining the world that operates according to definite laws and abstract design processes produced them by a mechanism which is entirely blind and has no particular objective other than survival. Although it is difficult for an evolutionary algorithm directly to suggest new principles, they may still be inferred by studying and analyzing an evolved series of examples. We proposed that by employing a blind evolutionary approach, assumptions and principles may be challenged, and thus, new concepts may emerge. A well-defined context in which to examine these issues is in the field of electronic circuit design. Here human designers have abstracted the world of binary (an alphabet of 0 and 1) and specified definite operations such as logical OR, AND, NOT, and XOR. The objective is to construct within this context an electronic or algebraic machine to carry out a definite function (e.g. addition, subtraction, multiplication) on a number of input variables, using either as few operations as possible, or a modular construction which can be used to build bigger systems.

The usefulness of these electronic machines is readily apparent in modern computers. There is a particular reason why attempting to develop arithmetic circuits might be a useful and illuminating exercise. Such circuits are modular in construction, so that very large systems may be constructed from smaller building blocks. For example, in this particular research project we emphasize that a half adder circuit could be used as a building block to solve the full adder circuit. It is very clear that the half adder circuit is a portion of the full adder circuit. Multiplication is a process of repeated addition and for this reason binary full adders connected in an arrangement are called a cellular array. This process is a classic example of human design. First of all we construct building blocks that carry out functions which we have abstracted for being fundamental. Second we build larger systems by manipulating these building blocks through a process of abstract reasoning. When we allow biologically inspired algorithms such as evolutionary algorithms to design the building blocks and assemble the parts, we discover an amazing number of new possibilities.

This leads us to one question that is analyzed in this research project: Can we, by evolving a series of circuits of increasing size, extract the general principle and hence discover new principles? Such a question immediately leads us to many other fundamental questions:
What is a subsystem or building block? What is a principle? How can we extract a principle?. To be honest there are no precise answers to these questions even in the context of human design. In a general form a building block is a subcomponent which has been empirically proved to be useful. A principle is just an observed or deduced rule which is found to be helpful in trying to form a synthesis of a wide range of data and may either directly or indirectly lead us to a prediction which can be tested. How are principles discovered? This is the unexplained process of discovery, which originates in the human mind and is called intuition or creativity. We hope to show in this research project that the specialized domain of electronic logic design is a perfect context in which to study some of these questions. We will give some examples of where we have been able to extract and rediscover some of the principles (laws and theorems of the Boolean algebra) and discover “new” principles. Another feature of this investigation is that we show some promising results that indicate how we might be able to extract principles from evolved circuit data.

In this research project we examine the idea of a genetic or evolutionary algorithm being an inspirational or knowledge discovery engine. Subsequently we propose that this knowledge can be stored in a knowledge base. When the system is facing an identical or similar new problem it can apply its “experience” in order to solve this new problem more rapidly. The proposed system is illustrated in the particular context of designing electronic circuits. We propose that by connecting pieces of logic building blocks together and testing them to see if they carry out the desired function, it may be possible to discover new principles of design, and new algebraic techniques. This is illustrated in the design of binary circuits, particularly arithmetic functions. We demonstrate in this research project that by applying an evolution process to a series of examples, it becomes possible to rediscover some of the well-known laws and theorems of Boolean algebra and discover “new” laws and theorems in the field of logic circuits.
2 Concepts

2.1 Combinational Logic Circuits Optimization

2.1.1 Justification for Logic Minimization

Logic minimization uses a variety of techniques to obtain the simplest gate-level implementation of a Boolean function. But simplicity depends on the means of measurement we use. One way to measure the complexity of a Boolean function is to count the number of literals it contains. Literals measure the amount of wiring needed to implement a function. For electrical and packaging reasons, gates in a given technology will have a limited number of inputs. While two-, three-, and four-input gates are common, gates with more than eight or nine inputs are rare. Thus, one of the primary reasons for performing logic minimization is to reduce the number of literals in the expression of the function, thus reducing the number of gate inputs.

An alternative means of measurement is the number of gates, which measures circuit area. There is a strong correlation between the number of gates in a design and the number of components needed for its implementation. The simplest design to manufacture is often the one with the fewest gates, not the fewest literals.

A third means of measurement is the number of cascaded levels of gates. Reducing the number of logic levels decreases overall delay, as there are fewer gate delays on the path from inputs to outputs. However, creating a circuit in a suitable way for minimum delay rarely yields an implementation with the fewest gates or the simplest gates. It is not possible to minimize all three means of measurements at the same time.

In this research project we emphasize the minimization of a combinational logic circuit thus reducing the number of gates in a given circuit.

2.1.2 Laws and Theorems of Boolean Algebra

Boolean algebra provides the foundation for the simplification techniques applied to logic circuits. Using the Boolean laws as a base, we can prove additional theorems that can be used as tools to simplify Boolean expressions. For example, if E1 and E2 are two expressions for the same Boolean function, we say that E2 is simpler than E1 if it contains fewer literals. This usually (but not always) means that the simpler expression will also contain fewer Boolean operations.
**Duality.** Before we provide a tabulation of useful laws and theorems, it is important to describe the concept of duality. Every Boolean expression has a dual. It is derived from the original by replacing AND operations by OR operations and vice versa, and replacing constant logic 0's by logic 1's and vice versa, while leaving the literals unchanged. It is a fundamental theorem of Boolean algebra, that any statement that is true about a Boolean expression is also true for its dual. Once we discover a useful theorem to simplify a Boolean expression, we obtain its dual as a bonus. For example, the dual of the Boolean theorem \( X + 0 = X \), written \( (X + 0)^D \), is the theorem \( X 1 = X \).

**Useful Laws and Theorems.** Table 2.1 is a list of frequently used laws and theorems of Boolean algebra. The second column shows the duals of the expression in the first column.

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Expression</th>
<th>Dual Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations with 0 and 1</td>
<td>( 1.X + 0 = X )</td>
<td>( 1D. X 1 = X )</td>
</tr>
<tr>
<td></td>
<td>( 2.X + 1 = 1 )</td>
<td>( 2D. X 0 = 0 )</td>
</tr>
<tr>
<td>Idempotent theorem</td>
<td>( 3.X + X = X )</td>
<td>( 3D. X X = X )</td>
</tr>
<tr>
<td>Involution theorem</td>
<td>( 4.(X)^{2} = X )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( 5.X + \bar{X} = 1 )</td>
<td>( 5D. X \bar{X} = 0 )</td>
</tr>
<tr>
<td></td>
<td>( 6.X + Y = Y + X )</td>
<td>( 6D. X Y = Y X )</td>
</tr>
<tr>
<td></td>
<td>( 7.(X + Y) + Z = X + (Y + Z) )</td>
<td>( 7D. (X Y) Z = X (Y Z) )</td>
</tr>
<tr>
<td>Distributive law</td>
<td>( 8.X (Y + Z) = X Y + X Z )</td>
<td>( 8D. X + (Y Z) = (X + Y) (X + Z) )</td>
</tr>
<tr>
<td>Simplification theorems</td>
<td>( 9.X Y + X Y' = X )</td>
<td>( 9D. (X + Y) (X + Y') = X )</td>
</tr>
<tr>
<td></td>
<td>( 10.X + X Y = X )</td>
<td>( 10D. X (X + Y) = X )</td>
</tr>
<tr>
<td></td>
<td>( 11.X (X^2 + Y) = X^2 + X Y X )</td>
<td>( 11D. (X Y) + Y = X + Y )</td>
</tr>
<tr>
<td>DeMorgan's theorem</td>
<td>( 12.X + Y + Z + ... = X' Y' Z' )</td>
<td>( 12D. (X Y Z ...) = X' + Y' + Z' + )</td>
</tr>
<tr>
<td></td>
<td>( 13.{(X1, X2, ... Xn, 0, 1, +, *)} = {(X1', X2', ... Xn', 1, 0, *, +)} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( 14.{(X + Y + Z + ... D = X Y Z )</td>
<td>( 14D. (X Y Z ...) = X + Y + Z + )</td>
</tr>
<tr>
<td></td>
<td>( 15.{(X1, X2, ... Xn, 0, 1, +, *)} = {(X1, X2, ... Xn, 1, 0, *, +)} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( 16.{(X + Y) (X' + Z) = X Z X Y )</td>
<td>( 16D. X Y + X Z' = X Z (X + Y) )</td>
</tr>
<tr>
<td>Theorem for multiplying and factoring</td>
<td>( 17.X Y + Z X' = X Y + Z X' )</td>
<td>( 17D. (X + Y) (Y + Z) (X + Z) = (X + Y) (X + Z) )</td>
</tr>
</tbody>
</table>

**Table 2.1 Useful laws and theorems to reduce combinational logic circuits.**

The notation \( (X1, X2, ..., Xn, 0, 1, +, *) \) used in theorems 13 and 15 represents an expression in terms of the variables \( X1, X2, Xn \), the constants 0, 1, and the Boolean operations + and *. Theorem 13 states that forming the complement of an expression, the variables are replaced by their complements, that is, 0 is replaced by 1 and 1 by 0, and + is replaced by * and * by +.

**DeMorgan’s Theorem.** DeMorgan’s theorem provides a procedure to complement a complex function. The complemented expression is formed from the original by replacing all literals by their complements; all 1’s become 0’s and vice versa, and ANDs become ORs and vice versa. This theorem indicates an interesting relationship between NOR, OR, NAND, and AND:

\[
(X + Y)' = X' * Y' \quad (X * Y)' = X' + Y'
\]
Note that \( (X+Y)' \neq X' + Y' \) and \( (X \ast Y)' \neq X' \ast Y' \). In other words, NOR is the same as AND with complemented inputs while NAND is equivalent to OR with complemented inputs.

This is demonstrated on the truth table in Table 2.2.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X’</th>
<th>Y’</th>
<th>(XY)’</th>
<th>X’+ Y’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2 Verifying DeMorgan’s theorem

For example, we can use DeMorgan's theorem to find the complement of the following expression:

\[ X = A'B'C + A'BC + AB'C + ABC' \]

Step by step, the complement is formed as follows:

\[ X' = (A'B'C + A'BC + AB'C + ABC')' \]
\[ X = (A'B'C)' (A'BC)' (AB'C)' (ABC')' \]
\[ X = (A+B+C')(A+B'+C')(A'+B+C')(A'+B'+C) \]

We can observe that duality and DeMorgan's law are not the same thing. The procedure to produce the dual is similar, but the literals are not complemented during the process. Thus, the dual of NOR is NAND (and vice versa); the dual of OR is AND (and vice versa). Any theorem that is true for an expression is also true for its dual.

2.1.3 Karnaugh maps.

2.1.3.1 Introduction.
We can always use the rules of Boolean algebra to simplify an expression, but this method has a few problems. Firstly, there is no algorithm that we can use to determine that we have obtained a minimum solution. Secondly, we often have to make the expression more complicated before we can simplify it. Sometimes we substitute \( (X+X') \) by 1 to add more terms. Then we rearrange the terms to obtain advantageous groups that help to simplify the expression in the next step. It is against human nature to climb out of a "local minimum" in the hope of finding a better global solution. But this is exactly what we often have to do.
And finally, it is just too cumbersome (and error prone) to manipulate Boolean expressions by hand.

Karnaugh maps (K-maps) provide an alternative technique for representing Boolean functions. For example, considering the Karnaugh map for a 2-input AND gate (Figure 2.1).

![Figure 2.1 Karnaugh map for a 2-input AND gate.](image)

The Karnaugh map comprises a box for every line in the truth table; the binary values above the boxes are those associated with the \( a \) and \( b \) inputs. Unlike a truth table, in which the input values typically follow a standard binary sequence (00, 01, 10, 11), the Karnaugh map's input values must be ordered in such a way that the values for adjacent columns vary by only a single bit, for example, 00, 01, 11, and 10. This ordering is known as a gray code, and it is a key factor in the functioning of Karnaugh maps.

The \( y \) column in the truth table shows all the 0 and 1 values associated with the gate's output. Similarly, all of the output values could be entered into the Karnaugh map. However, for reasons of clarity, it is common for only a single set of values to be used, typically the 1s.

Similar maps can be constructed for 3-input and 4-input functions (Figure 2.2). In the case of a 4-input map, the values associated with the \( c \) and \( d \) inputs must also be ordered as a gray code; that is, ordered in such a way that the values for adjacent rows vary by only a single bit.

![Figure 2.2 Karnaugh maps for 3-input and 4-input functions.](image)
2.1.3.2 Minimization Using Karnaugh Maps.
Karnaugh maps often prove useful in the simplification and minimization of Boolean functions. Consider an example of 3-input function represented as a black box with an associated truth table (Figure 2.3).

![Figure 2.3 Example 3-input function.](image)

![Figure 2.4 Karnaugh map minimization of example 3-input function.](image)
The equation extracted from the truth table in sum-of-products form contains four minterms, one for each of the 1s assigned to the output. Algebraic simplification techniques could be employed to minimize this equation, but this would require that every minterm is compared to each of the others, which can be somewhat time-consuming. By using Karnaugh maps we can reach a solution much more rapidly. The 1s assigned to the map's boxes represent the same minterms as the 1s in the truth table's output column. However, as the input values associated with each row and column in the map differ by only one bit, any pair of horizontally or vertically adjacent boxes corresponds to minterms that differ by only a single variable. Such pairs of minterms can be grouped together and the variable that differs can be discarded (Figure 2.4).

In the case of the horizontal group, input \( a \) is 0 for both boxes, input \( c \) is 1 for both boxes, and input \( b \) is 0 for one box and 1 for the other. Thus, for this group, changing the value on \( b \) does not affect the value of the output. This means that \( b \) is redundant and can be discarded from the equation representing this group. Similarly, in the case of the vertical group, input \( a \) is 1 for both boxes, input \( b \) is 0 for both boxes, and input \( c \) is 0 for one box and 1 for the other. Thus, for this group, input \( c \) is redundant and can be discarded.

### 2.1.3.3 Grouping Minterms

In the case of a 3-input Karnaugh map, any two horizontally or vertically adjacent minterms, each composed of three variables, can be combined to form a new product term composed of only two variables.

Similarly, in the case of a 4-input map, any two adjacent minterms, each composed of four variables, can be combined to form a new product term composed of only three variables. Additionally, the 1s associated with the minterms can be used to form multiple groups. For example, consider a new 3-input function (Figure 2.5).

![Karnaugh map minterms can be used to form multiple groups.](image)
Groupings can also be formed from four adjacent minterms, in which case two redundant variables can be discarded; consider the 4-input Karnaugh map example depicted in Figure 2.6. In fact, any group of $2^n$ adjacent minterms can be gathered together, where n is a positive integer. For example, $2^1 = \text{two minterms}$, $2^2 = \text{four minterms}$, $2^3 = \text{eight minterms}$, and so forth.

![Karnaugh Map Groupings](image)

**Figure 2.6 Karnaugh map groupings of four adjacent minterms.**

As was noted above, Karnaugh map input values are ordered so that the values associated with adjacent rows and columns differ by only a single bit. One result of this ordering is that a single bit only separates the top and bottom rows; similarly, a single bit only separates the left and right columns. It may help to visualize the map rolled into a horizontal cylinder so that the top and bottom edges touch, or into a vertical cylinder so that the left and right edges touch. This leads to some additional grouping possibilities (Figure 2.7).

Note especially the last example. Diagonally adjacent minterms generally cannot be used to form a group. However, remembering that the left-right columns and the top-bottom rows are logically adjacent, the four corner minterms can be used to form a single group.

### 2.1.3.4 Incompletely Specified Functions

In certain cases a function may be incompletely specified; that is, the output may be undefined for some of the input combinations. If the designer knows that certain input combinations will never occur, then the value assigned to the output for these combinations is irrelevant.
Alternatively, for some input combinations the designer may simply not be concerned about the value of the output. In both cases, the designer can represent the output values associated with the relevant input combinations as question marks on the map (Figure 2.8).

The ? characters indicate “don't care states” which can be considered to represent values of either 0 or 1, at the designer's discretion.
2.1.3.5 Populating Maps Using 0s Versus 1s.

When a Karnaugh map is populated using the 1s assigned to the truth table's output, the resulting Boolean expression is extracted from the map in the form of sum-of-products. As an alternative, the Karnaugh map can be populated using the 0s assigned to the output on the truth table. In this case, groupings of 0's are used to generate expressions in the form of product-of-sums (Figure 2.9).

Although the sum-of-products and product-of-sums expressions appear to be somewhat different, in fact they produce identical results. The expressions can be shown to be equivalent by using algebraic means, or by constructing truth tables for each expression and comparing the outputs.

![Figure 2.9 Karnaugh maps populated using 0s versus 1s.](image)

2.1.3.6 Advantages.

- The Karnaugh maps method is a completely visual approach.
- The Karnaugh maps method eliminates the necessity for the human designer to learn all the theorems and laws of Boolean algebra.
- The Karnaugh maps method is not dependent on the expertise of the human designer in applying the theorems and laws of Boolean algebra.

2.1.3.7 Disadvantages.

- Karnaugh maps are most often used to represent 3-input and 4-input functions. It is possible to create similar maps for 5-input and 6-input functions, but these maps can become unclear and difficult to use. The Karnaugh map technique is generally not
considered to have any practical use for functions with more than six inputs and more than a single output function at a time.

- Other disadvantages concerning the use of Karnaugh maps are that they are not easy to develop on computers and the interpretation of the maps is subjective.

2.1.4 Quine-McCluskey Method.

2.1.4.1 Description.
Except in special cases and for certain truth tables, the K-map method simply breaks down more than six variables. With the Quine-McCluskey method, developed in the mid-1950s, we can find the minimized representation of any Boolean expression. It provides a systematic procedure for generating all prime implicants and then extracting a minimum set of primes covering the on-set (input values when the output is 1).

In order to understand the previous explanation we need to be more precise about some of the concepts used in the description of the Quine-McCluskey method and the process for obtaining a minimized expression in expressions with more than six variables.

An implicant of a function F is a single element of the on-set or any group of elements that can be combined together in a K-map. A prime implicant is an implicant that cannot be combined with another one to eliminate a literal. In other words, you create implicants to cover as much of the on-set as possible (each prime implicant is an implicant with as few literals as possible). Each prime implicant corresponds to a product term in the minimum sum-of-products expression for the function. The trick is to find the fewest prime implicants that cover the elements of the on-set. If a particular element of the on-set is covered by a single prime implicant, it is called an essential prime implicant. All essential primes must be part of the minimized expression.

![Figure 2.10 Prime implicants](image)

Observe the example depicted in figure 2.10 to make these concepts more concrete.
The four-variable K-map in Figure 2.10 contains six prime implicants: A’B’D, BC’, AC, A’C’D, AB, and B’CD. Only AC and BC’ are essential. Adding the additional implicant A’B’D covers the entire on-set. Thus the minimized expression for the function becomes F = A’B’D + AC + BC’.

The Quine-McCluskey method finds the prime implicants by repeatedly applying the uniting theorem. The contribution of Quine-McCluskey is to provide a tabular method that ensures that all prime implicants are found. To understand how it works, we will use the example shown in table 2.3.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 2.3 Example for Quine-McCluskey method.

**Finding Prime Implicants.** The first step is to list all elements of the on-set and don’t-care set in terms of their minterm indices, represented as a binary number. The elements are grouped according to the number of 1's in the binary representation.

<table>
<thead>
<tr>
<th>Column I</th>
<th>Column II</th>
<th>Column III</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 √</td>
<td>0-00 *</td>
<td>01-- *</td>
</tr>
<tr>
<td>0100 √</td>
<td>010- √</td>
<td>-1-1 *</td>
</tr>
<tr>
<td>1000 √</td>
<td>010- √</td>
<td></td>
</tr>
<tr>
<td>0101 √</td>
<td>010- √</td>
<td></td>
</tr>
<tr>
<td>0110 √</td>
<td>010- √</td>
<td></td>
</tr>
<tr>
<td>1001 √</td>
<td>010- √</td>
<td></td>
</tr>
<tr>
<td>1010 √</td>
<td>010- √</td>
<td></td>
</tr>
<tr>
<td>0111 √</td>
<td>011- √</td>
<td></td>
</tr>
<tr>
<td>1101 √</td>
<td>1-01 *</td>
<td></td>
</tr>
<tr>
<td>1111 √</td>
<td>-111 √</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4 Structure of a Quine-McCluskey implicant table.
Table 2.4 shows the structure of a Quine-McCluskey implicant table. The first column contains the minterms of the on-set, that is, single points in the Boolean space. In the example, each of these represents a four-variable product term (a minterm). As a result of applying the method, the second column will contain implicants that form edges in the Boolean space: three-variable product terms. After another iteration of the method, the third column will contain larger implicants that represent planes in the Boolean space: two-variable terms. A third iteration will find implicant cubes in the space: one-variable terms.

We begin the method by filling in the first column of the table as follows. Each group of the minterm indices of the on-set and don't care set is separated by a blank line. The first group has no 1’s in the indices, the second has one 1 in each index, the third has two 1’s in each index, and so on.

To apply the uniting theorem systematically, compare the elements in the first group with each element in the second. If they differ by a single bit, it means that the minterms that the numbers represent are adjacent in n-dimensional Boolean space. For example, 0000 = A’B’C’D’ and 0100 = A’BC’D’ can be combined into the implicant A’C’D’ according to the uniting theorem. The latter term is represented symbolically by 0-0. Every time a new implicant is formed, it is placed in the next column. Since each group differs in its 1’s count by one, it is sufficient to restrict comparisons with adjacent groups to detect when the uniting theorem can be applied.

Now we will apply the Quine-McCluskey algorithm to the whole first column. We begin with the first group (no 1’s) compared with the second group (one 1’s). 0000 is compared with 0100 and 1000, yielding terms for the second column of 0-0 and –000 respectively. Every time a term contributes to a new implicant, it receives a check mark. This means that the implicant is not prime: it can be combined with some other element to form a larger implicant.

We repeat the procedure for the second group compared with the third group. 0100 combines with 0101 and 0110, giving 010- and 01-0 in the second column. 1000 combines with 1001 and 1010, resulting in 100- and 10-0.

Now we will try the third group compared with the fourth group. 0101 combines with 0111 and 1101 to give 01-1 and -101. 0110 combines with 0111 to yield 011-. 1001 combines with 1101 to give 1-01. 1010 does not combine with any element of the fourth group.

When we compare the fourth to the fifth group, two additional terms are added: -111 and 11-1.

The procedure is repeated for the groups in column II. In order for the elements to be combined, they must differ by a single bit and must have their "-" in the same bit position. The elements of the first group do not combine with any elements of the second group. We mark these with asterisks because they are prime implicants: we have expanded them as much as possible.
In the second and third groups, 010- can be combined with 011-, yielding 01-- for the third column. 01-0 and 01-1 are combined to derive the same condensed term. 100- and 10-0 cannot be combined further and are prime implicants.

Between the third and fourth groups only the additional term -1-1 is added to the third column, derived from the combinations of -101 and -111, and 01-1 and 11-1.

The elements of the third column cannot be reduced any further. Both are marked as prime implicants. Since no new implicants are added, we have found all the prime implicants, and the first phase of the algorithm can now terminate.

The algorithm has found the following prime implicants:

\[ 0-00 = A'CD' \quad -000 = BC'D' \]
\[ 100- = AB'C' \quad 10-0 = AB'D' \]
\[ 1-01 = AC'D \quad 01-- = A'B \]
\[ -1-1 = BD \]

These are shown circled in the K-map of Figure 2.11. They are exactly the same as the prime implicants found through applying Quine-McCluskey method. Note that in this portion of the procedure, the don't cares have been treated as though they were 1's.

![Figure 2.11 Prime implicants found by the Quine-McCluskey method](image)

**Finding the Minimum Cover.** The second step of the method is to find the smallest collection of prime implicants that cover the complete on-set of the function. This is accomplished through the prime implicant chart (as opposed to the implication chart used in the first phase), as shown in Figure 2.12.
The prime implicant chart is organized as follows. The columns are labeled with the minterm indices of the on-set. Note that don't cares are not included in this stage. The rows are labeled with the minterms covered by a given prime implicant. This is done by taking the indices of the prime implicant representation and replacing each "." by all possible combinations of 1's and 0's. For example, -1-1 becomes 0101, 0111, 1101, 1111, which are the indices of the minterms m5, m7, m13, and m15. An X is placed in the row/column location if the minterm represented by the column is covered by the prime implicant associated with the row. The initial configuration is given in Figure 2.12(a).

Next, we look for essential prime implicants. These are immediately apparent whenever there is a single X in any column. This means that there is a minterm that is covered by one and only one prime implicant. The essential prime implicants must participate in the final cover. We place a line through the column and row in which the essential prime implicant has been found and place a box around the prime. This is shown in Figure 2.12(b).

The essential prime implicants usually cover additional minterms. We cross out any columns that have an X in a row associated with an essential prime. These minterms are already covered by the essential primes. This is shown in Figure 2.12(c).
In the example, two minterms are still uncovered, in the columns 9 and 13. The final step is to find as few primes as possible that cover the remaining minterms. In our example, the single prime implicant 1-01 covers both of these. By adding it to the two essential prime implicants already found, we complete the cover. This is shown in Figure 2.12(d). The solution found is identical to the solution found through the application of K-maps.

2.1.4.2 Advantages.
- The Quine-McCluskey method is not limited to functions of 4 variables (as Karnaugh maps are, for all practical purposes)
- The method is easy to automate (i.e. it can be easily implemented on a computer).

2.1.4.3 Disadvantages.
- The main disadvantage of Quine-McCluskey method is that it is slower and less visual in comparison with the K-maps method.
2.2 *Evolutionary Algorithms.*

Before describing Evolutionary Algorithms (EAs) it would be better if we first look at where these algorithms are located within the Artificial Intelligence (AI) field. We can classify the evolutionary algorithms within the scope of AI in the way depicted in Figure 2.13.

![Evolutionary Algorithms within Artificial Intelligence](image)

*Figure 2.13 Evolutionary Algorithms within Artificial Intelligence.*

The proposed classification in Figure 2.13 considers the symbolic and subsymbolic nature of the processes that human beings appear to have at their disposal for performing cognitive tasks [Smolensky 1988]. It is possible to solve problems by conscious reasoning, following rules and algorithms. On the other hand, much of everyday processing appears to be immediate and intuitive, based on past experience. While a person is learning a task, such as driving a car, he often goes through a phase of conscious application of rules. After the skill has been perfected, the rules are no longer applied and the performance becomes intuitive. When the cognitive process is based on conscious rule application, the symbolic approach is an elegant way to model it. The knowledge structures and algorithms can be made explicit, and perhaps even be used to teach people such skills. It may well be that these tasks turn out to be implemented by subsymbolic processes when we really look, but the subsymbolic properties do not play a significant role in these behaviors. As long as our goal is to understand behavior, or perhaps replicate it on a computer, the symbolic procedures are more appropriate than the neural procedures. However, there are also processes that clearly have foundations that are subsymbolic. Most of everyday language...
processing appears to be this way: it is based firmly on associations that are opaque and immediate. Processing appears effortless and intuitive. These properties are very difficult to capture in the symbolic framework. It seems that a complete account of linguistic performance has to include subsymbolic mechanisms.

EAs are subsymbolic computer-based problem-solving systems, which use computational models of some of the known mechanisms of evolution as key elements in their design and implementation. A variety of evolutionary algorithms have been proposed. The major ones are: genetic algorithms, evolutionary programming, evolution strategies, classifier systems, and genetic programming. They all share a common conceptual base of simulating the evolution of individual structures via processes of selection, reproduction (sexual or asexual) and mutation. The processes depend on the perceived performance of the individual structures as defined by an environment.

More precisely, EAs maintain a population of structures, that evolve according to rules of selection and other operators, that are referred to as "search operators", (or genetic operators), such as: reproduction and mutation. Each individual in the population receives a measure of its fitness in the environment. Reproduction focuses attention on high fitness individuals, thus exploiting the available fitness information. Mutation perturbs those individuals, providing general heuristics for exploration. Although simplistic from a biologist's viewpoint, these algorithms are sufficiently complex to provide robust and powerful adaptive search mechanisms [Spears 1993].

To understand EAs, it is necessary to have some appreciation of the biological processes on which they are based. Firstly, we should note that evolution (in nature or anywhere else) is not a purposive or directed process. That is, there is no evidence to support the assertion that the goal of evolution is to produce Mankind. Indeed, the processes of nature seem to boil down to a haphazard generation of biologically diverse organisms. Natural selection or different individuals competing for resources in the environment determine some of evolution. Some are better than others. Those that are better are more likely to survive and propagate their genetic material [Beasley 2000].

In nature, we see that the encoding for genetic information (genome) is done in a way that admits asexual reproduction. Asexual reproduction typically results in offspring that are genetically identical to the parent. (Large numbers of organisms reproduce asexually; this includes most bacteria, which some biologists hold to be the most successful species known.)

Sexual reproduction allows the combination of chromosomes, producing offspring that contain a combination of information from each parent. At the molecular level what occurs is that a pair of almost identical chromosomes bumps into one another, exchange chunks of genetic information and drift apart. This is the recombination operation, which is often referred to as crossover, because of the way that biologists have observed strands of chromosomes crossing over during the exchange.
Figure 2.14 Block diagram for an Evolutionary Algorithm.
Reproduction occurs in an environment where the selection of the mate is largely a function of the fitness of the individual, i.e. the ability to compete in the environment. Sometimes a random effect is also involved. Some EAs use a simple function of the fitness measure to select individuals probabilistically to undergo genetic operations such as sexual or asexual reproduction (the propagation of unaltered genetic material). This is called fitness-proportionate selection. Other implementations use a model in which certain randomly selected individuals in a subgroup compete and the fittest is selected. This is called tournament selection and is the form of selection we see in nature when individuals with a high level of fitness have the privilege of mating with another individual of the same specie.

Much EA research has assumed that the two processes that most contribute to evolution are crossover and fitness based selection/reproduction. By definition evolution requires diversity in order to work. In nature, an important source of diversity is mutation. In an EA, a large amount of diversity is usually introduced at the start of the algorithm, by randomizing the genes in the population. The importance of mutation, which introduces further diversity while the algorithm is running, therefore continues to be a matter of debate. Some refer to it as a background operator, simply replacing some of the original diversity, which may have been lost, while others view it as playing the dominant role in the evolutionary process.

It cannot be stressed too strongly that an evolutionary algorithm (as a simulation of a genetic process) is not a random search for a solution to a problem (highly fit individual). EAs use stochastic processes, but the result is different from non-random (it is better than random).

Figure 2.14 depicts a block diagram showing the basic operation of an Evolutionary Algorithm.

2.2.1 Genetic Algorithms.

The famous naturalist Charles Darwin defined *Natural Selection or Survival of the Fittest* in his book [Darwin 1929] 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. Units called genes, which form sets called chromosomes, control such features. Over subsequent generations not only the fittest individuals survive, but also their fittest genes that are transmitted to their descendants during the sexual reproduction process, which is called crossover.

John H. Holland became interested in the application of natural selection to machine learning, and in the late 60s, 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 [Holland 1975] [Holland 1992].
In 1989, Goldberg published a book [Goldberg 1989] that provided a solid scientific basis for this area, and cited no less than 73 successful applications of the genetic algorithm. In the last few years the growing interest in this technique is reflected in a larger number of conferences, two new international journals, and an increasing amount of software and literature devoted to this subject.

According to Koza [1992] the genetic algorithm is a highly parallel mathematical algorithm that transforms a set (population of individual mathematical objects, typically fixed length character strings patterned after chromosome strings), each with an associated fitness value, into a new population (i.e., the next generation). The genetic algorithm uses operations patterned after the Darwinian principle of reproduction and survival of the fittest and after naturally occurring genetic operations (notably sexual reproduction).

In fact, the genetic algorithm derives its behavior from a metaphor of one of the nature’s mechanisms of evolution that is called hard selection [Heitkoetter 1995]. In this process, only the best available individuals are retained for generating descendants. This contrasts with soft selection, which offers a probabilistic mechanism for maintaining individuals to be parents of future progeny despite possessing relatively lower fitness values.

It has been argued [Heitkoetter 1995] that the term genetic algorithm (GA) is misleading, since natural selection is only one of the mechanisms of evolution. It would be more appropriate to call them hard selection (HS) algorithms in accordance with the characteristic that they deal with only that particular selection scheme. However, the term is so common today, that a change does not seem feasible, at least in the near future.

A genetic algorithm for a particular problem must have the following five components [Michalewicz 1992]:

a) A representation for potential solutions to the problem.
b) A way to create an initial population of potential solutions.
c) An evaluation function that plays the role of the environment, rating solutions in terms of their “fitness”.
d) Genetic operators that alter the composition of children.
e) Values for various parameters that the genetic algorithm uses (population size, generations, probabilities of applying genetic operators, etc.).

Some of the basic terminology used by the genetic algorithms (GAs) community is as follows [Heitkoetter 1995]:

A chromosome is a data structure that holds a “string” of task parameters, or genes. This string may be stored, for example, as a binary bit-string (binary representation) or as an array of integers (floating-point or real-coded representation) that represent a floating-point number. This chromosome is analogous to the base-4 chromosomes present in our own DNA. Normally, in the GA community, the haploid model of a cell is assumed (one-chromosome individuals). However, diploids have also been used in the past [Goldberg 1989]. A gene is a subsection of a chromosome that usually encodes the value of a single parameter. An allele is the value of a gene. For example, for a binary representation each...
gene may have an allele of 0 or 1, and for a floating-point representation, each gene may have an allele from 0 to 9.

A **schema** (plural schemata) is a pattern of gene values in a chromosome, which may include “do not care” states (represented by a # symbol). Thus in a binary chromosome, each schema can be specified by a string of the same length as the chromosome, with each character being one of (0, 1, #)\(^2\). A particular chromosome is said to contain a particular schema if it matches the schema (e.g. chromosome 01101 matches schema #1#0#). If the solution to a problem can be represented by a set of N real-valued parameters, then the job of finding this solution can be thought of as a search in an N-dimensional space. This region is simply referred as the **search space** of the problem.

The **fitness** of an individual is a value that reflects its performance (i.e., how well a certain task is performed). A fitness function is a mapping of the chromosomes in a population according to their corresponding fitness values. A fitness landscape is the hyper surface obtained by applying the fitness function to every point in the search space. A **building block** is a small, tightly clustered group of genes that have coevolved in such a way that their introduction into any chromosome will be likely to give increased fitness to that chromosome. The building block hypothesis [Goldberg 1989] states that GAs generate their solutions by first finding as many building blocks as possible, and then combining them to give the highest fitness. **Deception** is a condition under which the combination of good building blocks leads to reduced fitness, rather than increased fitness. Goldberg [1989] proposed this condition as a reason for the failure of GAs in certain tasks. **Epistasis** is the interaction between different genes in a chromosome. It is the extent to which the contribution to fitness of one gene depends on the values of other genes. Geneticists use this term to refer to a “masking” or “switching” effect among genes and a gene is considered to be epistatic if its presence suppresses the effect of a gene at another locus. This concept is closely related to deception, since a problem with a high degree of epistasis is deceptive, because building blocks cannot be formed. On the other hand, problems with little or no epistasis are very easy to solve (the Hill climbing method is sufficient).

**Elitism** (or an elitist strategy) is a mechanism that ensures that the chromosomes of the highly fit member(s) of the population are passed on to the next generation without being altered by any genetic operator. The use of elitism guarantees that the maximum fitness of the population never decreases from one generation to the next, and it normally produces a faster convergence of the population.

**Exploitation** is the process of using information gathered from previously visited points in the search space to determine which places might be profitable to visit next. Hill climbing is an example of exploitation, because it investigates adjacent points in the search space, and moves in the direction giving the greatest increase in fitness. Exploitation techniques

---

\(^2\) We can observe an analogy between the implicants in the Karnaugh maps method and the schemata in the schema theory [Goldberg 1989]. An implicant and a chromosome represented by the string 1001 characterize single points in the Boolean space. An implicant and a chromosome represented by 10#1 correspond to edges in the Boolean space. An implicant and a chromosome represented by 1##1 symbolize planes in the Boolean space and finally an implicant and a chromosome represented by 1#### correspond to cubes in the Boolean space.
are good at finding local minimum (or maximum) values. The GA uses crossover as an exploitation mechanism. **Exploration** is the process of visiting entirely new regions of a search space, to see if anything promising may be found there. Unlike exploitation, exploration involves leaps into unknown regions. Random search is an example of exploration. Problems that have many local minimum (or maximum) values can sometimes only be solved using exploration techniques such as random search. The GA uses mutation as an exploration mechanism. Figure 2.15 shows the differences between exploitation and exploration.

![Figure 2.15 Exploitation and Exploration differences.](image)

A **genotype** represents a potential solution to a problem, and is basically the string of values, also called chromosome, chosen by the user. A **phenotype** is the meaning of a particular chromosome, defined externally by the user. **Genetic drift** is the name given to the changes in gene/allele frequencies in a population over many generations, resulting from chance rather than from selection. It occurs most rapidly in small populations and can lead to some alleles becoming extinct, thus reducing the genetic variability in the population. A **niche** is a group of individuals that have similar fitness. Normally in multiobjective and multimodal optimization, a technique called **sharing** is used to reduce the fitness of those individuals who are in the same niche in order to prevent the population from converging on a single solution, so that stable subpopulations can be formed, each one corresponding to a different objective or peak (in a multimodal optimization problem) of the function.

The basic operation of a Genetic Algorithm is illustrated in the block diagram depicted in Figure 2.16
Figure 2.16 Block diagram for a Genetic Algorithm.
First, an initial population is randomly generated. The individuals of this population will be a set of chromosomes or strings of characters (letters and/or numbers) that represent all the possible solutions to the problem. We apply a fitness function to each one of these chromosomes in order to measure the quality of the solution encoded by the chromosome. Knowing each chromosome's fitness, a selection process takes place to choose the individuals (presumably, the fittest) that will be the parents of the following generation. The most commonly used selection schemes are the following [Goldberg 1991]:

**Proportionate Reproduction:** This term is used generically to describe several selection schemes that choose individuals for birth according to their objective function values \( f \). In these schemes, the probability of selection \( p \) of an individual from the \( i \)th class in the \( t \)th generation is calculated as:

\[
P_{i,t} = \frac{f_i}{\sum_{j=1}^{k} m_{i,j} f_j}
\]

Formula 2.1

Where \( k \) classes exist and the total number of individuals sums to \( n \) (population).

Several methods have been suggested for sampling this probability distribution, including Monte Carlo or roulette wheel selection [Jong 1975], stochastic remainder selection [Booker 1982] [Brindle 1981], and stochastic universal selection [Baker 1987] [Grefenstette 1989].

**Ranking Selection:** In this scheme, proposed by Baker [1985], the population is sorted from best to worst, and each individual is copied as many times as possible, according to a non-increasing assignment function, and then proportionate selection is performed according to that assignment.

**Tournament Selection:** The population is shuffled and then divided into groups of \( k \) elements from which the best individual (i.e., the fittest) will be chosen. This process has to be repeated \( k \) times because on each iteration only \( m \) parents are selected, where:

\[
m = \frac{\text{PopulationSize}}{k}
\]

Formula 2.2

For example, if we use binary tournament selection \( (k = 2) \), then we have to shuffle the population twice, since at each stage half of the parents required will be selected. The interesting property of this selection scheme is that we can guarantee multiple copies of the fittest individual among the parents of the next generation.

After being selected, crossover takes place. During this stage, the genetic material of a pair of individuals is exchanged in order to create the population of the next generation. The two
main ways of performing crossover are called single-point and two-point crossover. When a single-point crossover scheme is used, a position of the chromosome is randomly selected as the crossover point, as indicated in Figure 2.17.

When a two-point crossover scheme is used, two positions of the chromosome are randomly selected, as indicated in Figure 2.18.

![Figure 2.17 Use of a single-point crossover between two chromosomes.](image1)

![Figure 2.18 Use of a two-point crossover between two chromosomes.](image2)

**Mutation** is another important genetic operator that randomly changes a gene of a chromosome. If we use a binary representation, a mutation changes from 0 to 1 and vice versa. This operator allows the introduction of new chromosomic material to the population and, from the theoretical perspective, it ensures that given any population the entire search space is connected [Buckles 1992]. If we knew in advance the final solution, it would be
trivial to determine how to stop a genetic algorithm. However, as this is not normally the case, we have to use one of the two following criteria to stop the GA: either give a fixed number of generations in advance, or verify when the population has stabilized (i.e., all or most of the individuals have the same fitness).

GAs differ from traditional search techniques in several ways [Buckles 1992]:

a) GAs do not require problem specific knowledge to carry out a search.

b) GAs use stochastic instead of deterministic operators and appear to be robust in noisy environments.

c) In evaluating a population of $n$ strings, the GA implicitly estimates the average fitness of all schemas that are present in the population and increasing or decreasing their representation. This simultaneous implicit evaluation of a large number of schemas in a population of $n$ strings is known as implicit parallelism. This ability makes them less susceptible to local maximums.

The traditional representation used by the genetic algorithms community is the binary scheme according to which a chromosome is a string in the form $<b_1, b_2, \ldots, b_m>$, where $b_1, b_2, \ldots, b_m$ are called alleles (either zeros or ones). Since the binary alphabet offers the maximum number of schemata per bit of information of any coding [Goldberg 1989], its use has become very popular among scientists. This coding also facilitates theoretical analysis of the technique and allows elegant genetic operators. However, since the “implicit parallelism” property of GAs does not depend on using bit strings [Michalewicz 1992] it is worthwhile to experiment with larger alphabets, and even with new genetic operators. In particular, for optimization problems in which the parameters to be adjusted are continuous, a floating-point representation scheme seems a logical choice. According to this representation, a chromosome is a string of the form $<d_1, d_2, \ldots, d_m>$, where $d_1, d_2, \ldots, d_m$ are digits (numbers between zero and nine). Consider the examples shown in Figure 2.19, in which the same value is represented using binary and floating-point encoding.

![Figure 2.19 Representing the same number using binary and floating-point encoding.](image)
The term “floating” may seem misleading since the position of the implied decimal point is fixed, and the term “fixed point representation” seems more appropriate. However, the reason that the term “floating point” is preferred is because in this representation each variable (representing a parameter to be optimized) may have the point at any position along the string. This means that even when the point is fixed for each gene, is not necessarily fixed along the chromosome. Therefore, some variables could have a precision of 3 decimal places, while others are integers, and still they could all be represented with the same string. Nevertheless, the term real-coded GAs is also used in the literature [Goldberg 1990] [Wright 1991].

Floating-point representation is faster and easier to implement, and provides a higher precision than its binary counterpart, particularly in large domains, where binary strings would be prohibitively long. One of the advantages of floating-point representation is that it has the property that two points close to each other in the representation space must also be close in the problem space, and vice versa [Michalewicz 1992]. This is not generally true in the binary approach, where the number of different bit positions normally defines the distance in a representation.

Goldberg [Goldberg 1990] has presented a theory of convergence for real-coded or floating-point GAs, and also real numbers and other alphabets have been proposed [Wright 1991], particularly for numerical optimization, in a resemblance of the power of evolution strategies in this domain [Schwefel 1981]. As Eshelman and Schaffer [Eshelman 1993] point out, a lot of researchers in the GA community have agreed to use real-coded genetic algorithms for numerical optimization despite of the fact that there are theoretical arguments that seem to show that small alphabets should be more effective than large alphabets. Practitioners, on the other hand, have shown that real-coded genes work better in practice [Davis 1991][Coello 1999]. A few attempts have been made to develop a theoretical defense of this representation scheme, from which the recent investigation by Eshelman and Schaffer deserves special attention [Eshelman 1993]. One of the main abilities of real-coded GAs is their capacity to exploit the gradualness of functions of continuous variables (where gradualness is taken to mean that small changes in the variables correspond to small changes in the function) [Eshelman 1993] [Wright 1991].

2.2.2 Evolutionary Programming.

Evolutionary Programming (EP), originally conceived by Lawrence J. Fogel in 1960 [Fogel 1995], is a stochastic optimization strategy similar to genetic algorithms, but instead places emphasis on the behavioral linkage between parents and their offspring, rather than seeking to emulate specific genetic operators as observed in nature. Evolutionary programming is similar to Evolution Strategies (ES), although the two approaches were developed independently.

Like both ES and GAs, EP is a useful method of optimization when other techniques such as gradient descent or direct, analytical discovery are not possible. Combinatorial and real-valued function optimization problems in which the optimization surface or fitness landscape is difficult to explore, possessing many locally optimal solutions, are well suited for evolutionary programming.
For EP, like GAs, there is an underlying assumption that a fitness landscape can be characterized in terms of variables, and that there are optimum or multiple solutions in terms of those variables. For example, if we were trying to find the shortest path in a Traveling Salesman Problem, each solution would be a path. The length of the path could be expressed as a number, which would serve as the solution's fitness. The fitness landscape for this problem could be characterized as a hyper surface proportional to the path lengths in a space of possible paths. The goal would be to find the globally shortest path in that space, or more practically, to find very short tours very quickly.

The basic EP method involves 3 steps, which are repeated until a threshold for iteration is exceeded or an adequate solution is obtained:

a) **Choose an initial population of trial solutions at random.** The number of solutions in a population is highly relevant to the speed of optimization, but no definite answers are available as to how many solutions are appropriate and how many solutions are just wasteful.

b) **Each solution is replicated into a new population.** Each of these offspring solutions is mutated according to a distribution of mutation types, ranging from minor to extreme, with a continuum of mutation types between. The severity of mutation is judged on the basis of the functional change imposed on the parents.

c) **Computing its fitness assesses each offspring solution.** Typically, a stochastic tournament is held to determine N solutions to be retained for the population of solutions, although this is occasionally performed deterministically. There is no requirement that the population size be held constant, however, nor that only a single offspring be generated from each parent.

It should be pointed out that EP typically does not use any crossover as a genetic operator.

There are two important ways in which EP differs from GAs. First, there is no constraint on the representation. The typical GA approach involves encoding the problem solutions as a string of representative tokens, the genome. In EP, the representation follows from the problem. A neural network can be represented in the same manner as it is implemented, for example, because the mutation operation does not demand a linear encoding. In this case, for a fixed topology, real valued weights could be coded directly as their real values and mutation would operate by perturbing a weight vector with a zero mean multivariate Gaussian perturbation. For variable topologies, the architecture is also perturbed, often using Poisson distributed additions and deletions.

Second, the mutation operation simply changes aspects of the solution according to a statistical distribution which weights minor variations in the behavior of the offspring as highly probable and substantial variations as increasingly unlikely. Further, the severity of mutations is often reduced as the global optimum is approached. There is a certain tautology here: if the global optimum is not already known, how can the spread of the mutation operation be recognized when the solutions are approaching the optimum?
Several techniques have been proposed and implemented which address this difficulty, the most widely studied being the "Meta-Evolutionary" technique in which the variance of the mutation distribution is subject to mutation by a fixed variance mutation operator and evolves along with the solution.

Figure 2.20 Block diagram of Evolutionary Programming.
The first communication between the evolutionary programming and evolution strategy groups occurred in early 1992, immediately prior to the first annual EP conference. Despite their independent development over 30 years, they share many similarities. When implemented to solve real-valued function optimization problems, both typically operate on the real values themselves (rather than any coding of the real values as is often done in GAs). Multivariate zero mean Gaussian mutations are applied to each parent in a population and a selection mechanism is applied to determine which solutions should be removed from the population. The similarities extend to the use of self-adaptive methods for determining the appropriate mutations, to the use of methods in which each parent carries not only a potential solution to the problem at hand, but also information on how new trials will be distributed (offspring).

The main differences between ES and EP are:

1. **Selection:** EP typically uses stochastic selection via a tournament. Each trial solution in the population faces competition against a pre-selected number of opponents and receives a "win" if it is at least as good as its opponent in each encounter. Selection then eliminates those solutions with the least wins. In contrast, ES typically uses deterministic selection in which the worst solutions are purged from the population based directly on their function evaluation.

2. **Reproduction:** EP is an abstraction of evolution at the level of reproductive populations (i.e. species) and thus no reproduction mechanisms are typically used because reproduction does not occur between species. In contrast, ES is an abstraction of evolution at the level of individual behavior. When self-adaptive information is incorporated this is purely genetic information (as opposed to phenotypic) and thus some forms of reproduction are reasonable and many forms of reproduction have been implemented within ES. Again, the effectiveness of such operators depends on the problem at hand.

Figure 2.20 shows a block diagram of the operation of Evolutionary Programming.

### 2.2.3 Genetic Programming

Genetic Programming (GP) is the extension of the genetic model of learning into the space of programs (i.e., it is an extension of the genetic algorithm). That is, the objects that constitute the population are not fixed-length character strings that encode possible solutions to the problem at hand. They are programs that, when executed, are the candidate solutions to the problem. These programs are expressed in genetic programming as parse trees, rather than as lines of code. Thus, for example, the simple program "a + b * c" would be represented as in Figure 2.21; or, to be precise, as suitable data structures linked together to achieve this effect.

Due to the simplicity of the programming language Lisp (i.e., the parser is built-in as part of the language), many developers tend to use Lisp. However, this is simply an implementation detail. There are straightforward methods to implement GP using a non-Lisp programming environment.
The programs in the population are composed of elements from the function set and the terminal set, which are typically fixed sets of symbols selected to be appropriate to the solution of problems in the area concerned.

In GP the crossover operation is implemented by taking randomly selected sub-trees in the individuals (selected according to fitness) and exchanging them. It should be pointed out that GP usually does not use any mutation as a genetic operator.

One of the central challenges of computer science is to get a computer to do what needs to be done, without instructing it. Genetic programming addresses this challenge by providing a method for automatically creating a working computer program from a high-level problem statement of the problem [Koza 1992]. Genetic programming achieves this goal of automatic programming (also sometimes called program synthesis or program induction) by genetically breeding a population of computer programs using the principles of Darwinian natural selection and biologically inspired operations. The operations include selection, reproduction and mutation, and architecture-altering operations patterned after gene duplication and gene deletion in nature.

**Creation of Initial Population of Computer Programs.** Genetic programming starts with a primordial bunch of randomly generated computer programs. The set of functions that may appear at the internal points of a program tree may include ordinary arithmetic functions and conditional operators. The set of terminals appearing at the external points typically include the program’s external inputs (such as the independent variables X and Y) and random constants (such as 3.2 and 0.4). The randomly created programs typically have different sizes and shapes.

**Main Generational Loop of Genetic Programming.** The main generational loop of a run of genetic programming consists of the fitness evaluation, Darwinian selection, and the genetic operations. Each individual program in the population is evaluated to determine its adequacy for solving the problem at hand. Programs are then probabilistically selected from the population based on their fitness to participate in the various genetic operations (permitting reselection). While a more fit program has a better chance of being selected,
even individuals known to be unfit are allocated some trials in a mathematically principled way. That is, genetic programming is not a purely greedy hill-climbing algorithm.

The individuals in the initial random population and the offspring produced by each genetic operation are all syntactically valid executable programs. After many generations, a program may emerge to solve, or approximately solve, the problem at hand.

**Mutation Operation.** In the mutation operation (Figure 2.22), a single parental program is probabilistically selected from the population, based on fitness. A mutation point is randomly chosen, the subtree rooted at that point is deleted, and a new subtree is grown there, using the same random growth process that was used to generate the initial population. This asexual mutation operation is typically performed sparingly (with a probability of 1% during each generation of the run).

![Diagram](image)

**Figure 2.22 Mutation operation.**

**Sexual Reproduction Operation (Crossover).** In the crossover, or sexual reproduction operation (Figure 2.23), two parental programs are probabilistically selected from the population, based on fitness. The two parents participating in crossover are usually of different sizes and shapes. A crossover point is randomly chosen in the first parent and a crossover point is randomly chosen in the second parent. Then the subtree, rooted at the crossover point of the first, or receiving parent, is deleted and replaced by the subtree from
the second, or contributing parent. Crossover is the predominant operation in genetic programming (and genetic algorithm) and is performed with a high probability (85% to 90%).

![Crossover Operation Diagram]

**Figure 2.23** Crossover operation.

**Asexual Reproduction Operation.** The asexual reproduction operation copies a single individual, probabilistically selected on the basis of fitness, into the next generation of the population.

Figure 2.24 depicts a block diagram showing the operation of Genetic Programming.

**2.2.4 Evolution Strategies.**

In 1963 two students at the Technical University of Berlin (TUB) were introduced and were soon to collaborate on experiments that used the wind tunnel of the Institute of Flow Engineering. During the search for the optimal shapes of bodies in a flow, which was then a subject of laborious intuitive experimentation, the idea was conceived of proceeding strategically. However, attempts with the coordinate and simple gradient strategies were unsuccessful. Then one of the students, Ingo Rechenberg, hit upon the idea of trying random changes in the parameters defining the shape, following the example of natural mutations. This was the birth of evolution strategy. A third student, Peter Bienert, joined them and started the construction of an automatic experimenter, which would work
according to the simple rules of mutation and selection. The second student, Hans-Paul Schwefel began testing the efficiency of the new methods with the help of a Zuse Z23 computer; due to the numerous objections to these "random strategies."

![Figure 2.24 Block diagram of Genetic Programming.](image-url)
Evolution Strategies were invented to solve technical optimization problems (TOPs) like constructing an optimal flashing nozzle, and until recently the civil engineering community only knew ES, as an alternative to standard solutions. Usually no closed form analytical objective function is available for TOPs and hence no applicable optimization method exists, only the engineer's intuition. It is precisely for that kind of problem that techniques like ES are more suitable.

The first attempts to imitate principles of organic evolution on a computer still resembled the iterative optimization methods known up to that time. In a two-membered or (1+1) ES, one parent generates one offspring per generation by applying normally distributed mutations. For example, smaller steps are more likely to occur than big ones, until a child performs better than its ancestor and takes its place. Due to this simple structure, theoretical results for step-size control and convergence velocity could be derived. The ratio between successful mutations and the total number of mutations should be 1/5: the so-called 1/5 success rule was discovered. This first algorithm, using mutation only, was then enhanced to a (μ+1) strategy that incorporated reproduction due to several μ parents being available. The mutation scheme and the exogenous step-size control were carried over unchanged from (1+1) ESs.

Schwefel later generalized these strategies to the multimembered ES now denoted by (μ+λ) and (μ,λ) which imitates the following basic principles of organic evolution: a population, leading to the possibility of reproduction by random mating, mutation and selection. These strategies are termed plus strategy and comma strategy, respectively. In the plus case, the parental generation is taken into account during selection; while in the comma case only the offspring is taken into account during selection, and the parents die off. μ denotes the population size, and λ denotes the number of resulting offspring per generation.

However, dealing with ES is sometimes seen as "strong tobacco," because it requires a great amount of probability theory and applied statistics to understand the inner workings of an ES, as it navigates through the hyperspace of the usually n-dimensional problem space, by throwing hyperellipses into the deep.

Imagine a large black box, as large as, for example, a Coca-Cola vending machine. A single individual of the ES' population consists of the following genotype representing a point in the search space:

**Object Variables.** Real-valued \(x_i\) have to be tuned by reproduction and mutation in order that an objective function reaches its global optimum. Referring to the metaphor mentioned previously, the \(x_i\) represents the regulators of the alien Coca-Cola vending machine.

**Strategy Variables.** Real-valued \(s_j\) (usually denoted by a lowercase sigma) or mean step-sizes determine the mutability of the \(x_j\). They represent the standard deviation of a \((0, s_j)\) Gaussian Distribution (GD) being added to each \(x_j\) as an undirected mutation. With an "expectancy value" of 0 the parents will produce offspring similar to themselves on average. In order to make a doubling and a halving of a step-size equally probable, the \(s_j\) mutate log-normally, distributed, i.e. \(\exp(GD)\), from generation to generation. These step-
sizes hide the internal model the population has made of its environment, i.e. a self-
adaptation of the step-sizes has replaced the exogenous control of the (1+1) ES.

This concept works because selection sooner or later favors those individuals who have
built a good model of the objective function, thus producing better offspring. So, learning
takes place on two levels: (1) at the genotypic, i.e. the object and strategy variable level and
(2) at the phenotypic level, i.e. the fitness level.

Depending on an individual's \(x\), the resulting objective function value \(f(x)\), where \(x\) denotes
the vector of objective variables, serves as the phenotype (fitness) in the selection step. In a
plus strategy, the \(\mu\) best of all \((\mu+\lambda)\) individuals survive to become the parents of the next
generation. Using the comma variant, selection takes place only among the \(\lambda\) offspring. The
second scheme is more realistic and therefore more successful, because no individual may
survive forever, which could at least theoretically occur using the plus variant. Unypical
for conventional optimization algorithms and lavish at first sight, a comma strategy
allowing intermediate deterioration performs better. Only by forgetting highly fit
individuals can a permanent adaptation of the step-sizes take place and avoid long
stagnation phases due to misadapted \(s\). This means that these individuals have built an
internal model that is no longer appropriate for further progress, and thus should better be
discarded [Schwefel 1987].

By choosing a certain ratio \(\mu/\lambda\), one can determine the convergence property of the
evolution strategy: if one wants a fast, but local convergence, one should choose a small
hard selection, ratio, e.g. \((5,100)\), but looking for the global optimum, one should favor a
softer selection \((15,100)\).

Self-adaptation within ESs depends on the following agents [Schwefel 1987]:

**Randomness**: One cannot model mutation as a purely random process. This would mean
that a child is completely independent of its parents.

**Population size**: The population has to be sufficiently large. Not only the current best
should be allowed to reproduce, but a set of good individuals. Biologists have coined the
term “requisite variety” to mean the genetic variety necessary to prevent a species from
becoming genetically poorer and eventually dying out.

**Cooperation**: In order to exploit the effects of a population \((\mu > \lambda)\), the individuals should
recombine their knowledge with that of others (cooperate) because one cannot expect the
knowledge to accumulate in the best individual only.

**Deterioration**: In order to allow better internal models (step-sizes) to provide better
progress in the future, one should accept deterioration from one generation to the next. A
limited life-span in nature is not a sign of failure, but an important means of preventing a
species from freezing genetically.
ESs prove to be successful when compared to other iterative methods on a large number of test problems [Schwefel 1987]. They are adaptable to nearly all sorts of problems in optimization because they need very little information about the problem, especially no derivatives of the objective function. ESs are capable of solving high dimensional, multimodal, nonlinear problems subject to linear and/or nonlinear constraints. The objective function can also be the result of a simulation and it does not have to be given in a closed form. This also holds for the constraints, which may represent the outcome of, e.g. a finite elements method. ESs have been adapted to vector optimization problems [Kursawe 1992], and they can also serve as a heuristic for NP-complete combinatorial problems like the traveling salesman problem or problems with a noisy or changing response surface.
2.3 **Case Based Reasoning.**

Case Based Reasoning (CBR) is a problem-solving paradigm that in many respects is fundamentally different from other major AI approaches. 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 also is 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. The CBR field has grown rapidly over the last few years, as seen by its increased share of papers at major conferences, available commercial tools, and successful applications in daily use [Aamdot 1994].

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 have to be found by returning to the beginning.

The CBR could be divided in the following main stages (see Figure 2.25).

![Diagram of CBR process](image)

**Figure 2.25 General structure of the CBR.**

**Identifying the new problem:** The system receives the input case (new problem) and analyzes the 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 similitude between the cases are called indexes.
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 big. The selection of cases from the case base could be considered as an analog selection due to the fact it is based only on the distance measure (similarity) between the new case and each case in the case base.

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.

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.

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.

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.
2.4 Previous Works in Combinational Logic Circuits Optimization using EAs and/or CBR.

Coello et al [1996a][1997][1997a][2000][Hernandez 1999] have proposed an approach based on a genetic algorithm (GA) to design combinational logic circuits in which the objective is to minimize their total number of gates. They analyze the solutions found by the GA trying to find some clues as to how it reduces a Boolean expression, and they indicate that such a reduction is achieved by reusing common patterns within the circuit in ways that are sometimes completely non-intuitive for a human designer.

Zhang [1997] proposes a GA application to attribute selection and weighting (ASW), this being a key problem of case retrieval in CBR systems. With the use of the GA in ASW, the performance of the CBR could be improved.

Thomson [1999] explores the potential for evolving larger systems more quickly via a method of visualizing the sub-components of the final solution when they appear. Taking these partially evolved solutions from short runs and feeding these to a further evolution, originates that we have an approach consisting of a number of much shorter runs with evolution being assisted by sub-components that have already been identified, instead of long random evolutionary runs.

Miller [1999] examines the idea of an evolutionary algorithm being an inspirational or discovery engine, applying this idea to the context of designing electronic circuits. He argues that by connecting pieces of logic together and testing them to see if they carry out the desired function it may be possible to discover new principles of design and new algebraic techniques. He demonstrates that by evolving a hierarchical series of examples, it becomes possible to rediscover the well-known ripple-carry principle for building adder circuits of any size. Finally he shows that the genetic algorithm is able to produce fully working circuits that lie outside conventional algebra.

Louis and Johnson [1997] have proposed seeding a genetic algorithm's initial population with solutions to similar, previously solved problems and the genetic algorithm then adapts its seeded population to solving the current problem. They applied their program to the field of combinational circuit design and indicated the possibility of applying their approach to other fields. The results obtained indicate that seeding a small percentage of the population with “appropriate” cases improves performance on similar problems and that the combined system usually takes less time to provide a solution to a new problem as it gains experience (memory) from solving other similar problems.
Bibliography.

Combinational Logic Circuits Optimization


Evolutionary Algorithms


Case Based Reasoning:


45


Previous Works in Logic Circuits Optimization Using EAs and/or CBR


3 Proposed System

Based on the works mentioned in the previous chapter, we proposed the use of previous solutions in the optimization of combinational logic circuits and to place these individuals in the initial population of the GA. With this approach, we expect to obtain a reduction in the convergence time and we expect to find better solutions in comparison to those obtained from applying the GA with an initial population that is randomly generated only. With the combination of the GA and the CBR we expect to obtain a robust learning system. The proposed system is depicted in Figure 3.1.

![Figure 3.1 Proposed system to optimize combinational logic circuits using GAs and CBR.](image-url)
In the following paragraphs each module of the system is briefly described.

a) **New Cases Module.**
The system receives a new circuit in this module, and such a circuit is in a similar format to the stored cases in the case base. After receiving the new circuit, the information is sent to the weight assignment and attributes selection module.

b) **Weight assignment and attributes selection module.**
In this module the attributes selection is carried out using GAs in order to reduce the dimensionality of the attributes in the cases and to increase the accuracy in the case retrieval. With the use of GAs, higher weights are assigned to more relevant attributes.

c) **Genetic Algorithm 1.**
This module is used to obtain the information that helps to select and assign weights to the attributes. The application of the GA is justified due to the fact that it is very hard for a human expert to determine which are the attributes that should be used and conveniently assign the related weights to these attributes.

d) **Case Base.**
In this module the cases will be stored in two different ways:

- **Knowledge extraction 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 for the GA in a new problem.

- **Knowledge extraction during the evolutionary process.** The best individuals will be recognized during the evolutionary process in the early generations. Afterwards they will be stored as cases in the case base and retrieved in later generations.

e) **Generating the initial population.**
This module generates the initial population of the GA. The selection of individuals from the case base is carried out mixing the CBR and the GA. The module selects the individuals from the case base taking into account the fitness value of each individual and other attributes in order to evaluate the distance measure between the new case and each individual in the case base.

The number of individuals inserted from the case base should be balanced with the number of individuals randomly generated, in order to balance the exploration and the exploitation of the search space.

f) **Genetic Algorithm 2.**
This is the module where a solution to the input circuit using a GA was found. The GA takes individuals from the case base in two different stages:

- **At the beginning.** The GA takes the best individuals from the case base to seed the initial population.
During the evolution. The GA takes the best individuals found in early generations and seeds with these individuals later generations.

In the next generations the GA selects the best individuals from the previous population based only on the individual’s fitness value. To evaluate the fitness value of an individual the GA only takes into account the ability of this individual to solve a certain task.

g) Case validation.
The solution obtained using the GA represents the circuit that could be the potential solution. It is necessary to do an evaluation of the feasibility of this circuit before giving this solution to the user.

h) Case Storing.
Once the optimal design has been obtained and the feasibility of the circuit has been evaluated the solution is given to the user. Subsequently, if the solution is not in the case base, an index is assigned and the solution is stored in the case base. The new case is formed from the solution and the original circuit.

i) Obtained circuit.
This module gives the user the optimal circuit in the established format; it could be a binary or integer string, a symbolic representation of the gates or the combinational circuit diagram.

j) Explanation module.
This module gives the user an explanation regarding the way in which the GA arrives at the solution and gives the reason why the solution is considered a valid answer. The system explains where the answer comes from and which are the weakest and strongest building blocks. With this information the user corrects the system and does a post-analysis of the useful information, without losing any valuable information that the genetic operators could possibly destroy.
Bibliography.


4 Problem Description

4.1 The Knowledge that will be extracted from the evolutionary process of the circuit

The knowledge that will be extracted from the optimization process of a logic circuit, using evolutionary algorithms and stored in the case base, can be divided in the following way:

a. **Knowledge extraction at the end of the evolutionary process of the logic circuit.**

a.1 The knowledge that will be extracted are the Boolean Laws and the DeMorgan’s Theorem (these will be the cases), obtained after comparing the results obtained from two or more runs of the genetic algorithm with different parameters and with the solution obtained by a human expert.

a.2 The knowledge that will be extracted are “New Laws” (these will be the cases), obtained after comparing the results obtained from two or more runs of the genetic algorithm with different parameters and with the solution obtained by a human expert.

b. **Knowledge extraction during the evolutionary process of the logic circuit.**

The knowledge that will be obtained during the evolutionary process is that of the building blocks that the circuit structurally maintains during its evolutionary process. When some individuals arrive at a pre-defined 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 extract the knowledge and store it in a case base and seed later generations with these individuals.

In the following sections we show some examples, which are used to describe how the knowledge is extracted at the end and during the evolutionary process of optimizing a combinational logic circuit.
4.2 *Small Scale Integrated Circuit (SSI), Circuit A [Coello 2000]*.

### Knowledge extraction at the end of the evolutionary process of the logic circuit.

#### Truth Table

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Configuration data for the evolutionary process

<table>
<thead>
<tr>
<th>Representation used</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>3</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>1</td>
</tr>
<tr>
<td>Number of rows</td>
<td>5</td>
</tr>
<tr>
<td>Number of columns</td>
<td>5</td>
</tr>
<tr>
<td>Chromosome Length</td>
<td>225</td>
</tr>
<tr>
<td>Cardinality</td>
<td>5</td>
</tr>
</tbody>
</table>

#### Comparison of results

<table>
<thead>
<tr>
<th>Human Designer</th>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Population = 700</td>
<td>Population = 1000</td>
</tr>
<tr>
<td></td>
<td>Generations = 200</td>
<td>Generations = 400</td>
</tr>
<tr>
<td></td>
<td>Crossover probability = 0.5</td>
<td>Crossover probability = 0.5</td>
</tr>
<tr>
<td></td>
<td>Mutation probability = 0.007</td>
<td>Mutation probability = 0.007</td>
</tr>
<tr>
<td></td>
<td>Reed = 0.85</td>
<td>Reed = 0.85</td>
</tr>
<tr>
<td>Convergence Values</td>
<td>Generation = 18</td>
<td>Generation = 17</td>
</tr>
<tr>
<td></td>
<td>Fitness = 18</td>
<td>Fitness = 29</td>
</tr>
<tr>
<td>Equation</td>
<td>X = C(A ⊕ B) + B(A ⊕ C)</td>
<td>X = (A ⊕ B ⊕ C) + (AC ⊕ BC)</td>
</tr>
<tr>
<td>Gates Number</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Gates Count</td>
<td>2 AND, 1 OR, 2 XOR</td>
<td>3 AND, 1 OR, 2 XOR</td>
</tr>
</tbody>
</table>

#### Convergence graphs

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="SSI, Circuit A" /></td>
<td><img src="image2" alt="SSI, Circuit A" /></td>
</tr>
</tbody>
</table>
Diagrams obtained

<table>
<thead>
<tr>
<th>Human Designer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Analysis

**Analysis 1**
If we take the result obtained from the first run of the GA and we factorize the first element by B and the second element by C, we have:

\[ X = (AB \oplus BC) + (AC \oplus BC) = (B(A \oplus C)) + (C(A \oplus B)) = B(A \oplus C) + C(A \oplus B) \]  

We have the same result obtained by a human designer applying Karnaugh maps, so we can store this equality in the case base and we will have a reduction in the number of gates.

1. In order to obtain the equality we need to “rediscover” the distributive law mentioned in section 2.1 and apply this law to find the equality between the first run of the GA and the solution found by the human designer.

**Analysis 2**
If we take the solution 1 obtained from the GA and do the following:

\[ X = (AB \oplus BC) + (AC \oplus BC) = (AB \oplus BBC) + (AC \oplus BCC) \]

And when we take the term common in each operand, we have:
\[ X = (AB \oplus BC) + (AC \oplus BCC) = B(A \oplus BC) + C(A \oplus BC) \]

Finally we factorize the term \((A \oplus BC)\), and we have:
\[ X = B(A \oplus BC) + C(A \oplus BC) = (A \oplus BC)(B + C) \]

We have the same result obtained from the second run of the GA, which is a better solution, so we can store this equality in the case base and we will have a reduction in the number of gates.

1. Rediscovering the idempotent theorem mentioned in section 2.1
2. Rediscovering the distributive law mentioned in section 2.1
3. Rediscovering the distributive law mentioned in section 2.1

**Cases for storing in the case base**

<table>
<thead>
<tr>
<th>Original case</th>
<th>Solution</th>
<th>Description</th>
<th>Number of gates eliminated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1 (A'BC + AB' + BCB')</td>
<td>((C \oplus AB)(A + B))</td>
<td>Input data from the truth table and best solution found by the GA</td>
<td>(13 - 4 = 9)</td>
</tr>
<tr>
<td>Case 2 ((AB \oplus BC) + (AC \oplus BC))</td>
<td>(C(A \oplus B) + B(A \oplus C))</td>
<td>Comparison between the first run of the GA and the human designer</td>
<td>(6 - 5 = 1)</td>
</tr>
<tr>
<td>Case 3 ((AB \oplus BC) + (AC \oplus BC))</td>
<td>((C \oplus AB)(A + B))</td>
<td>Comparison between the two runs of the GA</td>
<td>(6 - 4 = 2)</td>
</tr>
</tbody>
</table>

### 4.2.2 Knowledge extraction during the evolutionary process of the logic circuit.

**Figure 4.1. Circuit obtained in generation 50.**

**Figure 4.2. Circuit obtained in generation 98.**

**Figure 4.3. Circuit obtained in generation 174.**
In the diagram depicted in Figure 4.1, we can see that in generation 50 the circuit has a fitness value of 13 and we can recognize the building blocks in the shadowed cells. From Figure 4.2, we can see that in generation 98 the circuit has increased its fitness value to reach 21, and we can observe that, despite the fact that the building blocks are in a different position, they still remain in the circuit (gates in the shadowed cells). Finally we have the last result obtained in generation 174, which has a fitness value of 29. We can see that the circuit has the same gates (building blocks) in this final result.
4.3 **SSI, Circuit B [Lam 1988].**

### 4.3.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.

**Truth Table**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Configuration data for the evolutionary process**

<table>
<thead>
<tr>
<th>Representation used</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>3</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>1</td>
</tr>
<tr>
<td>Number of rows</td>
<td>5</td>
</tr>
<tr>
<td>Number of columns</td>
<td>5</td>
</tr>
<tr>
<td>Chromosome Length</td>
<td>225</td>
</tr>
<tr>
<td>Cardinality</td>
<td>5</td>
</tr>
</tbody>
</table>

**Comparison of results**

<table>
<thead>
<tr>
<th>Human Designer</th>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Population = 500  Generation = 100  Crossover probability = 0.5  Mutation probability = 0.008  Rseed = 0.85</td>
<td>Population = 1500  Generation = 100  Crossover probability = 0.5  Mutation probability = 0.008  Rseed = 0.85</td>
</tr>
<tr>
<td>Convergence Values</td>
<td>Generation = 99  Fitness = 24</td>
<td>Generation = 94  Fitness = 29</td>
</tr>
<tr>
<td>Equation</td>
<td>X = AB + BC + ABC'</td>
<td>X = ((A+C) ⊕ B)'</td>
</tr>
<tr>
<td>Gates Number</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Gates Count</td>
<td>4 AND, 2 OR, 3 NOT</td>
<td>1 OR, 1 XOR, 2 NOT</td>
</tr>
</tbody>
</table>

**Convergence graphs**

**GA Run 1**

![Fitness vs. Generation for SSI, Circuit B](image1)

**GA Run 2**

![Fitness vs. Generation for SSI, Circuit B](image2)
Diagrams obtained

<table>
<thead>
<tr>
<th>Human Designer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Analysis

**Analysis 1**
If we rewrite the expression found by the human designer:

\[ X = AB' + B'C + A'BC' = B'(A + C) + B(A'C') \] ①

And if we apply DeMorgan’s theorem to the last operand, we have:

\[ X = B'(A + C) + B(A'C') = B'(A + C) + B(A + C)' \] ②

Then we can obtain the solution found by the second run of the GA:

\[ X = B'(A + C) + B(A' + C) = B \oplus (A + C) \] ③

We have the same result obtained by a human designer and the second run of the GA, which is a better solution, so we can store this equality in the case base and we will have a reduction in the number of gates.

① Rediscovering the distributive law
② Rediscovering the DeMorgan’s theorem
③ Rediscovering the operation of an XOR gate
Analysis 2
If we consider an equality for the solutions found by the two runs of the GA, we have:

\[ X = ((A + C) \oplus B') = (A + C) \oplus B \]

Then if we introduce the negation in the first term of the equality we would have:

\[ X = (((A + C) \oplus B')') = ((A + C)') \oplus B' \]

In order to maintain the equality it is necessary that:

\[ X = ((A + C)' \oplus B) = (A + C) \oplus B \]

If we apply the negation to the first term in \( \oplus \)

\[ X = (((A + C)' \oplus B)') = ((A + C)) \oplus B = (A + C) \oplus B \]

Then the equality is maintained and we can say that it would be the DeMorgan’s theorem applied to the XOR gates. This “new law” means that if we apply DeMorgan’s theorem to a XOR gate then we only need to negate one of the components in the original expression.

We found a “new” DeMorgan’s theorem applied to the XOR gates and we can store this “new” theorem in the case base.

<table>
<thead>
<tr>
<th>Case</th>
<th>Original case</th>
<th>Solution</th>
<th>Description</th>
<th>Number of eliminated gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>A'B'C'+A'B'C'+AB'C+AB'C</td>
<td>(A + C) \oplus B</td>
<td>Input data from the truth table and the best solution found by the GA</td>
<td>14 - 2 = 12</td>
</tr>
<tr>
<td>Case 2</td>
<td>AB'+B'C'+A'BC</td>
<td>(A + C) \oplus B</td>
<td>Comparison between a human designer and the best solution found by the GA</td>
<td>9 - 2 = 7</td>
</tr>
<tr>
<td>Case 3</td>
<td>(X \oplus Y)'</td>
<td>X \oplus Y</td>
<td>DeMorgan’s theorem applied to XOR gates obtained with the comparison between the two solutions found by the GA</td>
<td>3 - 1 = 2</td>
</tr>
</tbody>
</table>

4.3.2 Knowledge extraction during the evolutionary process of the logic circuit.

![Figure 4.4. Circuit obtained in generation 1.](image1)

![Figure 4.5. Circuit obtained in generation 73.](image2)
In the diagram depicted in Figure 4.4, we can see that in the first generation the circuit has a fitness value of 12 and in spite of being at an early stage, we can recognize the building blocks in the cells that are shadowed. From Figure 4.5, we can see that in generation 73 the circuit has increased its fitness value reaching 16, and we can observe that, despite the fact that the building blocks are in a different position, they still remain in the circuit (gates in the shadowed cells). Finally, we have the last result obtained in generation 94 and we have a fitness value of 23. We can see that the circuit has the same gates (building blocks) in this final result.
4.4 Medium Scale Integrated Circuit (MSI), subtraction circuit [Morris 1982].

4.4.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.

Truth Table

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Configuration data for the evolutionary process

<table>
<thead>
<tr>
<th>Representation used</th>
<th>Binary</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Number of rows</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Number of columns</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Chromosome Length</td>
<td>225</td>
<td>75</td>
</tr>
<tr>
<td>Cardinality</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Comparison of results

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Human Expert</th>
<th>AG Run 1 (Binary)</th>
<th>AG Run 2 (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Population = 200</td>
<td>Generation = 3675</td>
<td>Generation = 35</td>
</tr>
<tr>
<td></td>
<td>Generations ~2000</td>
<td>Fitness ~28</td>
<td>Fitness ~24</td>
</tr>
<tr>
<td>Crossover probability = 0.6</td>
<td>X = (B(C + ( ((A@B) ⊕ B') (A@BC) ) + (A@BC)) ) + (BC)</td>
<td>Y = (A@B) ⊕ C</td>
<td></td>
</tr>
<tr>
<td>Mutation probability = 0.001</td>
<td>Rseed = 0.85</td>
<td>Rseed = 0.85</td>
<td></td>
</tr>
<tr>
<td>Convergence Values</td>
<td>X = AB + AC + BC</td>
<td>Y = A@BC + A@BC + A@BC + ABC</td>
<td></td>
</tr>
<tr>
<td>Equation</td>
<td>8 AND, 5 OR, 3 NOT</td>
<td>3 AND, 1 OR, 2 XOR, 5 NOT</td>
<td>2 AND, 3 OR, 1 NOT, 2 XOR</td>
</tr>
<tr>
<td>Gates Number</td>
<td>16</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Gates Count</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>AG Run 3 (Float)</th>
<th>AG Run 4 (Float)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Population = 100</td>
<td>Population = 700</td>
</tr>
<tr>
<td></td>
<td>Generations ~2000</td>
<td>Generations ~400</td>
</tr>
<tr>
<td>Crossover probability = 0.5</td>
<td>X = (B ⊕ (A ⊕ C)) ⊕ (A ⊕ AC)</td>
<td>X = (B ⊕ (A ⊕ C) + BC</td>
</tr>
<tr>
<td>Mutation probability = 0.006</td>
<td>Y = (B ⊕ (A ⊕ C) ⊕ (B (A ⊕ C)</td>
<td>Y = B ⊕ (A@C)</td>
</tr>
<tr>
<td>Convergence Values</td>
<td>Generation = 1170</td>
<td>Generation = 398</td>
</tr>
<tr>
<td>Fitness ~34</td>
<td>Fitness ~35</td>
<td></td>
</tr>
<tr>
<td>Equation</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Gates Number</td>
<td>2 AND, 1 OR, 4 XOR</td>
<td>2 AND, 1 OR, 4 XOR</td>
</tr>
<tr>
<td>Gates Count</td>
<td>2 AND, 1 OR, 1 NOT, 2 XOR</td>
<td></td>
</tr>
</tbody>
</table>
Convergence graphs

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Convergence Graph" /></td>
<td><img src="image2.png" alt="Convergence Graph" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GA Run 3</th>
<th>GA Run 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Convergence Graph" /></td>
<td><img src="image4.png" alt="Convergence Graph" /></td>
</tr>
</tbody>
</table>

Diagrams obtained

![Human Designer Diagram](image5.png)
Analysis

Analysis 1
If we take the solution found by the human designer and we factorize C and C’ in Y, we have that:

\[ Y = A'B'C + A'BC + AB'C + ABC = C(A'B + AB') + C'(A'B + AB') \]

To transform this equation in terms of an XOR gate, it would be necessary that:

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:

\[ T' = (A'B + AB)' = (A'B)'(AB)' = (A + B)(A' + B') \]

Applying the distributive law and some basic theorems

\[ T' = (A + B)(A' + B') = AA' + AB + A'B + BB' = A'B + AB' \]

Verifying that \( T' = A'B + AB' \) is the negation of \( T = A'B + AB \). Then we can rewrite the equation \( \text{\( 1 \)} \) as follows:
Y = C(AB' + AB) + C'(A'B + AB') = C ⊕ (A'B + AB')

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

Y = C ⊕ (A'B + AB') = C ⊕ (A ⊕ B)

Applying the commutative law, we have

Y = C ⊕ (A ⊕ B) = (A ⊕ B) ⊕ C

We have the same result obtained from the solution found by the second run 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.

1. Rediscovering the distributive law
2. Rediscovering the DeMorgan’s theorem
3. Rediscovering the distributive law and some basic theorems
4. Rediscovering the operation of a XOR gate
5. Rediscovering the operation of a XOR gate
6. Rediscovering the commutative law

### Cases for storing in the case base

<table>
<thead>
<tr>
<th>Original case</th>
<th>Solution</th>
<th>Description</th>
<th>Number of gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>(A'B'C + A'B'C + AB'C + ABC)</td>
<td>Comparison between a human designer and the best solution found by the GA</td>
<td>13 - 2 = 11</td>
</tr>
<tr>
<td>Case 2</td>
<td>(A'B' + AB')</td>
<td>Case obtained with the comparison between a human designer and the best solution found by the GA</td>
<td>6 - 5 = 1</td>
</tr>
</tbody>
</table>

#### 4.4.2 Knowledge extraction during the evolutionary process of the logic circuit.

**Figure 4.7. Circuit obtained in generation 325.**

**Figure 4.8. Circuit obtained in generation 366.**
In the diagram depicted in Figure 4.7, we can see that the circuit has a fitness value of 23 in generation 325 and we can recognize the building blocks in the cells that are shadowed. From Figure 4.8, we can see that in generation 366 the circuit has increased its fitness value reaching 29, and we can observe that the building blocks remain in the same position in the circuit (gates in the shadowed cells). Finally, we have the last result obtained in generation 398 and we have a fitness value of 35. We can see that the circuit has the same gates (building blocks) in the same position as in earlier stages of the evolutionary process.
4.5 MSI, Parity bit checking [Morris 1982].

4.5.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.

Truth Table

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Configuration data for the evolutionary process

<table>
<thead>
<tr>
<th>Representation used</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>4</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>1</td>
</tr>
<tr>
<td>Number of rows</td>
<td>5</td>
</tr>
<tr>
<td>Number of columns</td>
<td>5</td>
</tr>
<tr>
<td>Chromosome Length</td>
<td>225</td>
</tr>
<tr>
<td>Cardinality</td>
<td>5</td>
</tr>
</tbody>
</table>

Comparison of results

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Human Expert</th>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population</td>
<td>600</td>
<td>3000</td>
<td>120</td>
</tr>
<tr>
<td>Generations</td>
<td>200</td>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.6</td>
<td>0.5</td>
<td>0.007</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>0.001</td>
<td>0.007</td>
<td>0.85</td>
</tr>
<tr>
<td>Population</td>
<td>35</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>Convergence</td>
<td>Generation = 117</td>
<td>Generation = 101</td>
<td></td>
</tr>
<tr>
<td>Fitness</td>
<td>35</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>Equation</td>
<td>X = ((A ⊕ B) ⊕ (C ⊕ D))'</td>
<td>X = ((A ⊕ C) ⊕ B) ⊕ ((B ⊕ B) ⊕ D)'</td>
<td>X = ((A ⊕ B) ⊕ (C ⊕ D))'</td>
</tr>
<tr>
<td>Gates Number</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Gates Count</td>
<td>3 NOT, 3 XOR</td>
<td>1 AND, 1 OR, 3 XOR, 4 NOT</td>
<td>1 NOT, 3 XOR</td>
</tr>
</tbody>
</table>
Convergence graphs

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Graph 1" /></td>
<td><img src="image" alt="Graph 2" /></td>
</tr>
</tbody>
</table>

Diagrams obtained

Human Designer

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram 1" /></td>
<td><img src="image" alt="Diagram 2" /></td>
</tr>
</tbody>
</table>
Analysis

Analysis 1

If we take the solution obtained by the human designer and apply the “new” DeMorgan’s theorem for XOR gates found in a previous example, we have:

\[
X = ((A \oplus B) \oplus (C \oplus D))' = ((A \oplus B) \oplus (C \oplus D))
\]

1

If we again apply the DeMorgan’s theorem for XOR gates to equation 1:

\[
X = ((A \oplus B) \oplus (C \oplus D)) = ((A \oplus B) \oplus (C \oplus D))'
\]

2

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

In general, we can establish the following equality and store it in the case base:

\[
(S' \oplus T')' = (S \oplus T)
\]

3

1. Rediscovering the DeMorgan’s theorem for XOR gates
2. Rediscovering the DeMorgan’s theorem for XOR gates
3. Applying two times the DeMorgan’s theorem for XOR gates

Cases for storing in the case base

<table>
<thead>
<tr>
<th>Original case</th>
<th>Solution</th>
<th>Description</th>
<th>Number of gates eliminated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>(((A \oplus B) \oplus (C \oplus D))')</td>
<td>Comparison between the second run of the GA and the human designer.</td>
<td>6 - 4 = 2</td>
</tr>
<tr>
<td>Case 2</td>
<td>(S' \oplus T)'</td>
<td>(S \oplus T)'</td>
<td>DeMorgan’s theorem applied to XOR gates obtained from the comparison between the solution found by the second run of the GA and the solution obtained by a human designer.</td>
</tr>
</tbody>
</table>

4.5.2 Knowledge extraction during the evolutionary process of the logic circuit.

Figure 4.10. Circuit obtained in generation 9.  
Figure 4.11. Circuit obtained in generation 67.
In the diagram depicted in Figure 4.10, we can see that the circuit has a fitness value of 17 in generation 9 and we can recognize the building blocks in the cells that are shadowed. From Figure 4.11, we can see that in generation 67 the circuit has increased its fitness value reaching 27, and we can observe the building blocks in a different position in the matrix (gates in the shadowed cells). Finally, we have the last result obtained in generation 101 and we have a fitness value of 37. We can see that the circuit has the same gates (building blocks) in different positions but with the same behavior as in the circuits in earlier stages.
4.6 MSI, 2-bit magnitude comparator circuit [Lam 1988]

4.6.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.

Truth Table

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Configuration data for the evolutionary process

<table>
<thead>
<tr>
<th>Representation used</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>4</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>3</td>
</tr>
<tr>
<td>Number of rows</td>
<td>6</td>
</tr>
<tr>
<td>Number of columns</td>
<td>7</td>
</tr>
<tr>
<td>Chromosome Length</td>
<td>126</td>
</tr>
<tr>
<td>Cardinality</td>
<td>6</td>
</tr>
</tbody>
</table>

Comparison of results

<table>
<thead>
<tr>
<th>Human Designer</th>
<th>GA Run 2</th>
<th>GA Run 1</th>
</tr>
</thead>
</table>
| Parameters     | Population = 600  
Generations = 2500  
Crossover probability = 0.5  
Mutation probability = 0.003  
Rseed = 0.249 | Population = 490  
Generations = 2000  
Crossover probability = 0.5  
Mutation probability = 0.003  
Rseed = 0.139 |
| Convergence Values | Generation = 1757  
Fitness = 78 | Generation = 1586  
Fitness = 79 |
| Equation        | X=AC’+BC’D+ABD’  
Y=’A’B’C’D+A’BCD+ABCD+AB’C’D+AB’CD  
Z=’A’C+A’B’D+B’CD  
X=’a’+a’  
Y=’a’  
Z=’a’ | X=’a’  
Y=’a’  
Z=’a’ |
| Where:          | a=(A ⊕ C) + (B ⊕ D)  
α=(A ⊕ C) + (A ⊕ B) ⊕ (A ⊕ C) + A’ | α=(D ⊕ (B ⊕ D’)) + (A ⊕ B) + (A’ ⊕ B) |
| Gates Number    | 21        | 12       |
| Gates Count     | 11 AND, 4 OR, 6 NOT | 1 AND, 3 OR, 5 XOR, 3 NOT | 2 AND, 3 OR, 3 XOR, 3 NOT |
Convergence graphs

![Convergence graph for GA Run 1](image1)

![Convergence graph for GA Run 2](image2)

Diagrams obtained

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3" alt="Diagram for GA Run 1" /></td>
<td><img src="image4" alt="Diagram for GA Run 2" /></td>
</tr>
</tbody>
</table>

Analysis

Analysis 1

For this circuit it was very difficult to find the relationship between the solutions found by the first and second run of the GA and the solution found by a human designer. However, we can see from the values in $\theta$ and $\alpha$ that the circuit uses the same set of gates many times, so this is the main reason that the solutions obtained in both runs of the GA considerably reduce the number of gates.

Cases for storing in the case base

<table>
<thead>
<tr>
<th>Case</th>
<th>Original case</th>
<th>Solution</th>
<th>Description</th>
<th>Number of gates eliminated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>X=AC+BC'D'+ABD' Y=A'B'C'D'+A'BC'D+ ABCD+AB'CD Z=A'C'+A'B'D+B'CD</td>
<td>X = 0' + $\alpha'$ Y = 0' Z = 0' $\alpha'$</td>
<td>Input data from the truth table and best solution found by the GA</td>
<td>21-12 = 9</td>
</tr>
</tbody>
</table>
4.6.2 Knowledge extraction during the evolutionary process of the logic circuit.

![Diagram of a circuit](image1)

**Figure 4.13. Circuit obtained in generation 663.**

![Diagram of a circuit](image2)

**Figure 4.14. Circuit obtained in generation 840.**

![Diagram of a circuit](image3)

**Figure 4.15. Circuit obtained in generation 1586.**

In the diagram depicted in Figure 4.13, we can see that the circuit has a fitness value of 57 in generation 663 and we can recognize the building blocks in the cells that are shadowed. From Figure 4.14, we can see that in generation 840 the circuit has increased its fitness value to reach 74, and we can observe the building blocks in the same position in the matrix (gates in the shadowed cells). Finally, we have the last result obtained in generation 1586 and we have a fitness value of 79. We can see that the circuit has the same gates (building blocks) in the same positions as in the circuits in earlier stages.
4.7  MSI, half-adder [Lam 1988].

4.7.1  Knowledge extraction at the end of the evolutionary process of the logic circuit.

Truth Table

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Configuration data for the evolutionary process

<table>
<thead>
<tr>
<th>Representation used</th>
<th>Binary</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Number of rows</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Number of columns</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Chromosome Length</td>
<td>225</td>
<td>75</td>
</tr>
<tr>
<td>Cardinality</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Comparison of results

<table>
<thead>
<tr>
<th>Human Designer</th>
<th>GA Run 1 (Binary)</th>
<th>GA Run 2 (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Population = 800</td>
<td>Population = 5000</td>
</tr>
<tr>
<td></td>
<td>Generations = 200</td>
<td>Generations = 100</td>
</tr>
<tr>
<td></td>
<td>Crossover probability = 0.6</td>
<td>Crossover probability = 0.25</td>
</tr>
<tr>
<td></td>
<td>Mutation probability = 0.001</td>
<td>Mutation probability = 0.001</td>
</tr>
<tr>
<td></td>
<td>Ratio = 0.85</td>
<td>Ratio = 0.85</td>
</tr>
<tr>
<td>Convergence Values</td>
<td>Generation = 90</td>
<td>Generation = 96</td>
</tr>
<tr>
<td></td>
<td>Fitness = 29</td>
<td>Fitness = 31</td>
</tr>
<tr>
<td>Equation</td>
<td>X = A ⊕ B</td>
<td>X = A ⊕ B</td>
</tr>
<tr>
<td></td>
<td>Y = AB</td>
<td>Y = AB</td>
</tr>
<tr>
<td>Gates Number</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Gates Count</td>
<td>1 AND, 1 XOR</td>
<td>1 OR, 2 XOR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GA Run 3 (Float)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Convergence Values</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Equation</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Gates Number</td>
</tr>
<tr>
<td>Gates Count</td>
</tr>
</tbody>
</table>
Convergence graphs

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Graph 1" /></td>
<td><img src="image2.png" alt="Graph 2" /></td>
</tr>
</tbody>
</table>
| ![Graph 3](image3.png) | GA Run 3 |}

Diagrams obtained

<table>
<thead>
<tr>
<th>Human Designer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Analysis

Analysis 1

For this circuit it was very difficult to find the relationship between the solution found by the first run of the GA and the solutions found by the human designer and the second and third run of the GA. Although we tried with many of the laws and theorems described in section 2.1 it was not possible to find the equality. However, we can see in the solution found by the first run of the GA that the only way for us to have a true value for the variable Y is when the values A and B are both 1. For any other value for A and B, the result for Y is zero. So we have the same truth table that we have in an AND gate where the only way for us to have a 1 in the output is when all of the values are 1. In any other case the output is 0.

Then we can establish the equality between the best solution found by the GA and and the solution found by the first run of the GA:

\[ Y = (A + (A \oplus B)) \oplus (A \oplus B) = AB \]

Discovering a “new” relation between the AND, XOR and OR gates
Cases for storing in the case base

<table>
<thead>
<tr>
<th>Original case</th>
<th>Solution</th>
<th>Description</th>
<th>Number eliminated of gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>X = A'B + AB'</td>
<td>Input data from the truth table and best solution found by the GA</td>
<td>5 - 1 = 4</td>
</tr>
<tr>
<td>Case 2</td>
<td>Y = (A + (A ⊕ B)) ⊕ (A ⊕ B)</td>
<td>Comparison between the first and second run of the GA</td>
<td>3 - 2 = 1</td>
</tr>
</tbody>
</table>

4.7.2 Knowledge extraction during the evolutionary process of the logic circuit.

![Figure 4.16. Circuit obtained in generation 0.](image)
![Figure 4.17. Circuit obtained in generation 21.](image)
![Figure 4.18. Circuit obtained in generation 93.](image)

In the diagram depicted in Figure 4.16, we can see that the circuit has a fitness value of 12 although the circuit was found in generation 0. This is because it is very easy to find a circuit that satisfies this very simple truth table. However, we can recognize only one building block in the cell that is shadowed. From Figure 4.17, we can see that in generation 21 the circuit has increased its fitness value to reach 23. In this circuit, we can observe the building blocks that help us to arrive to a solution in the matrix (gates in the shadowed cells). Finally we have the last result obtained in generation 93 and we have a fitness value of 31. We can see that the circuit has the same gates (building blocks) in the same positions that the circuit found in generation 21.
4.8 MSI, full-adder [Lam 1988].

4.8.1 Knowledge extraction at the end of the evolutionary process of the logic circuit.

Truth Table

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>X (Sum)</th>
<th>Y (Carrier)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Configuration data for the evolutionary process

<table>
<thead>
<tr>
<th>Representation used</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>3</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>2</td>
</tr>
<tr>
<td>Number of rows</td>
<td>5</td>
</tr>
<tr>
<td>Number of columns</td>
<td>5</td>
</tr>
<tr>
<td>Chromosome Length</td>
<td>75</td>
</tr>
<tr>
<td>Cardinality</td>
<td>5</td>
</tr>
</tbody>
</table>

Comparison of results

<table>
<thead>
<tr>
<th>Human Designer</th>
<th>GA Run 1</th>
<th>GA Run 2</th>
<th>GA Run 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generations = 100</td>
<td>Fitness = 29</td>
<td>Generations = 300</td>
</tr>
<tr>
<td></td>
<td>Crossover probability = 0.5</td>
<td></td>
<td>Crossover probability = 0.5</td>
</tr>
<tr>
<td></td>
<td>Mutation probability = 0.006</td>
<td></td>
<td>Mutation probability = 0.006</td>
</tr>
<tr>
<td></td>
<td>Rseed = 0.249</td>
<td></td>
<td>Rseed = 0.249</td>
</tr>
<tr>
<td>Convergence</td>
<td>Equation</td>
<td></td>
<td>Equation</td>
</tr>
<tr>
<td>Values</td>
<td>X = A @ B @ C</td>
<td></td>
<td>X = A @ B @ C</td>
</tr>
<tr>
<td></td>
<td>Y = AB + C(A @ B)</td>
<td></td>
<td>Y = AB + B(A @ C)</td>
</tr>
<tr>
<td></td>
<td>X = (B' @ (A @ C))'</td>
<td></td>
<td>Y = ((A @ C) @ ((A @ C) + (A @ B'))) @ (B' @ (B @ C))</td>
</tr>
<tr>
<td>Gates Number</td>
<td>5</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Gates Count</td>
<td>2 AND, 1 OR, 2 XOR</td>
<td>1 OR, 5 XOR, 2 NOT</td>
<td>2 AND, 1 OR, 2 XOR</td>
</tr>
</tbody>
</table>

GA Run 3

| Parameters     | Population = 700 | Generation = 124 |
|                | Generations = 250 | Fitness = 36 |
|                | Crossover probability = 0.5 | |
|                | Mutation probability = 0.006 | |
|                | Rseed = 0.049 | |
| Convergence     | Equation | |
| Values          | X = A @ B @ C | |
|                | Y = ((A @ B) @ C) @ ((A @ (A @ B) @ C)) + (A @ B) | |
| Gates Number    | 5         |
| Gates Count     | 1 OR, 4 XOR |
Convergence graphs

<table>
<thead>
<tr>
<th>GA Run 1</th>
<th>GA Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSI, Full-Adder</td>
<td>MSI, Full-Adder</td>
</tr>
<tr>
<td><img src="image1" alt="Graph 1" /></td>
<td><img src="image2" alt="Graph 2" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GA Run 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSI, Full-Adder</td>
</tr>
<tr>
<td><img src="image3" alt="Graph 3" /></td>
</tr>
</tbody>
</table>

Diagrams obtained

<table>
<thead>
<tr>
<th>Human Designer</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Analysis

Analysis 1
If we take the first run of the GA and we apply the DeMorgan’s theorem for an XOR gate in the term X, we have:

\[ X = (B' \oplus (A \oplus C))' = B \oplus (A \oplus C) \]  

If we rearrange the variables applying the commutative law and associative law, we have:

\[ X = B \oplus (A \oplus C) = A \oplus C \oplus B \]  

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

1. Rediscovering the DeMorgan’s theorem for XOR gates
2. Rediscovering the commutative and associative laws
Cases for storing in the case base

<table>
<thead>
<tr>
<th>Original case</th>
<th>Solution</th>
<th>Description</th>
<th>Number of gates eliminated</th>
</tr>
</thead>
</table>
| Case 1        | X = A'B'C' + A'BC' + ABC' + ABC  
Y = A'BC + AB'C + ABC' + ABC | X = A ⊕ C ⊕ B  
Y = AB + B(A ⊕ C) | Input data from the truth table and best solution found by the GA (equal to the solution found by a human designer) | 19 - 5 = 14 |
| Case 2        | X = (B' ⊕ (A ⊕ C))'  
Y = ((A ⊕ C) ⊕ ((A ⊕ C) + (A ⊕ B'))) ⊕ (B' ⊕ (B ⊕ C)) | X = A ⊕ C ⊕ B  
Y = AB + B(A ⊕ C) | Comparison between the first run and the second run of the GA | 8 - 5 = 3 |

4.8.2 Knowledge extraction during the evolutionary process of the logic circuit.

![Figure 4.19. Circuit obtained in generation 44.](image1)

![Figure 4.20. Circuit obtained in generation 84.](image2)

![Figure 4.21. Circuit obtained in generation 98.](image3)

In the diagram depicted in Figure 4.19, we can see that the circuit has a fitness value of 22. This solution was found in generation 44. We can recognize most of the building blocks that compose the full adder. From Figure 4.20, we can see that in generation 84 the circuit has increased its fitness value to reach 28, in this circuit we can observe all the building blocks that compose the full adder in the cells that are shadowed. Finally, we have the last result obtained in generation 98 where we reached a fitness value of 36. We can see that the circuit has the same gates (building blocks) in the same positions that the circuit found in generation 84.
4.9 How the knowledge will be automatically extracted

Knowledge extraction at the end of the evolutionary process of the logic circuit: Once the solution to a new problem is found by a run of the GA, a new case is formed with the solution and the original problem. The original problem will make 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 retrieve the most similar cases in a more efficient way.

Knowledge extraction during the evolutionary process of the logic circuit [Louis 1993]: The GA records data for each individual in the population as it is created and evaluated. These data include 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 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.

4.10 How the extracted knowledge will be stored in the knowledge base

4.10.1 Representation of combinational logic circuits as strings

![Diagram](image.png)

Figure 4.22 Three different ways to represent the same circuit: (a) The logic circuit graphically represented (using two-input gates), (b) The logic circuit represented by a symbolic matrix, (c) The logic circuit represented as a string of integers.
Figure 4.22 shows an example of the three different representations of a logic circuit that we normally adopt. They range from the use of integers (genotypic representation) to a graphic display of the Boolean function.

In the previous representation, each integer string is divided into triplets that represent each element of the matrix. For example, the first triplet (0 2 0) in Figure 4.22 (c) represents the following:

<table>
<thead>
<tr>
<th>Input 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input 2</td>
<td>2</td>
</tr>
<tr>
<td>Type of gate (AND in this case)</td>
<td>0</td>
</tr>
</tbody>
</table>

The following table maps integers with types of gates:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AND</td>
</tr>
<tr>
<td>1</td>
<td>OR</td>
</tr>
<tr>
<td>2</td>
<td>NOT</td>
</tr>
<tr>
<td>3</td>
<td>WIRE</td>
</tr>
<tr>
<td>4</td>
<td>XOR</td>
</tr>
<tr>
<td>5</td>
<td>NOT</td>
</tr>
<tr>
<td>6</td>
<td>WIRE</td>
</tr>
<tr>
<td>7</td>
<td>XOR</td>
</tr>
</tbody>
</table>

4.10.2 Representation of combinational logic circuits as cases in the case base.
The extracted knowledge represented by cases in the case base will be stored in two different ways:

Knowledge extraction 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 for the GA. The attributes contained in this part of the case base are the following:

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

Some cases stored in the case base are shown in table 4.1:
Knowledge extraction during the evolutionary process. The best individuals are recognized during the evolutionary process in early generations. 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 cases stored in the case base are shown in table 4.2.

<table>
<thead>
<tr>
<th>Case ID</th>
<th>Distance</th>
<th>Schema</th>
<th>Order</th>
<th>Fitness</th>
<th>Weight</th>
<th>Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>710<em>13</em>2a</td>
<td>6</td>
<td>30</td>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td><strong>4<em>50</em>2a</strong></td>
<td>4</td>
<td>60</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>163<em>14</em>41</td>
<td>7</td>
<td>15</td>
<td>4</td>
<td>67</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>350610*7a</td>
<td>7</td>
<td>65</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>214*16169</td>
<td>8</td>
<td>30</td>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td><strong>3<em>10</em>2a</strong></td>
<td>4</td>
<td>60</td>
<td>8</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 4.2 Cases for knowledge extraction during the evolutionary process.
Bibliography


5 Applying CBR and GAs to find the solution for the full adder circuit

The application of CBR and GAs for the full adder circuit took place in several steps:

a) **First step.** 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.

b) **Second step.** Some solutions to different logic circuits including the full adder, the half-adder, the comparator and other circuits were stored in a case base, and retrieved as the most similar cases to seed a portion of the initial population of a GA before running it.

c) **Third step.** 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 to seed a part of the initial population of a GA were retrieved before running it.

5.1 **First step.**

In this step the full adder circuit was solved by generating a percentage (10%) of the initial population with individuals from previous solutions found for the full adder circuit and another percentage (90%) of individuals randomly generated.

In this step the individuals stored in the case base were only previous solutions to the full adder circuit with different fitness values. The individuals stored in the case base did not include the optimal solution to the full adder circuit, because the optimal solution has a fitness value of 36 and the solutions stored in the case base for this step had a maximum fitness value of 22. This was done to show how individuals with a higher fitness in the initial population improve the convergence time of the GA, in comparison with the convergence time using individuals randomly generated.

**Comparison of results**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Human Designer</th>
<th>GA Run 1 (initial population randomly generated)</th>
<th>GA Run 2 (initial population generated with previous solutions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Population = 700</td>
<td>Population = 700</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generations = 300</td>
<td>Generations = 300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Crossover probability = 0.5</td>
<td>Crossover probability = 0.5</td>
</tr>
<tr>
<td>Equation</td>
<td>Gates Number</td>
<td>Gates Count</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>X = A ⊕ B ⊕ C</td>
<td>5</td>
<td>2 AND, 1 OR, 2 XOR</td>
<td></td>
</tr>
<tr>
<td>Y = AB + C(A ⊕ B)</td>
<td>5</td>
<td>2 AND, 1 OR, 2 XOR</td>
<td></td>
</tr>
<tr>
<td>X = A ⊕ C ⊕ B</td>
<td>5</td>
<td>2 AND, 1 OR, 2 XOR</td>
<td></td>
</tr>
<tr>
<td>Y = AB + B(A ⊕ C)</td>
<td>5</td>
<td>2 AND, 1 OR, 2 XOR</td>
<td></td>
</tr>
</tbody>
</table>

All the values obtained for these two runs are shown (in tabular form) in Appendix A.

In the graph depicted in Figure 5.1 we show the evolution for the full adder circuit with an initial population randomly generated.

![Full adder circuit](image)

**Figure 5.1 Full adder circuit with an initial population randomly generated.**

In the graph depicted in Figure 5.2 we show the evolution for the full adder circuit with a percentage of the initial population generated from individuals in the case base. In this first step, the case base had only cases with solutions to the full adder circuit with a maximum fitness value of 22. We can observe in the graph that a fitness value of 22 is the first value for an individual in generation 1; this is because it is the maximum fitness value for individuals stored in the case base and retrieved as the most similar case for the case base reasoning.

In the graph depicted in Figure 5.3 we show the evolution for the full adder circuit comparing both the evolution of the circuit with the initial population randomly generated and the evolution of the circuit with the initial population seeded from individuals in the case base.

Conclusions from step 1:
• As we expected, the circuit arrives at the solution more quickly for this first step due to the fact that the system is retrieving from the case base an individual that previously solved the full adder circuit with a fitness value of 22.

![Full adder circuit](image)

Figure 5.2 Full adder circuit with a percentage of the initial population generated from individuals in the case base.

![Full adder circuit](image)

Figure 5.3 Comparison for the full adder circuit for step 1.

• The individuals taken from the case base cause the GA evolves from a valid circuit, as we can observe in figure 5.3.
• Although both runs have a fitness value of 22 in generation 45; the GA, with an initial population with previous solution, evolves in a different way and arrives at optimal solution in generation 87, while the GA with an initial population randomly generated arrives at optimal solution in generation 98.

• We can conclude for this first step that if the system has to solve a new problem and it has solved the same problem previously, the system uses the previous solution to find the new solution for this new problem more quickly.

5.2 Second step.
In this step the full adder circuit was solved generating a percentage (10%) of the initial population with individuals retrieved from the case base, which are the most similar cases, found for the full adder circuit and another percentage (90%) with individuals generated randomly.

In this step the individuals stored in the case base were previous solutions to different circuits including solutions for the full adder circuit with a maximum fitness value of 22 (it is not the optimal solution), the half-adder circuit, the comparator circuit and other circuits. This was done to show how the system discriminates between the solutions found for the full adder circuit and the solutions found for other circuits. Finally, we retrieve the most similar case to the full adder circuit and discard the half adder and the other circuits. This step improves the convergence time as step 1 does.

Comparison of results

<table>
<thead>
<tr>
<th>Human Designer</th>
<th>GA Run 1 (initial population randomly generated)</th>
<th>GA Run 2 (initial population generated with previous solutions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Population = 700</td>
<td>Population = 700</td>
</tr>
<tr>
<td></td>
<td>Generations = 300</td>
<td>Generations = 300</td>
</tr>
<tr>
<td></td>
<td>Crossover probability = 0.5</td>
<td>Crossover probability = 0.5</td>
</tr>
<tr>
<td></td>
<td>Mutation probability = 0.006</td>
<td>Mutation probability = 0.006</td>
</tr>
<tr>
<td></td>
<td>Reed = 0.249</td>
<td>Reed = 0.249</td>
</tr>
<tr>
<td>Initial values</td>
<td>Fitness = 14</td>
<td>Fitness = 22</td>
</tr>
<tr>
<td>Convergence</td>
<td>Generation = 98</td>
<td>Generation = 87</td>
</tr>
<tr>
<td>Values</td>
<td>Fitness = 36</td>
<td>Fitness = 36</td>
</tr>
<tr>
<td>Equation</td>
<td>X = A ⊕ B ⊕ C</td>
<td>X = A ⊕ C ⊕ B</td>
</tr>
<tr>
<td></td>
<td>Y = AB + C(A ⊕ B)</td>
<td>Y = AB + B(A ⊕ C)</td>
</tr>
<tr>
<td>Gates Number</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Gates Count</td>
<td>2 AND, 1 OR, 2 XOR</td>
<td>2 AND, 1 OR, 2 XOR</td>
</tr>
</tbody>
</table>

All the values obtained for these two runs are shown (in tabular form) in Appendix B.

The graph for the full adder circuit with an initial population randomly generated is the same in the three steps described in this section (see figure 5.1 to compare the results obtained for this second step).

In the graph depicted in Figure 5.4 we show the evolution for the full adder circuit with a percentage of the initial population generated from individuals in the case base. In this second step the case base had cases with solutions to many circuits including the full adder, the half adder, the comparator and other circuits. We can observe in figure 5.4 that a fitness
value of 22 is the first value for an individual in generation 1. This is because the system retrieves as the most similar case the solution found for the full adder circuit with the maximum fitness value in the case base (fitness value = 22). Although the half adder circuit is a similar problem, the system does not retrieve it because it retrieves the previous solution to the full adder circuit due to the fact it is identical to the new problem.

**Figure 5.4 Full adder circuit with a percentage of the initial population generated from individuals in the case base.**

In the graph depicted in Figure 5.5 we show the evolution for the full adder circuit comparing both, the evolution of the circuit with the initial population randomly generated and the evolution of the circuit with the initial population seeded from individuals in the case base.

**Figure 5.5 Comparison for the full adder circuit for step 2.**
Conclusions from step 2:

- As in the first step the GA found the solution more quickly because the system retrieves from the case base a previous solution found for the full adder circuit. It found that a previous solution for the full adder circuit is an identical problem solved previously, and retrieved this case and seeded it with a percentage of the initial population.

- The system discriminates the other circuits including the half adder circuit because the most similar case found in the case base is the same previously solved problem.

- As in step 1, the GA with an initial population with previous solutions begins to evolve from a valid circuit (fitness = 22) and arrives at an optimal solution more quickly in comparison with the GA with an initial population randomly generated.

- We can conclude for this second step that, if the system has to solve a new problem and it has solved a similar problem and an identical problem previously, the system uses the previous solution found for the identical problem in order to solve more quickly the new problem.

5.3 Third step.
In this step the full adder circuit was solved generating a percentage (10%) of the initial population with individuals retrieved from the case base, which are the most similar cases, found for the full adder circuit, and another percentage (90%) with individuals randomly generated.

In this step the individuals stored in the case base were previous solutions to different circuits including solutions to the half-adder circuit, the comparator circuit and other circuits, but without including solutions to the full adder circuit. This was done to show how the system discriminates between the solutions to different circuits and retrieves the half adder as the most similar case. Although this step does not improve convergence time as was the case in steps 1 and 2, it converges more quickly on the solution in comparison with an initial population randomly generated.

Comparison of results

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Human Designer</th>
<th>GA Run 1 (Initial population randomly generated)</th>
<th>GA Run 2 (Initial population generated with previous solutions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population</td>
<td>700</td>
<td></td>
<td>700</td>
</tr>
<tr>
<td>Generations</td>
<td>300</td>
<td></td>
<td>300</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.5</td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>0.006</td>
<td></td>
<td>0.006</td>
</tr>
<tr>
<td>Rseed</td>
<td>0.249</td>
<td></td>
<td>0.249</td>
</tr>
<tr>
<td>Initial values</td>
<td>Fitness = 14</td>
<td>Fitness = 14</td>
<td></td>
</tr>
<tr>
<td>Convergence Values</td>
<td>Generation = 98</td>
<td>Fitness = 36</td>
<td></td>
</tr>
<tr>
<td>Equation</td>
<td>X = A ⊕ B ⊕ C</td>
<td>X = A ⊕ B ⊕ C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y = AB + C(A ⊕ B)</td>
<td>Y = AB + B(A ⊕ C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X = A ⊕ C ⊕ B</td>
<td>X = A ⊕ C ⊕ B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y = AB + B(A ⊕ C)</td>
<td>Y = AB + B(A ⊕ C)</td>
<td></td>
</tr>
</tbody>
</table>
All the values obtained for these two runs are shown (in tabular form) in Appendix C.

![Full adder circuit](image)

**Figure 5.6** Full adder circuit with a percentage of the initial population generated from individuals in the case base.

![Full adder circuit](image)

**Figure 5.7** Comparison for the full adder circuit for step 3.
In the graph depicted in Figure 5.6 we show the evolution for the full adder circuit with a percentage of the initial population generated from individuals in the case base. In this third step the case base had cases with solutions to many circuits including the half adder, the comparator and other circuits, but without including the full adder circuit. We can observe in figure 5.6 that the GA begins to evolve from a fitness value of 14 in generation 1 in the same way as the initial population randomly generated does. 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. It is true because the half adder circuit, as the name indicates, is the half of the solution for the full adder circuit.

In the graph depicted in Figure 5.7 we show the evolution for the full adder circuit comparing both, the evolution of the circuit with the initial population randomly generated and the evolution of the circuit with the initial population seeded from individuals in the case base.

Conclusions from step 3:

- The circuit using the GA with an initial population with previous solution converges to the optimal solution somewhat more quickly (generation 95) in comparison with the GA with the initial population randomly generated (generation 98). The system retrieves from the case base as the most similar case, the solution obtained for the half adder circuit. It found that a previous solution for the half adder circuit is not an identical problem, but it is the most similar problem solved previously and retrieved and seeded with this case a percentage of the initial population.

- The system discriminates the other circuits and selects the half adder circuit as the most similar case found in the case base.

- Although the GA with an initial population with previous solutions begins to evolve in the same way as the GA with initial population randomly generated, it evolves in a completely different way. The former finds a valid circuit in generation 34, in contrast with generation 45 for the latter. So we can say that if the system seeds an initial population with a similar problem previously solved it causes the GA to explore the search space in a more efficient way.

- We can conclude for this third step that if the system has to solve a new problem and it has solved only a similar problem previously, the system uses this similar previous solution in order to solve the new problem more quickly.
6 Conclusions

For a new problem (inputs and outputs in a truth table) when we populate a given generation with a similar case solved previously we are modifying the exploration of the search space (a kind of “mutation”) but without losing the benefits related to the exploitation of the search space. Mainly this is because this kind of “mutation” is done in such a way that the search is directed towards the most promising areas of the search space.

With this research we have shown an alternative to the design of optimal combinational logic circuits using a GA, and then we have analyzed the additional use of CBR to solve the full adder circuit. The GA is able to find better circuits (in terms of number of gates) in comparison with circuits obtained from Karnaugh maps. The use of CBR improves convergence time. We empirically show that the input parameters of the GA are very important in order to obtain better solutions (i.e., the GA is very sensitive to such parameters).

With the analysis of the obtained solutions with the GA, we can see that the reduction in the number of gates is mainly due to the fact the GA reuses a lot of components. This is normally very hard for a human designer.

From the analysis of the evolutionary design process of a logic circuit we can conclude that if we can recognize and retrieve the building blocks that appear at the end of and during the evolutionary process of the circuit and then store this information in a knowledge base, we can improve the convergence time of the GA. Previously generated solutions are used to feed the initial population with individuals from this knowledge base. With the use of its own experience the system saves and retrieves solutions to similar or identical problems to seed the initial population of the GA increasing its performance in this way, without discarding the information generated in previous solutions.

In this investigation we have put forward the view that the analysis of evolutionary algorithms results can be regarded as a discovery engine of well-known laws or theorems, and new principles or theorems. We studied this idea in the context of digital logic. We suggested that new principles can be discovered by examining a series of evolved designs, in our case, for some arithmetic logic circuits. We examined the concept of the space of all circuit representations, but we feel that similar ideas may be well carried over the general field of design.
Some of the areas for future work are the following:

a) Automatic retrieval and automatic storage of cases in the case base.

b) Improvement in the efficiency of the retrieval and storage of cases.

c) Verification of the propagation of good building blocks and tracking them, as a way to improve performance of the GA.

d) Evaluation of how the learning platform converges when the algorithm does not choose the individuals with the higher fitness in the initial population due to the fact it considers other attributes to be more important (genotype, number of inputs, number of outputs, etc.)

e) Development of the Genetic Algorithm 1 module to implement the attribute selection and weighting.

f) Implementation of the storage and retrieval of cases during the evolutionary process of the logic circuit.

g) The addition of the option of using gates with any number of inputs and interconnected with gates in any column.

h) The addition to the platform of the option of using flip-flops in order to use the system in the optimization of sequential logic circuits.

i) The application of this system to solve circuits in the field of LSI and VLSI.

j) The application of the Learning Platform in another field (for example in structural design).
Data obtained in the first step for applying CBR and GA to the full adder circuit

<table>
<thead>
<tr>
<th>Generation Number</th>
<th>Initial population randomly generated</th>
<th>Initial population with previous solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fitness</td>
<td>Violations</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>39</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>46</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>50</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>52</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>53</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>55</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>56</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>56</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>58</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>61</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>65</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>66</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>67</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>68</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>69</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>70</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>71</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>72</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>73</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>74</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>76</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>77</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>78</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>79</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>81</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>82</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>83</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>85</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>86</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>87</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>88</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>89</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>92</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>93</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>94</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>95</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>96</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>98</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>36</td>
<td>0</td>
</tr>
</tbody>
</table>
Data obtained in the second step for applying CBR and GA to the full adder circuit

<table>
<thead>
<tr>
<th>Generation Number</th>
<th>Initial population randomly generated</th>
<th>Initial population with previous solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fitness</td>
<td>Violations</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>39</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>46</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>52</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>53</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>55</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>56</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>58</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>61</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>65</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>66</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>67</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>68</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>69</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>70</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>71</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>72</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>73</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>74</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>76</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>77</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>78</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>79</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>81</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>82</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>83</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>85</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>86</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>87</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>88</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>89</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>92</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>93</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>94</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>95</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>96</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>98</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>36</td>
<td>0</td>
</tr>
</tbody>
</table>
Data obtained in the third step for applying CBR and GA to the full adder circuit

<table>
<thead>
<tr>
<th>Generation Number</th>
<th>Initial population randomly generated</th>
<th>Initial population with previous solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fitness</td>
<td>Violations</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>39</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>46</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>52</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>53</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>55</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>56</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>58</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>61</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>65</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>66</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>67</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>68</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>69</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>70</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>71</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>72</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>73</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>74</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>76</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>77</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>78</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>79</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>81</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>82</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>83</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>85</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>86</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>87</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>88</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>89</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>92</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>93</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>94</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>95</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>96</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>98</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>36</td>
<td>0</td>
</tr>
</tbody>
</table>
D.1 Source code.

D.1.1 Designer.h

```c
/* Global structures and variables */

#define LINELENGTH 80 /* width of printout */

#include <stdio.h>
#include <string.h>
#include <math.h>

struct individual
{
  unsigned int *chrom; /* chromosome string for the individual */
  int parent[2];
  long int viol;
  double fitness; /* fitness of the individual */
  long int f[4];
};

struct bestever
{
  unsigned int *chrom; /* chromosome string for the best-ever individual */
  double fitness; /* fitness of the best-ever individual */
  long int viol;
  long int f[4];
  int generation; /* generation which produced it */
};

/* Functions prototypes */

void read_input();
void compute_code_size();
void memory_allocation();
void memory_for_selection();
void free_all();
void free_selection();
void nomenclature(char *string);
void initialize_pop();
void initial_report();
void statistics(struct individual *);
void warm_up_random(float);
float rndreal(float, float);
int rnd(int,int);
float randomperc();
double randomnormaldeviate();
```

102
void randomize();
double noise(double, double);
void initrandomnormaldeviate();
int flip(float);
void advance_random();
void generation();
int select();
void recharr(FILE *, char *, int);
void skip(FILE *, int);
void select_memory();
void select_flee();
void mutation(struct individual *);
void crossover(unsigned int *, unsigned int *, unsigned int *, int *, int *);
void report();
void winklechrom(unsigned int *);
void cls(int);
void create_string(int);
long int potencia(int, int);
void report_pop(unsigned int *);
void objfun(struct individual *ind);
void create_string(int);
void print_tips(int);
void decode(unsigned int *, int);

static int *tourlist, tourpos, toursize; /* Tournament list, position in list */
struct individual *oldpop; /* last generation of individuals */
struct individual *newpop; /* next generation of individuals */
struct beast evolve; /* fittest individual so far */
double sum_fitness; /* summed fitness for entire population */
unsigned int *localmax; /* String corresponding to the local maximum */
double avg; /* average fitness of population */
double min; /* minimum fitness of population */
float Pc; /* probability of crossover */
float Pm; /* probability of mutation */
int gen; /* current generation number */
int Gmax; /* maximum number of generations */
int nmutation; /* number of mutations */
int ncross; /* number of crossovers */
float Rseed; /* Random number seed */
double oldrand[55]; /* Array of 55 random numbers */
int jrand; /* current random number */
double rndx2; /* used with random normal deviate */
int rndcallflag; /* used with random normal deviate */
FILE *input, *output, *ideal;
int popsize, codes[as], run, maxruns, size[500];
char REPRESENTATION[16], VAR_TYPEP[500];
int lower_int_range[500], upper_int_range[500], precision[500], tipo[500],
input[500], input2[500];
int num_vars[100], num_vars;
int truth_int[300][100], truth_out[300][100];
double lower_real_range[500], upper_real_range[500], discrete[500][500];
double h[20]; /* specific to the problem */
int printstrings = 1; /* print strings (=1) or not (=0) */
int num_objs; /* Required by VEGA and similar approaches */
int obj_num, cardinality; /* index of corresponding objective */
int feasible_flag = 0; /* Indicate if a solution is feasible or not */
int num_weight = 10, num_entr1es, num_in1puts, num_out1uts;
long int f[100];
double f[200][100], threshold;
int table_size, num_rows, num_cols, num_gates, num_input1, num_input2;
int fitness_flag = 1;
int num_objs = 33; /* f(x) + constraints */
D.1.2 Cases.c

/* Circuit Design using Genetic Algorithms */
/* Implemented by: Carlos A. Coelho Coelho, Ph.D. */
/* Tulane University, Department of Computer Science */
/* Last update: February 22, 1997 */

/*******************************************************************************/
/* Case Based Reasoning Module in the Initial Population*/
/* Implemented by: Eduardo Islas Perez, M.Sc. */
/* Universidad Veracruzana - LANIA, IA Department */
/* Last update: August 19, 2000 */
/*******************************************************************************/

/* Attempt to use lower mutation rates */
#include "designer.h"

int which;

main(int argc, char *argv[]) {
    struct individual *temp;
    float Reseed; /*
    if(argc < 3) {
        printf("Usage: designernew <input file> <output file> {<# of runs> <# of generations> <pop size> <mutation rate>}
        <crossover rate> <seed> }\n\n"");
        exit(1);
    } else {
        if((input=fopen(argv[1],"r")==NULL) {
            printf("Error opening file %s in read mode\".argv[1]);
            exit(1);
        }
    read_input();
    compute codersize();

    if((output=fopen(argv[2],"w")==NULL) {
        printf("Error opening file %s in write mode\".argv[2]);
        exit(1);
    }

    } /* Default parameters */

    Gmax=200; /* 200 */
    popsize=700; /* 700 */
    Pm=0.5/75; /* 0.5 */
    Pc=0.5;
    Rseed=0.35;
    maxruns=1;

    if(argc > 3) {
        Rseed=atof(argv[3]);
        if(((Rseed<0.0) || (Rseed>1.0)) {
            printf("The random number seed, %f, must be between 0.0 and 1.0\". Rseed);
            exit(1);
        }
    }

    if(argc > 4) {
        popsize=atoi(argv[4]);
    }
if (popsize<10) {
    print("Population size, \"%d, \must be at least ten\", popsize);
    exit(1);
}

if (argc>5) {
    Gmax=atoi(argv[5]);
}

if (Gmax<1) {
    print("There must be at least one generation\n");
    exit(1);
}

if (argc>6) {
    Pm=atof(argv[6]);
}

if ((Pm<0.0) || (Pm>1.0)) {
    print("Mutation rate, \"%f, \must be between 0.0 and 1.0\", Pm);
    exit(1);
}

if (argc>7) {
    Pc=atof(argv[7]);
}

if ((Pc<0.0) || (Pc>1.0)) {
    print("Crossover rate, \"%f, \must be between 0.0 and 1.0\", Pc);
    exit(1);
}

if (argc>8) {
    maxruns=atoi(argv[8]);
}

if (maxruns<0) {
    print("There must be at least one run\n");
}

for (run=1; run<maxruns; run++) {
    /* Perform the previous memory allocation */
    memory_allocation();
    /* Initialize random number generator */
    randomize();
    nmutation = 0;
    ncrass = 0;
    bestfit.fitness = 0.0;
    bestfit世代 = 0;
    max=0.0;
    min=1.0e38;
    /* Initialize the populations and report statistics */
    initialize_pop();
    statistics(oldpop);
    initial_report();
    for (gen=0; gen<Gmax; gen++) /* Iterate the given number of generations */{

"* Create a new generation */

    printf("Generation \# %d --> Fitness = %f, Violations = %d\"", gen, bestfit.fitness, bestfit.viol);
/* printf("Generation \# %d\"", gen); */
    generation();

/* Compute fitness statistics on new populations */

    statistics(newpop);

/* Report results for new generation */

    report();

/* Advance the generation */

    temp = oldpop;
    oldpop = newpop;
    newpop = temp;

    fclose(output);
    free_all(r;

} /* Read file containing information about representation chosen, and ranges
   of each one of the variables */

void read_input()

    int i,j;

    /* Choose representation:

        BINARY = Use binary representation
        FLOAT = Use floating-point representation

    */
    fgets(REPRESENTATION,16,input);

    /* Give the number of inputs */
    fscanf(input,"%d", &num_inputs);
    printf(\"There are %d inputs\n\", num_inputs);

    /* Give the number of outputs */
    fscanf(input,"%d", &num_outputs);
    printf(\"There are %d outputs\n\", num_outputs);

    /* Read cardinality of the alphabet */
    fscanf(input,"%d", &cardinality);
    printf(\"The cardinality of the alphabet is %d\n\", cardinality);

    /* Read number of rows of the array of gates */
    fscanf(input,"%d", &num_rows);
    printf(\"The array of gates has %d rows\n\", num_rows);

    /* Read number of columns of the array of gates */
    fscanf(input,"%d", &num_cols);
print("The array of gates has %d columns/n", num_cols);

/* Read the number of gate types available */
scanf(input, "%d", &num_gates);
printf("There are %d types of gates available/n", num_gates);

/* Read the number of possible first inputs */
scanf(input, "%d", &num_input1);
printf("There are %d possible first inputs/n", num_input1);

/* Read the number of possible second inputs */
scanf(input, "%d", &num_input2);
printf("There are %d possible second inputs/n", num_input2);

/* Read the printout threshold */
scanf(input, "%f", &threshold);
printf("Results will be printed only when the fitness >= %lf/n", threshold);

/* Read the truth table */
num_entries = int(pow(2, num_inputs);

/* The truth table has %d entries/n", num_entries);

/* Read inputs */
printf("Inputs/n");
for (i=0; i<num_entries-1; i++) {
   for (j=1; j<num_inputs; j++) {
      fscanf(input, "%d", &truth_in[i][j]);
      printf("%d ", truth_in[i][j]);
   }
   printf("n");
}

/* Read outputs */
printf("Outputs/n");
for (i=0; i<num_entries-1; i++) {
   for (j=1; j<num_outputs; j++) {
      fscanf(input, "%d", &truth_out[i][j]);
      printf("%d ", truth_out[i][j]);
   }
   printf("n");
}

/* Give the number of variables to use */
num_vars = num_rows*num_cols+3; /* 3 = Type + Imp1 + Imp2 */

/* fscanf(input, "%d", &num_vars); */
/* INTEGER = Provide lower and upper limit
REAL = Provide lower and upper limit, and precision
DISCRETE = Provide list of values */

printf("Representation = %s",REPRESENTATION);
printf("There are %d variables\n",num_vars);
for (i=1; i<=num_vars; i++) {
    strcpy(VAR_TYPE[i],"INTEGER");
}
for (i=1; i<=num_vars; i+3) {
    lower_int_range[i]=0;
    upper_int_range[i]=num_vars-1;
}
for (i=2; i<=num_vars; i+3) {
    lower_int_range[i]=0;
    upper_int_range[i]=num_input1-1;
}
for (i=3; i<=num_vars; i+3) {
    lower_int_range[i]=0;
    upper_int_range[i]=num_input2-1;
}
for (i=1; i<=num_vars; i++) {
    printf("Variable %s is of type %s\n",i,VAR_TYPE[i]);
    printf("The lower limit is %d\n",lower_int_range[i]);
    printf("The upper limit is %d\n",upper_int_range[i]);
}
fclose(input);

/* Define the population for generation # 0 */
void initialize_pop()
{
    int bits;
    int j, k, m;

    FILE *cases;
    char namefile[]="cases.txt";
    int numcases=(int)(0.10*popsize);
    int totalcases=10;
    int numcases=10;
    int attribute=4;
    int attributes=3;
    int case[totalcases][attributes+codesize];
    int hold, pass;
    int switched = 1;
}
if (!fopen(namefile, "r")) == NULL) {
    printf("Error opening file %s in mode '%s', namefile
exit(1);
}

/* To read the case base and assign the attributes in a bidimensional array, including the individual genotype */
/* The number of cases are 10% of the population */

for (j=0; j < totalcases; j++) {
    printf("Individual %d ", j);
    for (k=0; k < attributes; k++) {
        fscanf(cases, "%d", &bits);
        cases[j][k]=bits;
        printf("%d ", cases[j][k]);
    }
    for (k=attributes; k < attributes+codesize; k++) {
        fscanf(cases, "%d", &bits);
        cases[j][k]=bits;
        printf("%d ", cases[j][k]);
    }
    printf ("n");
}

/* To read from the case base (array) every individual and get the new fitness values */
for (j=0; j < totalcases; j++) {
    printf("Individual %d ", j);
    for (k=attributes; k < attributes+codesize; k++) {
        if (strcmp(REPRESENTATION,"BINARY",6)==0)  
            oldpop[j].chrom[k-attributes]=cases[j][k];
        else
            oldpop[j].chrom[k-attributes]=cases[j][k];
    }
    printf("n");
    if (cases[j][1]==3 & cases[j][2]==2) {
        decode(oldpop[j].chrom, codesize);
        objfunc(&oldpop[j]);
        cases[j][attribfitness]=oldpop[j].fitness;
    } else cases[j][attribfitness]=0;
}

/* Determine in the case base the individuals with better fitness values */
for(j=0; j < numcases-l & & switched==1; pass++){
    switched = 0;
    for (k=0; k<numcases-l ; k++){
        if (cases[k][3] < cases[k+1][3]){
            switched = 1;
            for (m=0; m<attributes+codesize; m++){
                hold = cases[k][m];
                cases[k][m] = cases[k+1][m];
                cases[k+1][m]= hold;
            }
        }
    }
}
for (j=0; j < numcases; j++) {
    printf(Individual %d: ", j);
    for (k=0; k < attributes; k++) {
        printf ("%d: %s, cases[j][k]);
    }
    for (k=attributes; k < attributes+codesize; k++) {
        printf ("%d: %s, cases[j][k]);
    }
    printf ("n");
}
/* Randomly generating the rest of the population (90% of the population) */
for (j=numcases; j < popsize; j++) {
    for (k=0; k < codesize; k++) {
        if (strcmp(REPRESENTATION,"BINARY",6)==0)
            oldpop[j][chrom[k]] = rand(0,1);
        else
            oldpop[j][chrom[k]] = rand(0,cardinality-1);
    }
    decode(oldpop[j],chrom, codesize);
    objfunc(&oldpop[j]); /* Evaluate initial fitness */
}
/* Print the initial report of the parameters given by the user */
void initial_report()
{
    fprintf(output,"n Parameters used with the Genetic Algorithm:\n\n");
    fprintf(output,"Total population size = %d\n",popsize);
    fprintf(output,"Chromosome length = %d\n",codesize);
    fprintf(output,"Maximum number of generations = %d\n",Gmax);
    fprintf(output,"Crossover probability = %f\n",Pc);
    fprintf(output,"Mutation probability = %f\n",Pm);
    fprintf(output,"\n\n");
}
/* Compute the chromosome length, based on the information provided by the user */
void compute_codesize()
{
    int i;
    long int lower_range, upper_range;
    for (i=1; i<=num_vars; i++) { /* Loop through all variables */
        if (strcmp(REPRESENTATION,"BINARY",6)==0) {
            if (strcmp(VAR_TYPE[i],"INTEGER",7)==0) {
                size[i] = (int)(log((double)(upper_int_range[i]-lower_int_range[i]))/log(2.0)+0.9);
                if (size[i]==0) size[i]=1; /* end INTEGER */
            }
            if (strcmp(VAR_TYPE[i],"REAL",4)==0) {
                lower_range = lower_real_range[i] * potencia(10, precision[i]);
                upper_range = upper_real_range[i] * potencia(10, precision[i]);
                size[i] = (int)(log((double)(upper_range-lower_range))/log(2.0)+0.9);
            }
        }
    }
if (size[i]==0) size[i]=1;
    } /* end REAL */
if (strcmp(VAR_TYPE[i], "DISCRETE", 8)==0) {
    size[i] = (int)(log((double)num_vals[i])/log(2)+0.9);
if (size[i]==0) size[i]=1;
    } /* end DISCRETE */
printf("Variable %s is of type %s. ",VAR_TYPE[i], VAR_TYPE[i]);
printf("and it needs %d genes under binary representation\n", size[i]);
} /* end BINARY REPRESENTATION */
else { /* begin FLOATING-POINT REPRESENTATION */
if (strcmp(VAR_TYPE[i], "INTEGER", 7)==0) {
    size[i] = (int)(log((double)(upper_int_range[i]-lower_int_range[i]))/log(10.0)+0.9);
if (size[i]==0) size[i]=1;
    } /* end INTEGER */
if (strcmp(VAR_TYPE[i], "REAL", 4)==0) {
    lower_range = lower_real_range[i]*potencia(10, precision[i]);
    upper_range = upper_real_range[i]*potencia(10, precision[i]);
    size[i] = (int)(log((double)(upper_range-lower_range))/log(10.0)+0.9);
if (size[i]==0) size[i]=1;
    } /* end REAL */
if (strcmp(VAR_TYPE[i], "DISCRETE", 8)==0) {
    size[i] = (int)(log((double)num_vals[i])/log(10.0)+0.9);
if (size[i]==0) size[i]=1;
    } /* end DISCRETE */
printf("Variable %s is of type %s. ",VAR_TYPE[i], VAR_TYPE[i]);
printf("and it needs %d genes under floating-point representation\n", size[i]);
} /* end FLOATING-POINT REPRESENTATION */
codesize+=size[i];
printf("Codesize = %d\n", codesize);
} /* End loop */

long int potencia(int base, int exponente)
{
    int j;
    long int result=1;
    for (j=0; j<exponente; j++) result*=base;
    return result;
}
/* Perform the memory allocation needed for the strings that will be generated */

void memory_allocation()
{
    unsigned int numbytes, nbytes;
    char *malloc();
    int i;

    /* Allocate memory for old and new populations of individuals */
    numbytes = popsize*sizeof(struct individual);
    if ((oldpop = (struct individual *) malloc(numbytes)) == NULL)
        nomainery("old population");
    if ((newpop = (struct individual *) malloc(numbytes)) == NULL)
        nomainery("new population");

    /* Allocate memory for chromosome strings in populations */
    nbytes = codesize*sizeof(unsigned int);
    for (i=0; i < popsize; i++) {
        if ((oldpop[i].chrom = (unsigned int *) malloc(nbytes)) == NULL)
            nomainery("old population chromosomes");
        if ((newpop[i].chrom = (unsigned int *) malloc(nbytes)) == NULL)
            nomainery("new population chromosomes");
        if ((bestfit.chrom = (unsigned int *) malloc(nbytes)) == NULL)
            nomainery("bestfit chromosomes");
    }
    select_memory();
}

/* When done, free all memory */

void free_all()
{
    int i;
    for (i=0; i < popsize; i++) {
        free(oldpop[i].chrom);
        free(newpop[i].chrom);
    }
    free(oldpop);
    free(newpop);
    select_free();
}

/* Notify if we run out of memory when generating a chromosome */

void nomainery(char *string)
{
    printf("ERROR! out of memory making %s
", string);
    exit(1);
}

/* Perform all the work involved with the creation of a new generation of chromosomes */

void generation()
{
    int mate1, mate2, jcross1, jcross2, j = 0;
}
/* perform any preselection actions necessary before generation */
preselect();

/* select, crossover, and mutation */
do {
    /* pick a pair of mates */
    mate1 = select();
    mate2 = select();

    /* Crossover and mutation */
crossover(oldpop[mate1].chrom, oldpop[mate2].chrom, newpop[j].chrom,
            newpop[j+1].chrom,&jcross1,&jcross2);
    mutation(&(newpop[j]));
    mutation(&(newpop[j+1]));
    /* Decode string, evaluate fitness, & record */
    /* parentage data on both children */
decode(newpop[j].chrom, codesize);
decode(newpop[j+1].chrom, codesize);
    objfunc(&(newpop[j]));
    objfunc(&(newpop[j+1]));

    /* Increment population index */
    j = j + 2;
}
while(j < (popsize-1));

/* Perform a mutation in a random string, and keep track of it */
void mutation(struct individual *ind)
{
    int placemut,k;
    /* Do mutation with probability Pm */
    /*
if (flip(Pm)) {
    placemut=rnd(0,(codesize-1));
    nmutation++;
    for (k=0;k<codesize;k++) {
        if (placemut==k) {
            if (strcmp(REPRESENTATION,"BINARARY")==0)
                ind->chrom[k]=rnd(0,1);
            else
                ind->chrom[k]=rnd(0,cardinality-1);
        }
    }
*/
    for (k=0; k < codesize; k++) {
        if (flip(Pm)) {
            nmutation++;
        }
    }
}
if(strcmp(REPRESENTATION,"BINARY")!=0)
    ind->chrom[k]=rnd(0,1);
else
    ind->chrom[k]=rnd(0,cardinality-1);
}

/* Cross 2 parent strings, place in 2 child strings */

void crossover (unsigned int *parent1, unsigned int *parent2,
                unsigned int *child1, unsigned int *child2,
                int *c1, int *c2)
{
    int i, j;
    unsigned temp;

    /* Do crossover with probability Pc */
    if(flip(Pc))
    {
        *c1 = (unsigned int)rand(0,(unsigned int)(codenumber-1)); /* Crosspoint 1 between 1 and length-1 */

        /* Define crosspoint 2 between one child1 and length-1 */
        if(*c1 < (unsigned int)(codinglen-1)) *c2 = (unsigned int)rand(*c1+1,(unsigned int)(codenumber-1));
        else
        {
            *c2 = (unsigned int)rand(0,(*c1-1)); /* Make sure that c1 is */
            temp = *c1; /* always less than c2 */
            *c1 = *c2;
            *c2 = temp;
        }

        ncross++;

        /* Keep track of the number of crossovers */
        for (i=(unsigned int)0; i<codinglen; i++)
        {
            child1[i] = parent1[i];
            child2[i] = parent2[i];
        }
    }
    else
    {
        for (i=0; i<codinglen; i++)
        {
            child1[i] = parent1[i];
            child2[i] = parent2[i];
        } /*cross 1 = 0; *cross 2 = 0; */

    }

    /* Create next batch of 55 random numbers */
    void advance_random()
    {
        int i;
        double new_random;

    }

}
```c
for(j1 = 0; j1 < 24; j1++)
{
    new_random = oldrand[j1] - oldrand[j1+31];
    if(new_random < 0.0) new_random = new_random + 1.0;
    oldrand[j1] = new_random;
}
for(j1 = 24; j1 < 55; j1++)
{
    new_random = oldrand[j1] - oldrand[j1-24];
    if(new_random < 0.0) new_random = new_random + 1.0;
    oldrand[j1] = new_random;
}

/* Flip a biased coin - true if heads */
int flip(float prob)
{
    float randomperc();
    if(randomperc() <= prob) return(1);
    else return(0);
}

/* initialization routine for randomnormaldeviate */
void initrandomnormaldeviate()
{
    mdcalcflag = 1;
}

/* normal noise with specified mean & std dev: mu & sigma */
double noisy(double mu, double sigma)
{
    double randomnormaldeviate();
    return((randomnormaldeviate()*sigma) + mu);
}

/* Initialize random numbers batch */
void randomize()
{
    int j1;
    for(j1=0; j1<54; j1++)
        oldrand[j1] = 0.0;
    jrand=0;
    warmup_random(Rseed);
}

/* random normal deviate after ACM algorithm 267 / Box-Muller Method */
double randomnormaldeviate()
{
    double sqrt(), log(), sin(), cos();
```
float randomperc();
  double t, rndx1;
  if(rndcalcflag)
    { 
      rndx1 = sqrt(-2.0*log((double) randomperc()));
      t = 6.2831853072 * (double) randomperc();
      rndx2 = sin(t);
      rndcalcflag = 0;
      return(rndx1 * cos(t));
    } 
  else 
    { 
      rndcalcflag = 1;
      return(rndx2);
    } 
/* Fetch a single random number between 0.0 and 1.0 - Subtractive Method */
/* See Knuth, D (1969). v. 2 for details */
/* name changed from random() to avoid library conflicts on some machines */
float randomperc()
  { 
    jrand++;
    if(jrand >= 55)
      { 
        jrand = 1;
        advance_random();
      }
    return((float) oldrand[jrand]);
  }
/* Pick a random integer between low and high */
int rnd(int low, int high)
  { 
    int i;
    float randomperc();
    if(low >= high)
      i = low;
    else 
      { 
        i = (randomperc() * (high - low + 1)) + low;
        if(i > high) i = high;
      }
    return(i);
  }
/* real random number between specified limits */
float rndreal(float lo, float hi)
  { 
    return((randomperc() * (hi - lo)) + lo);
  }
/* Get random off and running */
void warmup_random(float random_seed)
{
    int j1, ii;
    double new_random, prev_random;

    oldrand[54] = random_seed;
    new_random = 0.0000000001;
    prev_random = random_seed;

    for(j1 = 1; j1 <= 54; j1++)
    {
        ii = (21*ii)%54;
        oldrand[ii] = new_random;
        new_random = prev_random-new_random;
        if(new_random<0.0) new_random = new_random + 1.0;
        prev_random = oldrand[ii];
    }
    advance_random();
    advance_random();
    advance_random();
    jrand = 0;
}

/*--------------------------*/
/* tourneylist, toruencypos, tournamentsize;
/* tournament list, position in list */
void select_memory()
{
    unsigned nbytes;
    char *malloc();
    int j;

    nbytes = popsize*sizeof(int);
    if(tourneylist = (int *)malloc(nbytes)) == NULL)
        nomemory("tourneylist");

    /* if(numfiles == 0)
        fprintf(output,"Enter tournament size for selection --> ");
        scanf(intp,"%d", &tournamentsize); */
    tournamentsize = 2;
    if(tourneysize > popsize)
    {
        fprintf(output,"FATAL: Tournament size (%d) > popsize (%d)\n",
            tournamentsize, popsize);
        exit(-1);
    }
}

void select_free()
{
    free(tourneylist);
}

void preselect()
{
    reset();
    tourneypos = 0;
```c
int select()
{
    int pick, winner, i;
    /* If remaining members not enough for a tournament, then reset list */
    if((popsize - tourneypos) < tourneysize)
    {
        reset();
        tourneypos = 0;
    }
    /* Select tourneypos structures at random and conduct a tournament */
    winner = tourneylist[tourneypos];
    for(i = 1; i < tourneysize; i++)
    {
        pick = tourneylist[i + tourneypos];
        if(oldpop[pick].fitness > oldpop[winner].fitness) winner = pick;
    }
    /* Update tourneypos */
    tourneypos += tourneysize;
    return(winner);
}
void reset()
{ /* Shuffles the tourneylist at random */
    int i, rand1, rand2, temp;
    for(i = 0; i < popsize; ++i)
    {
        for(i = 0; i < popsize; ++i)
        {            rand1 = rd(1, popsize);            rand2 = rd(1, popsize);
            temp = tourneylist[rand1];            tourneylist[rand1] = tourneylist[rand2];
            tourneylist[rand2] = temp;
        }
    }
    /* Calculate population statistics */
    void statistics(struct individual *pop)
    {
        int i, j, c;
        sumfitness = 0.0;
        /* min = pop[0].fitness; max = pop[0].fitness; */
        /* Loop for max, min, sumfitness */
        for(j = 0; j < popsize; j++)
        {            sumfitness = sumfitness + pop[j].fitness; /* Accumulate */
                if(pop[j].fitness > max) { max = pop[j].fitness; /* New maximum */
                        localmax = pop[j].chrom; /* Store string */
                }
                if(pop[j].fitness < min) min = pop[j].fitness; /* New minimum */
        }
    /* Define new global best-fit individual */
```

118
if (pop[j].fitness > bestfit.fitness) {
    for (i=0; i < codesize; i++) bestfit.chrom[i] = pop[j].chrom[i];
    bestfit.fitness = pop[j].fitness;
    bestfit[1] = pop[j][1];
    bestfit[2] = pop[j][2];
    bestfit.viol = pop[j].viol;
    bestfit.generation = gen;
}

/* Calculate average */

avg = sumfitness/popsize;

void repchar (outfp, repcount)
/* Repeatedly write a character to stdout */
FILE *outfp;
char *ch;
int repcount;
{
    int j;

    for (j = 1; j <= repcount; j++) fprintf(outfp,"%s", ch);
}

void skip(outfp, skipcount)
/* Skip skipcount lines */
FILE *outfp;
int skipcount;
{
    int j;

    for (j = 1; j <= skipcount; j++) fprintf(outfp,"\n");
}

/* ---------------------------------------------------------------*/
/* report.c - generation report files */
/* ---------------------------------------------------------------*/

void report()
/* Write the population report */
{
    void repchar(), skip();
    int i, j, k, c;

    struct individual *pind;

    repchar(output,"-", LINELENGTH);
    skip(output, 1);
    /* if (gen == Gmax-2) */
    {
        repchar(output," ",(LINELENGTH-17)/2);
        fprintf(output,"Population Report\n");
        fprintf(output,"Generation %d\n", gen);

        fprintf(output," Old Strings\n");
        for(j=0; j<popsize; j++)
        {
            /* Old string*/
            pind = &oldpop[j]);
            decode(oldpop[j].chrom, codesize);
        }
    }
...

119
objfunc(\&oldpop[j]);
if (oldpop[j].viol == 0 && oldpop[j].fitness >= threshold) {
  i+=1; k-=1; c+=1;
  while (k<num_rows*num_cols) {
    fprintf(output,"n");
    fprintf(output,"\%d\%d\%d", input[i], input[j], k);
    if (k % num_cols == 0) {
      fprintf(output,"n");
    } else {
      c+=1;
    }
    i+=num_rows;
    k++;
  }
}
/* } */
/* if (gen==Gmax-1) { */
  fprintf(output, "Generation \%3d\%3d", (gen+1));
  fprintf(output, " New Strings\n");
for (j=0; j<popsize; j++)
  {
    /* New string */
    pind = &newpop[j];
    decode(newpop[j].chrom, code_size);
    objfunc(\&newpop[j]);
    if (newpop[j].viol == 0 && newpop[j].fitness >= threshold) {
      i=1; k=1; c=1;
      while (k<num_rows*num_cols) {
        fprintf(output,"n");
        fprintf(output,"\%d\%d\%d", input[i], input[j], k);
        if (k % num_cols == 0) {
          fprintf(output,"n");
        } else {
          c+=1;
        }
        k++;
      }
    }
  } /* if (gen==Gmax-1) */
}
i = num_rows;
k++;

fprintf(output,"n");
fprintf(output,"f1[1]=%d\n",newpop[j][f1[1]]);
fprintf(output,"f2[1]=%d\n",newpop[j][f2[1]]);
fprintf(output,"fitness=%d\n",newpop[j].fitness);
fprintf(output,"violations=%d\n",newpop[j].viol);
fprintf(output,"String=");
for (k=0; k < codesize; k++) {
    fprintf(output,"%d ",newpop[j].chrom[k]);
}
fprintf(output,"n");
repchar(output,"-",LINELENGTH);
skip(output,1);

/* */
/* write the summary statistics in global mode */
fprintf(output,"Generation %d Accumulated Statistics:\n",gen);
fprintf(output,"Total Crossovers = %d, Total Mutations = %d\n", ncross,mutation);
fprintf(output,"min = %d max = %d avg = %d sum = %d\n", min,max,avg,sumfitness);
fprintf(output,"Global Best Individual so far, Generation %d\n", bestfit.generation);
fprintf(output,"Fitness = %d \n", bestfit.fitness);
writechrom((&bestfit).chrom);
skip(output,1);
report現實(&bestfit).chrom);

}

void print_type(int i)
{
    switch(i) {
    case 0 : fprintf(output," AND\n", break;
    case 1 : fprintf(output," OR\n", break;
    case 2 : fprintf(output," NOT 1\n", break;
    case 3 : fprintf(output," WIRE 1\n", break;
    case 4 : fprintf(output," XOR\n", break;
    case 5 : fprintf(output," NOT 2\n", break;
    case 6 : fprintf(output," WIRE 2\n", break;
    case 7 : fprintf(output," XOR\n", break;
    }
void writepop(FILE *outp)
{
    struct individual *pind;
    int j;
    for (j=0; j<popsize; j++)
    {
        fprintf(outp, "%3d\n", j+1);
        /* Old string */
        pind = &olddop[j];
        writechrom(pind->chrom);
        fprintf(outp, "%8f\n", pind->fitness);
        /* New string */
        pind = &newpop[j];
        writechrom(pind->chrom);
        fprintf(outp, "%8fn\n", pind->fitness);
    }
}

void writechrom(unsigned int *chrom)
/* Write a chromosome as a string of ones and zeroes    */
/* note that the most significant bit of the chromosome is the    */
/* RIGHTMOST bit, not the leftmost bit, as would be expected...    */
{
    int k;
    for (k=0; k < codesize; k++)
    {
        fprintf(output, "%d \n", chrom[k]);
    }
}

void report_es(unsigned int *chrom)
{
    decode(chrom, codesize);
}

void decode(unsigned int *chrom, int length)
{
    int i, j, accum, val;
    i=0; val=1;
    while (i<codesize-1) {
        /*
        * if (val<~num_rows) input1[val]~(chrom[i] % num_inputs) + 1;
        * else input1[val]~chrom[i]+1;
        */
        input1[val]~(chrom[i] % num_inputs) + 1;
        i+=3;
        val++;}
    i=1; val=1;
    while (i<codesize-1) {
        /*
        * if (val<~num_rows) input2[val]~(chrom[i] % num_inputs) + 1;
        * else input2[val]~chrom[i]+1;
        */
        input2[val]~(chrom[i] % num_inputs) + 1;
        i+=3;
        val++;}
}
i=2; val=1;
while (i<=codsize-1) {
    tipo[val] = (chrom[i] % num_gates);
    i+=2;
    val++;
}
}

void object(struct individual **ind)
{
    int i, j, k, val;
    int output[50], b[100][50], ty, kp, lw, total;
    long int viol, right, bonus, z;
    viol=0; right=0; bonus=0; total=num_rows*num_cols;
    for (kp=0; kp<=num_entries-1; kp++)
    for (lw=1; lw<=num_inputs; lw++)
    { b[kp][lw]=truth_tab[kp][lw];
    }
    if (num_input1 > num_inputs) {
        for (kp=0; kp<=num_entries-1; kp++)
        for (lw=num_inputs+1; lw<=num_input1; lw++)
        { b[kp][lw]=b[kp][lw-num_inputs];
        }
    }
    if (num_input2 > num_inputs) {
        for (kp=0; kp<=num_entries-1; kp++)
        for (lw=num_inputs+1; lw<=num_input2; lw++)
        { b[kp][lw]=b[kp][lw-num_inputs];
        }
    }
    for (i=0; i<=num_entries-1; i++)
    { j=1; val=1;
      while (j<=total) {
        switch(tipo[j]) {
        case 0: output[val]=b[i][input1[j]] & b[i][input2[j]];
            break;
        case 1: output[val]=b[i][input1[j]] | b[i][input2[j]];
            break;
        case 2: output[val]=~(b[i][input1[j]]);
            break;
        case 3: output[val]=~(b[i][input1[j]]);
            break;
        case 4: output[val]=~(b[i][input1[j]]);
            break;
        case 5: output[val]=~(b[i][input2[j]]);
            break;
        }
    }
case 6 : output[i][j] = b[i][input[2][j]];
    break;

    case 7 : output[i][j] = b[i][input[j] ^ b[i][input2[i]]];
    break;

    } /* end switch */

    if (output[val] == 1) output[val] = 1;
    if (output[val] == 2) output[val] = 0;

    if (j % num_rows == 0) {
        val = 1;
        for (ty = 1; ty < num_rows; ty++) b[i][ty] = output[i];
    }
    else val++;
    j++;

    } /* end while */

    for (which = 1; which < num_outputs; which++) {
        /* if (output[num_outputs - 1 - which] == truth_out[i][which]) right++;
        if (output[which] == truth_out[i][which]) right++;
        else viol++;
        */
    }

    } /* end for */

    f[1] = right;
    if (viol == 0) { for (i = 1; i < total; i++) { if (((type[i] == 3) || (type[i] == 6)) bonus++; }
    f[2] = bonus;

    ind->fitness = (double)(f[1] + f[2]);
    /* printf("%f", ind->fitness); */
    ind->viol = f[1];
    ind->f[1] = f[1];
    ind->f[2] = f[2];
}