An Improved Genetic Algorithm for the Inference of Finite State Machine

Nattee Niparnan, Prabhas Chongstitvatana

Computer Engineering Department

Chulalongkorn University

Bangkok, Thailand

nattee.n@student.chula.ac.th, prabhas@chula.ac.th

Abstract--**We propose a new method for the use of genetic algorithms to synthesize a finite state machine consistent with a given input/output sequence set. Our approach improves the method previously used by many researchers by omitting the evolution of complete FSM but rather evolve only the state transition function. The output will later be inferred from the input/output sequence set. The results show that our method is able to solve a much larger problem size than the previous approach.**

Keywords: **Grammatical Inference, Finite State Machine, Genetic Algorithm**

I. INTRODUCTION

We propose the use of genetic algorithms to solve the problem of synthesizing a finite state machine consistent with a given input/output sequence set. Our motivation is to mimic the target machine by synthesizing a machine that is consistent with partial input/output sequences observed from the target machine. Some researchers have attacked this problem. Manovit [1] show that longer length of sequence yields higher percentage of the correct machine. Chongstitvatana and Aporntewan [2] also show that multiple input/output sequences should be used to improve the correctness percentage. Aporntewan and Chongstitvatana [3] also made an online evolvable hardware using GAs to search for the circuit that satisfying the input/output sequence set.

Finding a FSM consistent with given input/output sequences is a problem in the field of grammatical inference (GI). GI is about the inference of any structure that can recognize a language. These structure may be a finite state automaton, a grammar or a push down automaton, etc. Many researchers have worked on the GI. Some related works are summarized here.

It is well known that finding a minimum size deterministic finite automaton consistent with a set of given samples is NP-complete [4]. Furthermore, Pitt and Warmuth [5] show that even finding a DFA whose size is polynomial on the size of minimum solution is also NP-complete.

Oliveira and Silva [6] propose an improved search method over a method proposed by Biermann [7]. Their method is known to be the current state of the art on finding a minimum size FSM that is consistent with a given input/output sequence set.

Lang, *et al* [8], propose an algorithm called “blue-fringe” which is based on the idea of state merging. This method constructs a prefix tree that describes the given positive and negative example and then folds it up by merging compatible pairs of state. This method is able to handle a large size of problem (512 states).

Fogel, *et al* [9], use the evolutionary programming to infer finite-state automata. Some researcher also try other form of automata, e.g. Langhorst [10] used GAs for the induction of pushdown automata to recognized the given language

Recently, Chongstitvatana and Aporntewan [2] attacked on more realistic problem by using GAs to synthesize a Mealy mode FSM which has multi-bit input/output. However, their method is able to solve only small sizes of FSM. Our work intends to improve the method of using GAs to synthesize a FSM.

The next section gives a basic definition used in this paper. Section 3 presents the use of GAs in the synthesis method and our improvement. Section 4 describes the experiment and its result. Finally, the last section provides the conclusion of this work.

II. BASIC DEFINITION

In this section, we give the problem statement and the definitions that will be used in this paper.

A. Definition

We use the definition of finite state machine as used by Hopcroff and Ullman [11] as described here.

Definition 1 *A Mealy machine is a six-tuple* $(Q,\Sigma,\Delta,\delta,\lambda,q_0)$ *where* Q *is a set of states,* Σ *is the input alphabet,* Δ *is the output alphabet,* $\delta(q,a) : Q \times \Sigma \to Q$ *is the transition function, and* $\lambda(q,a) : Q \times \Sigma \to \Delta$ *is the output function.*

We assume that the size of Q is m, the size of Σ is n and the size of Δ is o. We use q to denote a particular state, a to denote a particular input and b to denote a particular output.

Definition 2 *An input/output sequence* S *of length* n *is a set of pairs* $\{(i_0,o_0), (i_1,o_1),\ldots, (i_n,o_n)\}$ *where* $(i_i,o_i) \in \Sigma \times \Delta$. *An input/output sequence set* ζ *is a set of* S.

In this paper, we use M to denote a target FSM and M' to denote a evolving FSM. A finite state machine $M = (Q,\Sigma,\Delta,\delta,\lambda,q_0)$ is said to be consistent with an input/output sequence S iff $o_i = \lambda(q_0,i_0i_1i_2\ldots i_i)$ for all $0 \le i \le n$.

B. Problem Statement

Given the input/output sequence set ζ, which composes of a number of input/output sequences randomly generated from a target machine M, the task is to find a Mealy machine M' that is consistent with all elements in the set ζ.

III. Genetic Algorithm For The Problem

In this section, we firstly describe the previous method of using GAs to attack the problem, then we describe our improvement of the method.

A. Basic Approach

The most obvious choice is to use GAs to evolve the entire machine. Assume that we know the number of state in the target machine in advance. Input and output symbols are observed from the input/output sequence. We need to consider are the transition function and the output function. We encode the next state and the output for every transition of M'. The next part to be considered is the fitness evaluation. We want to reward FSMs that is consistent with the given input/output set. The more that FSM is consistent, the higher score it should get. The most obvious choice is to feed the input sequences to the synthesized FSM then collects the output produced by that FSM. The output produced will be compared with the output in the sequence, the fitness is the number of similar bit in both sequences. If every bit in both sequences is the same, the synthesized FSM will be consistent with the given sequence.

B. The Improvement

The main drawback of the basic approach described in previous subsection is that the evaluation counts the number of similar bits of the output on every step. This method, while reasonable, is not effectively evaluate the FSM as it should be. Let us consider a target FSM $M = (Q,\Sigma,\Delta,\delta,\lambda_0,q_0)$ with binary input and output and a synthesized FSM $M' = (Q,\Sigma,\Delta,\delta,\lambda_1,q_0)$ which its output function $\lambda_1 = \neg\lambda_0$. When we evaluate M' with sequences set ζ generated from M, the result will always be very bad because the output of M' always differs from the sequences. But, in fact, M' is very similar to M except the output label is inversed.

Actually we do not need to evolve the output of M'. Consider again the M' in last paragraph and let λ_1 be undefined. If we feed the sequence of S to M', the output sequence of M' will be $\lambda_1(q_0,i_0)$, $\lambda_1(q_1,i_1)$, $\lambda_1(q_b,i_2)$,… $\lambda_1(q_n,i_n)$ where n is the length of sequence and $q_a,q_b,\ldots q_n$ are the appropriate states that M' steps over due to M' transition function. Next, considering the output of sequence S, the output sequence is $o_0, o_1,\ldots,o_n$; o_0 is $\lambda_0(q_0,i_0)$, o_1 is $\lambda_0(q_a,i_1)$ and o_n is $\lambda_0(q_n,i_n)$. Since the Q,Σ,Δ,δ and q_0 of M and M' are the same, it is easy to see that for arbitrary state q and q' and arbitrary input a and a', the output generated by M' and the output of sequence S (which was generated by M), $\lambda_1(q,a)$ will correspond to only one $\lambda(q',a')$. We can define $\lambda_1(q,a)$ as $\lambda(q',a')$.

In conclusion, if M' transition is the same as M, we can determine the output of M' by feeding the input/output sequence generated from M and define the output of M' from the output of the feeding sequence.

This situation led us to find a new way to synthesize M'. The basic idea is that we omit the synthesis of the output function completely and concentrate on the state transition function. The output function for the M' will be inferred from the sequences.

C. Representation

Since we synthesize only the transition function, we will represent only the transition function in our chromosome. For a machine with m states and n input alphabets, the number of transition of that machine is $m \times n$. In this work, we use binary representation for the transition function. For each transition, the next state is encoded. The number of bits required for each transition is $\lceil \log|\Sigma| \rceil$.

A chromosome consists of the concatenation of m by n transition. Let m be the number of states and let n be the number of input alphabets. The arrangement of transition is shown in Figure 1

D. Fitness Evaluation

We evaluate the transition function by counting the "conflict" of the output. Let λ_1 be the output function of M'. If the transition function is good, each of $\lambda_1(q,a)$ will be mapped to only one value, if not $\lambda_1(q,a)$ may map to more than one value, we call this as a "conflict".

We maintain the output count table OC. The OC is a three dimensions array by size of $m \times n \times o$ for counting the number of output which should be b for the transition of state q on input a. The OC is initially zero for every element. For each input/output sequence, assume that the FSM currently is in the state q, if the next input in the sequence is a and the corresponding output is b, we will increase the $OC[q,a,b]$ by one. When all sequences are exhausted, $OC[q,a,b]$ will be the frequency of output that is b for transition of state q on input a.

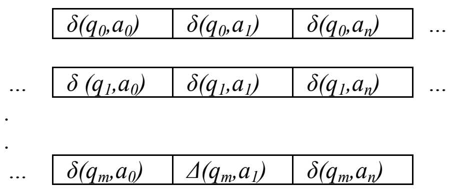

Fig. 1. Chromosome Representation

Formally, if for all $x \in \Delta$ there is one or none of $OC[q,a,x]$ that is more than 0, we will say that there is no conflict on $\lambda(q,a)$. On the other hand, if there is more than one of $OC[q,a,x]$ that is more than 0, we will say that there is a conflict on $\lambda(q,a)$.

We will illustrate the process of calculating OC on a small example. Let us look at Figure 2 that shows an evolving state machine M'. Assume that we have an input/output sequence $\{(0,0),(0,1),(1,1),(0,0),(1,0),(0,0),(1,0)\}$. We start at state A. The first input/output pair is (0,0) so we increase $OC[A,0,0]$

by one. From the transition of *M'*, we proceed to state *B*. The next input/output pair is (0,1) so we increase *OC[B,0,1]* by one. From the transition of *M'*, our machine is still in state *B*. We repeat this process until all input/output pairs are consumed. Table 1 shows the *OC* after the evaluation.

Next, we must infer the output of *M'*. We must define $\lambda_1(q,a)$ for every state *q* on every input *a*. *OC* is used to infer the output. Consider an arbitrary state *q* and an arbitrary input *a*, and *OC[q,a,x]* for all $x \in \Delta$. The value of *OC[q,a,x]* can be divided in to 3 cases. First case: for all $x \in \Delta$, $OC[q,a,x] = 0$, it means that the input sequences never use the transition of state *q* on input *a*, so we can set $\lambda_1(q,a)$ to anything. Second case: for all $x \in \Delta$, there is only one *OC[q,a,x]* that is more than 0. It means that every time the transition of state *q* on input *a* is used, there is only one value of output. Let *x'* be the *x* that made $OC[q,a,x] > 0$, the required output is *x'*. So we set $\lambda_1(q,a)$ to *x'*. The last case: if for all $x \in \Delta$, there is more than one *OC[q,a,x]* that is more than 0, this case is a conflict. Let *x'* be the *x* that maximize *OC[q,a,x]*, that is, the most frequent output of the transition of state *q* on input *a* is *x'*. Setting the output the transition of state *q* on input *a* to *x'* will cause the least error. So, we set $\lambda_1(q,a)$ to *x'* for this case.

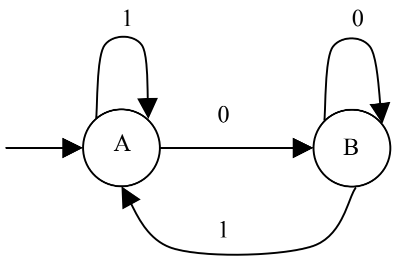

Fig. 2. Evolving State Machine

From all cases described, we infer the output for each transition as follow. Let *x'* be the *x* that maximize *OC[q,a,x]* for all $x \in \Delta$, define $\lambda(q,a)$ as *x'*.

TABLE I
OUTPUT COUNT TABLE

Input	State A		State B	
	Output 0	Output 1	Output 0	Output 1
0	3	0	0	1
1	0	0	2	1

Let us look again at the previous example. This time we will define the output function λ_1 for the machine *M'*. After the process of feeding the input/output sequence, we have the output count table as shown in Table 1. For the transition of state *A* on input 1, we see that both the *OC[A,1,0]* and the *OC[A,1,1]* are zero. Therefore we can set $\lambda_1(A,1)$ to anything. Next, consider the transition of state *A* on input 0 and the transition of state *B* on input 0. Both of them have only one output count that is more than zero. Therefore we set $\lambda_1(A,0)$ and $\lambda_1(B,0)$ to 0 and 1 respectively. Finally, the transition of state *B* on input 1, we have a conflict. Both output counts are more than zero. Therefore we set $\lambda_1(B,1)$ to 0, since 0 is the most frequent output used.

The output count table also indicates the goodness of the transition function on the input/output sequence set. We define fitness function as follow, let *T* be the transition function.

$$F(T) = \sum_{i=1}^{m} \sum_{j=1}^{n} \max(OC[i,j,b_1], OC[i,j,b_2], \ldots OC[i,j,b_o])$$

The idea is that for a transition with no conflict, we reward a score on how frequent that transition is used. For the transitions that generate a conflict, we reward a score on how severe is the conflict. If the conflict is severe (the required output is uniformly distributed among the output alphabet Δ), we will reward low score. If the conflict is small (the required output is mostly b_1 and few for b_2,b_3, etc.), we will reward high score.

For input/output sequence set ζ of *x* strings of length *y* each, this function would yield a value of $x \times y$ for the transition function *T* that generate no conflict at all. When we assign the output function λ for *T* as described in the previous paragraph, we will get a FSM that is consistent with *S*.

IV. EXPERIMENT AND RESULT

We test our method with three set of problems, each of which composes of randomly generated finite state machines. These FSMs were reduced and their unreachable states were removed. The reduction resulted in FSMs with their states vary from 2 to 32 states. All machines in the set 1 have one bit output, all machines in the set 2 have two bits output and all machines in the set 3 have three bits output. Every machine has one bit input. For each FSM, we randomly generate input/output sequence set. Each set composes of twenty strings of length 30 resulting in 600 pairs of input/output mapping. We applied our method on every set. For each set, 10 runs were performed with at most 5,000 generations per each run for problem in the set 1 and 10,000 generations for problems in the set 2 and 3.

From our preliminary study, we noticed that to evolve a transition function capable of handling a sequence generated from a FSM of size *m*, if the size of *M'* is slightly larger than *m*, the solution could be found faster. If the size of *M'* is too large, this gain of speed will drop. So, we assigned a size of *M'* according to the size of *M* as shown in Table 2.

TABLE II
SIZE OF GENE

Size of M	Size of M'	Gene's Size
3-15	16	128
16-31	32	320
32	64	768

We use a rank-based selection method. The number of individuals in each generation is 100. The single point crossover is used with probability of 0.5 and the mutation rate is 0.1.

Our experiments were performed on 833MHz Pentium with 256 Megabytes of memory running Linux. The problem which we could not find the solution took approximately 1 hour of execution time.

We compare our method with the method described in Section 3.1 called the reference method. For both methods, we use the same set of problems and parameters. Figure 3 shows the number of average generation used by our method (labeled Imp.) and the reference method (labeled Ref.) to produce the answer for the problem set 1. The run which does not produce a correct solution is also included in the calculation of the average value. A value of 5,000 indicates that the method is not able to solve the problem. For ease of comparison, the problems were sorted by the number of generations used by the reference method, so the graph of the reference method is a smooth and increasing function. Problem set 1 has 400 problems. Our method was able to solve 192 problems while the reference method solved only 162 problems.

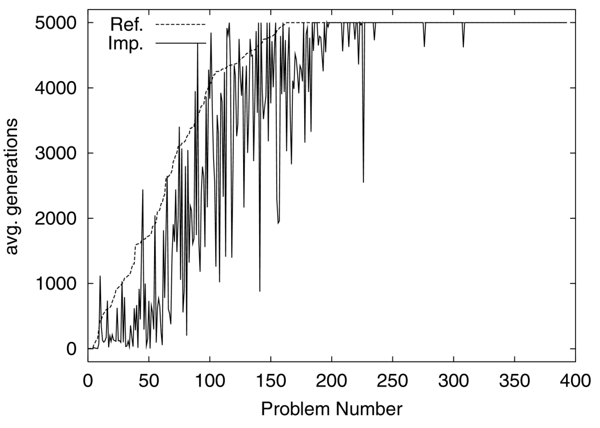

Fig 3. Average Generation Used for Set 1

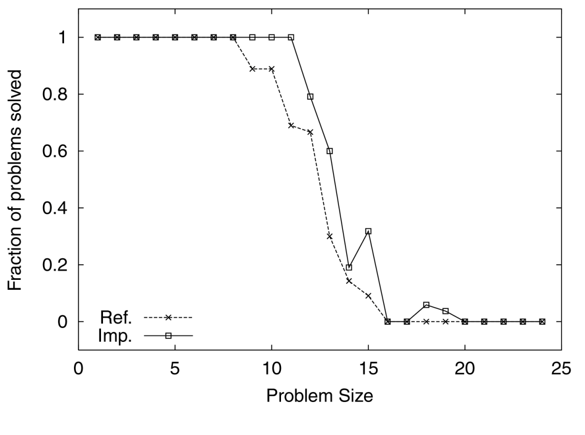

Fig. 4. Fraction of Problem Solved for Set 1

Figure 4 shows the percentages of problems solved according to the number of state in the target machine. The largest size that we are able to solve is 19 states while the reference method is able to solve only up to the size of 15 states. For one bit input and output, our method is slightly better than the reference method. However, the improved method does not show much of its power in the case of one bit output.
Next, we perform an experiment on the problem set 2, which has 2 bits output.

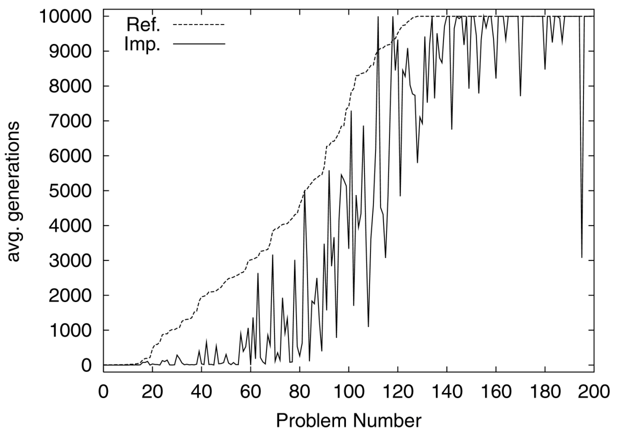

Fig. 5. Average Generation Used for Set 2

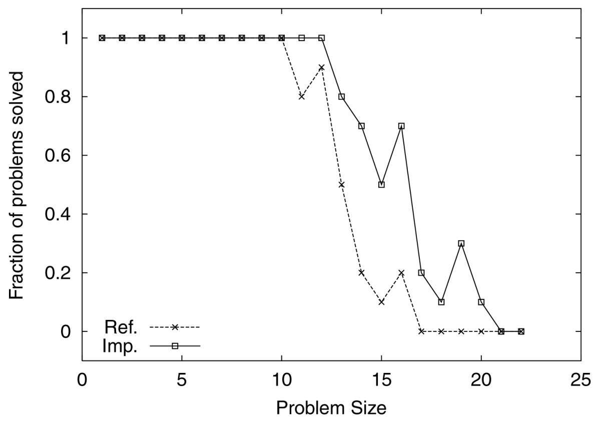

Fig. 6. Fraction of Problem Solved for Set 2

Figure 5 and 6 show the average generation used and the fraction of problem solved for the problem set 2 respectively. A value of 10,000 indicates that the solution could not be found. For the 2 bits output problem set, our method shows more improvement on the number of generations used. Problem set 2 has a total of 200 problems. Our method solves 154 problems while the reference method solves 127 problems. The largest size that our method is able to solve is 20 while the reference method's largest size is 16.

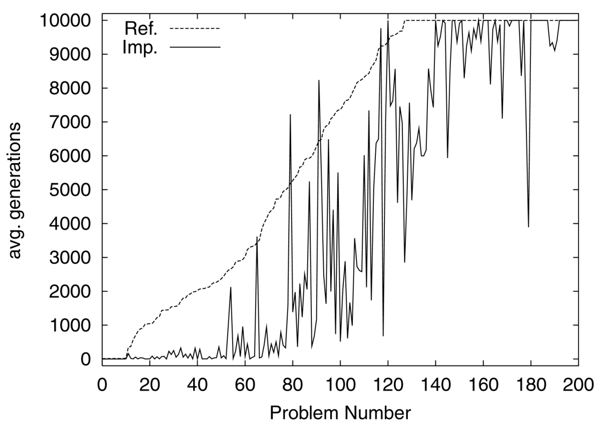

Fig. 7. Average Generation Used for Set 3

Figure 7 and 8 show the average generation used and the fraction of problem solved for the problem set 3. A value of 10,000 indicates that the solution could not be found. Problem set 3 composed of 200 machines with 3 bits output. Again, we can see clearly the improvement. Our method solves 166 problems while the reference method solves 127 problems. The largest size that our method and the reference method can solve are both 20.

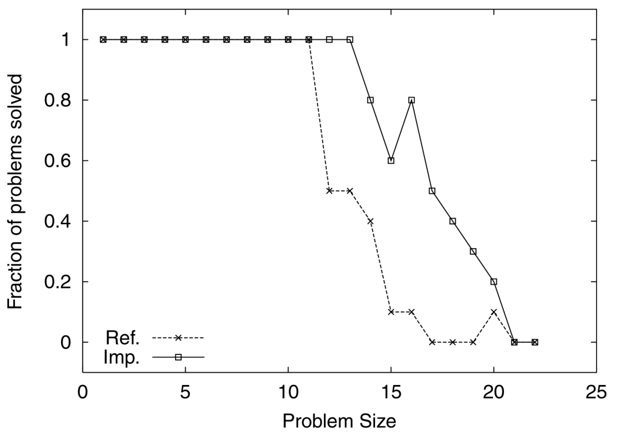

Fig. 8. Fraction of Problem Solved for Set 3

V. CONCLUSION

In this work, we presented a new approach of using the GA in the problem of inferring the finite state machine that is consistent with a given input/output mapping. This approach addresses some limitations in the previous methods used by many researchers. Our method mainly reduces the search space in the problem and guides the search process in a more accurate direction.

We attack realistic problems by picking the Mealy FSM, since most digital circuit design use the Mealy FSM. We also interest in the multi-bit output over the single bit output.

From the experiment, the improved method reduces the average generation used to find a solution. It also increases the problem size that GA can handle. The improved method can solve the problem of larger size than the reference method. The improvement of our method can be clearly seen on the multi-bit output since we reduce the effort used for the output part.

Please note that, the goal of this method is to synthesize a machine that is consistent with a given input/output sequence set. We do not aim to synthesize a machine that is equivalent with the target machine. We do not measure the correctness of the solution and the target machine. However, we can use larger and longer size of input/output sequence set to increase the correctness of the solution (Chongstitvatana and Aporntewan 1999).

REFERENCES

[1] C. Manovit, C. Aporntewan, and P. Chongstitvatana, "Synthesis of synchronous sequential logic circuits from partial input/output sequence," in *Proc. of Int. Con. on Evolvable Systems,* pp.98-105, 1998

[2] P. Chongstitvatana, and C. Aporntewan, "Improving correctness of Finite-state machine synthesis from multiple partial input/output sequences," in *Proc. of the 1st NASA/DoD Workshop of Evolvable Hardware*, pp.262-266, 1999

[3] C. Aporntewan and P. Chongstitvatana, "An on-line evolvable hardware for learning finite-state machine," in *Proc. of Int. Conf. on Intelligent Technologies,* pp.125-134, 2000

[4] E. M. Gold, "Complexity of automaton identification from given data," *Inform. Control*, Vol. 37, pp.302–320, 1978

[5] L. Pitt and M. Warmuth, "The minimum consistent DFA problem cannot be approximated within any polynomial," *Journal of ACM*, Vol. 40(1) pp.95–142, 1993

[6] A. L. Oliveira and J. P. M. Silva, "Efficient Algorithms for the Inference of Minimum Size DFAs," *Machine Learning 44,* pp.93-119, 2001

[7] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers,* Vol. 21 pp.592–597, 1972

[8] K. L. Lang, B. A. Pearlmutter and R. Price, "Results of the Abbadingo One DFA learning competition and a new evidence driven state merging algorithm," in *Fourth International Colloquium on Grammatical Inference, volume 1433 of Lecture Notes in Computer Science*, pp 1-12, 1998

[9] L. J. Fogel, A. J. Owens and M. J. Walsh, *Artificial Intel-ligence through Simulated Evolution*, John Wiley & Sons, 1966

[10] M. M. Lankhorst, "A genetic algorithms for the induction of pushdown automata," in *Proc. of Int. Conf. on Evolutionary Computation,* Vol. 2, pp.741–746, 1995

[11] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation,* 1979