An Approach Based on the Use of the Ant System to Design Combinational Logic Circuits

Benito Mendoza García
MIA, Universidad Veracruzana
Sebastián Camacho 5
Xalapa, Veracruz 91090, MEXICO
bmendoza@uv.mx

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

September 6, 2002

Abstract

In this paper we report the first attempt to design combinational logic circuits using the ant system. In order to design circuits, a measure of quality improvement in partially built circuits is introduced and a cost metric (based on the number of gates) is adopted in order to optimize the feasible circuits generated. The approach is compared to a genetic algorithm and to a human designer using several examples and the sensitivity of the algorithm to its parameters is studied using analysis of variance. The results indicate that the ant system is a viable alternative to design combinational logic circuits.

1 Introduction

Design is a task that normally requires creativity and it is therefore traditionally difficult to automate. In this paper, we deal with a domain well-known by humans: the design of combinational logic circuits. Despite the existence of several standard aids and methodologies for designing combinational circuits, no method exists that allows to produce optimal designs for any arbitrary truth table given.

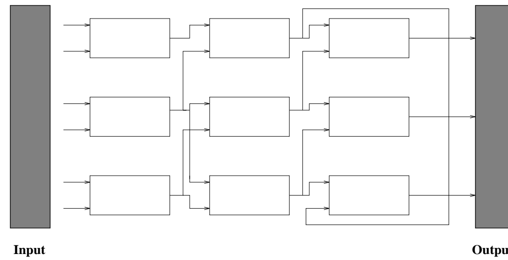

Figure 1: Matrix used to represent a circuit to be processed by an agent (i.e., an ant). Each gate gets its inputs from either of the gates at the previous column.

Several heuristics have been used in the past to design combinational circuits (see for example [9, 2]. However, so far, no one had attempted to use the ant system in this domain. This is perhaps because circuit design is a task that does not have much resemblance with the traveling salesperson problem (TSP) and therefore the application of the ant system algorithm is not straightforward. In this paper we present a methodology that allows to restate the circuit design problem in such a way that the ant system can be applied to its solution. Our results indicate the suitability of the approach, since it produces competitive results (with respect to other heuristics and with respect to human designers), in a resonable amount of time.

2 Description of the Approach

In this section, we will describe the way in which the circuit design problem had to be reformulated in order to be able to use the AS to solve it.[1] The main problem that we faced was how to make an analogy (as much as possible) between circuit design and the TSP. The main issues are: the representation to be adopted, the notion of state in that representation, the way in which a path would be built, and the way of updating the trails of each ant. Each of these issues will be discussed in this section

2.1 Representation

Since we need to view the circuit optimization problem as one in which we want to find the optimal path of a graph, we will use a matrix representation for the circuit as shown in Figure 1. This matrix is encoded as a fixed-length string of integers from 0 to $N-1$, where N refers to the number of rows allowed in the matrix.

[1] We did not use the ant colony system because the problem to be solved does not present all the properties normally associated with the application of such algorithm [5]. In particular, we do know how many states remain to be visited from a certain point in the path that leads to a certain circuit. We do, however, use the exploration mechanism of the ant colony system (i.e., a proportional selection procedure to choose the next state).

Input 1	Input 2	Gate Type

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

More formally, we can say that any circuit can be represented as a bidimensional array of gates $S_{i,j}$, where j indicates the *level* of a gate, so that those gates closer to the inputs have lower values of j. (Level values are incremented from left to right in Figure 1). For a fixed j, the index i varies with respect to the gates that are "next" to each other in the circuit, but without being necessarily connected. Each matrix element is a gate (five types of gates were considered in our work: AND, NOT, OR, XOR and WIRE[2]) that receives its 2 inputs from any gate at the previous column as shown in Figure 1. We have used this representation before with a genetic algorithm (GA) [1, 2].

A chromosomic string encodes the matrix shown in Figure 1 by using triplets in which the 2 first elements refer to each of the inputs used, and the third is the corresponding gate as shown in Figure 2 (only 2-input gates were used in this work).

2.2 Building a path

The path of an ant in our case is a full circuit. In other words, each ant traverses a path and, in the process, it builds a circuit. In the TSP, the ants also traverse a path and try to find the shortest way to the goal. In our case, "shortest" relates to "less gates". However, in the TSP, any permutation is a valid solution, whereas in our case, an arbitrary string encodes a circuit that may or may not be feasible. We only try to minimize the number of gates of feasible circuits.

The aim is to maximize a certain payoff function. Since our code was built upon our previous GA implementation [2], we adopted the use of fixed matrix sizes for all the agents, but this needs not be the case (in fact, we could represent the Boolean expressions directly rather than using a matrix, and other representations are currently a matter of further research). The matrix containing the solution to the problem is built in a column-order fashion as indicated next.

Each state is, in our case, a column of the matrix, which is composed of several elements. A certain state is selected element by element (gate by gate). Each of these column elements is called a substate. A substate is a triplet in which the first two elements refer to each of the inputs used (taken from the previous level or column of the matrix) and the third is the corresponding gate (chosen from AND, OR, NOT, XOR, WIRE) as shown in Figure 2. For the gates at the first level (or column), the possible inputs for each gate were those defined by the truth table given by the user (a modulo function was implemented to allow more rows than available inputs). The gate and inputs to be used for each element of the matrix

[2] WIRE basically indicates a null operation, or in other words, the absence of gate.

are chosen randomly from the set of possible gates and inputs (a modulo function is used when the relationship between inputs and matrix rows is not one-to-one).

The distance (between cities or states), which we denote by h, is measured in our case as the increment or decrement in the fitness value of the circuit when we move from one level to the next. By level, we refer to a column in the matrix. Since our algorithm builds the circuit progressively (starting from the leftmost column), as we move to the right, levels increase and fitness values change. Fitness in this domain is measured according to the amount of hits achieved (i.e., matches between the outputs of the circuit and the outputs defined in the truth table). Feasible circuits get an extra increase in their fitness measured as the amount of WIREs that they contain. This allows us to perform a fair comparison between feasible and infeasible designs (i.e., feasible designs always get a higher reward than infeasible designs).

One important difference between the statement of this problem and the TSP is that in our case not all the states within the path have to be visited, but both problems share the property that the same state is not to be visited more than once (this property is also present in some routing applications [4]).

When we move from one substate to another in the path, a value is assigned to all the substates that have not been visited yet and the next substate (i.e., the next triplet) is randomly selected using a certain selection factor p^k. This selection factor determines the chance of going from state i to state j at the iteration t, and is computed using the following formula that combines the pheromone trail with the heuristic information used by the algorithm:

$$p^k_{i,j,l} = f_{j,l} \times h_{i,j,l} \quad (1)$$

where k refers to the ant whose pheromone we are evaluating (the ant that is building the path), $f_{j,l}$ is the amount of pheromone at state j at row l (this value is initialized to zero), and $h_{i,j,l}$ is the score increment between substate i and substate j for row l (each row is associated with an output in the truth table). This score is measured according to the number of matches between the output produced by the current circuit and the output desired according to the truth table given by the user. The value of $h_{i,j,l}$ is given by the number of hits that the partially-built circuit produces so far with respect to the l outputs of the truth table provided by the user. This value is therefore a score increment analogous to the distance between nodes used in the TSP. Note that the previous transition rule is the same normally adopted with the ant system, but in our case $\alpha = \beta = 1$. Obviously, changing α and β would affect the behaviour of the algorithm, but so far we have not experimented with different values for these parameters.

Once every combination has been assigned a selection factor, we choose one of them. At this point, we apply roulette-wheel selection. We do this for every substate that belongs to one of the rows representing an output of the circuit. The other substates are randomly chosen.

The previous process is repeated until we finish a path (i.e., until we reach the last state of the circuit, or the last column of the matrix).

2.3 Updating the trails

The amount of pheromone is updated each time an agent builds an entire path (i.e., once the whole circuit is built). This is done in two steps:

1. First, we simulate the evaporation of the pheromone trails in all substates, such as they occur with real ants (over time). For the simulation, we adopt the following formula:

$$f_{i,l} = \rho \times f_{i,l} \tag{2}$$

   where $0 < \rho < 1$ ($\rho = 0.5$ was used in all the experiments reported in this paper) is the trail persistence and its use avoids the unlimited accumulation of pheromone in any path, and $f_{i,l}$ is the amount of pheromone at state i at row l.

2. Then, we deposit the pheromone in the substates through which the ants passed, using the following formula:

$$f_{i,l} = f_{i,l} + \sum_{k=1}^{m} f_{i,l}^{k} \tag{3}$$

   where m refers to the number of agents (or ants), $f_{i,l}^{k}$ corresponds to the amount of pheromone deposited by ant k at state i at row l. This value is obtained in the following way:

   - If the circuit is not feasible (i.e., if not all of its outputs match the truth table), then:

$$f_{i,l}^{k} = \text{payoff} \tag{4}$$

   - If the circuit is feasible (i.e., all of its outputs match the truth table), then:

$$f_{i,l}^{k} = \text{payoff} \times 2 \tag{5}$$

   - If it is the circuit with the highest fitness (i.e., the best path found):

$$f_{i,l}^{k} = \text{payoff} \times 3 \tag{6}$$

   - If the ant k did not pass through substate i of row l:

$$f_{i,l}^{k} = 0 \tag{7}$$

The value of payoff is given by the following expression:

$$\text{payoff} = hits + ((Cols \times Rows) - TotCirc) \tag{8}$$

where: $hits$ is the number of matches produced between the outputs generated by the circuit produced by the AS and the truth table given by the user; $Cols$ is the number of columns in the matrix; $Rows$ is the number of rows in the matrix, and $TotCirc$ is the number of gates used by the circuit generated by the AS.

To build a circuit, we start by placing a gate (randomly chosen) at a certain matrix position and we fill up the rest of the matrix using WIREs. This tries to compute the effect produced by a gate used at a certain position (we compute the score corresponding to any partially built circuit). The distance is computed by subtracting the hits obtained at the current level (with respect to the truth table) minus the hits obtained up to the previous level (or column). When we are at the first level, we assume a value of zero for the previous level.

3 Comparison of Results

We used several examples taken from the literature to test our AS implementation. Our results were compared to those obtained by a human designer (using Karnaugh maps plus simplification using Boolean rules) and by a genetic algorithm using binary representation (BGA). In all the examples presented, the matrix used was of size 5×5, and the length of each string representing a circuit was 75. Since 5 gates were allowed in each matrix position, then the size of the intrinsic search space (i.e., the maximum size allowed as a consequence of the representation used) for all of the examples is 5^l, where l refers to the length required to represent a circuit ($l = 75$ in our case). Thefore, the size of the intrinsic search space is $5^{75} \approx 2.6 \times 10^{52}$. Also, the parameters of the BGA were chosen so that they approximated the total number of fitness function evaluations required by the AS[3]. For each of the following examples, we performed 20 runs with each technique.

The experiments described next were performed on a PC with a Pentium III processor (running at 550 Mhz), with 128 Mbytes in RAM and a 13 Gbytes hard disk. Although the original version of our ant system implementation was developed using Borland C++ Builder 4, to allow a direct comparison with the binary genetic algorithm, we migrated the code to GNU C under Red Hat Linux (version 7). Thus, both programs were run on the same hardware and software platform (a single computer with identical workload was used for all the experiments rather than a distributed system, so that time could be measured more accurately).

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

A	**B**	**W**	**X**	**Y**	**Z**
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Table 2: Comparison of the Boolean expressions produced by the AS, a GA with binary representation (BGA), and a human designer for the circuit of the first example.

Human Designer
$X = A'B'$, $Y = A'B$, $Z = AB'$, $W = AB$
6 gates
4 ANDs, 2 NOTs
BGA
$W = (AB)A$, $X = A \oplus (AB)$, $Y = ((A \oplus B) + A) \oplus A$, $Z = ((A \oplus B) + A)'$
7 gates
3 XORs, 1 OR, 2 ANDs, 1 NOT
Ant System
$X = AB \oplus A$, $Y = BA'$, $Z = A' \oplus BA'$, $W = AB$
5 gates
2 XORs, 2 ANDs, 1 NOT

3.1 Example 1

Our first example has 2 inputs and 4 outputs (it is a decoder 2-4) as shown in Table 1. Our AS algorithm used the following parameters: population size = 30, maximum number of iterations = 30, ρ=0.5. The BGA used the following parameters: population size = 200, maximum number of generations = 1000, crossover rate = 0.5, mutation rate = 0.5/L, chromosomic length = 225 bits. In this case, the BGA performed 200,000 fitness function evaluations per run, and the AS performed 185,400 fitness function evaluations per run.

The results for the AS were the following: best and worst fitness = 36, average fitness = 36, standard deviation = 0.0, CPU time = 13 seconds. The results for the BGA are the following: best fitness = 34, worst fitness = 15, average fitness = 28.45, standard deviation = 5.072889, CPU time = 30 seconds. The AS was able to find a solution with a fitness of 36 (i.e., a circuit with 5 gates) in all the runs performed. The best solution that the BGA could find had a fitness of 34 (i.e., a feasible circuit with 7 gates). In 10% of the runs, the BGA converged to an infeasible solution.

The comparison of the Boolean expressions produced by the AS, the BGA, and a human designer are shown in Table 2. It can be clearly seen that the AS produced better solutions than both the human designer and the BGA for this example. However, note in Table 2, that some of the Boolean expressions generated by the BGA can be easily simplified (e.g., $W = (AB)A = AB$). Nevertheless, we were interested in comparing the solutions generated by the AS and the BGA without any extra human intervention.

3.2 Example 2

Our second example has 4 inputs and 1 output, as shown in Table 3. The parameters used by the Ant System and the BGA are the same adopted for the previous example. Again, the BGA performed 200,000 fitness function evaluations per run, and the AS performed 185,400 fitness function evaluations per run.

The results for the AS were the following: best fitness = 34, worst fitness = 32, average fitness = 33.15, standard deviation = 0.5871429, CPU time = 13 seconds. The results for the BGA are the following: best fitness = 34, worst fitness = 13, average fitness = 21.3, standard deviation = 8.3986214, CPU time = 30 seconds. The best solution that the AS could find had a fitness of 34 (i.e., a circuit with 7 gates). In all cases, the AS converged to a feasible circuit and 25% of the time a fitness value of 34 was achieved. The best solution that the BGA could find had a fitness of 34 (i.e., a feasible circuit with 7 gates), but it appeared only once in the 20 runs performed. The BGA converged to an infeasible solution 60% of the time. The comparison of the Boolean expressions produced by the AS, the BGA, and a human designer are shown in Table 4.

---

[3]In this work, the term "fitness function evaluation" refers to a unit used to compare the performance of our algorithm against others. In terms of computational effort, a fitness function evaluation is the amount of time required to evaluate a solution (i.e., a circuit).

Table 3: Truth table for the circuit of the second example.

A	**B**	**C**	**D**	**F**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Table 4: Comparison of the Boolean expressions produced by the AS, a GA with binary representation (BGA), and a human designer for the circuit of the second example.

Human Designer	**BGA**	**Ant System**
$F = (D' + (A \oplus B))$ $((AC)' + (B \oplus D))$ 8 gates 2 XORs, 2 ANDs, 2 ORs, 2 NOTs	$F = (AC + (B + D))' +$ $(AD \oplus B)$ 7 gates 2 ANDs, 3 ORs, 1 XOR, 1 NOT	$F = (AC + D)' +$ $(A \oplus B) \oplus D'$ 7 gates 2 XORs, 1 AND, 2 ORs, 2 NOTs

Table 5: Truth table for the 2-bit multiplier of the third example.

A_1	A_0	B_1	B_0	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

3.3 Example 3

Our third example is the 2-bit multiplier (4 inputs and 4 outputs) whose truth table is shown in Table 5. Our AS algorithm used the following parameters: population size = 30, maximum number of iterations = 30, ρ=0.5. The BGA used the following parameters: population size = 800, maximum number of generations = 1000, crossover rate = 0.5, mutation rate = 0.5/L, chromosomic length = 225 bits. In this case, the BGA performed 800,000 fitness function evaluations per run, and the AS performed 725,400 fitness function evaluations per run.

The results for the AS were the following: best fitness = 82, worst fitness = 81, average fitness = 81.4, standard deviation = 0.50262469, CPU time = 55 seconds. The results for the BGA are the following: best fitness = 80, worst fitness = 62, average fitness = 68.25, standard deviation = 7.731544, CPU time = 253 seconds. The best solution that the AS could find had a fitness of 82 (i.e., a feasible circuit with 7 gates). In all cases, the AS converged to a feasible circuit and 40% of the time a fitness value of 82 was achieved. The best solution that the BGA could find had a fitness of 80 (i.e., a feasible circuit with 9 gates), and it appeared only once in the 20 runs performed. For 55% of the runs, the BGA converged to an infeasible solution. The comparison of the Boolean expressions produced by the AS, the BGA, and a human designer are shown in Table 6. The solution produced by the AS is better (i.e., it uses less gates) than those produced by the human designer and the BGA. In fact, these last two solutions are really the same, although the BGA was not able to eliminate a double NOT in the Boolean expression.

Table 6: Comparison of the Boolean expressions produced by the AS, a GA with binary representation (BGA), and a human designer for the circuit of the third example (a 2-bit multiplier).

Human Designer	BGA	Ant System
$C_0 = A_0B_0$	$C_0 = ((A_0B_0)')'$	$C_0 = A_0B_0$
$C_1 = A_0B_1 \oplus A_1B_0$	$C_1 = A_0B_1 \oplus A_1B_0$	$C_1 = A_1B_0 \oplus A_0B_1$
$C_2 = A_1B_1(A_0B_0)'$	$C_2 = A_1B_1(A_0B_0)'$	$C_2 = A_1A_0B_1B_0 \oplus A_1B_1$
$C_3 = A_1A_0B_1B_0$	$C_3 = A_1A_0B_1B_0$	$C_3 = A_1A_0B_1B_0$
8 gates	9 gates	7 gates
6 ANDs, 1 XORs, 1 NOT	1 XOR, 6 ANDs, 2 NOTs	2 XORs, 5 ANDs

Despite the good results reported here, it is important to clarify that our algorithm presents problems when trying to solve larger circuits. Our current results indicate that there is an important increase in the computational cost when solving larger circuits (mainly because the size of the matrix has to be increased). Also, the quality of the solutions produced tends to decrease as the size of the circuit increases, and our algorithm tends to present solutions with more gates than a genetic algorithm.

4 Sensitivity Analysis

Our approach uses several parameters and, ideally, we would like to know how to setup their values such that we can solve any arbitrary circuit. Trying to deal with this problem, we performed a sensitivity analysis of the most significant parameters adopted designing a factorial experiment which is described next.

4.1 Statement of the Problem

Our application combines several sub-circuits (either gates or portions of circuits) built by the ants. From this, it is obvious that at a larger population size, we will have more opportunities to recombine sub-circuits at each iteration, and therefore we would expect a higher convergence rate. However, a higher population size also implies higher running time for the algorithm. Something similar happens with the number of cycles. A higher number of cycles allows more recombination of sub-circuits, but also requires more running time.

The matrix represents the space where the sub-circuits built are placed. Therefore, if the size of the matrix is increased, we will have more space to place sub-circuits which we hope to be benefitial (i.e., this should increase our convergence rate). However, a larger matrix size also requires that an ant takes longer to build a path. Thus, larger matrix size also implies larger running time for the algorithm.

We can determine, beforehand, that the evaporation factor does not have any

Table 7: Values adopted for the parameters of our algorithm in the sensitivity analysis conducted.

Parameter	**Alias**	**Value 1**	**Value 2**	**Value 3**
Matrix Size	Matrix	8×8	10×10	12×12
Evap. Factor	Evap	0.10	0.5	0.75
Pop. Size	PopSize	10	30	50
# of Iterations	Max_Cic	10	30	50

impact on the running time of the algorithm. However, we consider its effect on the convergence rate of the algorithm.

Based on the previous analysis, we formulated the eight following hypothesis:

1. The size of the matrix has an impact on the convergence rate.
2. The size of the matrix has an impact on the running time of the algorithm.
3. The size of the population has an impact on the convergence rate.
4. The size of the population has an impact on the running time of the algorithm.
5. The number of cycles has an impact on the convergence rate.
6. The number of cycles has an impact on the running time of the algorithm.
7. The evaporation factor has an impact on the convergence rate.
8. The evaporation factor does not have an impact on the running time of the algorithm.

Note that in the analysis proposed, the values of α and β used in the transition rule are not included. This was done to reduce the number of runs to be performed, but their incorporation in the sensitivity analysis of the algorithm is part of our future work.

4.2 Choosing Factors and Levels

The factors are in this case the parameters used by our algorithm. We decided to adopt three sets of values which represent the variability that each parameter may have. The values chosen are shown in Table 7 (the selection was based on our own experience using the algorithm).

4.3 Selection of the Response Variables

In our case, we were interested in knowing how often would our algorithm converge to a feasible solution when adopting a certain combination of parameter values. Additionally, we were interested in the time taken by the algorithm to finish a run.

Furthermore, we also tried to determine the combination of parameters that provided the best convergence rate and the lowest computational cost. Therefore, our response variables are the *convergence rate* and the *average time* required to finish a run.

4.4 Experimental Design

Our experimental design consisted in choosing the circuits with which we would perform our analysis. We chose five circuits that have been adopted by other authors to validate their own algorithms and that are representative of the type of circuits solved with heuristics in the literature. The circuits are the following: Sasao's circuit [11], Katz's circuit with one output [6], a full two-bit adder a full two-bit multiplier, Katz's circuit with three outputs [6]. For each circuit, we performed 40 runs for each combination of parameters (from those indicated in Table 7). Since we have four parameters and three values for each of them, there are $3^4 = 81$ possible combinations of values. Therefore, we have 3240 runs for each circuit (81×40). Since we have five circuits and 3240 runs for each of them, we have a total of 16200 runs (or experiments) to be performed. Since the output of each run depends of the random numbers seed, we adopted a table of random numbers to setup these seeds properly. From each run, we obtained the best solution (or path) found, plus the following: fitness, number of gates if the circuit was feasible or -1 if not, iteration or cycle in which the solution was found, number of violations if the circuit found was infeasible or zero otherwise, time needed to finish a run. Once all the runs were finished, we computed the following: average time required to finish a run, convergence rate (i.e., the percentage of runs that produced a feasible circuit), number of gates of the best solution found after performing 40 runs.

4.5 Statistical Analysis of the Results

In order to study the effect of the values of the parameters on our response variables, we implemented an analysis of variance (ANOVA) [10]. The rationale of the method is that the response variables are modified by the variation of the independent variables (in this case: Matrix, Evap, PopSize and Max_Cic) and their possible interaction or combination.

The behavior of the variable "time" was affected by any increase in Max_Cic, Matrix or PopSize, whereas Evap did not influenced it. Therefore, we calculated the average time for each value of Max_Cic, and for each combination of PopSize and Matrix. Table 8 shows the average time for each of the combinations tried. In general, as the number of cycles increased, the average time required by the algorithm also increased (see Table 8). The same can be observed within each cycle: as the size of the matrix or the size of the population increases, the average time increases as well.

The conclusions of the analysis regarding *time* are then very clear:

- The *evaporation factor* DOES NOT affect the time required by the algorithm to finish a run, such as we expected from our hypothesis 8 previously stated

Table 8: Average time (in seconds) for each of the circuits used in our experiments, using the different combinations of parameters chosen.

Parameter			**Time**		
Comb. No.	**PopSize**	**Matrix**	**Max_Cic = 10**	**Max_Cic = 30**	**Max_Cic = 50**
1	10	8×8	16.00	45.33	76.00
2	10	10×10	45.67	138.00	228.33
3	10	12×12	112.33	337.67	562.33
4	30	8×8	45.67	137.33	228.00
5	30	10×10	137.00	411.33	685.33
6	30	12×12	337.33	1012.33	1687.33
7	50	8×8	76.33	228.67	380.00
9	50	12×12	562.67	1688.33	2813.00
Max_Cic Avg			**173.52**	**520.44**	**866.93**

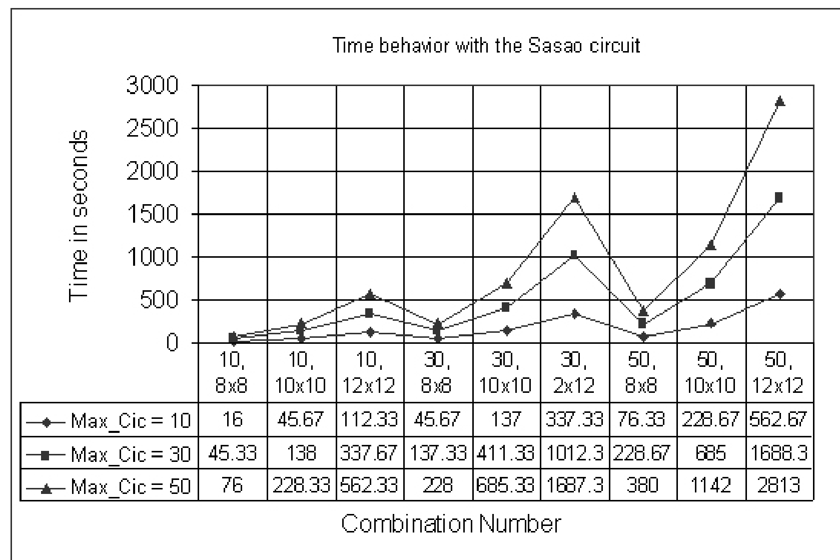

Figure 3: Plot that shows the running times required by our algorithm for different combinations of parameters when using Sasao's circuit.

(see Section 4.1).

- An increase in the *population size* of the algorithm increases the time required to finish a run. This increase is practically the same produced if the *number of iterations* is increased. This is what we expected from our hypothesis 6 and 4 previously stated.

- The *matrix size* adopted has an impact on the time required by the algorithm to finish a run as expected from our hypothesis 2 previously stated.

To analyze the effect of the parameters over the convergence rate of the algorithm, we also performed a variance analysis for each level or value of the variable Max_Cic (10, 30, 50) for the dependent variable *convergence rate* with the independent variables Matrix, PopSize and Evap. Thus, we obtained three analysis per circuit (one for each value of Max_Cic) such that: a) we could prove if the values or levels of the parameters involved and their combination really impacted the average convergence rate, and b) if from the previous point we could find results with an statistical significance, then we would proceed to identify the combination of parameters that provided the highest average convergence rate in the lowest time possible. In other words, we wanted to find the combination of parameters capable of making the algorithm to reach the feasible region with the lowest possible computational cost.

Regarding convergence, the conclusions were the following:

- The matrix size impacts convergence rate as expected from hypothesis 1 (see Section 4.1). This impact was observed both when considered combined with other factors and when considered independently.

- The population size impacts convergence rate. Within the analysis, this factor indicated that an increase in its value was reflected by a higher convergence rate. This was expected from our hypothesis 3 previously stated.

- As expected from hypothesis 5 (Section 4.1), the number of cycles or iterations impacts convergence rate. If this parameter is increased we obtain a higher convergence rate.

- Although not observed in all circuits, the evaporation factor impacts the convergence rate as stated in our hypothesis 7 from Section 4.1. In some circuits, this parameter had a significant impact on convergence independently, but in others only when used with certain matrix sizes.

From our experiments we concluded that the following parameters provided the most consistent results are are, therefore, recommended for any given circuit (with similar characteristics to those presented here) if no further information is available: matrix size = 10×10, Evap = 0.50, Max_Cic = 30, PopSize = 30.

5 Conclusions and Future Work

We have proposed an algorithm based on the ant system which was found effective in the design of combinational logic circuits at a gate-level. Our results indicate that our algorithm is competitive with respect to those produced by human designers and with respect to a genetic algorithm. It should be indicated, however, that the genetic algorithm tends to produce better circuits (i.e., with less gates) in a shorter time than the ant system when dealing with larger combinational circuits (see for example [3, 8]). This indicates that our implementation requires further refinements. One of the possible research paths considered in our current work involves the use of a tree-encoding such as that adopted in genetic programming [7].

We are also considering the introduction of α and β (from the transition rule transition adopted with the ant system) in our sensitivity analysis. Another interesting aspect to analyze in our future work is the role of the heuristic information in the performance of the algorithm.

Acknowledgements

We thank the anonymous reviewers for their comments that greatly helped us to improve the contents of this paper. The second author acknowledges support from the Consejo Nacional de Ciencia y Tecnología (CONACyT) through project number 32999-A.

References

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

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

[3] Carlos A. Coello Coello, Rosa L. Zavala Gutiérrez, Benito Mendoza García, and Arturo Hernández Aguirre. Ant Colony System for the Design of Combinational Logic Circuits. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pages 21–30, Edinburgh, Scotland, April 2000. Springer-Verlag.

[4] G. Di Caro and M. Dorigo. AntNet: Distributed Stigmergetic Control for Communications Networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[5] Marco Dorigo and Luca M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

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

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

[8] Benito Mendoza García and Carlos A. Coello Coello. Uso del Sistema de la Colonia de Hormigas para el Diseño de Circuitos Lógicos Combinatorios. In E. Alba, F. Fernández, J.A. Gómez, F. Herrera, J.I. Hidalgo, J. Lanchares, J.J. Merelo, and J.M. Sánchez, editors, *Primer Congreso Español de Algoritmos Evolutivos y Bioinspirados (AEB'02)*, pages 294–301, Mérida, España, 2002. Universidad de la Extramadura.

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

[10] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, New York, fifth edition, 2000.

[11] Tsutomu Sasao, editor. *Logic Synthesis and Optimization*. Kluwer Academic Press, 1993.