A Genetic Algorithm for Finite State Machine Inference

Nattee Niparnan

10 September 2002

Abstract

This thesis aims to improve the use of genetic algorithm in the problem of inferring a finite state machine which is consistent with a given input/output sequences set. The thesis uses the work of Aporntewan as a reference model and then analyzing the problem in Genetic Algorithm point of view to find various ways to improve the method. The final goal is to find a better way to infer a finite state machine from a given input/output set

1 Introduction

This work attacks the problem of inferring a compact finite state machine that is consistent with a given set of input/output sequences. The problem has been proven to be an NP-Complete problem by many researchers and captures lots of interest from subsequences researchers. The motivation behind this problem for this thesis is to construct a machine that resembles the target machine by mimicking the target machine's partial behavior (input/output.) Although the generalization and the compact consistent inference is not the same, they are somewhat connected. Usually, when the number of examples is large enough, a behavioral equivalence can be achieved from the compact inference.

This specific problem falls into a class of theoretical problems called inductive inference. Inductive inference is a method of extracting a general rules from given examples. Also, it belongs to the field of artificial intelligence and machine learning. There are wide range of situations when the problem of inferring a finite state machine might be useful. Anything that its behavior can be represented by finite state machine can have benefit from the task. An obvious example is a machine that could mimic another machine. Consider a case when we need a fault-tolerance machine. One could have spares of every parts of the machine or one might have some learnable part that can mimic any of other part. The latter approach is much more economical than the first. However, the application of the problem is broad and not limited to the fault-tolerance machine.

Of course, the problem of inferring a finite state machine, or more general, problem of inferring any form of language has been attacked by many researchers. This thesis

concentrates to the study of the inference of a Mealy mode finite state machine by using genetic algorithm method.

Genetic algorithms are widely used in the field of Machine Learning. This problem is also the case. There are various approaches for the problem, beside the genetic algorithm, and lots of them show many promising result. However, genetic algorithm approach has it own unique goodness which is described in later section. Improvement of genetic algorithm for the problem will yield a good alternative solution.

Some researchers had used genetic algorithm in this specific problem, e.g., a work done by Aporntewan [Apo99]. The goal of this thesis is to improve the genetic algorithm method used in Aporntewan's work. The definition of the problem is presented in Section 3.4

The work of Aporntewan emphasizes on the hardware implementation of the problem rather than the performance of the genetic algorithm method. Genetic algorithms are quite new and have many interesting characteristic. There are various issues which one could optimize a genetic algorithm to perform better on a specific problem. These issues present an interesting theoretical aspect of the genetic algorithm.

2 Related Work

This thesis proposes the use of genetic algorithms to solve the problem of synthesizing a finite state machine consistent with a given input/output sequence set. Finding a finite state machine consistent with given input/output sequences is a problem in the field of grammatical inference. Grammatical inference is an inference of any structure that can recognize a language. The inferred structure can be anything ranging from as simple as a finite state automaton to as complex as Turing machine. Many researchers have worked on the finite state machine inference problem. This section describes some of interesting works.

It is well known that finding a minimum size deterministic finite automaton consistent with a set of given samples is NP-complete. This is shown in Gold's work [Gol78]. Furthermore, Pitt and Warmuth [PW93] shows that even finding a deterministic finite automaton whose size is polynomial in the size of minimum solution is also NP-complete.

For the solution of the problem, Biermann proposes algorithms in [BF72] and [BBP75]. The algorithm based on an extensive search method. The algorithm starts by constructing a prefix tree acceptor which described all given examples. The algorithm then tries to find a mapping of the states prefix tree acceptor to states finite state machine of size n. If no possible mapping is found, n is increased by one and the algorithm restarts the search. By starting from a small value of n which is guaranteed to be less than minimum, the method is guaranteed to find the minimum solution. Oliveira and Silva [OS01] made notable improvements over the method of Biermann. Their method incorporates many techniques in searching, such as *non-chronological backtracking* and *conflict diagnosis*. When the search method reach the point where there is unsatisfactory in search criteria, non-chronological backtracking tries to find the cause of dissatisfaction and backtrack

all the way back to the beginning of the cause. This avoid unnecessary search in all contradictory branch below the beginning point of the cause. Conflict analysis improves the method furthermore by remembering a set of parameters that caused dissatisfaction. Whenever these parameters are detected again in another branch, the algorithm will then omits the entire branch below that point. This method prunes out a lots of branches and results in much less number of backtracking. Biermann's algorithm does not include such a technique. Although the time used per node of Oliveira's method is higher, it can reduce the number of backtracking effectively. In fact, the work of Oliveira and Silva 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.

Since the problem is NP-Complete, i.e. there is no known efficient algorithm for the problem, Genetic Algorithms are among the choices that are used by many researchers. Some works are summarized here. Many forms of evolutionary computation were used. Fogel, et al, [FOW65] (reprinted in [FOW98]) use an evolutionary programming to infer finite-state automata. Dupont [Dup94] proposed the GIG method which based on genetic algorithm to infer automata accepting 15 different regular languages. The GIG method construct a *maximal canonical automaton* or *MCA* for the given positive examples. The MCA is a non-deterministic form of a prefix tree acceptor. So, the MCA itself can correctly recognize the positive example but not the negative examples. The GIG method then uses genetic algorithm to find a *partition* of states in the MCA. A partition is used to group and merge states in MCA. The method tries to find a partition such that the partition can correctly identify the negative examples. The evaluation function of the method is designed to promote a small machine.

Aporntewan and Chongstitvatana [AC00] use Genetic Algorithm to synthesize a Mealy mode finite state machine which has multi-bit input/output. Their work proposes a hardware implementation for the problem. Their genetic algorithm encodes transition functions and output functions of a mealy machine in the bit string. The consistency of the machine and the given input/output sequence is used as an evaluation function. However, their method is able to solve only small size of finite state machine. The motivation behind the work is to construct a hardware that can mimic other hardware.

Manovit, et al, [MAC98] study the relation between length of the given sequence and the correctness percentage of the solution in the case of genetic algorithm method. The correctness percentage is the percentage of a consistency machine that is behavioral equivalent to the target machine. The work defines the lowerbound and upperbound length of the sequence for the selected test set. The work shows that a sequence that is shorter than the lowerbound length can not yield correct solution and the correctness percentage saturates at the upperbound length. The correctness percentage can not be raised to 100 percent. In general, longer sequence of example yields higher correctness percentage that the inferred machine can correctly identify unseen data. Chongstitvatana and Aporntewan [CA99] extends the previous work by showing that multiple input/output sequences should be used to improve the correctness percentage and it is better to have multiple short sequences than one long sequence. One long sequence, while statistically long enough to exercise all transition, might not really exercise every

transition of the target machine, especially the first state. The sequence might walk pass one state only once and never go back to that state again. In this case, only one out path of that state is exercised. This problem can be addressed by using multiple sequences. The work also shows that multiple sequences can be used to raise the correctness percentage to 100 percent.

The use of genetic algorithm is not limited to inference of finite state machine but it extends to inference of many kind of structures. There are various works that use genetic algorithm to generate a structure capable of recognize a language. Lucas [Luc94] attacked a context-free grammar inference problem by using biased chromosome representation. He succeed in inference three alphabets palindromes. Lankhorst [Lan95] used Genetic Algorithm for the induction of nondeterministic pushdown automata that can recognize a language from given positive and negative examples. The automata uses the acceptance by empty stack approach. The algorithm encodes fixed number of transitions of the pushdown automata. The key point in his algorithm is the evaluation function which is based on the number of correctly identified examples, prefix lengths of consumed incorrectly identified example, and size of stack used. The method is compared with the GIG method of Dupont [Dup94] and other method that use neural network as an acceptor. The result shows better achievement over the other method. Pereira, et al, [PMCC99a, PMC$^+$00] attacked the problem of constructing a Turing Machine to perform a specific task. The task is to write as much "1" as possible on the initially blank tape and halt after that. These works use genetic algorithm with special techniques designed for the problem such as special chromosome presentation [MPCC99] and graph based crossover [PMCC99b].

Another related problem is the task of learning a deterministic finite automata from given positive and negative examples. The goal is a generalization, which means that the learned DFA must be able to correctly identify *unseen* test data. Lang and Pearlmutter set up the Abbadingo One competition to promote the development of better algorithm for this particular problem. There are many interesting results from the work. Notable one is the Even Driven State Merging algorithm and its variation described in [LPP98]. The algorithm is based on state merging method introduced by Trakthenbrot and Barzdin [TB73] This method constructs a prefix tree acceptor that describes the given positive and negative example and then folds it up by merging compatible pairs of state. However, one could not know whether the state that is to be merge is exactly the same state as in target automata or not. A merging of states usually introduces another constrains to a merging of later states. It is crucial that early merges should be correct. So, merging is done according to some measurement on the evident that the two merging states should be the same.It is very important to merge states that is most likely to be the same first. There are various measurement for the problem. The new EDSM based algorithm introduced in [LPP98], called blue-fringe, improves the old state merging algorithm by using a more accurate measurement of evident. It also reduces a pool of candidates pairs of state to be merged which is not likely to be the same state (though sometime the discarded pair does really is the same state) to reduce search time.

The goal of generalization and the goal of finding a minimal consistent rule is quite

different. Although they share some common attribute, the results of both can not be compare as noted in [OS01]. The blue-fringe algorithm, while results in more generalized hypothesis, makes no warranty on the size of conjectured machine and normally it is not the minimum size. The state merging based algorithm also lose the ability to correctly identify the finite automaton when the size of the training set gets smaller. However, blue-fringe algorithm is faster and much more scalable.

2.1 Approach of this Thesis

In conclusion, the work on the problem of inferring a finite state machine can be divided into three major approaches.

A Genetic Algorithm Approach: genetic algorithm is used to construct a finite state machine. An example is [Apo99].

An Exact Minimum Search Approach: This approach is based on the algorithm proposed in [BF72]. It tries to map states in prefix tree acceptor of the given example into minimum possible finite state machine.

An Inexact Heuristic Approach: This approach constructs a finite state machine with a heuristic method. The goal in this approach is slightly different from the goal of previous two approaches.

It seems that the third approach is the most successful one if considers the size of the problem. The approach is fast and highly scalable. However, this approach does not put constraint on the number of states of the result machine in its goal. Furthermore, when the training set gets smaller, this approach becomes ineffective when compared to the second approach [OS01].

The second approach, in contrast, put a very strong constraint on the size of the result. The result is guaranteed to be minimum. The speed of the algorithm and the problem size that it can cope with is fairly good, though it is far from the third method. The method is based on search method that tries to map states in prefix tree acceptor which is created from given examples into a minimum finite state machine. Thus, when the size of the problem is fixed, the time complexity of the method is exponentially dependent on the number of given examples. This is a major problem since more examples will yield more accurate hypothesis result as noted in [CA99].

A genetic algorithm approach, though it is not as fast as the third method nor it guarantees with the minimum size, plays a good role by filling a gap which give a fairly small size with a speed that is polynomially (not exponentially) dependent on the number of examples, when the size of the target machine is fixed.

This thesis knows of that advantage and intends to improve the work in a genetic algorithm approach.

3 Background Theory

Since this thesis deals with the application of genetic algorithm for finite state machine inference. there are three main subjects needed to be discussed. The first is genetic algorithm, which is the main tool of the thesis. The second subject is the inference problem and the last one is the finite state machine.

3.1 Genetic Algorithms

Genetic algorithms are a class of function optimizer algorithms, though their actual application span a wide range of problems from machine learning to intractable problems. Genetic algorithms, introduced by John Holland [Hol92] since 1975, are search procedures inspired by evolution. In general, they maintain a collection of potential solutions called a population. Each potential solution, usually called an individual, is evaluated and its "goodness" is measured in some way. Some of solutions are selected according to their "goodness" values to undergo a series of operations to create a new collection of solutions. This new population represents new points in the search space. The selection and operation are designed to drive a population better and better by implicitly identifying and assembling of critical information. Genetic algorithms have a bias on which of the new sampling point area are to be search next in the search space.

Genetic algorithms assume few assumptions about the problem being solved. This renders genetic algorithms very general. This is both advantage and disadvantage. The generality allows genetic algorithms to be used on a broad range of problems. On the other hand, generality comes with the cost that genetic algorithms could not exploit special structure of the problem. Genetic algorithms usually do not do well in a field in which there are known specialized algorithms for the problem. For example, though genetic algorithms are known as function optimizer, it could not even match with typical numerical methods, i.e., Newton-Raphson method, in optimization of the convex function. This is due to the fact that those numerical methods rely on differentiable property of the function while genetic algorithms do not assume and do not use such information.

So, where could genetic algorithms be a good choice? Genetic algorithms are regarded as a global search method that can be applied to non-differentiable or multiple local optima problems. NP-Complete and intractable problems are good candidates as a problem instant where genetic algorithms may be superior.

Genetic algorithms are inspired by evolution. They can be thought of as a simulation of evolution in the nature. A collection of solution can be regarded as a population of an arbitrary specie. Each individual in the population has a chance to survive. A highly fit individual has a better chance to breed or to pass its gene or characteristic to the next generation of individual. Many researcher use a hybrid method that combine existing method with genetic algorithms.

Genetic algorithm procedure is described briefly as follows. Genetic algorithms begin with a collection of (typically random) potential solutions encoded in chromosome-like data structure (a bit string). After that, genetic algorithms enter a loop. The loop repeats until the stopping criteria is met. The criteria might be a limit or a discovery of

solution. There are two main steps in the loop. First step is an evaluation of the population. It is a process that measures the goodness of each individual. The next step is to select candidates from the population and then apply some recombination operators on those candidates to create new individuals. These new individuals replace the old ones in the population. There are various selection scheme and replacement scheme but all of them share one thing in common, an individual with high "goodness" has more chance of surviving to the new population than the lower one.

Genetic algorithms assume few assumptions. Normally, there are two things that genetic algorithms require and are problem specific. There are the encoding of the solution and the evaluation function. Genetic algorithms encode potential solutions in bit strings. It implies that every parameters in a solution must be discretizable. The other assumption is that there must be an evaluation function for the problem. This is not difficult since the evaluation function is usually given as a part of a problem definition itself.

There are some issues on both assumption. When some parameter of a solution must be discretized into *exactly* some numbers that is not a power of 2. For example, if value of X has admissible presentation only in the range of precisely 0–20. The number of bits required to map it into 21 distinct values is 5 bits. However, 5 bits can be decoded into 32 distinct values. What should be done on remaining 11 values? Most obvious (and worst) solution is just to ignore an individual that possess inadmissible value but this usually results in lower performance of the algorithm. Another interesting issue is about an evaluation. Some problems do not have an "exact" evaluation function. This is usually the case when a solution is in a form of program. This kind of problem needs simulation-based evaluation. In such case, an evaluation function is just an approximation and a partial performance of the solution. To have more accurate evaluation function, the simulation process must be more precise hence takes more time There is a trade off between time and accuracy.

3.1.1 Canonical Genetic Algorithm

There are various variations of genetic algorithm. This section describes an implementation of one form of Genetic Algorithms originally proposed by Holland [Hol92] known as Canonical Genetic Algorithm.

Canonical Genetic Algorithm starts with a collection of potential solution. Each individual is randomly created and encoded in a fixed-length binary string. The length of string depends on the encoding of the problem. With starting population, the algorithm enters a loop of evaluation and reproduction. The loop is run until the answer is found or the predefined number of iterations is met.

There are two steps in the loop. First step is an evaluation of every individuals. The algorithm evaluates every individuals by specific evaluation function depends on the problem. After that, each individual is assigned a *probability of selection* which is calculated from the value of the evaluation function. The *probability of selection* tells how much chance that individual has on being selected to reproduce itself. The function that calculates the probability is called a ***fitness function*** and it is equal to $f_i/\overline{f}$, where f_i

denotes the evaluation value associated with the individual i and $\overline{f}$ denotes an average evaluation value over all individuals in the population.

This thesis adopts notions used in Whitley's work [Whi93]. That is, an *evaluation function* means the function that is used to measure "goodness" of each individual while a *fitness function* converts that measurement into a probability of selection. *Evaluation function* takes only one individual as a parameter and its value is independent to the other individuals. On the other hand, *fitness function* takes all individuals into account and each individual's value is depend on the others'.

The next step in the loop is a reproduction step. A pair of individuals is selected with probability according to their *fitness function* ($f_i/\overline{f}$) Various implementation of this style of selection has been proposed. A widely used one is a simulation of a roulette wheel. Each individual has its own slot in the wheel. Each slot has different size and is proportion to its *fitness value*. A selection is done by spinning the wheel. This resembles a "stochastic sampling with replacement" [Whi93].

With two individuals in hand, the next step is to determine whether to apply a recombination operator or not. There is a parameter p_c which indicates probability of the pair of individual is to undergo an recombination operator. An operator called *1-point crossover* is used to recombine that pairs of individual.

Remember that each individual is encoded in a fixed-length bit string, crossover is to swap a portion of two individuals. First, a cross site is randomly chosen with uniform distribution. A cross site indicates a boundary of portion which is to be swapped. Assume that a string `110011` is going to be crossed with a string `baabba`. Assume that a cross site is chosen to be between the second and the third bit, A crossover happens like this.

```
11|0011
ba|abba
```

swapping portion of these two parent string produces following offspring

```
11abba
ba0011
```

After a crossover is done, The mutation operator will be applied on the offsprings. Each bit in the string will be flipped with probability p_m. Normally p_m is set to a low value, e.g., 1% or less. In the case that crossover is not operated, mutation operator is applied to the selected pair of strings directly.

When all crossover and mutation is done, this pair of strings is added into the next population. The process of selection, crossover and mutation is repeated until the same numbers of individual is created for the new population. With new generation of population, Canonical Genetic Algorithm proceed to another iteration of the loop, i.e., it goes on an evaluation of the new population and then recombine again and again until the loop is stopped.

3.1.2 Schemata Theorem

Why does genetic algorithm work, after all? Or more precisely, what make genetic algorithm an effective search algorithm? Most widely accepted explanation is the theorem laid down by John Holland as described in [Gol89]

The theory suggests that, although Genetic Algorithms seem to search on a population of individuals, in fact, they *implicitly* search in a much larger number of hyperplanes of the solution space. Genetic Algorithms benefit from much larger number of hyperplanes sampled than the number of individuals in the population. The implicit searching is so important that it receives a special name as *implicit parallelism* or *intrinsic* [Whi93]. Genetic Algorithms, generation by generation, use the information stored in the population to implicitly search in the space of hyperplanes.

To illustrate a hyperplane, it is best to consider a sample problem of 3 bits encoding. There are eight possible individuals in the search space. Imagine a cube in a 3-dimensional space with every corners are labelled as depicted in Fig. 1. The cube is labelled `000` at the front lower left corner. The front plane of the cube contains all strings ended with "`0`". A special notion is introduced to describe portion of the cube. A notion "`*`" is used to present a *don't care* value. The string "`**0`", which means "any string ended with `0`", describes the front plane. Another example is "`*1*`" which describes the top plane and "`*00`" which describes the front lower line. A hyperplane is a partition of strings that is represented by a string containing "`*`". A string containing "*" has a special name as *schema*.

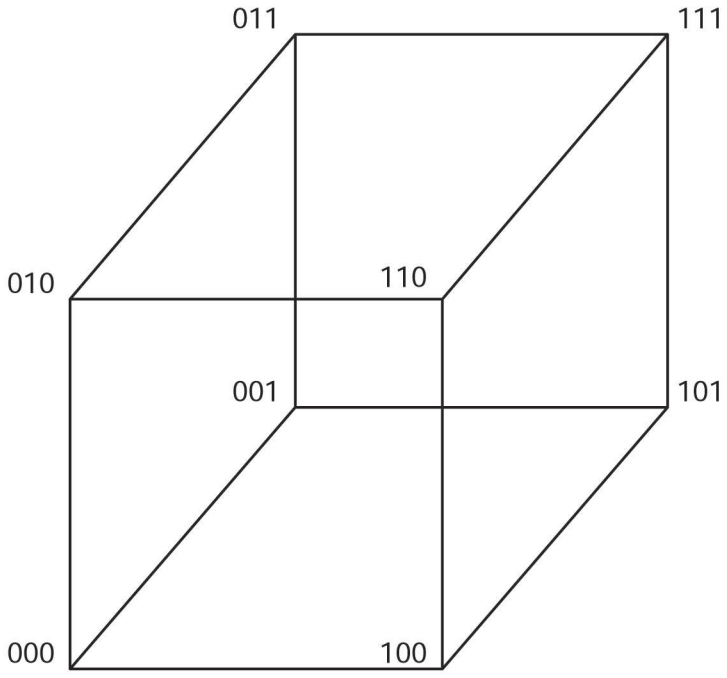

Figure 1: A 3-dimensional cube

A bit string matches a schema when every bit in the schema that is not "`*`" is the same as the corresponding bit in the bit string. Since a schema represents a hyperplane, any string that match a particular schema is contained in the hyperplane represented by that schema. A bit string of length L is contained in $2^L - 1$ different schemata (the schema with "`*`" in every bits is not counted as a partition of the search space).

The idea of implicit parallelism comes from the fact that while Genetic Algorithms work with a population of individuals, genetic algorithm also work with a lot more number of hyperplanes possessed in those individuals. In fact, a population of individuals contains information about how many representative does each particular hyperplane has in the population. When a population is evaluated, a lot more number of hyperplanes are also implicitly evaluated in parallel. A recombination and selection process affect schemata by changing their numbers of representatives in the population due to the average fitness of bit strings that are contained in those schemata. It implies that genetic algorithms use a population of bit strings to estimate fitness of hyperplane partitions. The key of genetic algorithms is that the population is driven toward a region of highly fit schemata. High fitness schemata will have more and more representatives in the next generation while low fitness schemata will die off.

A selection alone is enough to change a number of representative of each hyperplane but selection itself does not generate new sampling points in a search space. This is where recombination operators and mutation come into play.

Crossover affects the representative of hyperplane as it disrupts some schemata and sometimes it introduces new ones. A disruption by crossover occurs when crossover point lies within critical parts of a schema. A chance that crossover will affect any particular schema depends on value of that schema. For example, let us consider a schema `***11***` of length $L = 8$. The probability that crossover will separate this schema is $1/(L-1)$ since there is only one cross site (between the first and the second `1`) in the total of $L - 1$ possible sites. Let us consider another schema `1******1`. The probability is $(L-1)/(L-1)$, every crossover site will separate the fixed value of the schema. In general, the longer that the fixed bits span in a schema, the higher chance that the schema will be separated by crossover. This behavior reduces a chance that each schemata has on surviving into the next generation. This effect should be minimized. It can be done by a *compact representation* with respect to schemata. A compact representation suggests that highly fit schemata should have low distance of the fixed bits since it will has less chance to be affected by crossover.

Mutation also disrupts schemata since mutation flips bits in a schema. However, mutation should not be omitted entirely because it prevents every bit from permanently loss from a population. Usually, it is possible that a population converges into some schemata. This cause some positions in all bit strings to be only `0` or `1`. There is a chance that those positions converge into wrong bit value (this situation is called *premature convergence*). Mutation is used to prevent such a case. It is also suggested that mutation should be used with very low probability.

A formal definition of schema theorem is presented as follows. The theorem tells a lowerbound number of a particular schema's representations in next generation. Let

$M(H,t)$ be the number of strings in the current generation t that represents schema H. The change of particular schema H (assume that the fitness function is $f_i/\overline{f}$) will be:

$$M(H,t+1) = M(H,t)\frac{f(H,t)}{\overline{f}}$$

Now, the effect of crossover is to be added into the equation. Crossover executed with probability p_c will sometimes destroy schema H. Crossover sometimes creates schema H by operating on a pair of strings that coincidently contain *only* portion of H. To make things simple, a creation of schema H by crossover is neglected. Let p_d denotes the probability that schema H is destroyed by crossover. The equation above changes into inequality:

$$M(H,t+1) \geq (1-p_c)M(H,t)\frac{f(H,t)}{\overline{f}} + (p_c)\left[M(H,t)\frac{f(H,t)}{\overline{f}}(1-p_d)\right]$$

p_d can be computed from properties of schema H. Let $\Delta(H)$ denotes the *defining length* of H. The *defining length* of a schema is the distance between the leftmost and the rightmost bit in the schema that is not "*". For example, schemata `****11***` has a defining length of 1 since leftmost position is the `1` in fifth bit and the rightmost is the `1` in the sixth bit; $6-5=1$. Schema `****11*0*` has a defining length of $8-5=3$. The probability that a cross site will lie in this critical section and will destroy this schema is $\Delta(H)/(L-1)$ where L is the length of this schema. There is one exception, crossover will not destroy schemata H even a cross site is on this $\Delta(H)$ bit long portion when both of the parent strings represent H. Let $P(H,t)$ be the proportional representation of H. $P(H,t)$ is equal to $M(H,t)$ divided by the population size. The probability that the selected parent will represent H is $P(H,t)$. p_d can be described as:

$$p_d = \frac{\Delta(H)}{L-1}(1-P(H,t))$$

Divides the last inequality by the population size to change M into P and then rearranges it a little.

$$P(H,t+1) \geq P(H,t)\frac{f(H,t)}{\overline{f}}\left[1-p_c\cdot p_d\right]$$

Substitutes p_d by its value yield:

$$P(H,t+1) \geq P(H,t)\frac{f(H,t)}{\overline{f}}\left[1-p_c\cdot\frac{\Delta(H)}{L-1}\left(1-P(H,t)\right)\right]$$

Mutation also affect the proportion of the schema. Let $o(H)$ be the *order* of schemata H. An order of schema H is the number of bits in H that is not "`*`". Mutation, which flips bits in strings, will destroy a schema if the bit that is not "`*`" (don't care) is flipped. Thus, the probability that mutation will destroy particular schema H is $(1-p_m)^{o(H)}$. This yields in the final inequality:

$$P(H,t+1) \geq P(H,t)\frac{f(H,t)}{\overline{f}}\left[1-p_c\cdot\frac{\Delta(H)}{L-1}(1-P(H,t))\right](1-p_m)^{o(H)}$$

3.2 Inductive Inference

Inductive inference is the method of "guessing" general rules from given examples. A lot of problems in machine learning fall in inductive inference field. For instance, a method of finding a finite automata that can correctly recognize given positive and negative examples or a method of finding a function $f(x)$ given pairs of $f(x)$ and its arguments x are inductive inference method.

An inductive inference is not learning though it is quite similar. Angluin [AS83] point out the distinction that "to learn" is "to gain knowledge, understanding, or skill by study, instruction, or experience" while "induction" is "the act, process, or result of an instance of reasoning from a part to a whole, from particulars to generals, or from the individual to the universal."

3.2.1 An Example

To better understand a problem of inference, let us consider a problem of polynomial guessing. The problem consists of a program that has a sequence of numbers generated by predetermine polynomial. The program starts by displaying the first number in the sequence. A user is required to "guess" the next number in the sequence. After the user made a guess, a program displays the correct number in the sequence and checks whether the guess of user is correct or not. The program goes on indefinitely. The problem is considered solved when the user eventually "catch on" with the program, i.e., the user can always make a correct guess. To make the problem not too hard, the polynomial is one variable and its coefficient is positive integer. There are two obvious method to solve the problem. Each of them represents a concept of inductive inference which will be presented later in this section.

The first method is a very general one, though it is inefficient. The method searches through all possible polynomials systematically until it finds one that agrees with the sequence so far. The search is done on the enumeration of $d = 0, 1, 2 \ldots$. The method generates all possible $(d+1)$-tuples of integers in the range of $[-d, d]$ and assigns them as the coefficients of $1, n, n^2, \ldots$. For example, when the program runs to the point where the program produced t numbers, says $y_1, y_2, y_3, \ldots, y_t$, the method searches through an enumeration of polynomial to find one, says P, such that $P(n) = y_n$ for $n = 1, 2, \ldots, t$ and guesses the next value as $P(t+1)$. For a correct polynomial of degree d, this method will search through all d^d difference polynomials to find one that agrees with the examples. The d^d search space is big, so big that this method usually impractical on a large value of d.

The second method, which uses less computational power, takes initial sequence $y_1, y_2, y_3, \ldots, y_t$ and interpolates a polynomial of degree at most $t-1$ by using this values. To correctly interpolate a polynomial of degree d, this method uses a number of arithmetic operations polynomially in d while the first method uses a number of operations exponential of d. Although the second method is much faster, it is not as general as the first one. For example, if the program is allow to use not only a polynomial of integer coefficient but also a sum of integers exponent in n, for example, $2^n + 3^n$, the

first method can be easily adjusted to cover the additional rule while the second method is not as easy as the first.

However, both two methods never know whether it comes to the correct polynomial or not since no bound on degree of polynomial is specified. There is no way to tell when the guessed number is correct since the polynomial produced by the method might not agree with next unseen value which belongs to a bigger polynomial.

3.2.2 Two Basic Concepts

There are two basic concepts in inductive inference [AS83] which characterize a method of inference. One is *identification in the limit* and the other one is *identification by enumeration*

Identification in the limit regards an inference method as evolving process. In other words, the method changes its hypothesis while it receives more and more examples. The method's behavior *in the limit* is used to determine the method's success. When a method stop changing its hypothesis eventually and that hypothesis is the correct one, that method is success in inference. This concept view an inference method as a continuous learning process. A method successively modifies its conjectured hypothesis while it learns more and more on unknown rules. When the method stops modifying its hypothesis after some finite time and the last hypothesis is correct, the method correctly identify the rules in the limit on this sequence of examples.

Formally, let M be an inference method trying to infer some unknown rules R and M runs on larger and larger collections of examples. M conjectures sequences of hypothesis on each run, say, $g_1, g_2, g_3, \ldots$. If there exists some number n such that g_n correctly describe R and

$$g_m = g_{m+1} = g_{m+2} = \ldots$$

then M is said to identify R correctly in the limit on this sequence of examples.

Identification by enumeration represents a general, powerful class of inference method. Identification by enumeration method solves inference problem by searching throughout all possible rules systematically until one that agree with the given examples is found. An enumeration of polynomials in the example above is identification by enumeration method and so is the genetic algorithm method used in this thesis. This is because both of them search among the possible description of rules.

Suppose that some unknown rule R is in some particular domain and there is an enumeration of description, says, $d_1, d_2, d_3, \ldots$ such that each rule in the domain has at least one description in this enumeration. A method that searches in this enumeration to find a description, say d_i, that agrees with the given examples and then conjectures d_i is said to be an identification by enumeration method.

Although identification by enumeration is very general, it usually is impractical since the size of the search space is exponential on the size of correct rule.

3.3 Finite State Machine

The study of computer and computation begins with a basic question, what is a computer? To answer the question, one must understand the computer. Instead of dealing with *real computer*, i.e., one with specific CPU, memory, input/output peripheral, etc., a work on theory of computation involves an abstract *model of computation.* A finite state machine is one form of model of computation. To understand a model of computation it is required first to understand the "language", which ,in turn, requires the understanding of the idea of *decision problem*. A decision problem is a problem of which a solution can be only "yes" or "no". For example, a problem of determining whether a positive integer n is prime or not is a decision problem. Not all problems are decision problems; However for one that is not, there usually is a decision problem whose solution is comparable to original one [Mar97]. For a decision problem, an input can be encoded into a string of alphabets. This leads to another aspect of decision problems, a *language recognition problem*. A language recognition problem is a problem of whether a given string is in a specific set of string or not. For example, let us consider again the prime problem. Instead of checking whether n is prime, the problem transforms into a problem of checking whether string encoding n is in the language containing all primes or not.

A finite state machine is a very simple computational model. It can determine whether a given string is in the machine's specific set of string or not. A finite state machine resembles a system that, in any moment, is in one of finite states (hence the name) and has a deterministic way of action in response to input.

This work uses the definitions in Hopcroft and Ullman's book [HU79]. Some of definition is presented here.

Definition 1 *A finite automaton is a 5-tuple* $(Q, \Sigma, \delta, q_0, A)$ *where*

Q *is a finite set (which represent all states of the machine)*

Σ *is a finite input alphabet*

$q_0 \in Q$ *(*q_0 *represent starting state)*

$A \subseteq Q$ *(*A *is a set of accepting state)*

δ *is a function mapping* $Q \times \Sigma$ *to* Q *(the transition function) that is, for a particular state* q *and an input alphabet* a*,* $\delta(q, a)$ *is the state to which the finite automata moves, if it is in state* q *and gets input* a*.*

This thesis deals mostly with a variation of finite automaton, *a finite state machine with output*. A finite automaton defined earlier has an implicit output, that is, it can tell whether the input string is accepted or not. This can be thought of a binary output signal of "yes" or "no". For *a finite state machine with output*, outputs of the machine are defined explicitly. There are two types of machine with output, one with output defined for every states. It is called as *Moore machine* The other one with output defined for every transitions is called *Mealy machine*. These two kinds of machine is equivalent, though a Moore machine tends to have more states than its equivalent Mealy machine.

Definition 2 *A Moore machine M is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where Q, Σ, δ and q_0 are the same as defined in definition 1*

Δ is finite output alphabet

λ is a function mapping Q to Δ, the output function that give the output associated with each state.

The output of the machine in response to input $a_1, a_2, \ldots, a_n$ where $n \geq 0$ is $\lambda(q_0)\lambda(q_1)\ldots\lambda(q_n)$ where $q_0, q_1, \ldots, q_n$ is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$

Note that a Moore machine produces first output $\lambda(q_0)$ in response to null-input. A finite automaton might be viewed as a special case of a Moore machine whose output alphabet is $0, 1$ and state q is accepting state if and only if $\lambda(q) = 1$.

Definition 3 *A Mealy machine M is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where $Q, \Sigma, \Delta, \delta$ and q_0 are the same as defined in definition 2*

λ is a function mapping $Q \times \Sigma$ to Δ, that is, $\lambda(q, a)$ is a output associated with the transition from state q on input a.

The output of the machine in response to input $a_1, a_2, \ldots, a_n$ where $n \geq 0$ is $\lambda(q_0, a_1)\lambda(q_1, a_2)\ldots\lambda(q_{n-1}, a_n)$ where $q_0, q_1, \ldots, q_n$ is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$

An output sequence of Mealy machine is one alphabet shorter than an output from its Moore counterpart. This is because the first output of Mealy machine is generated from the first input while the first output of Moore machine is generated from null-input (the output from the starting state).

3.4 Problem Definition

This section presents the formal definition of the problem.

Definition 4 *Let $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a Mealy machine. An input/output sequence S_M of length n is a set of pairs $\{(i_1, o_1), (i_2, o_2), \ldots, (i_n, o_n)\}$ where $(i_0, o_0) \in \Sigma \times \Delta$. An input/output sequence set ζ_M is a set of S_M.*

Given the input/output sequence set ζ_M, which composes of a number of input/output sequences randomly generated from a target machine M whose number of state is m, the task is to find a Mealy machine M' that is consistent with all elements in the set ζ_M. The number of states of M' must less than $2m$.

4 Work Plan

This section presents draft of the plan for the work.

1. Study works in the field of genetic algorithm, inductive inference and automata theory.

2. Set up a reference method to be used as a baseline for the thesis.
3. Develop methods which might be used to improve the efficiency of the reference method.
4. Set up experiments to compare the developed method with the reference method.
5. Validate and summarize the result from the experiments.
6. Publish a scientific paper from the work.
7. Write a thesis and engage in defense.

4.1 Completed Works

1. Study works in the field of genetic algorithm, inductive inference and automata theory.
2. A C++ Program for the experiment was written.
3. One new method dealing encoding is developed and is shown to be better.
4. A paper for the new method is accepted as a poster by one conference and as a full presentation in another one which is
   - (Full presentation) “An improved genetic algorithm for the inference of finite state machines”, IEEE Int. Conf. on Systems, Man and Cybernetics, to be held in Tunisia, 6-9 Oct, 2002.
   - (Poster) “An improved genetic algorithm for the inference of finite state machines”, Genetic and Evolutionary Computation Conference, New York, July 9-13, 2002.

4.2 Ongoing Works

1. Designing a new crossover operator which utilizes the fact that the individual is graph based.
2. Designing a new representation coping with the problem of machine behavioral redundancy.
3. Validation of previous result.
4. Writing a thesis.

5 Purpose of the Thesis

1. To develop a method that can improve performance of genetic algorithm approach for the problem described in section 3.4

6 Scope of the Thesis

1. To compare the new method with the method used by [Apo99]
2. The new method must be shown to be better than reference method which is used in [Apo99]. The measurement is described in section 6.1
3. The solutions from the new method must be shown to be consistency.

6.1 Measurement of Improvement

This work aims to develop a better method for the problem. However, due to stochastic nature of the algorithm, it is hard to do a formal proof on the result. Hence, this work uses an empirical procedure to check the improvement of the new method. To compare the new method with the reference one, this work uses a test bed of problem instances consists of both real world instances and random instances. Two measurements are used to compare the method. One is the number of problem which is solved and the other one is the time used in solving all the problems. A new method is said to be better if and only if it uses less time than the reference method and it solves greater or equal number of problems.

7 Benefit from the Work

1. To have a genetic algorithm that solve the problem and the method is empirically shown to be better than former method.
2. To have a finite state machine inference algorithm which can result in compact size of hypothesis, scale well under larger number of examples and works in reasonable time

References

[AC00] Chatchawit Aporntewan and Prabhas Chongstitvatana. An on-line evolvable hardware for learning finite-state machine. In *Proceedings of International Conference on Intelligent Technologies*, pages 125–134, 2000.

[Apo99] Chatchawit Aporntewan. An mimetic evolvable hardware for sequencial circuits. Master's thesis, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, 1999.

[AS83] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

[BBP75] Alan W. Biermann, Richard I. Baum, and Frederick E. Petry. Speeding up the synthesis of programs from traces. *IEEE Transactions on Computers*, 24(C):122–136, 1975.

[BF72] Alan W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21:592–597, 1972.

[CA99] Prabhas Chongstitvatana and Chatchawit Aporntewan. Improving correctness of finite-state machine synthesis from multiple partial input/output sequences. In *Proceedings of First NASA/DoD Workshop of Evolvable Hardware*, 1999.

[Dup94] Pierre Dupont. Regular grammatical inference from positive and negative samples by genetic search: the GIG method. In *Proceedings of the International Colloquium on Grammatical Inference*, pages 236–245, 1994.

[FOW65] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. Artificial intelligence through a simulation of evolution. In M. Maxfield, A. Callahan, and L.J. Fogel, editors, *Biophycisc and Cybernetic System: Proceedings of the 2nd Cybernetic Sciences Symposium*, pages 131–155, Washington, DC, 1965. Spartan Books.

[FOW98] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. Artificial intelligence through a simulation of evolution. In David B. Fogel, editor, *Evolutionary Computation : the Fossil Record*, pages 230–254. IEEE Press, Piscataway, NJ, 1998.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information Control*, 37:302–320, 1978.

[Gol89] David E. Goldberg. *Genetic algorithm in search, optimization and machine learning*. Addison–Wesley, Reading, MA, 1989.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, Cambridge, MA, 1992.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison–Wesley, Reading, MA, 1979.

[Lan95] Marc M. Lankhorst. A genetic algorithm for the induction of pushdown automata. In *Proceedings of the Second IEEE Conference on Evolutionary Computation*, pages 741–746, 1995.

[LPP98] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. *In Fourth International Colloquium on Grammatical Inference, Lecture Notes in Computer Science*, 1433, 1998.

[Luc94] Simon Lucas. Structuring chromosomes for context-free grammar evolution. In *IEEECEP: Proceedings of The IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, 1994.

[MAC98] Chaiyasit Manovit, Chatchawit Aporntewan, and Prabhas Chongstitvatana. Synthesis of synchronous sequential logic circuits from partial input/output sequences. In *Proceedings of 2nd International Conference on Evolvable Systems*, pages 98–105, 1998.

[Mar97] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw–Hill, Singapore, 1997.

[MPCC99] Penousal Machado, Francisco B. Pereira, Amilcar Cardoso, and Ernesto Costa. Busy beaver – the influence of representation. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598, pages 29–38, Goteborg, Sweden, 26-27 1999. Springer-Verlag.

[OS01] Arlindo L. Oliveira and Joao P. Marques Silva. Efficient algorithms for the inference of minimum size DFAs. *Machine Learning*, 44(1/2):93–119, 2001.

[PMC+00] Francisco B. Pereira, Penousal Machado, Ernesto Costa, Amilcar Cardoso, Alberto Ochoa-Rodriguez, Roberto Santana, and Marta Soto. Too busy to learn. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 720–727. IEEE Press, 6-9 2000.

[PMCC99a] Francisco B. Pereira, Penousal Machado, Ernesto Costa, and Amilcar Cardoso. Busy beaver: An evolutionary approach. In *Proceedings of the second Symposium on Artificial Intelligence*, 1999.

[PMCC99b] Francisco B. Pereira, Penousal Machado, Ernesto Costa, and Amilcar Cardoso. Graph based crossover-A case study with the busy beaver problem. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1149–1155, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[PW93] Leonard Pitt and Manfred K. Warmuth. The minimum consistent dfa cannon be approximated within any polynomial. *Journal of ACM*, 40(1):95–142, 1993.

[TB73] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite Automata*. North-Holland, Amsterdam, 1973.

[Whi93] Darell Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Department of Computer Science, Colorado State University, 1993.