Designing Genetic Algorithms for the State Assignment Problem

José Nelson Amaral, Kagan Tumer and Joydeep Ghosh

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 involved for the SAP due to a large 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. Experiments with several examples show that the GA approach yields results that are often comparable to, or better than those obtained using established heuristics that embody extensive domain knowledge.

Keywords— **Genetic Algorithms, State Assignment, Heuristic Search, Multi-Cost Functions.**

I. Introduction

This study investigates the suitability of genetic algorithms for finding *good* solutions for Combinatorial Optimization Problems (COPs) [8], selecting the state assignment problem (SAP) as a case study. The SAP entails the codification of states in a Finite State Machine (FSM), and is a well studied NP-complete problem. Moreover, several well-known COPs such as the traveling salesman problem, can be formulated as special cases of the SAP. For these reasons, we use SAP in this paper as a testbed to investigate the optimization capabilities of Genetic Algorithms.

A wide variety of heuristics are available for the SAP. For example, KISS (Keep Internal State Simple) works with symbolic minimization and multivalued logic [4]. Varma and Trachterberg [14] use partition theory and spectral translation techniques to search for a good state assignment. Amaral and Cunha developed an algorithm that uses a weighted graph to sort a set of heuristic rules [1]. Villa and Sangiovanni-Vincentelli created algorithms for state assignment based on the solution of face hypercube and ordered face hypercube embedding [16]. Devadas and Newton introduce an "exact" algorithm for encoding problems that obtains minimum number of product terms in an optimized PLA implementation [7]. Several related research in the areas of multilevel implementations, testable machines, FSM decomposition, FSM verification, use of signal transition graph for asynchronous circuit synthesis, and logic minimization have been published recently [5], [11], [12].

Manuscript received August XX, 1993.

The authors are with the Department of Electrical and Computer Engineering at the University of Texas, Austin, TX 78712.

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.

In this paper, we present the state assignment problem, and through an example show the importance of proper codification. A GA approach is then proposed along with a set of operators needed for its implementation. The selection of these operators is crucial for the efficiency of the algorithm. Finally, we present the results obtained by the GA and compare them with established methods.

II. State Assignment 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).

A. 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 I.

TABLE I
State Table with State Assignments.

Present State	Next State		Output		Asgn # 1	Asgn # 2
	$I_0 = 0$	$I_0 = 1$	Z_0	Z_1		
S_0	S_1	S_2	0	0	000	000
S_1	S_4	S_3	1	1	100	100
S_2	S_4	S_3	1	0	011	110
S_3	S_4	S_4	0	1	010	011
S_4	S_0	S_0	0	0	111	010

Definition 1: A **literal** is a boolean variable or its complement.

The cost of a boolean equation B_i represented in sum of product form, is given by equation 1.

$$C(B_i) = \sum_{j=0}^{k-1} P_j(B_i) + O(B_i), \tag{1}$$

where k is the number of product terms in the equation, and with $m > 1$:

$$P_j(B_i) = \begin{cases} m & \text{if the } j^{th} \text{ term of } B_i \text{ has } m \text{ literals} \\ 0 & \text{if the } j^{th} \text{ term of } B_i \text{ has 1 literal,} \end{cases}$$

$$O(B_i) = \begin{cases} m & \text{if } B_i \text{ has } m \text{ terms} \\ 0 & \text{if } B_i \text{ has 1 term.} \end{cases}$$

Given a set of boolean equations $S = \{B_0, B_1, ..., B_{n-1}\}$, its cost is computed by equation 2.

$$C(S) = \sum_{i=0}^{n-1} \left[C(B_i) - \sum_{j=0}^{k_i-1} R_j(B_i) \right], \tag{2}$$

where $C(B_i)$ is the cost of equation B_i, k_i is the number of product terms in equation B_i, and with $(l < i)$:

$$R_j(B_i) = \begin{cases} P_j(B_i) & \text{if the } j^{th} \text{ term of } B_i \text{ is in } B_l \\ 0 & \text{otherwise.} \end{cases}$$

Definition 2: The cost of an SSC is equal to the cost of the set of boolean equations formed by the equations that generate the next state plus the equations that generate the output value in the current state.

In the example presented, for the two different state assignments proposed in Table I, Assignment 1 has a cost of 31, whereas Assignment 2 has a cost of 13. This example illustrates that choosing an appropriate state assignment greatly reduces the cost of implementation.

B. 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], [3]. Before presenting these rules, some definitions are necessary:

Definition 3: The bit string assigned to state S_i is called the **attribution** of state S_i and is denoted by $A(S_i)$. In an FSM with p input signals, there are $c = 2^p$ **input conditions**. An input condition I_a is a binary representation formed by p bits. Each bit indicates the state of one of the input signals.

Definition 4: 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 $Suc(S_k)$. 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 for any input condition I_a. The set of all predecessors of a state S_i with a given input condition I_a, is denoted by $Pred(S_i, I_a)$. Each output of an FSM is said to partition the states of the FSM into two subsets. The set of partitions of an output Z_k is denoted by $O(Z_k)$. For Moore machines an output is denoted by $Z_k(S_i)$, in a Mealy machine the output is denoted by $Z_k(S_i, I_a)$[1]. 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 I_a, or if both of them are in the same partition of an output Z_m.

[1]In a machine type Moore the output is a function of the state, while in a machine type Mealy the output is also dependent on the values of the input.

Definition 5: 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 $D(S_i, S_k)$.

The heuristic rules used in this research state that the cost of the SSC is reduced 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 last section, we have:

$Suc(S_0) := \{S_1, S_2\}$; $Suc(S_1) := \{S_3, S_4\}$;
$Suc(S_2) := \{S_3, S_4\}$; $Suc(S_3) := \{S_4\}$; $Suc(S_4) := \{S_0\}$;
$Pred(S_4, I_0 = 0) := \{S_1, S_2, S_3\}$;
$Pred(S_3, I_0 = 1) := \{S_1, S_2\}$;
$O(Z_0) := \{(S_1, S_2); (S_0, S_3, S_4)\}$;
$O(Z_1) := \{(S_1, S_3); (S_0, S_2, S_4)\}$.

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 I, the Hamming distance between these states should be small. Indeed the Assignment # 1 of Table I 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.

C. 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 an 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 3.

$$\begin{aligned} DAG_{ij} = DAG_{ji} &= R_1 \sum_{l=0}^{s-1} \alpha_{li}\,\alpha_{lj}\,\delta_{ij} \\ &+ R_2 \sum_{a=0}^{c-1}\sum_{l=0}^{s-1} \beta_{lia}\,\beta_{lja}\,\delta_{ij} \\ &+ R_3 M \sum_{b=0}^{v-1} \gamma_{ijb}\,\delta_{ij} \\ &+ R_3(1-M) \sum_{a=0}^{c-1}\sum_{b=0}^{v-1} \phi_{ijab}\,\delta ij \\ &+ R_4\,(\alpha_{ij} + \alpha_{ij})\,\delta_{ij}, \end{aligned} \tag{3}$$

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

$$\alpha_{lm} = \begin{cases} 1 & \text{if } S_m \in Suc(S_l) \\ 0 & \text{otherwise} \end{cases}$$

$$\beta_{lma} = \begin{cases} 1 & \text{if } S_m \in Pred(S_l, I_a) \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{ijb} = \begin{cases} 1 & \text{if } Z_b(S_i) = Z_b(S_j) \\ 0 & \text{otherwise} \end{cases}$$

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

$$\phi_{ijab} = \begin{cases} 1 & \text{if } Z_b(S_i, I_a) = Z_b(S_j, I_a) \\ 0 & \text{otherwise} \end{cases}$$

$$M = \begin{cases} 1 & \text{for Moore machines} \\ 0 & \text{for Mealy machines} \end{cases}$$

The first term of equation 3 refers to pairs of states that are common successors to a given state (rule i). For example, in Table I states S_3 and S_4 are successors of state S_1, therefore we have to add R_1 to DAG_{34}. The second term refers to pairs of states that have a common predecessor with a given input condition (rule ii). In table I, states S_2 and S_3 are predecessors of state S_4 with the input condition $I_0 = 0$, therefore R_2 needs to be added to DAG_{23}. The third and fourth terms refer to pairs of states that are in the same output partition for a given output (rule iii). The machine in table I is a Moore machine, therefore the outputs are independent of the input conditions. For example, state S_1 and S_3 are in the same partition for output $Z_1 = 1$, therefore we should add R_3 to DAG_{13}. The last term indicates transitions between two states, and is used as a tie breaker when the previous terms fail to indicate the relative position of each state. There are transitions between S_0 and S_4 in table I, therefore R_4 should be added to DAG_{04}. Since the DAG is an undirected and fully connected graph, the values of its connections can 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 II shows the matrix representation of the connections in the DAG obtained for the FSM of Table I, using equation 3.

For example, we obtain $DAG_{12} = R_1 + R_2 + R_2 + R_3 = 3 + 4 + 4 + 1 = 13$. The factor R_1 is produced because states S_1 and S_2 are common successors of state S_0 (first line on table I). The two factors R_2 appear because S_1 and S_2 are common predecessors of state S_4 under input condition $I_0 = 0$, and are common predecessors of state S_3 under input condition $I_0 = 1$. The factor R_3 reflects the fact that $Z_0(S_1) = Z_0(S_2)$. There is no factor R_4 because there is no transition between S_1 and S_2, that is $\alpha_{12} = \alpha_{21} = 0$.

The SSC cost is lowered when two states with strong connections in the DAG are close to each other. Thus, if 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

TABLE II
MATRIX OF DAG CONNECTIONS.

	S_0	S_1	S_2	S_3	S_4
S_0	0	1	3	2	5
S_1	1	0	13	7	1
S_2	3	13	0	5	3
S_3	2	7	5	0	9
S_4	5	1	3	9	0

this is given by:

$$Fitness = \sum_{i=0}^{s-1} \sum_{j=0}^{s-1} (k + 1 - D(S_i, S_j)) DAG_{ij} \qquad (4)$$

where k is the number of bits used for the state codification. Equation 4 can be expressed as

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

Since DAG_{ij} is fixed for a given FSM formulation and k is a positive constant, the first sum results in a constant term. Therefore $Fitness$ is maximum when the sum over i and j of the product $D(S_i, S_j) DAG_{ij}$ is minimum. The sum $\sum_{i=0}^{s-1} \sum_{j=0}^{s-1} D(S_i, S_j) DAG_{ij}$ is lower when pairs of state with large DAG_{ij} have a smaller distance $D(S_i, S_j)$. Therefore a state assignment with maximum fitness abides by the heuristic rules of section II-B. The goal of this research is to find a state assignment with maximum fitness that, according to the heuristic rules, results in an SSC with low cost. From now on we shall assume that the DAG is given and our task is to find an assignment that maximizes the fitness.

III. GENETIC ALGORITHMS

Algorithmic approaches to COPs can have a constructive approach, an improvement approach, or a combination of both. 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 [9]. Some considerations made by Whitley for the Traveling Salesman Problem (TSP) are also valid for the SAP [17].

A. Genetic Algorithm applied to SAP

This section describes the representation of an individual, the fitness function, the mutate operator, the crossover operator, and the selection procedure, when GA is applied to SAP.

A.1 Genotype

The "genotype" of a problem is the representation of an individual in the GA. The standard technique is to represent each individual by a single bit string encoding a solution. However, for certain problems, a matrix representation is more suitable than a bit-string representation in terms of both naturalness and quality of results [15]. 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 I constitute representations of individuals.

A.2 Crossover

As pointed out in section III, 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 [17]. 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 4, 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 $D(S_i, S_j)$, 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 III-A.1, each column j of the binary matrix contributes to the value of the fitness function with $\sum_{i=0}^{s-1} D(S_i, S_j) DAG_{ij}$. This enable us to think of the fitness function as being formed by the sum of individual contributions of 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 head and from the second parent otherwise. The result might be an invalid solution in case of *conflicts* among states. These conflicts consist of two or more states with the same attribution. In this case the offspring generated is converted into a valid solution by modifying the conflicting assignments preserving information that came from the parent with better fitness. Usually these conflicts can be eliminated with few changes in the offspring.

In the example presented in Table III, the first two columns of the transition 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.

TABLE III
CROSSOVER 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

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 if l columns are taken from the parent with better fitness, at most 2^{k-l} assignments can be identical in those l columns. To eliminate the conflict, it is enough to find at most 2^{k-l} different combinations for the $k-l$ remaining columns, which is obviously possible. In the example of Table III, there are at most 2 assignments which share the same combination in the first two columns, and we can obtain two different combinations (namely 1 and 0) for the third column. 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.

A.3 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.

For the sake of the mutation operator, every pattern P_l formed by k bits is assigned to a given state. If the FSM has $s < 2^k$ states, then $2^k - s$ dummy states are assigned to the patterns not utilized by the real states of the FSM. 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 $A(S_j) = P_l$[2].

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.

TABLE IV
MUTATION EXAMPLE.

	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

In the example of Table IV 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.

A.4 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 [9], 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 whose offset is large. 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:

$$\begin{aligned} Roul\_Wheel\_Fitness(I_k) &= Fitness(I_k) \\ &- (q+1)\, Fitness(Min) \\ &+ q\, Fitness(Max) \end{aligned} \tag{5}$$

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:

$$P(worst) = \frac{q}{q+1} P(best). \tag{6}$$

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.

B. 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 [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[3].

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

$$Weight(C_j) = \sum_{i=0}^{s-1} B_{ij} 2^i \tag{7}$$

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 a factor of $s!$. An example of these operations is presented in Table V. 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 $A(S_0) =$

[2]In the case that either S_j or S_i is a dummy state, the swapping is reduced to a change in the assignment of a single state. If both states S_i and S_j are dummy states, the swapping has no effect on the assignment.

[3]Actually, the column complement operation affects the cost of the SSC. However the Fitness function defined by equation 4 is insensitive to this effect. For more details on equivalent assignments see [10].

TABLE V
REDUCTION IN SOLUTION SPACE.

	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

0. After this operation, $Weight(C_0) = 12$, $Weight(C_1) = 46$, $Weight(C_2) = 22$. 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 distances among their 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.

IV. EXPERIMENTAL RESULTS

Two separate sets of experiments were conducted during the testing of the GA. Firstly, characteristics of the GA were explored in depth using structured FSMs. Secondly, the performance of the GA was compared to those of leading algorithms on previously published FSMs.

A. Characteristics of GA for SAP

To test the GA characteristics we used FSMs with 32, 33 and 64 states, with different degrees of connectivity among states and developed regular structures to enable the prediction of an assignment close to the best one. In order to assess the solution quality, we solved the same FSMs using a heuristic algorithm previously developed by Amaral [1], and considered the fitness of that solution to be unity. Amaral's algorithm invariably finds a close-to-optimal solution for structured machines with a small number of states. The optimal solution for such machines is known through exhaustive search. To allow comparisons between different machines and obtain a relative measure, the fitness computed by the GA was normalized using equation 8. MIN is the worst solution ever obtained by random search. MAX is the solution obtained by the heuristic algorithm mentioned above.

$$Norm\_Fitness = \frac{Fitness - MIN}{MAX - MIN} \qquad (8)$$

The results obtained with the GA for varying population sizes, number of generations and mutation rates are presented in Figures 1-4[4]. The curve labeled "random" represents the best fitness obtained after 100 individuals were randomly selected. These tests were done with a 32 state FSM, called "c32", using five bits to code each state. The values used for MIN and MAX to compute the *Norm_Fitness* were $MIN = 14382$, and $MAX = 17146$.

Figures 1-2 show the effect of varying the population size. As expected, GAs with large populations produce better results.

[4]Since GA is non-deterministic, each data point in these figures is based on the average of twenty independent runs. Therefore slight variations may be observed among curves representing runs with similar parameters.

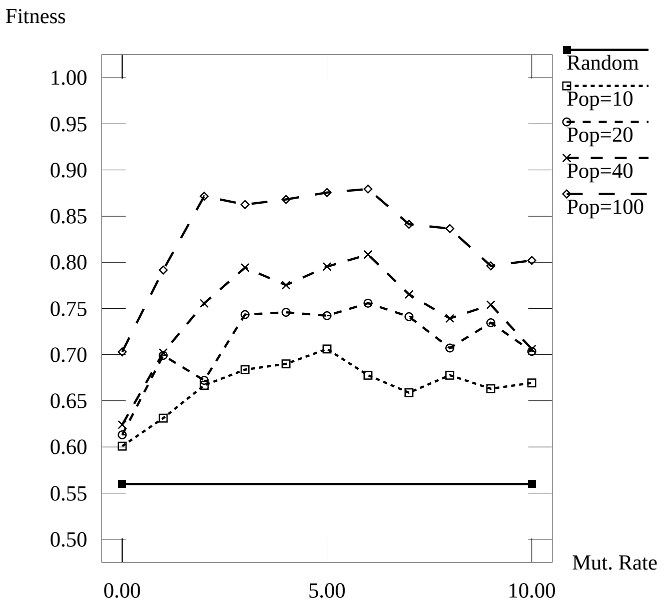

Fig. 1. Effect of the population size on solution quality (40 generations) - c32.

The influence of the population size in the results is smaller if the GA runs for more generations. The effect of the mutation rate is significant. In the absence of mutation, GA produces results that are only marginally better than the ones obtained by random selection[5]. On the other hand, large mutation rate has a negative effect on the performance. A GA with high mutation rate introduces an excessive amount of new information in the generation of offsprings and does not preserve good characteristics in the population. A small mutation rate introduces sufficient new information to allow the search of the entire solution space without excessively disturbing the population. It is encouraging to observe from Figure 2 (also see Figure 4) that the fitness is relatively insensitive to mutation rate between 2 and 8. This suggests that fine-tuning of the rate is not needed so long as it is in the appropriate ranges.

Figures 3-4 show the effect of the number of generations on the solution quality. Without mutation, the population size dictates the solution quality without much influence from the number of generations. Except for this difference, these curves are strikingly similar to the previous ones, suggesting that the determining factor for solution quality is the product of the population size and the number of generations. The population size is limited by storage space, while the number of generations is limited by processing time. Therefore, there is a time/space tradeoff. As long as both numbers are large enough, similar results can be achieved by either running the GA for a large number of generations with a modest population size, or by running the GA for a moderate number of generations with a large population size.

[5]Since the random measure reflects the best of 100 random solutions, a GA with a population of 10 and without mutation can provide a worse solution since it draws from a smaller pool of solutions

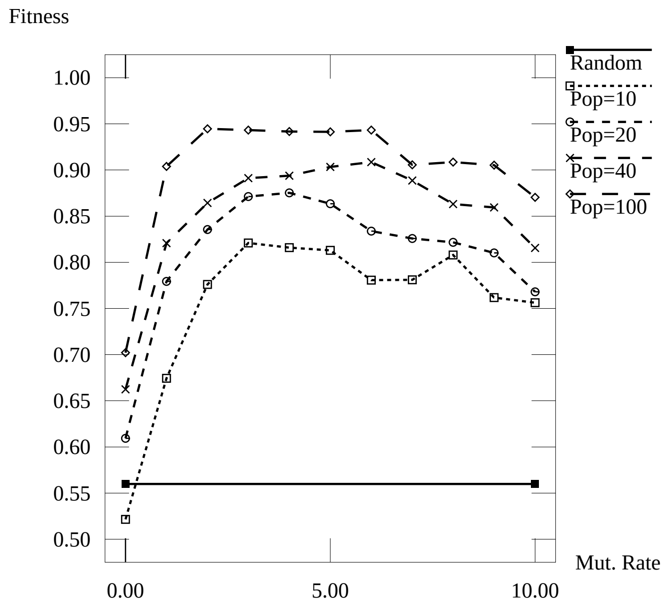

Fig. 2. Effect of the population size on solution quality (100 generations) - c32.

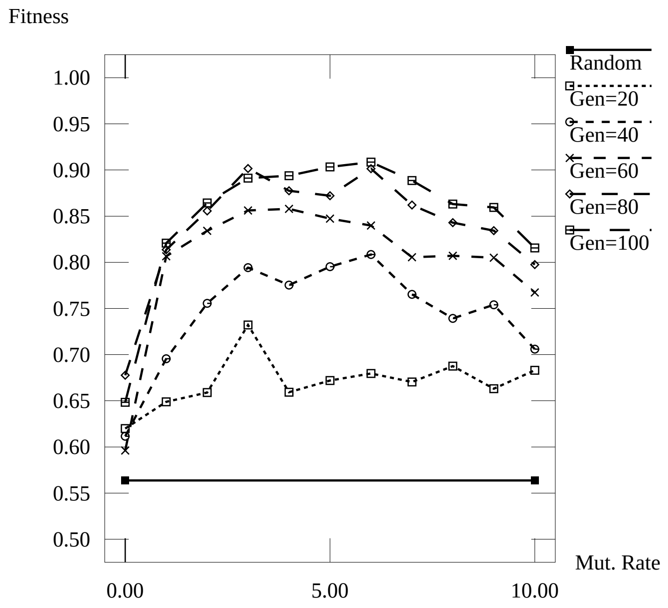

Fig. 3. Effect of the number of generations on solution quality (Population = 40) - c32.

One important conclusion from these experiments is that if either the population size or the number of generations is too small the GA cannot search the entire solution space well. Moreover, a large mutation rate prevents the information transfer from one generation to the next, seriously inhibiting the solution improvement process. Therefore the best GA results are obtained for a small amount of mutation rate and a large product of number of generations and population size, as long as neither of these two factors is too small.

When working with 33 and 64 state machines we observed that the quality of the solutions deteriorates with the number of bits used to code the states. This is mainly due to the fact that the number of bits determines the size of the search space. Variations in the number of states have no 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. This is a salient feature since it removes the need for tuning the algorithm for each new FSM design.

B. Comparative results.

The second set of experiments consisted of comparing the GA results on different machines with some established, specialized algorithms for SAP. These comparisons were based on the number of literals for the sum-of-product form—lit(sop)—and for the factored form—lit(fac). For NOVA and GA the cost was computed using the system “sis” distributed by the Dept. of EECS, UC Berkeley [13]. For verification purposes, the actual assignments (i.e. the solutions obtained) that resulted in the number of literals shown on tables VI[6] [7] and VII(a) are presented in Table VIII[8]. The GA solutions were obtained with eight hundred generations and a population of two hundred individuals.

We used benchmarks presented in [6] and [16] to compare the performance of the GA with those of the competing algorithms. Comparative results are presented in Tables VI and VII(a). Results for MUSTANG-P, MUSTANG-N, and KISS were obtained from [6][9]. The size of the examples are given in Table VII(b).

Algorithms for SAP are based on heuristics, and the experimental results reflect this fact. For each of the six algorithms compared above, there is at least one circuit for which it gives the best result and another for which it yields the worst! Even for a small set of examples, no algorithm can be singled out as providing the best solutions. General trends from Table VI shows that the GA compares favorably to any other algorithm, and for one example —lion9— outperforms all of them. This is remarkable since NOVA, KISS and MUSTANG are very sophisticated and incorporate a lot of domain knowledge as compared to the GA. Table VII(a) shows the cost as computed using the sum-of-products form[10]. GA results are as good as, or better than the best NOVA result in 5 of the 9 machines, but fall short on 2 machines.

The results obtained by the GA are encouraging and the unpredictability of the results emphasize the need for diverse meth-

[6]NOVA_1 is NOVA executed with the default option -e ig, that causes NOVA to be driven by input constraints.

[7]NOVA_2 is NOVA executed with options -e ioh -r, that causes NOVA to be driven by input and output constraints, and all possible rotations of the codes are tried.

[8]Each assignment is presented as a table of decimal numbers that represent the binary code of the states of the FSM in crescent numerical order.

[9]We always report the best result between the two logic optimizations reported in that paper — #lit1 and #lit2.

[10]This cost was not available for MUSTANG and KISS.

TABLE VI
COMPARATIVE RESULTS — LIT(FAC) FORM.

Example	GA	NOVA_1	NOVA_2	KISS	MUSTANG-P	MUSTANG-N
shiftreg	10	9	3	8	16	8
train11	47	64	49	–	–	–
lion9	21	40	32	67	73	38
donfile	257	206	178	548	259	148
bbara	86	84	100	129	67	84
tav	26	29	29	21	21	21
bbsse	180	207	214	151	125	141
dk14x	159	165	178	132	123	112
c32	243	421	274	–	–	–

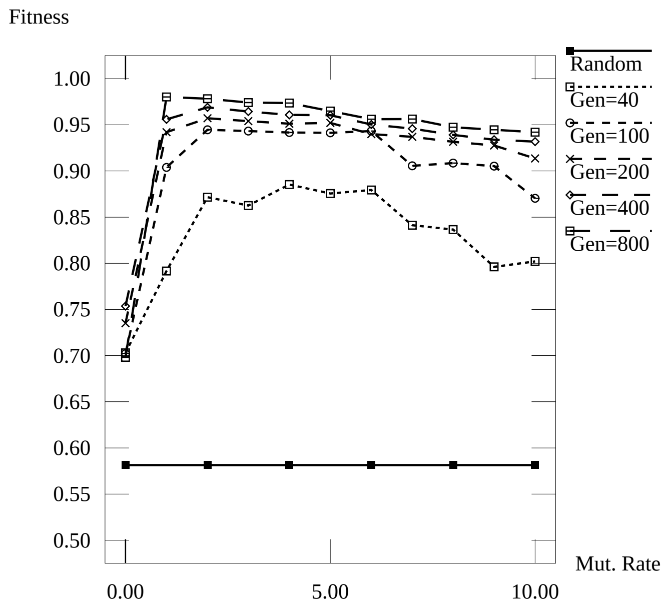

Fig. 4. Effect of the number of generations on solution quality (Population = 100) - c32.

ods, since the best for any given machine can't be known in advance.

V. 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 captured the inherent properties of the search space and led to satisfactory solutions for the state assignment problem.

The experiments with FSMs of various sizes show that the optimum parameters do not change significantly from one machine to the next. The GA also shows some robustness to the mutation rate as long as one stays away from extreme values.

For most of the examples considered, the GA compared favorably to leading specialized algorithms for SAP that incorporate extensive domain knowledge about the problem. An added advantage of the GA is that it can also be used to improve solutions obtained with other algorithms. For example, the initial population can consist of solutions obtained with other algorithms, guiding the GA to the area of the search space where the optimum solution is expected to lie.

REFERENCES

[1] J. N. Amaral and W. C. Cunha. State assignment algorithm for incompletely specified finite state machines. In *Fifth 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] D. J. Comer. *Digital Logic and State Machine Design*. CBS College Publishing, New York, 1984.

[4] 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.

[5] S. Devadas and K. Keutzer. A unified approach to the synthesis of fully testable sequential machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10:39–50, January 1991.

[6] S. Devadas, H.-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vicentelli. Mustang: State assignment of finite state machines for optimal multi-level logic implementations. In *International Conference on Computer Aided Design*, pages 16–19, 1987.

[7] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment, and four-level boolean minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10:13–27, January 1991.

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

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

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

[11] S. H. Hwang and A. R. Newton. An efficient verifier for finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10:326–334, March 1991.

[12] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vicentelli. Solving the state assignment problem for signal transition graph. In *Proceedings of 29th ACM/IEEE Design Automation Conference*, pages 568–572, June 1992.

[13] E. M. Sentovich et. al. Sis: A system for sequential circuit synthesis. Memorandum No. UCB/ERL M92/41, Dept. of EECS, Univ. of Berkeley.

[14] 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.

[15] 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.

[16] T. Villa and A. Sangiovanni-Vincentelli. Nova: State assignment of finite

TABLE VII
(A) COMPARATIVE RESULTS — LIT(SOP) FORM. (B) EXAMPLE SIZES.

Example	GA	NOVA_1	NOVA_2
shiftreg	10	9	3
train11	53	79	48
lion9	22	51	39
donfile	408	321	280
bbara	130	134	154
tav	32	35	35
bbsse	345	312	381
dk14x	252	252	268
c32	401	768	450

Example	inputs	outputs	states
shiftreg	1	1	8
train11	2	1	11
lion9	2	1	9
donfile	2	1	24
bbara	4	2	10
tav	4	4	4
bbsse	7	7	16
dk14x	3	5	7
c32	3	4	32

TABLE VIII
STATE ASSIGNMENTS.

Example	GA	NOVA_1	NOVA_2
shiftreg	0-2-5-7-4-6-1-3	0-4-2-6-3-7-1-5	0-2-4-6-1-3-5-7
train11	0-8-2-9-13-12-4-7-5-3-1	0-8-2-9-1-10-4-6-5-3-7	0-13-11-5-4-7-6-10-14-15-12
lion9	0-4-12-13-15-1-3-7-15	2-0-4-6-7-5-3-1-11	0-4-12-14-6-11-15-13-7
donfile	0-12-9-1-6-7-2-14-11-17- -20-23-8-15-10-16-21-19- -4-5-22-18-13-3	12-14-13-5-23-7-15-31-10-8- -29-25-28-6-3-2-4-0-30-21- -9-17-12-1	6-30-11-28-25-19-0-26-1- -2-14-10-31-24-27-15-12- -8-29-23-13-9-7-3
bbara	0-6-2-14-4-5-13-7-3-1	4-0-2-3-1-13-12-7-6-5	9-0-2-13-3-8-15-5-4-1
tav	0-2-3-1	0-3-1-2	0-3-2-1
bbsse	0-4-10-5-12-13-11-14-15- -8-9-2-6-7-3-1	12-0-6-1-7-3-5-4-11-10- 2-13-9-8-15-14	2-3-6-15-1-13-7-8-12-4- -9-0-5-10-11-14
dk14x	0-4-2-1-5-7-3	5-7-1-4-3-2-0	7-2-6-3-0-5-4

state machines for optimal two-level logic implementation. *IEEE Transactions on Computer-Aided Design*, 9:905–924, September 1990.

[17] 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.