Applying Genetic Algorithms to the State Assignment Problem: A case Study

Jose Nelson Amaral, Kagan Tumer, and Joydeep Ghosh
Department of Electrical and Computer Engineering,
University of Texas at Austin,
Austin, Texas 78712

Abstract

Finding the best state assignment for implementing a synchronous sequential circuit is important for reducing silicon area or chip count in many digital designs. This State Assignment Problem (SAP) belongs to a broader class of combinatorial optimization problems than the well studied traveling salesman problem, which can be formulated as a special case of SAP. The search for a good solution is considerably more involved for the SAP than it is for the traveling salesman problem due to a much larger number of equivalent solutions, and no effective heuristic has been found so far to cater to all types of circuits.

In this paper, a matrix representation is used as the genotype for a Genetic Algorithm (GA) approach to this problem. A novel selection mechanism is introduced, and suitable genetic operators for crossover and mutation, are constructed. The properties of each of these elements of the GA are discussed and an analysis of parameters that influence the algorithm is given. A canonical form for a solution is defined to significantly reduce the search space and number of local minima. Simulation results for scalable examples show that the GA approach yields results that are comparable to those obtained using competing heuristics. Although a GA does not seems to be the tool of choice for use in a sequential Von-Neumann machine, the results obtained are good enough to encourage further research on distributed processing GA machines that can exploit its intrinsic parallelism.

1 INTRODUCTION

The purpose of this study is to investigate the suitability of genetic algorithms for finding *good* solutions for Combinatorial Optimization Problems (COP). A COP is defined as the search for an optimal solution of a problem described with *discrete* variables. The set of feasible solutions is finite—or possibly countably infinite, and is called *the search space*[13]. Some COPs are characterized by having a number of solutions that grows at least exponentially with the size of the problem. These problems are called *intractable*, since the time necessary to find an optimal solution for a given instance of such problems grows combinatorially with the problem size [7]. However, for many practical engineering situations, it is not necessary to obtain the actual *best* solution for the problem. It is sufficient to find a *good* solution that differs only slightly in quality from the best one. This procedure is specially interesting if the tradeoff between the time necessary to find a solution and the cost associated with that search is significantly better that for the optimum solution. In this study the state assignment problem is used as a testbed to investigate the use of Genetic Algorithms to find practical solutions to COPs.

The state assignment problem, which entails the codification of states in a Finite State Machine (FSM), is a well studied NP-complete problem. Micheli *et al* developed a system called KISS (Keep Internal State Simple) at Berkeley [6]. KISS works with symbolic minimization and multivalued logic. A rule-based system called ASYL was developed in France for control logic design [14]. ASYL includes a solution to the state assignment problem. A system for implementation of sequential circuits in Programmable Logic Array was developed by Varma and Trachterberg [15]. In this solution, partition theory and spectral translation techniques were used in the search of a good state assignment. Amaral and Cunha developed an algorithmic solution based on a set of heuristic rules [1]. Thus a wide variety of heuristics based on diverse approaches, are available for this problem. Moreover, several well-known COPs such as the traveling salesman problem, are special cases of the SAP. For these reasons, we use SAP in this paper as a testbed to investigate the use of Genetic Algorithms to find practical solutions to COPs.

In this paper, we present the state assignment problem, and through an example show the importance of proper codification. After reviewing the available heuristics methods to obtain good solutions, we present a GA approach, and propose a set of operators needed for its implementation. The efficacy of these operators over alternative GA formulation is highlighted. Finally, we present the results obtained by the GA and compare it with competing conventional methods.

2 STATE ASSIGNMENT PROBLEM

2.1 Statement of the problem

The behavior of a Synchronous Sequential Circuit (SSC) can be represented by an FSM. In this representation, each state is identified by a symbol, i.e., a string of characters. In the actual implementation of an SSC, the states are represented by bit strings. In the process of realizing an SSC from its FSM specification, it is necessary to *assign* a bit string to each state. The cost of the SSC realization depends heavily on this assignment. The problem of finding the association between states and bit strings that results in minimal cost is called the State Assignment Problem (SAP).

The number of distinct state assignments for a machine with s states, each of which is encoded by $k \geq \lceil log_2 s \rceil$ bits, is given by [10]:

$$N = \frac{(2^k)!}{k!(2^k - s)!} \tag{1}$$

Even for a moderate size problem ($s = 16, k = 4$), the number of distinct assignments is large enough to discourage any attempt at obtaining the solution by enumeration ($N > 10^{11}$).

2.2 A Motivating Example

We begin by presenting an example that illustrates how the state assignment can influence the cost of an SSC. This example will also be used later on to illustrate the genetic algorithm operators. An FSM with five states (S_1, S_2, S_3, S_4, S_5), one input (I_0), and two outputs (Z_0, Z_1), is given in Table 1.

Present State	Next State		Present Output	
	$I_0 = 0$	$I_0 = 1$	Z_0	Z_1
S_0	S_1	S_2	0	0
S_1	S_4	S_3	1	1

Table 1: State table.

Table 2: State Assignments.

Two different state assignments are proposed in Table 2. If the cost to implement an assignment is taken to be the number of inputs to logic gates [3] in the correspondent set of Boolean equations, Assignment 1 has a cost of 33, whereas Assignment 2 has a cost of 11. This example illustrates that choosing an appropriate state assignment greatly reduces the cost of implementation.

2.3 Heuristic Rules

Given an FSM specification, determining the state assignments leading to an SSC implementation with minimum cost is a non-trivial problem. A set of heuristic rules compiled along the years, have been proven to lead to good SSC implementations for many designs [2, 4]. Before presenting these rules, some definitions are necessary:

The bit string assigned to state S_i is called the *attribution* of state S_i and is denoted by $A(S_i)$.
A state S_i is called a *successor* of a state S_k if there is a transition from state S_k to state S_i. The set of all successors of a state S_k, is denoted by $S(S_k)$.
A state S_i is called a *predecessor* of a state S_k if there is a transition from state S_i to state S_k. The set of all predecessors of a state S_i with a given input condition, is denoted by $P(S_i, < input\ condition >)$.

Each output is said to partition the states of an FSM into two subsets. The set of partitions of an output Z_i is denoted by $O(Z_i)$.
States S_i and S_j are said to be *associated* with each other if both of them are a successor of a given state S_k, if both of them are in the set of predecessors of a state S_l with a given input condition, or if both of them are in the same partition of an output Z_m.
The *distance* between two states S_i and S_k is defined as the Hamming distance between $A(S_i)$ and $A(S_k)$, and is denoted by $D(S_i, S_k)$.

According to the heuristic rules, the cost of the SSC will be minimized when the state assignment is done in a way that minimizes the distance between states that:

i. are in the same set of successors of a given state;
ii. are in the same set of predecessors of a given state with a given input condition; or
iii. are in the same partition for a given output.

Returning to the FSM used in the previous section, we have: $S(S_0) := \{S_1, S_2\}$; $S(S_1) := \{S_3, S_4\}$; $S(S_2) := \{S_3, S_4\}$; $S(S_3) := \{S_4\}$; $S(S_4) := \{S_0\}$; $P(S_4, I_0 = 0) := \{S_1, S_2, S_3\}$; $P(S_3, I_0 = 1) := \{S_1, S_2\}$; $O(Z_0) := \{(S_1, S_2); (S_0, S_3, S_4)\}$; $O(Z_1) := \{(S_1, S_3); (S_0, S_2, S_4)\}$.

Observe that the pairs of states (S_1, S_2), (S_3, S_4) and (S_0, S_4) are associated with each other more frequently than other pairs. Therefore, in a good state assignment for the FSM in Table 1, the Hamming distance between these states should be small. Indeed the Assignment # 1 of Table 2 has $D(S_1, S_2) = 3$, $D(S_3, S_4) = 2$, and $D(S_0, S_4) = 3$; while Assignment #2 has $D(S_1, S_2) = 1$, $D(S_3, S_4) = 1$, and $D(S_0, S_4) = 1$. Clearly Assignment 2 achieves this task while Assignment 1 does not, explaining the significant difference in the respective SSC costs.

2.4 Desired Adjacency Graph

Based on a paper by Armstrong [2], Amaral introduced the Desired Adjacency Graph (DAG) as a tool for applying heuristic rules to any given FSM [1]. The DAG is a undirected, weighted, fully connected graph that has as its nodes the states of the FSM. The weight on an arc connecting two nodes of the DAG represents the strength of that connection, and indicates the "desirability" of having these states "close" to each other in the SSC implementation. To have a low cost SSC, it is necessary to minimize the distance between states that are strongly connected in the DAG. The connection between state i and state j in the DAG is given by the multi-objective function expressed in equation 2.

$$\begin{aligned} DAG_{ij} = DAG_{ji} &= R_1 \sum_{i=0}^{s-1}\sum_{j=0}^{s-1}\sum_{k=0}^{s-1} \alpha_{jk,i}\,\delta_{jk} + R_2 \sum_{\alpha=0}^{c-1}\sum_{i=0}^{s-1}\sum_{j=0}^{s-1}\sum_{k=j}^{s-1} \beta_{jk,i,a}\,\delta_{jk} \\ &+ R_3 \sum_{l=0}^{v-1}\sum_{i=0}^{s-1}\sum_{j=0}^{s-1} \gamma_{ij,l}\,\delta_{ij} + R_4 \sum_{i=0}^{s-1}\sum_{j=0}^{s-1} \psi_{ij}\,\delta_{ij}, \end{aligned}$$

where c is the number of input conditions, v is the number of output variables, s is the number of states, and

$$\alpha_{jk,i} = \begin{cases} 1 & \text{if } S_j \in S(S_i) \text{ and } S_k \in S(S_i) \\ 0 & \text{otherwise} \end{cases}$$

$$\beta_{jk,i,a} = \begin{cases} 1 & \text{if } S_j \in P(S_i, I_a) \text{ and } S_k \in P(S_i, I_a) \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{ij,l} = \begin{cases} 1 & \text{if } Z_j(S_i) = Z_l(S_j) \\ 0 & \text{otherwise} \end{cases}$$

$$\psi_{ij} = \begin{cases} 1 & \text{if } S_j \in S(S_i) \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_{ij} = \begin{cases} 1 & \text{if } j \neq k \\ 0 & \text{if } j = k \end{cases}$$

The first term of equation 2 sums all pairs of states that are common successors to a given state (rule i). The second term sums all pairs of states that have a common predecessor with a given input condition (rule ii). The third term sums the pairs of states that are in the same output partition for a given output (rule iii). The last term sums the number of transitions between two states, and is used as a tie breaker when the previous terms fail to indicate the relative position of each state. Since the DAG is an undirected and fully connected graph, the values of its connections might be represented by a symmetric square matrix. The coefficients R_i are constants which are set according to the importance of each individual rule. In this study $R_1 = 3$, $R_2 = 4$, $R_3 = 2$, and $R_4 = 1$ were used. Table 3 shows the matrix representation of the connections in the DAG obtained for the FSM of Table 1, using equation 2.

	S_0	S_1	S_2	S_3	S_4
S_0	0	5	1	4	2
S_1	5	0	15	9	1
S_2	1	15	0	5	1
S_3	4	9	5	0	8
S_4	2	1	1	8	0

Table 3: Matrix of DAG Connections.

The SSC cost is lowered when two states with strong connections in the DAG are close to each other. Thus, if the DAG_{ik} is large, $D(S_i, S_k)$ should be small. Given an FSM specification and a state assignment, it is possible to quantify the "fitness" of this specific assignment. The *fitness* function which achieves this is given by:

$$FITNESS = \sum_{i=0}^{s-1} \sum_{j=0}^{s-1} (k + 1 - D(S_i, S_j)) DAG_{ij} \tag{2}$$

where k is the number of bits used for the state codification. A state assignment with maximum fitness results in an SSC with minimum cost. From now on we shall assume that the DAG is given and our task is to find an assignment that maximizes the fitness.

2.5 Comparison of SAP with TSP

There are some similarities between the State Assignment Problem (SAP) as formulated in this paper and the well-studied Traveling Salesman Problem (TSP) [11]. Just as the goal of the TSP is to find a path that minimizes the traveled distance, the goal of the SAP is to find an implementation that minimizes the implementation cost. The equivalent of the distance between two cities in the TSP is the connection value of a DAG arc in the SAP. However in the TSP only the distance between adjacent cities in a tour is computed to form the fitness of a given path. In the SAP the connection of each state with *all* other states must be weighted by the corresponding distances between these states, and then summed to form the fitness. In other words, the TSP can be seen as a special case of the SAP where the Hamming distance is reduced to a binary function whose value is one if the states are adjacent, and zero otherwise.

In the TSP, when the position of two cities on a tour are swapped, only the connections with their neighbors change. Defining this property as *locality*, we observe that the SAP does not have locality: if two state assignments are swapped, their connections with all other states are affected.

The fitness function used for an n-city TSP is the length of the tour. Due to the symmetry of this function, for each path there are at least $2n - 1$ other paths with the same length. In the SAP the symmetry of the fitness function results in $2^k k!$ equivalent solutions, where k is the number of bits used in the SSC. This large number of equivalent solutions results in a search space with many local minima. Therefore, it is harder to find a good solution for the SAP then for the TSP.

3 Genetic Algorithms

3.1 Introduction

Algorithmic approaches to COPs can have a constructive approach, an improvement approach, or a combination of both [11]. A Genetic Algorithm (GA) can be classified as an improvement type algorithm. It starts with a *population* of randomly generated *individuals* (solutions to the problem), from which individuals are selected for the application of a *crossover* operator. Given two *parents* (selected individuals), a crossover generates an *offspring*. A *mutate* operator introduces some random information in the offspring which is then inserted back into the population. When the population reaches a given size, usually twice that of the initial one, one *generation* is completed. A *selection* procedure is then used to reduce the size of the population, typically to its original size, and a new generation starts. All the selections are done in a probabilistic fashion and according to the *fitness* of each individual. A good introduction to GAs and their applications is provided in [8], and the use of a GA at a meta-level to obtain control parameters for another GA is explored in [9]. Some considerations made by Whitley for the TSP are also valid for the SAP [18]:

1. The crossover operator should preserve as much information from the parents as possible. Moreover, the solutions generated by crossover should be valid. Typically an additional procedure is needed, and care should be taken to ensure that information not present in either parent is not introduced at this stage.
2. In the TSP the specific position of a given city in the tour does not matter. What matters is sequence in which the cities are visited. Similarly, in the SAP the particular attribution of a given state is irrelevant to the fitness function. Rather, the fitness is determined by the Hamming distance among state attributions. Thus the crossover operator should preserve the distance among attributions in the parents, rather than any particular value of attribution.
3. The mutate operator must be able to introduce enough random information to enable search over the entire solution space. In other words, the probability of reaching any solution must be non-zero.
4. The probability of selecting a given individual as a parent for the crossover operator must be proportional to its fitness.

3.2 Genetic Algorithm applied to SAP

To apply a GA to a problem like the SAP, it is necessary to precisely define an individual, a fitness function, a mutate operator, a crossover operator, and a selection procedure.

3.2.1 Genotype

The "genotype" of a problem is the representation of an individual in the GA. De Jong and Spears have some considerations about the difficulty in finding a suitable genotype for a given problem [5]. They speculate that if there is a fairly natural mapping of the problem to GAs, robust performance might be achieved. GAs have been applied to many problems, where each individual was represented by a single bit string encoding a solution. This is also the way GAs have been presented by Goldberg [8]. However, for certain problems, a matrix representation is more suitable than a bit-string representation in terms of both naturalness and quality of results [16]. For the SAP, if the individual representation contains the underlined structure of a solution, i.e. represents clearly each state attribution and the distance among them, it is easier to define genetic operators and compute the fitness function. In this sense, a binary matrix is a very natural and suitable mapping for an individual in the SAP. In this study an individual will be represented by a binary matrix with s rows and k columns, where s is the number of states in the FSM and k is the number of bits used in the SSC, thus $k \geq \lceil log_2 s \rceil$. Assignments in Table 2 constitute representations of individuals.

3.2.2 Selection

A selection mechanism is necessary to select the individuals that will generate offsprings, and also to select the individuals that will *survive* to the next generation. In this study the roulette wheel method was chosen. In the method presented in [8], the probability of selecting a given individual is given by its fitness divided by the "length" of the roulette wheel—the length of the wheel is the sum of the fitnesses of all individuals. However, for some problems, the fitness varies in a narrow interval, with a large offset. Therefore, if the very same method is used, the selectiveness of the roulette wheel becomes very poor. For instance, an FSM used in our tests has individual fitnesses within the interval $[47694, 53346]$. Using the simple roulette wheel selection, the probability of selecting the best individual would be just 1.12 times the probability of selecting the worst one. To get around this problem, the actual value used in building the roulette wheel is given by:

$$Roul\_Wheel\_Fitness(I_k) = Fitness(I_k) - (q+1)Fitness(Min) + qFitness(Max) \tag{3}$$

where $Roul\_Wheel\_Fitness(I_k)$ is the fitness used in the roulette wheel for the individual I_k, $Fitness(I_k)$ is the actual fitness of the individual I_k, $Fitness(Min)$ is the fitness of the worst individual in the population and $Fitness(Max)$ is the fitness of the best individual in the population. The constant q is arbitrary, and is used to define the *selectiveness* of the roulette. The relationship between the probability of choosing the best individual $P(best)$ and the probability of chosing the worst individual $P(worst)$ is given by:

$$P(worst) = \frac{q}{q+1}P(best). \tag{4}$$

If q is zero, the probability of selecting the worst individual is reduced to zero. This is not recommended in terms of genetic procedures, where the probability of selecting any individual should be strictly positive. In this study, $q = 0.01$ is used, which makes the best individual two orders of magnitude more likely to be selected then the worst one. A final observation in this modification to the roulette wheel procedure is that the distribution of the individuals in the roulette is still proportional to their fitness, and the selectiveness of the roulette is independent of the particular population. However, this procedure does not work properly in a completely homogeneous population because this causes $Roul\_Wheel\_Fitness(I_k) = 0$ for all I_k.

3.2.3 Crossover

As pointed out in section 3.1, an important characteristic of a crossover operator is that it should preserve as much information as possible from the parents while creating an offspring. Whitley et al. devised a crossover operator for the TSP that generates only legal tours, and preserves connections among cities [18]. Defining edges as the connections between the cities, Whitley argues that operators that break fewer edges are more successful in finding good solutions. To design an operator for the SAP, it is necessary to find a parameter that influences the fitness value and can be manipulated easily. Examining equation 2, one can notice that the fitness depends on the Hamming distance between the state attributions, and the DAG. The DAG is equivalent to the distance map in the TSP, and is fixed for a given FSM. Therefore the fitness of a particular individual will be determined by the $D(S_i, S_j)$s, the Hamming distances among states. The Hamming distance between two bit strings is the sum of the Hamming distances between individual bits that form the string. With the genotype defined in section 3.2.1, the fitness function might be considered the sum of contributions from each binary matrix column. Therefore, if *bit columns* from the parents are preserved, the information relevant to the fitness is preserved.

The crossover operator suitable for the SAP consists of randomly selecting columns from the parents in order to create an offspring. This selection is done by independent flippings of a fair coin wherein a column is selected from the first parent if the outcome is a head and from the second parent otherwise. The result might be an invalid solution in case of *conflicts* among states. In this case the offspring generated is converted into a valid solution by restoring the information that came from the parent with better fitness. These conflicts do not occur very often, and usually can be eliminated with few changes in the offspring.

In the example presented in Table 4, the first two columns are taken from parent #1 and the third column is taken from parent #2. This yields a transition solution which is invalid because states S_2 and S_3 have the same attribution. To resolve the conflict, the attribution of state S_3 is changed in such a way that preserves the first two columns, taken from parent #1. Parent #1 was assumed to have a better fitness in this example.

	Parent # 1	Parent # 2	transition	Offspring
S_0	0 0 0	0 0 1	0 0 1	0 0 1
S_1	1 0 0	0 1 1	1 0 1	1 0 1
S_2	0 1 0	0 1 0	0 1 0	0 1 0
S_3	0 1 1	1 1 0	0 1 0	0 1 1
S_4	0 0 1	1 0 0	0 0 0	0 0 0
S_5	1 1 0	1 0 1	1 1 1	1 1 1

Table 4: Crossover example.

Since the "correction" is made by taking information from one of the parents, new information is not introduced by the crossover operator. Also, it is always possible to resolve collisions by deciding in favor of the parent with better fitness. This is because the number of states with a given combination of bits in the columns taken from a parent cannot be greater then the possible number of combinations in these columns. A final consideration about this crossover operator is that by preserving the information from the strongest parent, it benefits the survivability of better characteristics in the population.

3.2.4 Mutation

Since the crossover operator preserves information existing in the parents, if it is used all by itself, it will hinder the emergence of new traits and the diversity of the population will vanish. Only patterns present in the current population will be passed on to the next generation and the GA will be heavily biased by the initial population. The search will not encompass the entire solution space and the probability of finding a good solution will be limited. It is therefore necessary to introduce some random information in the offsprings generated by crossover. This random information is introduced by a mutation operator. Two properties are desirable in this operator:

- Given any individual, it must be possible to obtain any other individual within the solution space by a finite number of successive applications of the mutate operator.
- There must be a way of controlling the amount of random information introduced by the mutation operator.

These properties guarantee that with a minimum amount of random information, it is possible to reach all the states in the solution space.

Given two patterns of bits P_l and P_m, and two states S_i and S_j, such that $A(S_i) = P_l$ and $A(S_j) = P_m$, a *swapping* operation between P_l and P_m results in $A(S_i) = P_m$ and $A(S_j) = P_l$[1].

The mutate operator created for the SAP works by applying a sequence of swapping operations to the state assignment. The mutation rate controls the number of operations to be applied and in this way controls the amount of random information introduced into an individual.

In the example of Table 5 two swapping operations are performed during mutation. The first one swaps the states with assignments 110 and 010, changing assignments of states S_1 and S_2. The second swapping is between the patterns 011 and 111, changing the assignments of states S_3 and S_5.

The mutation operator defined above fulfills the two properties stated earlier. It works by "breaking edges" in a well-controlled fashion in the individuals obtained by crossover. All solutions are reachable by this operator because given an assignment, a finite number of single swapping operations can transform it to any other assignment in the solution space. Finally, the result of the application of this operator is always a valid assignment.

[1] In the case that one of the patterns is not assigned to any state, the swapping is reduced to a change in the assignment of a single state. If both patterns P_l and P_m are not used, the swapping has no effect on the assignment.

	Before mutation	After mutation
S_0	0 0 1	0 0 1
S_1	1 1 0	0 1 0
S_2	0 1 0	1 1 0
S_3	0 1 1	1 1 1
S_4	0 0 0	0 0 0
S_5	1 1 1	0 1 1

Table 5: Mutation example.

3.3 Reducing the solution space

Two state assignments for an FSM are said to be *equivalent* if one can be obtained from the other by a finite sequence of column complement and column permutation operations [12, 17, 10]. Due to the symmetry of the fitness function, given an FSM, for each state assignment, there are $2^k k!$ equivalent assignments, where k is the number of bits used in the SSC[2].

Reducing the solution space improves the probability of getting a good solution in a smaller number of generations. This reduction of space is accomplished in the SAP by expressing each individual in a canonical form. This procedure is applied to the individuals generated randomly for the first generation, as well as to those obtained through crossover and mutation in the subsequent generations.

To specify the canonical form, a weight function is defined for each column of the binary matrix that represents an individual. Let B_{ij} be the bit value of $A(S_i)$ in column C_j. Then, we define

$$Weight(C_j) = \sum_{i=0}^{s-1} B_{ij} 2^i \tag{5}$$

	Original Individual	After complement	After permutation
S_0	0 0 1	0 0 0	0 0 0
S_1	0 1 0	0 1 1	1 1 0
S_2	1 1 0	1 1 1	1 1 1
S_3	1 1 1	1 1 0	1 0 1
S_4	0 0 0	0 0 1	0 1 0
S_5	0 1 1	0 1 0	1 0 0

Table 6: Reduction in solution space.

The canonical form is defined by having $A(S_0) = 0$ and all columns C_j fully ordered in descending order of $Weight(C_j)$. Any arbitrary solution can be reduced to the canonical form by complementing and permuting columns. The complement operations reduce the solution space by a factor of 2^s, and the permutations reduce it by another factor of $s!$. An example of these operations is presented in Table 6. Assuming the columns of the binary matrix are numbered as C_0, C_1, and C_2 from left to right, the column C_2 is complemented to enforce that $A(S_0) = 0$. After this operation, $Weight(C_0) = 12$, $Weight(C_1) = 46$, $Weight(C_2) = 14$. To enforce descending order C_0 is permuted with C_1, and subsequently C_1 is permuted with C_2.

Two distinct solutions in canonical form are *nonequivalent* and cannot be reduced to each other while preserving distance among its states. The solution space reduced in this way contains at least one solution with the same fitness as any other solution in the actual state space. Therefore the reduced space contains the absolute optimum solution.

[2] Actually the column complement operation affects the cost of the SSC. However the Fitness function defined by equation 2 is insensitive to this effect. For more details on equivalent assignments see [10].

4 EXPERIMENTAL RESULTS

The Genetic Algorithm described in this paper was tested against FSM specifications with 32, 33 and 64 states. Different degrees of connectivity among states were used. Some regular structures were developed to enable the prediction of an assignment sufficiently close to the best one. For comparison purposes, an assignment using an heuristic algorithm previously developed by Amaral was also performed [1], and the fitness of that solution was considered to be unity. For machines with small number of states, the algorithm invariably finds a solution that is optimal or close to optimal. To allow comparisons between different machines and obtain a relative measure, the fitness computed by the GA was normalized using equation 6. MIN is the worst solution ever obtained by random search. MAX is the solution obtained by the heuristic algorithm mentioned above.

$$Norm\_Fitness = \frac{Fitness - MIN}{MAX - MIN}. \tag{6}$$

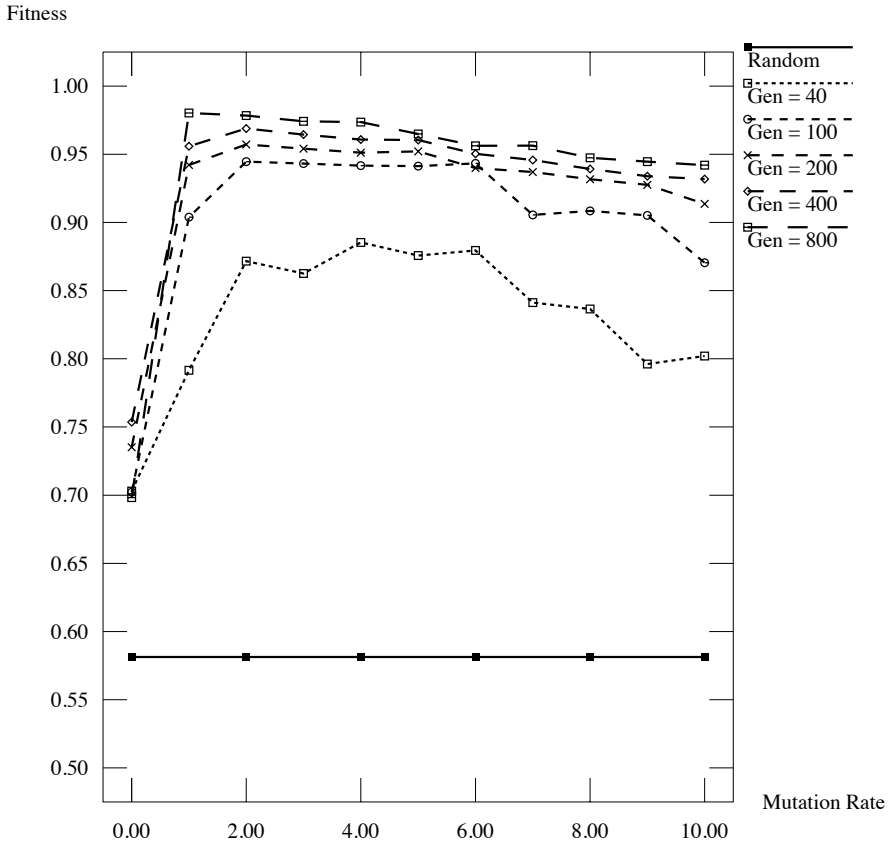

Figure 1: Effect of the number of generations on solution quality (Population = 100).

The first set of tests fixed the population to 100 individuals and varied the number of generations and the mutation rate. The results are plotted in Figure 1. The test was repeated with a smaller population (40 individuals) and the results are presented in Figure 2. In Figure 3, the number of generations is fixed at 40 and the population size is changed. These tests were done with a 32 state FSM, using five bits to code each state.

The results of the experiment presented in Figure 1 show that GAs produce results comparable to those obtained by the heuristic algorithm, if both the number of generations and the population size are properly chosen. The results

obtained are clearly better than the ones given by random search (flat line in the graph). Also when the solution approaches the best one, even significant increases in the number of generations have little effect in the quality of the results.

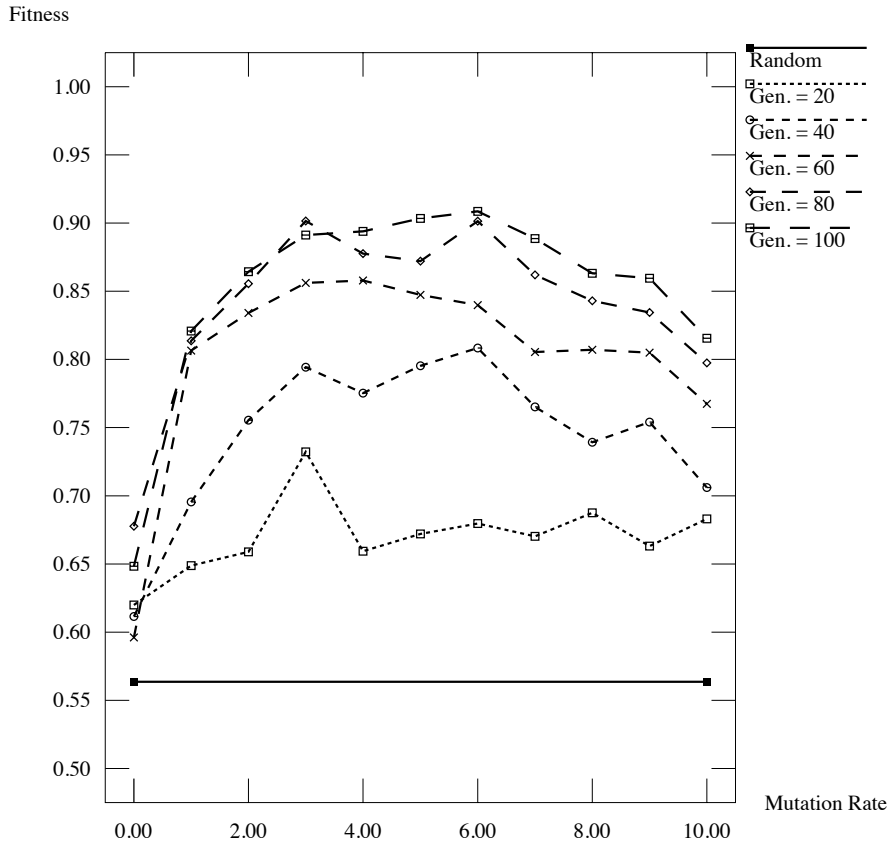

Figure 2: Effect of the number of generations on solution quality (Population = 40).

Figure 2 presents results obtained for a reduced population size and fewer generations. The results are still better than a random search, but clearly not as good as those previously presented. The results also appear to be more sensitive to variations in the mutation rate.

Figure 3 shows how the population size can influence the quality of the solutions. Increasing the population size compensates for the reduction in the number of generations. However further tests showed that this compensation is not linear. If the number of generations is too small, even a very large population will not be able to get the same result quality as that of a GA with more generations.

When working with 33 and 64 state machines it was observed that the quality of the solutions deteriorates with the number of bits used to code the states. We believe that this is due to the fact that the number of bits determines the size of the search space. Variations in the number of states does not seems to have a big impact on the results as long as the number of bits needed for codification remains constant.

The mutation rate producing the best results seems not to change significantly with the definition of the FSM or its number of states. If a sufficient number of generations and a large enough population is used, the result of the GA is close to the best one for a reasonably large range of mutation rate. This is a salient feature since it removes the need for tuning the algorithm for each new FSM design.

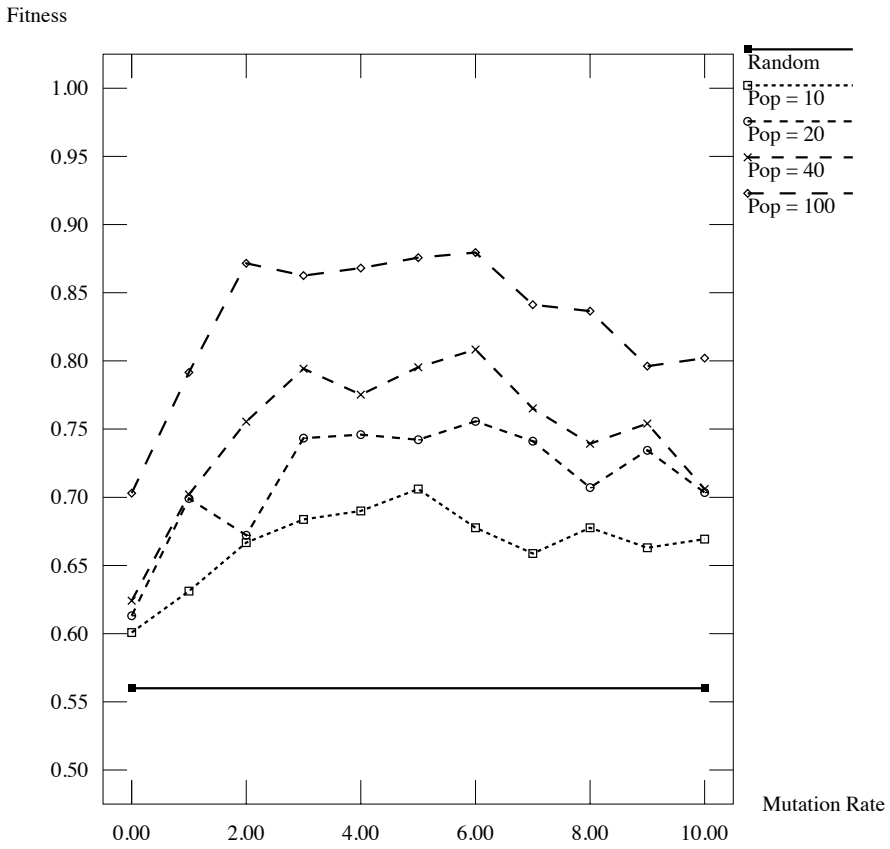

Figure 3: Effect of the population size on solution quality (40 Generations).

5 CONCLUSIONS

This paper explored the use of Genetic Algorithms for the solution of Combinatorial Optimization Problems. The research highlights the importance of having good insights into a problem before defining an adequate genotype, and designing the operators necessary to apply a GA. By using a natural matrix representation and state space reduction techniques, the GA proposed in this paper consistently outperformed alternative GA formulations. From our experiments with various FSMs of different size, it seems that the optimum parameters of this GA do not change much for different FSM specifications. However, it took extensive simulations and "educated guesses" to find an adequate set of parameters for SAP.

Although good results were obtained for practical size State Assignment Problems, GAs still do not seem to be the tool of choice for these kinds of problems. It is true for the SAP, where the GA does not yield better solutions than the competing heuristic approach. However, the results are good enough to encourage further research in distributed processing GA machines. The GA technique has intrinsic parallelism in the generation and selection of offsprings. The speed gained with parallel processing might be used to further improve the results, or to tackle larger problems that at this stage appear out of range.

Acknowledgements: This research is supported in part by Conselho Nacional de Pesquisa Cientifica e Tecnologica (CNPq), and Pontificia Universidade Catolica do Rio Grande do Sul (PUCRS) - Brazil, by an NSF Initiation grant MIP 9011-787, and by a Faculty Development Award from TRW Foundation.

References

[1] J. N. Amaral and W. C. Cunha. State assignment algorithm for incompletely specified finite state machines. In *Fitfth Congress of the Brazilian Society of Microelectronics*, pages 174–183, July 1990.

[2] D. B. Armstrong. A programmed algorithm for assigning internal codes to sequential machines. *IRE Transactions on Electronic Computers*, pages 466–472, August 1962.

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

[4] D. J. Comer. *Digital Logic and State Machine Design*. CBS College Publishing, New York, 1984.

[5] K. A. De Jong and W. M. Spears. Using genetic algorithms to solve np-complete problems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 124–132, 1989.

[6] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Trans. Comp.-Aided Design*, pages 269–284, July 1985.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-completeness*. W. H. Freeman, San Francisco, 1979.

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

[9] J. Grefenstette. Optimization of control parameters for genetic algorithms. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 122–128, Jan/Feb 1986.

[10] M. A. Harrison. On equivalence of state assignments. *IEEE Transactions on Computers*, C-17:55–57, January 1968.

[11] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shimoys. *The Travelling-Salesman Problem*. John Wiley & Sons, Chichester, 1985.

[12] E. J. McCluskey and S. H. Unger. A note on the number of internal variable assignments for sequential switching circuits. *IRE Transactions Electronics Computers*, pages 439–440, December 1959.

[13] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[14] G. Saucier, M. C. Paulet, and P. Sicard. Asyl: A rule-based system for controller synthesis. *IEEE Trans. Comp.-Aided Design*, pages 1088–1097, November 1987.

[15] D. Varma and E. A. Trachtenberg. A fast algorithm for the optimal state assignment of large finite state machines. In *International Conference on Computer-Aided Design*, pages 152–155, 1988.

[16] G. A. Vignaux and Z. Michalewicz. A genetic algorithm for the linear transportation problem. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 445–452, Jan/Feb 1991.

[17] F. R. Weiner and E. J. Smith. On the number of distinct state assignments for synchronous sequential machines. *IEEE Transactions on Electronics Computers*, EC-16:220–221, April 1967.

[18] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 133–140, 1989.