ขั้นตอนวิธีเชิงพันธุกรรมสำหรับการอนุมานเครื่องจักรสถานะจำกัด

นายนัทที นิภานันท์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2545
ISBN xxx-xxx-xxx-x

A GENETIC ALGORITHM FOR FINITE STATE MACHINE INFERENCE

Mr. Nattee Niparnan

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering in Computer Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2002
ISBN xxx-xxx-xxx-x

Thesis Title	A Genetic Algorithm for Finite State Machine Inference
By	Nattee Niparnan
Field of Study	Computer Engineering
Thesis Advisor	Associate Professor Prabhas Chongstitvatana, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

..... Dean of Faculty of Engineering
(Professor Somsak Panyakaew, D.Eng.)

THESIS COMMITTEE

..... Chairman
(Professor Chidchanok Lursinsap, Ph.D.)

..... Thesis Advisor
(Associate Professor Prabhas Chongstitvatana, Ph.D.)

..... Member
(Assistant Professor Boonserm Kijsirikul, D.Eng.)

..... Member
(Assistant Professor Nachol Chaiyaratana, Ph.D.)

นัททที นิภานันท์ : ขั้นตอนวิธีเชิงพันธุกรรมสำหรับการอนุมานเครื่องจักรสถานะจำกัด (A GENETIC ALGORITHM FOR FINITE STATE MACHINE INFERENCE). อาจารย์ที่ปรึกษา : รศ. ดร. ประภาส จงสถิตย์วัฒนา, 99 หน้า. ISBN xxx-xxx-xxx-x

วิทยานิพนธ์ฉบับนี้ได้ศึกษาปัญหาการอนุมานเครื่องจักรสถานะจำกัด โดยมีเป้าหมายเพื่อที่จะสร้างเครื่องจักรสถานะจำกัด ที่สามารถลอกเลียนแบบพฤติกรรมของเครื่องจักรเป้าหมายโดยการสังเกตอินพุตและเอาต์พุตของเครื่องจักรเป้าหมาย วิทยานิพนธ์นี้ได้นำเสนอขั้นตอนวิธีเชิงพันธุกรรมสำหรับปัญหาดังกล่าว และได้ทำการทดลองเปรียบเทียบขั้นตอนวิธีดังกล่าวกับขั้นตอนวิธีต่าง ๆ ที่ใช้ในการแก้ไขปัญหาเดียวกัน ผลจากการทดลองแสดงให้เห็นว่า ขั้นตอนวิธีที่ได้นำเสนอนั้น มีประสิทธิภาพในการทำงานที่ดีกว่าวิธีอื่น ๆ ที่นำมาเปรียบเทียบ นอกจากนี้วิทยานิพนธ์ฉบับนี้ยังได้ทำการวิเคราะห์ขั้นตอนวิธีดังกล่าว เปรียบเทียบกับขั้นตอนวิธีอื่น ๆ ซึ่งผลจากการวิเคราะห์ได้ชี้ให้เห็นถึงแง่มุมต่าง ๆ ที่น่าสนใจในเรื่องของขั้นตอนวิธีเชิงพันธุกรรม

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต .....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่ออาจารย์ที่ปรึกษา .....
ปีการศึกษา	2545	ลายมือชื่ออาจารย์ที่ปรึกษาร่วม .....

##4470370121 : MAJOR COMPUTER ENGINEERING
KEYWORDS : GENETIC ALGORITHM / FINITE STATE MACHINE / INDUCTIVE INFERNECE

NATTEE NIPARNAN : A GENETIC ALGORITHM FOR FINITE STATE MACHINE INFERENCE. THESIS ADVISOR : ASSOC. PROF. PRABHAS CHONGSTITVATANA, Ph.D., 99 pp. ISBN xxx-xxx-xxx-x

This thesis attacks the problem of finite state machine inference. The objective of the problem is to synthesize a finite state machine that can mimic the target machine by passively inspecting the input/output of the target machine. This work proposes a genetic algorithm for the problem. The experiments are carried out to compare the performance and the efficiency of the proposed algorithm. The result indicates that the proposed algorithm outperforms other methods. This work also gives an analysis of the algorithm in comparison with other algorithms. The analysis shows interesting issues in Genetic Algorithms.

Department	Computer Engineering	Student's signature .....
Field of study	Computer Engineering	Advisor's signature .....
Academic year	2002	Co-advisor's signature .....

Acknowledgements

This work can not become like this present stage without numerous contributions, effects and side-effects from many people, either unintentionally or on purpose. I would like to use this invaluable section to express my gracious gratitude to all of them.

First and foremost, I would like to thank my advisor, Assoc. Prof. Dr. Prabhas Chongstitvatana, for his valuable advice and continuous support, for his encouragement in many ways and, above all, for my first impression in Genetic Algorithms shown by him since my early days of computer science education several years ago.

I also would like to show my gratefulness to all my colleagues who live, work, endure and graduate in this same master of computer engineering curriculum. They are invaluable friends. Moreover, I could not express enough gratitude that could match with many things done by all members of Intelligent System Laboratory that always support me in many ways.

Many friends who work and live in another different world, a world without tons of research papers to read, a world without everyday anxiety about how to publish even one small paper, also deserve lots of gratitude for providing a ran-off place for me when life of graduate students are too overwhelming. Although they are tired of me when I start my conversation with the abstract and end it by giving a future research. Above all, they always give inestimable advice from another point of view. I love them all.

Most important, I would like to thank Miss Nipaporn Vitsalapong for not only be a good friend but also motivate me in the despair days of thesis writing, though she might not aware of her motivation, directly. Without her, this thesis surely take much longer time to complete. I would like to bestow all my appreciations to her.

Finally, I would like to thank my parents for having continuous support and encouragement, though sometimes indirectly. They are very patient for years and months that I spent as a graduate student, for nights that they are anxious what kept me up so late at night in the dark cold laboratory. Of course, some of those nights are spent for no good but they do not seem to mind that much. It might be because that I never told them.

This research was not sponsored by any grant nor funding from any institution, except from time to time endorsement from my parent, though sometime unwillingly, of course.

Nattee Niparnan

December 20, 2002

Contents

Contents (cont.)

Contents (cont.)

List of Tables

Page

List of Figures

Page

CHAPTER I

Introduction

Someone had said that most theses can be classified according to their propositions. Under such classification, this particular thesis is considered to be in a group which says "the process X is a better way to do the task Y than any previously known method." In this thesis, the *task Y* is the inference of finite state machine which is consistent with a given set of input/output sequences and the process X is the Genetic Algorithm newly proposed in this thesis.

The problem of finite state machine inference is a kind of induction. When looking up a famous dictionary, "induction" is "the act or process of reasoning from a part to a whole, from particulars to generals, or from the individual to the universal; also, the result or inference so reached." A *part*, *particulars* or *individual* in this problem are the given input/output sequences while a *whole*, *generals* or *universal* are finite automata. The importance of inference, one that makes inference useful, is that the inferred things, though based on only parts or particulars, do capture the sense of the whole or the general. The problem in this thesis is to infer a finite state machine from some given examples such that the inferred machine has high accuracy in identifying other *unseen* examples generated from the same source.

The algorithm presented in this thesis infers a machine by a method that is based on a fundamental concept in inference, an *identification by enumeration*. It searches systematically in the space of possible solutions to find one that agrees with given example. Finding a machine that agrees with given examples without any restriction is arbitrary easy and usually does not possess abilities to identify unseen data. The algorithm that restricts the size of the result could better identify unseen data. The problem can be said more precisely as an "inference of a *compact* finite state machine from the given examples". This inference problem is considered to be a passive inference. Passive inference means that the learner does not and can not control the data it receives. There are many proof showing that the problem is NP-Complete.

Why one should try to learn finite automaton? A more general question is, why one should interest in inventing something that has capability of learning? The answer lies within the fact that learning is adaptive in nature. One good example is the case of

hand writing recognition. When faced with hand writing recognition, one could write a program with fully predetermined set of rules which describe hand writing recognition, assuming that these rules are known in advance. Another choice is to write a learning method that takes some sample handwriting text and let that method determine rules by itself. The second approach is more robust. For examples, rules might differ from person to person. The first approach might suffer from this problem while the second one can adapt itself to fit any handwriting. Of course, the learning approach has some disadvantages. It must undergoes a learning period which might takes very long time. Moreover, it usually could not learn rules as complex as ones that can be determined by human. However, adaptation is very useful and can solve many problems. After all, the existence of a program that could learn, one that possess some sense of intelligence, is one ultimate goal in computer science.

Then comes another question, why finite automaton is a good representation. Finite automaton is a fundamental model of computing. Its applications span in a broad range such as sequential logic, parser, lexical analyzer, etc. Many control models are represented by finite automaton such as communication protocol. To be able to infer finite automaton for such task could be very beneficial.

The problem of inferring a finite state machine, or more general, a problem of inferring any form of language, has been attacked by many researchers. This thesis concentrates on the study of the inference of a Mealy mode finite state machine by using Genetic Algorithms.

Genetic Algorithms are widely used in the field of Machine Learning. This problem is also one of a case. There are various approaches for the problem, beside Genetic Algorithms, and many of them show very promising results. Of course, one algorithm does not fit all, especially in the broad problem like this one. There are many cases with different restrictions and criteria that make one method more preferable than other methods. Genetic Algorithms have their own unique goodness which are argued to be suitable in many situations. Improvement over previous Genetic Algorithms for the problem can result in a good alternative algorithms that is beneficial. This thesis proposes a new genetic algorithm for the problem that is better than former Genetic Algorithms.

Some researchers had used Genetic Algorithms in this specific problem, e.g., a work in (Aporntewan, 1999). The work of Aporntewan emphasizes the hardware implementa-

tion 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.

1.1 Claim

A thesis has to claim something and this very thesis is not an exception. As stated very early in this chapter, a proposition made in this thesis is in the kind of *"the process X is a better way to do the task Y than any previously known method"*. The process X is the Genetic Algorithm method proposed in this thesis and the previously known method is limited to Genetic Algorithm-based methods.

Formally, the claim made in this thesis is "A new Genetic Algorithm method proposed in this thesis is a better way to solve the problem of finite state machine inference than the former Genetic Algorithms."

1.2 Scope of the Work

This thesis is about the improvement of Genetic Algorithm methods. Though it attacks the well-known problem, it does not set up to propose a grand new method that beats every other methods. It is intended to outperform the methods that use the same approach, Genetic Algorithms. In the subsequence chapter, the goodness of Genetic Algorithm approaches is discussed to show that there are the cases that Genetic Algorithms are suitable. The thesis limits itself to the improvement of Genetic Algorithms for the problem. The goal of the thesis is to introduce a new genetic algorithm method, to compare it with a former method and to show that the new method is correct and better than the prior one.

The claim made in the previous section must be accompanied by some evidence that supports it. It is hard, according to nature of Genetic Algorithms, to show mathematically that any method is better than any other methods. This work limits itself to *empirically* demonstrate that the proposed method is better than other method on some test data. Of course, the empirical result itself is not strong enough to support the claim especially with the presence of a theorem like No Free Lunch theorem (Wolpert and Macready, 1995, 1997). To further strengthen the support of the claim, this work also gives some discussion and analysis of the result.

1.3 Organization of the thesis

The thesis is organized as follows. Chapter 1 provides the overview of the thesis and the problem. It describes what the problem is and why it is important. After that, it states the contribution of the thesis, its scope of the work and the hypothesis. This chapter is an introduction that is intended to be readable by non-specialist.

Chapter 2 is a survey of related works and works which are relevant to this thesis. Mainly, there are three categories of works in this chapter. The first category is about the work that studies the characteristic of the problem. The second one is the work that attacks the problem and the final category is the work that uses genetic algorithm to solve same class of problems.

Genetic Algorithms, which are the conceptual model of method used in this thesis, are described in Chapter 3. The introduction, basics of Genetic Algorithms are discussed along with the implementation of the simplest form of Genetic Algorithms. The fundamental theory of Genetic Algorithm, the Schema Theory, is also briefly described.

Chapter 4 defines technical terms and definitions that are used throughout this thesis. Most definitions follow ones that are given in standard text books. Some new definition is needed to be defined, nevertheless. It provides a precise, unambiguous definition for terms before they are used. However, some new terms might be introduced later on in the following chapter when it is appropriate.

Chapter 5 is the exhaustive detail of two genetic algorithm methods which are compared in this thesis. They are described at the level of implementation detail. Some of pseudo-codes of the methods are also presented in this chapter.

Chapter 6 describes the framework and the detail of the experiments. The setup of the experiments are given in this chapter. It discusses how the experiments are designed and created, what the experiments want to measure, how the measurement is measured. Finally, the results of the experiments are shown.

Chapter 7 gives the analysis of results and the method presented in this thesis. It conjectures on the theory of the behaviour of the proposed method. Finally, the conclusion is given in Chapter 8 along with recommendation for future research.

CHAPTER II

Literature Review

This section gives a survey of works relating to this thesis. The works are divided into three categories. There are works involving in characterizing the problem, algorithms for the problem, and Genetic Algorithms for related problems.

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 something as simple as a finite state automaton to something as complex as Turing machine. Grammatical inference itself is a sub-field of *Inductive Inference*. In summary, inductive inference is a process that takes some examples and conjectures a general rule that describes the given examples. For instance, a process is given some pairs of a value x and a value of some particular function at the point x, $(x, f(x))$ and the process is required to hypothesize what that particular function f is. The problem of finding a finite state machine which is consistent with given input/output sequence set can be thought of as an inductive inference problem by considering that the given set is examples and a machine that can correctly identify it is a conjectured rule. A good survey of inductive inference can be found in (Angluin and Smith, 1983).

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, 1978). Furthermore, Pitt and Warmuth (1993) shows that even finding a deterministic finite automaton whose size is polynomial in the size of the minimum solution is also NP-complete. If a uniform-complete sample up to some particular length n is given, the problem can be solved in polynomial time on the size of samples but the size of a uniform-complete samples itself is exponential on the number of states. It is also shown that even when some fixed small fraction of uniform-complete samples is missing, the problem still be an NP-complete (Oliveira and Silva, 2001). The work in (Dupont et al., 1994) is a good short summary on the complexity of automaton identification. Thought the problem is shown to be intractable, it seems to be practical on average as there are many algorithms that do well in the task which are discussed in the following paragraphs.

For the solution of the problem, Biermann proposes algorithms in (Biermann and Feldmann, 1972) and (Biermann et al., 1975). The algorithm uses an extensive search approach. The algorithm starts by constructing a prefix tree acceptor which describe all given examples. The algorithm then tries to find a mapping of the states of the prefix tree acceptor to states of a 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. A notable improvement over the method of Biermann has been recently done in (Oliveira and Silva, 2001). Their method incorporates many techniques in searching, such as *non-chronological backtracking* and *conflict diagnosis*. When the search method reaches 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 avoids unnecessary search in all contradictory branches below the beginning point of the cause. Conflict analysis improves the method furthermore by analyzing and memorizing a set of parameters that causes dissatisfaction. Whenever these parameters are detected again in any other branch of the search, 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. Although the time used per node of Oliveira's method is higher than Biermann's, Oliveira's can reduces the number of unnecessary backtracking effectively. In fact, the work of Oliveira and Silva is recognized as 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. A story of finite state machine inference by evolutionary computation are dated far back in the history. Fogel, et al, (1965) (reprinted in (Fogel et al., 1998)) use an evolutionary programming to infer a finite state automaton. Dupont (1994) proposed the GIG method which is based on Genetic Algorithms to infer automata accepting 15 different regular languages. The GIG method constructs a *maximal canonical automaton* or *MCA* for the given positive examples. The MCA is a non-deterministic form of a prefix tree acceptor. The MCA itself can correctly recognize the positive examples but not the negative examples. The GIG method then uses Genetic Algorithms to find a *partition* of states in the constructed 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 also designed to have preference on a small machine.

Aporntewan and Chongstitvatana (2000), use Genetic Algorithms 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 a 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 machines. The motivation behind the work is to construct a hardware that can mimic other hardwares.

The work in (Manovit et al., 1998) studies the relation between the length of given sequence and the accuracy in identifying unseen data of the solution. The method used in the work is Genetic Algorithms that produces the GAL structure circuit that is consistent with a given sequence. The accuracy is a correctness percentage that a consistency machine (GAL circuit) is behavioral equivalent to the target machine that generates the input/output sequence. The work defines the lower bound and upper bound lengths of the sequence for the selected test set. The work shows that a sequence that is shorter than the lower bound length can not yield accurate solutions and the correctness percentage saturates at the upper bound length. The correctness percentage can not be raised to 100 percent. In general, the longer sequence of examples yields higher correctness percentage. Chongstitvatana and Aporntewan (1999) 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 Algorithms is not limited to an inference of finite state machines, but it also extends to an inference of many kind of structures. There are various works that use genetic algorithms to generate a structure capable of recognizing a language. Lucas (1994) attacked a context-free grammar inference problem by using biased chromosome representation. He succeeded in inference three alphabets palindromes. Lankhorst (1995)

used Genetic Algorithms for the induction of nondeterministic pushdown automata that can recognize a language from given positive and negative examples. The automata use the acceptance by empty stack approach. The algorithm encodes a 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 examples, and the size of stack used. The method is compared with the GIG method in (Dupont, 1994) and other methods that use neural networks as an acceptor. The result shows better achievement over the other methods. Pereira, et al, (1999,2000) attacked the problem of constructing a Turing Machine to perform a specific task. The task is to write as much "1" as possible on an initially blank tape and halt after that. These works use Genetic Algorithms with special techniques designed for the problem such as special chromosome presentation (Machado et al., 1999) and graph-based-crossover (Pereira et al., 1999b).

Another related problem is the problem of learning a deterministic finite automaton from given positive and negative examples. The goal emphasizes in a generalization, which means that the learned DFA must be able to correctly identify *unseen* data. Lang and Pearlmutter set up the *Abbadingo One* (Lang, 1997) competition to promote the development of better algorithms for this particular problem. There are many interesting results from the work. Notable one is the Event Driven State Merging (EDSM) algorithm and its variation described in (Lang et al., 1998). The algorithm is based on the *state merging* method introduced by Trakhtenbrot and Barzdin (1973). This method constructs a prefix tree acceptor that describes the given positive and negative examples and then folds it up by merging compatible pairs of states. However, one could not know whether the states that are going to be merged are exactly the same state in target automata or not. A merging of states usually introduces other constrains to a merging of later states. It is crucial that early merges should be as correct as possible. So, merging are done according to some measurement on the evidence that the merging states are the same. It is very important to merge states that is most likely to be the same first. There are various measurement for the evidence. The original algorithm by Trakhtenbrot and Barzdin merges states in breadth-first order. Breadth-first order of merge is a coarse estimate of evidence of merging since near-rooted nodes have large subtree beneath them, and that subtree can be used as an evidence for compatibility as pointed out in (Lang, 1992). However, a better strategy is to measure the evidence directly instead of using some predetermined order that correlates with the evidence. The EDSM algorithm calculates the

evidence measurement of every pair of nodes before performing any single merge. However, extensive calculation of every pair takes too much time. A winner in *Abbadingo One* (there are two winners) use a modified version of EDSM that filters some pairs out of evidence calculation. It is possible that the filtered out candidates of merging might be good ones but it is usually unlikely. The filtered EDSM algorithm, called "blue-fringe", turns out to be much faster and simpler while it suffers only slightly inaccuracy when compared to the reference EDSM.

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 compared as noted in (Oliveira and Silva, 2001). The blue-fringe algorithm, while results in more generalized hypothesis, makes no warranty on the size of the conjectured machine and normally it is not the minimum size. The state merging based algorithm also loses the ability to correctly identify the finite automaton when the size of the training set gets smaller as noted in (Oliveira and Silva, 2001). However, blue-fringe algorithm is faster and much more scalable.

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

1. A Genetic Algorithm Approach: This approach uses Genetic Algorithms to search among space of possible machines of limited size to find one that is consistent with given data. An example is the method in (Aporntewan, 1999).

2. An Exact Minimum Search Approach: This approach is based on the algorithm proposed in (Biermann and Feldman, 1972). It tries to map states in prefix tree acceptor of the given example into a minimum possible finite state machine.

3. 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. This approach does not limit the size of the result but it can find the result in very short amount of time. It also scales well when the size of the problem increases.

It seems that the third approach is the most successful one if main consideration is on the size of the problem. Methods in this approach are 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 method that has size restriction as noted in (Oliveira and Silva, 2001).

The second approach, in contrast, puts 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 a 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 (Chongstitvatana and Aporntewan, 1999).

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 between two previous approaches. It gives 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 recognizes the advantage and intends to improve the work in a genetic algorithm approach.

In conclusion, when a user is faced by the automaton inference problem, that user can choose among these three approaches. One way of determining the appropriate algorithm is to consider the constraint of the problem at hand. First, if the problem does require that the result automaton must be as small as possible, the exact minimum approach is the best. If the problem does not have size restriction, the possible choice are the heuristic approach and the Genetic Algorithms approach. Which one should be used can be considered from the size of the training set. If the size is quite small, the GAs approach is more promising. If not, Biermann's-based algorithms should be used instead. the GAs approach also has other benefits. It is easily customizable. For examples, if the problem requires that the size of the machine must be fixed to a particular size which is smaller than minimum, a GAs based method can be easily modified to find near-consistency solutions but it is not quite clear on how to modify the minimum search approach. Figure 2.1 shows a guideline of the methods choosing for the problem.

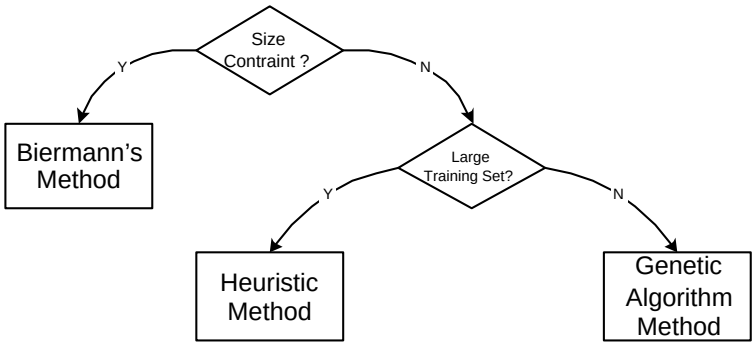

Figure 2.1: Method Choosing Guideline

2.1 Summary

This section reviewed some interesting researches that are related to the work. Three fields of works are presented here. They are the characteristic of the work, the algorithms for the problem and the Genetic Algorithms for the problem. The problem of automaton inference captures a lots of interest from many researchers. Many works from theoretical and practical point of view have been done. In summary, the problem is known to be very hard in theory but there are many promising algorithm that can solve the problem on average cases, for a reasonable small size, of course. Genetic Algorithms are a choice of many researchers, not only for this problem but for many problems alike. For this specific problem, algorithms for it are reviewed here and a recommendation for choosing of algorithms is made according to some characteristic of the problem.

This chapter and the previous chapter have laid out some introduction of the work. After this point, the thesis will concentrate on describing the essence of the work. The next chapter starts by providing details about a conceptual model of the work, the Genetic Algorithms.

CHAPTER III

Genetic Algorithms

This chapter describes the conceptual model of the method used in this thesis, the Genetic Algorithms. This chapter starts by explaining Genetic Algorithms in practical sense. A basic form of Genetic Algorithms is described in detail along with its implementation. After that, the fundamental theory of Genetic Algorithms, the Schema Theorem, is discussed. This chapter is intended to provide basic introduction to Genetic Algorithms. A reader who is familiar with Genetic Algorithms might skip this chapter entirely.

Genetic Algorithms are classified to be in a group of method called *evolutionary computation* which refers to a method that uses some form of evolution as a major part of the process. Original Genetic Algorithms were introduced since 1975 by John Holland (1992) but evolution-based computation approaches are dated back much prior than that. However, the most popular forms of evolutionary computation are Genetic Algorithms and its variants called Genetic Programming which is introduced by Koza in (Koza, 1992). Genetic Programming differs from Genetic Algorithms in a way that they manipulate their data. Genetic Algorithms use fixed-length representation while Genetic Programming use dynamic representation. This chapter focuses entirely on Genetic Algorithms.

3.1 Basics of Genetic Algorithms

Genetic Algorithms are a class of function optimizer algorithms, though their actual applications span in a wide range of problems from machine learning to solving of intractable problems. Genetic algorithms, introduced by John Holland since 1975, are search procedures inspired by evolution. In general, Genetic Algorithms maintain a collection of potential solutions called population. Each potential solution, usually called as an individual, is evaluated to check how "goodness" it is. The measurement of each individual is problem dependent. Some of solutions are selected according to their "goodness" values to survive to the next generation. Some of survived solutions, if not all, then undergo series of operations to create new solutions. The newly created solution is called as an offspring. A collection of offsprings forms a new population. The process is repeated until some conditions are met. The selection method and the operators are designed to drive population better and better by implicitly identifying and assembling of critical information.

Genetic Algorithms assume few assumptions about the problem being solved. This renders Genetic Algorithms to be a very general approach. This has both advantages and disadvantages. The generality allows Genetic Algorithms to be applicable on a broad range of problems. On the other hand, generality comes with the cost of performance, Genetic Algorithms do not exploit much of information of a problem that the problem might provides. Genetic Algorithms usually are not competitive in a field in which there are known specialized algorithms for the problem. For example, though genetic algorithms are known as function optimizers, they perform very poorly when compared with typical numerical methods, e.g., 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 global search methods that can be applied to non-differentiable or multiple local optima problems. NP-Complete and intractable problems are good candidates as a problem instance where genetic algorithms may be superior.

The Genetic Algorithm procedure is described briefly as follows. Genetic Algorithms begin with a collection of (typically random) potential solutions encoded in a chromosome-like data structure (a bit string). The word *chromosome* and *bit string* refers to a potential solution which is encoded into a form that Genetic Algorithms can perform their operators on. In contrast, The word *solution* and *individual* refer to the potential solution. Whether that solution is in an encoded form or a decoded form is not specific. With a collection of individuals, Genetic Algorithms enter a loop. The loop is repeated until the stopping criteria is met. The criteria might be a limit of time or a discovery of the solution. There are two main steps in the loop. The first step is the evaluation of population. This step measures the goodness of each individual. The next step is to select some candidates from the population and then applies some recombination operators on the selected 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 one with low “goodness”.

Genetic Algorithms are inspired by evolution. They can be thought of as a simulation of evolution in nature. A collection of solutions can be regarded as a population of an arbitrary specie. That specie lives for a period of generations. Some individuals die

and some individuals are born. The entire specie evolves through their living and dying. Each individual in the population has a chance to survive. A highly fit individual has a better chance to mate, breed and pass its gene which describes its characteristic to the next generation. In the end of generation, a “good” individual is hoped to emerge.

Genetic Algorithms assume very few assumptions. Normally, there are two facts which are problem dependent that genetic algorithms require. The first is how the solution of the problem is encoded for processing. The second is how “goodness” value of each solution is measured. Genetic Algorithms encode potential solutions in bit strings. It implies that every parameters in a solution must be discretizable. The second assumption implies 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. On first assumption, when some parameters of a solution must be discretized into *exactly* some numbers that is not a power of 2, for example, if the value of the parameter X has an admissible presentation only as an integer in the range of precisely 0–20. The number of bits required to map these 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 an inadmissible value. This method usually results in low performance. For such problem, the user has to use some techniques to cope with this issue. The other issue is evaluation. Some problems do not have “exact” evaluation function. This is usually a case where a solution is some kind of program or code. This kind of problem needs simulation-based evaluation. In such case, an evaluation function is just an approximation of partial performance of the solution. To have more accurate evaluation function, the simulation process must be precise hence takes more time. There is a trade off between time and accuracy of evaluation.

3.2 Canonical Genetic Algorithm

There are various variations of Genetic Algorithms. This section describes an implementation of one form of Genetic Algorithms originally proposed by Holland (1992) known as Canonical Genetic Algorithm or sometimes as Simple Genetic Algorithm. The section concentrates mainly on the implementation of the algorithm.

Canonical Genetic Algorithm starts with a collection of potential solutions. 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 runs 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 individual. The algorithm evaluates every individual by the specific evaluation function depending on the problem. After that, each individual is assigned with a *probability of selection* which is calculated from the value of the evaluation function. The *probability of selection* influences 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, 1993). 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 dependent on the others'.

The next step in the loop is a reproduction step. It consists of two sub-processes, a selection and a recombination. Selection chooses some individuals with probability according to their *fitness function* ($f_i/\overline{f}$). It is possible that some individual is selected more than once. 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." The newly selected population is called *intermediate population*.

The recombination is applied with some probability. Two individuals from the previously created intermediate population are selected and then the operator called *crossover* is applied to produce new individuals. The individuals which are selected to be recombined are called *parents* and the generated individuals are called *offsprings*. The parameter p_c indicates how often that the operator is applied. When the operator is applied, the offsprings replace their parents. Searching in Genetic Algorithms is mainly done by

recombination because it generates new points in the search space. The parameter p_c is usually set to a high value.

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

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

Swapping portion of these two parents string produces following offsprings:

```
11abba
ba0011
```

After a crossover is done, the mutation operator is applied to the intermediate population (which some of its individuals are replaced by new offsprings). Mutation randomly flips some bit in the chromosome with probability p_m. Mutation is used to ensure that every bit position is never fixed to some value. Normally p_m is set to a very low value, e.g. 1% or less.

Reproduction step results in a new population. The old population is discarded and the new one acts as the population in hand. Canonical Genetic Algorithm proceeds to another iteration of the loop, i.e., it goes on an evaluation of the new population and then recombines again and again until the iteration is stopped.

3.3 Schema Theorem

Why does genetic algorithm work, after all? Or more precisely, what makes the Genetic Algorithm an effective search algorithm? Most widely accepted explanation is the theorem laid down by John Holland as described in (Goldberg, 1989).

The theory suggests that, although Genetic Algorithms seem to search on a population of individuals, in fact, they *implicitly* search in a larger search space of hyperplanes. Genetic Algorithms benefit from the fact that there are a much larger number of

hyperplanes than the number of individuals in the population itself. The hyperplanes are implicitly sampled in the population. The implicit searching receives a special name as *implicit parallelism* or *intrinsic* (Whitley, 1993). 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 labeled as depicted in Figure 3.1. The cube is labeled `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 represent 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 *schema*.

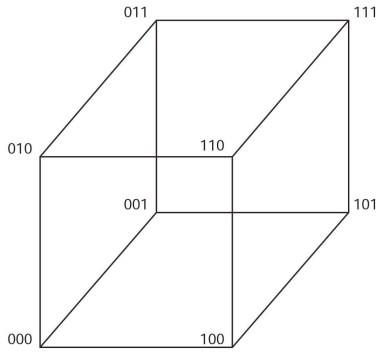

Figure 3.1: A 3-dimensional cube

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 matches 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 bit 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 Algorithms also work with much larger number of hyperplanes possessed in those individuals. In fact, a population of individu-

als contains information about how many representatives does each particular hyperplane have 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 idea 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.

Selection alone is enough to change the number of representatives of each hyperplane but selection itself does not generate new sampling point in the search space. Recombination operator and mutation are the ones that are responsible.

Crossover affects the representative of hyperplane as it disrupts some schemata and sometimes it introduces new ones. Disruption by crossover occurs when crossover point lies within critical parts of a schema. For example, consider a schema `***11***`, when a cross site falls between the first and the second `1`'s, this schema is destroyed, except that the other parent incidentally has `1` in that position. A chance that crossover affects any particular schema depends on the value of that schema. For example, consider the 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 for this schema is $(L-1)/(L-1) = 1$, any cross site separates 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 destroyed by crossover. This behavior reduces a chance on surviving into the next generation that each schema has. This effect should be minimized. It can be done by using a *compact representation* with respect to schemata. A compact representation suggests that highly fit schemata should have short distance of the fixed bits since it will have less chance to be affected by crossover.

Mutation also disrupts schemata since mutation flips bit in a schema. However, mutation should not be omitted entirely because it prevents some schema from permanently loss from a population. Usually, it is possible that a population converges into some schemata. This causes 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 *pre-*

mature 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 predicts a lower bound 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 a 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 which is executed with probability p_c sometime destroys a schema H. In fact, crossover sometime creates schema H by operating on a pair of strings that coincidently contain *only* portion of H. To make things simple, a creation of a schema H by crossover is neglected. Let p_d denote the probability that a schema H is destroyed by crossover. The equation above changes into inequality:

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

A value p_d can be computed from properties of a 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, schema `****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 is $\Delta(H)/(L-1)$ where L is the length of this schema. There is one exception, crossover will not destroy a schema 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 representative 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:

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

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

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

Substitution 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 affects the proportion of the schema. Let $o(H)$ be the *order* of a schema H. An order of a 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 a particular schema H is $(1 - p_m)^{o(H)}$. This yields the final inequality:

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

In conclusion, this inequality says that the proportion of schema H in the next generation changes from the current generation according to its fitness value. Above average schema will grow while under average schema will die off. Its growth and dead rate are perturbed by its order and defining length. The higher the order and the defining length, the higher that it is perturbed.

3.4 Summary

This chapter described the basics of Genetic Algorithms from both practical and theoretical points of view. It started by giving the overview of Genetic Algorithms, its characteristic, its potential and usefulness. The simplest form of Genetic Algorithm, the Canonical Genetic Algorithms was described next. In general, Genetic Algorithms are a class of search algorithms inspired by evolution from nature. The Schema Theorem which is the fundamental theory of Genetic Algorithms was also presented. The theorem said that a low order, short defining length schema grows according to its fitness value. For more detail about Genetic Algorithms, the reader is suggested to read the standard book such as (Goldberg, 1989).

The next chapter will formally define definitions and terminologies that are going to be used in this work.

CHAPTER IV

Formal Definition

This section defines many definitions which will be used throughout this thesis. It also formally describes the problem statement. This work follows the same definitions as in (Hopcroft and Ullman, 1979). Some important definitions are presented in this chapter.

4.1 Definition

Definition 1 *A Mealy machine M is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where Q is the set of states, Σ is the input alphabet, Δ is the output alphabet, $\delta(q, a) : Q \times \Sigma \rightarrow Q$ is a transition function, $\lambda(q, a) : Q \times \Sigma \rightarrow \Delta$ is an output function and q_0 is the starting state.*

This work assumes that the size of Q is m, the size of Σ is n and the size of Δ is o. Unless state otherwise, q denotes a particular state, i denotes a particular input and o denotes a particular output. λ is a function mapping $Q \times \Sigma$ to Δ, that is, $\lambda(q, a)$ is an output associated with the transition from state q on input a.

For simplicity, it is assumed without loss of generality that states of the machine is a set of number $\{0, 1, \ldots, m\}$ where m is the number of states of the machine. The input and output alphabets are assumed in the same way. For a set of alphabet of size a, the set is assumed to consist of $\{0, 1, \ldots, a-1\}$. For example in the case of binary input/output, input and output alphabets are $\{0, 1\}$ both.

Usually, there are many cases that a transition function and an output function apply to a state and a string. It is convenient to define a new transition function and an output function that can apply to both a string and a symbol.

Definition 2 *A function $\hat{\delta}$ is a function from $Q \times \Sigma^*$ to Q. The function $\hat{\delta}$ is defined as*
1) $\hat{\delta}(q, \epsilon) = q$
2) $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$
where ϵ is an empty string, w is a string and $a \in \Sigma$ is an input alphabet.

Definition 3 *A function $\hat{\lambda}$ is a function from $Q \times \Sigma^*$ to Δ. The function $\hat{\lambda}$ is defined as*
1) $\hat{\lambda}(q, \epsilon) = q$

2) $\hat{\lambda}(q, wa) = \lambda(\hat{\lambda}(q, w), a)$
where ϵ is an empty string, w is a string and $a \in \Sigma$ is an input alphabet.

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 $1 \leq i \leq n$

Definition 4 *Let $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a Mealy machine. An input/output sequence S of length n is an ordered pair (I, O) where $I = i_0 i_1 \ldots i_{n-1} \in \Sigma^n$ and $O = o_0 o_1 \ldots o_{n-1} \in \Delta^n$. An input/output sequence set $\zeta = \{S_0, S_1, \ldots, S_{|\zeta|-1}\}$ is a set of input/output sequences.*

Definition 5 *Let $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a Mealy machine. M is said to be consistent with input/output sequence S = $(i_0 i_1 \ldots i_{n-1}, o_1 o_2 \ldots o_{n-1})$ iff $o_j = \lambda(q_0, i_0 i_1 \ldots i_j)$ for all $0 \leq j \leq n$. M is said to be consistent with input/output sequence set ζ iff M is consistent for all $S_i \in \zeta$.*

Definition 6 *A walk W on machine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ by input/output sequence S of length n is a sequence $q_0, q_1, q_2, ..., q_n$ where $q_i \in Q$ such that, $q_i = \hat{\delta}(q_0, i_0 i_1 \ldots i_j)$. In other words, a walk W is a sequence of states that was visited when an input/output sequence S_M is fed to machine M.*

4.2 Problem Statement

Given the input/output sequence set ζ_M, which is composed of a number of input/output sequences where the input is randomly generated and the corresponding output are collected from a target machine M whose number of states 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 be less than $2m$. The number of input and output alphabets of a machine is assumed to be a power of two.

Basically, an input/output sequence is a continuous string of input/output observed from the target machine. There can be more than one sequence per one problem instance. A problem instance is considered solved when we find a Mealy machine that produces the same output as the sequence when we feed the input from the sequence to it.

4.3 Terminologies and Symbols

This section defines some terminologies that are used in this thesis. Unless it is stated otherwise in the context, the words listed in this section assume the meaning as described here.

Target machine refers to a machine M where the input/output sequence set is generated from.

Evolving machine denotes an intermediate machine that is in the process of searching for a consistent machine. The *evolving machine* is usually inconsistency except at the end of the search that yields the correct result. In that case, it is preferred to call that machine as an *evolved machine*.

Hypothesis machine denotes a machine that is a result of an inference method. It is *not* necessary that the machine is consistent since it is possible that an inference method might wish to yield an inconsistent machine.

Example Data refers to the data that are given as a training set for learning, sometime it is called *given example* or *training data*.

Individual, sometimes referred to as a **solution**, is a point in the search space. Usually, it refers to a member of population in general with no specification in its state. That member can be in encoded form which is ready to be manipulated by operators of Genetic Algorithms or can be in decoded form ready to be evaluated.

Chromosome refers to a string used to represent an individual in Genetic Algorithms, usually a binary string. It is an individual in an encoded form. It can be used interchangeably with the term **bit string**.

Gene refers to a group of bits in chromosome that contributes one parameter value when decoded. For example, A bit position 0, 1 and 2 in a chromosome of some particular encoding might be decoded into one parameter with 8 possible values that determine some particular characteristic of that individual. That group of bits at position 0 to 2 is called a gene.

In the upcoming chapters, some pseudo-code is presented. There are new symbols that are used often. These symbols are not widely used in general. They are defined as

follows.

The first one is the random assignation symbol $\xleftarrow{\$}$. The $\$$ indicates randomness. Formally, a statement

$$x \xleftarrow{\$} P$$

where P is a set means that one element of P is randomly selected with equal probabilities and assigned that element to the variable x.

Another symbol is the size of input/output example in the sequence set $\| \zeta \|$. The symbol $\| \zeta \|$ represents the summation of the lengths of all sequence in the set ζ. This symbol is formally defined as follows.

$$\| \zeta \| = \sum_{S \in \zeta} (|S|)$$

4.4 Summary

This chapter stated many definitions that are going to be used in upcoming chapters. It also formally stated the problem and gave some terminology that will be used throughout the thesis.

The next chapter will describe two methods which are subjects of this thesis. These methods, Genetic Algorithm-based methods, are used to solve the problem of finite state machine inference. The first method is the reference method used by many researcher and the next method is the newly proposed method.

CHAPTER V

Genetic Algorithm Methods for the Problem

This chapter describes two genetic algorithm methods which are used to solve the problem of finite state machine inference. These methods are the reference method which is used by other researchers and the newly proposed method. Both of them are described from the implementation perspective. The explanation is in step-wise manner starting from the encoding, the evaluation, the operators, and finally, the initialization and stopping criteria.

5.1 Reference GA Method for the problem

This section describes the reference Genetic Algorithms for the problem. The method is used in (Aporntewan, 1999), (Aporntewan and Chongstitvatana, 2000) and (Chongstitvatana and Aporntewan, 1999). Essentially, the method searches among the space of all Mealy finite state machines for a consistent machine. A machine is evaluated by taking the given input sequence and then comparing the generated output with the give output sequence. The method uses standard operators, which are single point crossover and standard mutation.

The reference method searches for a Mealy machine. A Mealy machine can be described by specifying $Q, \Sigma, \Delta, \delta, \lambda$ and q_0. So, the reference method has to search for all that variable. For this problem, an input and output alphabets (Σ and Δ) can be observed directly from the given input output sequence. So, Σ and Δ are lefted out from the search. Since a state labelling does not affect consistency of the machine, Q is assumed to be $\{0, 1, \ldots, |Q| - 1\}$ and the starting state q_0 is also assumed to be the state which is labelled as 0. By using this assumption, it turns out that Q, Σ, Δ and q_0 are already described. The method is then have to search for a transition function δ and an output function λ.

5.1.1 Encoding Scheme

The problem statement states that the required solution must has at most $2m$ states where m, the number of states of the target machine, is known in advanced. One way to achieve this is to set the size of the evolving machine to be $2^{\lceil \log_2(|Q|+1) \rceil}$ states. This makes the number of states of an evolving machine to be a power of two which is guaranteed

to be more than m and less than $2m$. Being a power of two has a benefit that when a chromosome is decoded back to a machine, it will always result in a valid machine. The same benefit also applies to the input and output values since it is stated in the problem statement in Section 4.2 that the number of output and input alphabets be a power of two.

A transition function and an output function are encoded in a bit string. The encoding is done in a transition-wise manner. A machine is encoded transition by transition, starting from the first transition of the state 0 and then the next transition of the state 0 and so forth. The order of transitions and states are determined by the value of input alphabets. Since the input alphabets and state labels are starting from 0 to the appropriate value, the order of transitions and states can be easily determined. For each transition, its next state value and output value are encoded into an individual. Figure 5.1 shows the encoding scheme.

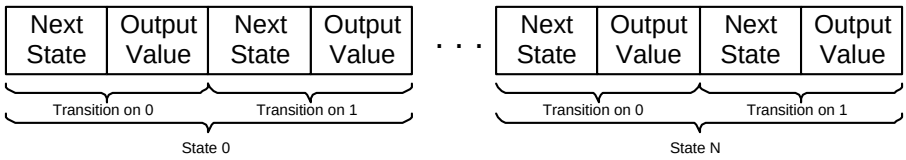

Figure 5.1: Reference Encoding Scheme

The number of bits required to encode next state value and output value is determined by the number of states and the size of output alphabets, respectively. Let $|Q|$ be the number of states of an evolving machine, then $\lceil \log_2 |Q| \rceil$ is the number of bits required to encode a next state value. As same as next state value, the number of bits required to encode an output value is $\lceil \log_2 |\Delta| \rceil$. The length of a chromosome for encoding a machine can be computed from Q, Σ and Δ. Consider an evolving binary input/output machine with 8 states for example. The number of bits required to encode a next state value and an output value is $3 + 1$. There are two transitions for each state so it requires $(3 + 1) \times 2$ bits for each state. This machine has 8 states which means that the length of a chromosome is $(3 + 1) \times 2 \times 8$ bits.

Generally, the length of a chromosome for encoding a machine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ is

$$(\lceil \log_2(|Q|) \rceil + \lceil \log_2(|\Delta|) \rceil) \times |\Sigma| \times |Q|$$

Table 5.1 shows the length of chromosome used to encode a binary input/output Mealy machine at different size.

Table 5.1: Chromosome Length of the Reference Method

Target Size	Evolving Machine Size	Bits for Encoding Q Plus Δ	Chromosome Length
2–3	4	2+1	24
4–7	8	3+1	64
8–15	16	4+1	160
16–31	32	5+1	384

5.1.2 Evaluation Function

Evaluation is done straightforwardly. The method takes all input sequences of the problem instance and feeds them to an evolving machine. The machine then produces its output sequences from the given input sequences. The output sequences generated from the machine are compared bit-wise with the associated given output sequences from the problem instance. Oen point of evaluation value is awarded for each similar bit between the output of the machine and the given output sequence. The maximum score of this evaluation is the size of input/output sequence set $\| \zeta \|$. Figure 5.2 and Figure 5.3 illustrate the evaluation process and the pseudo-code of the evaluation function, respectively.

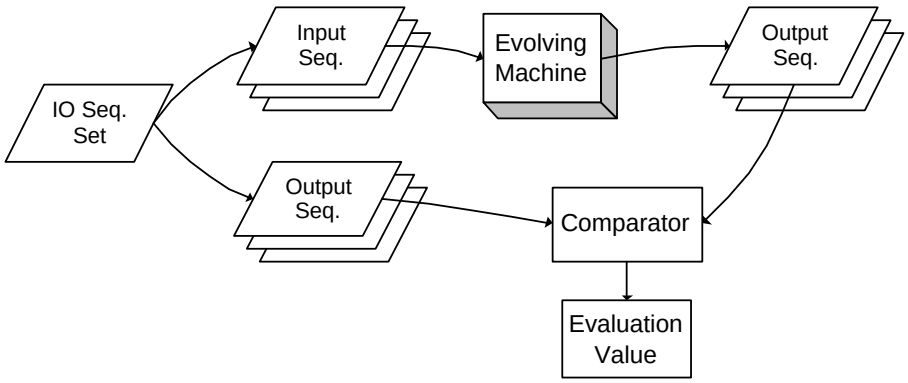

Figure 5.2: Reference Evaluation Function

5.1.3 Operators

This section describes the selection operator which is used to determine which individual is going to survive to the next generation. This section also describes the genetic

```
1  funct Evaluation(M', ζ)
2    begin
3      score ← 0
4      for j := 0 to |ζ| − 1 do
5          reset M
6          Let I be the I component of S_j = (I, O)
7          Let O be the O component of S_j = (I, O)
8          for k := 0 to |I| − 1 do
9              Let p and q be the k^th character of I and O, respectively
10             feed p to M, let y be the output of M
11             if (y == q) then score ← score + 1 fi
12         od
13     od
14     return score
15   end
```

Figure 5.3: Reference Evaluation Function Pseudo-code

operator which is used to manipulate a bit string.

Originally, the work of Aporntewan (1999) uses a modified version of *combined rank selection* (Mitchell, 1997) with the intention to preserve the diversity. The modified version sorts individuals many times. This makes the selection process a very time consuming process. However, the preliminary study indicates that, when using tournament selection instead of combined rank selection, the method could find the required solution in less real world time. So, the reference method in this work uses tournament selection instead of combined rank selection. The tournament selection is associated with a *tournament size* parameter t. The tournament selection works by picking up t individuals at random from the population. After that, it selects one individual which has maximum evaluation value out of the chosen t individuals. That selected individual is going to survive to the next generation. Figure 5.4 lists the pseudo-code of the tournament selection where P is a set of individuals and $E(x)$ is an evaluation function. The function returns an intermediate population P' as a result.

The next operator to be discussed is a genetic operator. The reference method uses the standard one point crossover and standard mutation operator which are used in Canonical Genetic Algorithm described in 3.2. The one point crossover cuts chromosome of parents at some random point and swaps the portions of those parents to create two new offsprings. The operator is executed at probability p_c. The standard mutation mutates every bit in a chromosome with probability p_m.

```
1  funct TournamentSelection(P, t, E)
2    begin
3      score ← 0
4      P' ← ∅
5      for j := 1 to |P| − 1 do
6          ind_1, ind_2, ..., ind_t ←$ P
7          ind ← ind_j such that E(ind_j) ≥ E(ind_1) ... E(ind_t)
8          P' ← P' ∪ {ind}
9      od
10     return P'
11   end
```

Figure 5.4: Tournament Selection Pseudo-code

5.1.4 Starting and Stopping Criteria

The method starts from random individuals. Each individual is randomly created in a form of a chromosome. Each bit of a chromosome is determined by flipping a fair coin, i.e., every bit in a chromosome has an equal chance to be `1` or `0`.

Two criteria identify the stopping condition of the method. First, a method stops when a consistent machine is found. A machine is known to be consistent when it does not produce any mis-value output which means that its evaluation score is equal to the summation of the length of all sequences in the input/output sequence set. Another stopping criteria is a generation limit. A method is given some fixed amount of generations it could produce. When a method produces that many generations and it still can not find a consistent machine, the method is forced to stop.

5.2 New Genetic Algorithm Method for the Problem

Basically, the new method is the modified version of the reference method. The modification is done to improve some aspects of the reference method. The new method differs from the reference method in the encoding scheme, the evaluation function and the operators. The initialization and stopping criteria of this method are the same as the reference method. This section describes these modifications.

5.2.1 New Evaluation Function

This section describes the new evaluation method. It starts by describing the process of the new evaluation function and then it go on describing the main idea and meaning of

the new evaluation function.

The new evaluation function works just like the old evaluation function. Its feeds the given input sequence to the machine alphabet by alphabet. For each alphabet, it remember the internal state of the machine so that it can exactly know which transition is being use on the current alphabet. Now, instead of comparing the generated output alphabet of the machine with the corresponding given output alphabet, it counts that the transition being used is associated with the corresponding given output alphabet. By doing this on all given input output sequence, it knows that how often that each particular transition is associated with each particular output alphabet. This information will be used to calculate the evaluation value.

To calculate the evaluation value, it simply scans every transition of the machine. For each transition, it knows that how many time that each output value is associated with that transition. It picks the most frequently associated output alphabet and remembers the frequency of that alphabet. The summation of the frequency of the most often associated output alphabets on every transition is the final evaluation value.

The idea is implemented by maintaining an *Output Count Table*, OC. The OC is a three-dimension array of size $|Q| \times |\Sigma| \times |\Delta|$. The $OC[q, i, b]$ stores the frequency that the output b is associated to the transition of the state q on the input value i. OC is initialized to zero for all elements at start. On feeding of the sequence S to the evolving machine M', for all $0 \leq j \leq |S| - 1$, $OC[\hat{\delta}(q_0, i_0 i_1 \dots i_{j-1}), i_j, o_j]$ is increased by one. When all $S \in \zeta$ is fed to M', $OC[q, i, o]$ will represent the frequency of output o that is mapped to the transition of state q on the input i. To calculate the evaluation value for M', the evaluation function scans OC through all output values on each transition to find the maximum value of that transition and sums the maximum of all transitions. Formally, the evaluation function F can be defined as

$$F(M') = \Sigma_{q=0}^{|Q|-1} \Sigma_{i=0}^{|\Sigma|-1} \max_{b \in \Delta}(OC[q, i, b])$$

Figure 5.5 shows the pseudo-code of the new evaluation function. The evaluation function takes the evolving Mealy machine $M' = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ and input/output sequence set $\zeta = \{S_0, S_1, \dots, S_{|\zeta|-1}\}$ as inputs. It returns integer value as the evaluation value.

```
1  funct Evaluation(δ, ζ)
2    begin
3      for j := 0 to |ζ| − 1 do
4          state ← q_0
5          Let I be the I component of S_j = (I, O)
6          Let O be the O component of S_j = (I, O)
7          for k := 0 to |I| − 1 do
8              Let p and q be the k^th character of I and O respectively
9              OC[state, p, q] ← OC[state, p, q] + 1
10             state ← δ(state, p)
11         od
12     od
13     score ← 0
14     for j := 0 to |Q| − 1 do
15         for k := 0 to |Σ| − 1 do
16             score ← score + max_{l∈Δ}(OC[j, k, l])
17         od
18     od
19     return score
20   end
```

Figure 5.5: New Evaluation Function Pseudo-code

The following paragraph shows the execution of the evaluation function on a small example. Figure 5.6 shows the example evolving machine. The output of the machine is not displayed because it does not involve in the evaluation. Let the given input/output sequence set consist of only one sequence $S = (0010101, 0110000)$. The evaluation starts by resetting the machine to an initial state, making the current state as the state A. The first input in the sequence is 0 and the corresponding output is 0, so $OC[A, 0, 0]$ is increased by one and the current state of the machine is changed to B, according to its transition function. The next input and output value is 0 and 1 respectively so $OC[B, 0, 1]$ is increased by one. The current state of the machine still is B since the transition of state B on input 0 directs to B itself. The rest of the sequence is fed to the machine in the same way. The OC table after all input/output is consumed is shown in Table 5.2.

Table 5.2: Sample 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

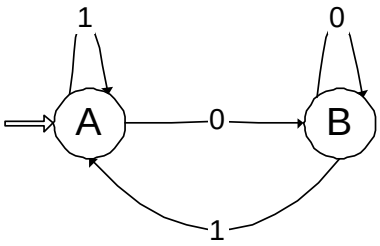

Figure 5.6: Sample Mealy Machine

The evaluation value of this machine under the given sequence is the summation of maximum frequency on every transition, which is 3 (from the transition of state A on input 0) plus 1 (from the transition of state B on input 0) plus 0 (from the transition of state A on input 1) plus 2 (from the transition of state B on input 1) the total is 6.

At this point, the new evaluation function is described formally. However, two questions might arise. The first is what exactly is the meaning of the new evaluation function. The second one is why does the output function of the evaluating machine play no role in evaluation. To answer these questions, the main idea of the new evaluation function is described in the followings.

First, let us consider the reference evaluation function more closely. The main drawback of the reference method described in the previous subsection is that the evaluation value, which is the number of similar output bits between the generated output of the evolving machine M' and the given output sequence, is not effectively evaluate the FSM as it should be.

For example, consider two binary input/output finite state machine, one is a target machine $M = (Q, \Sigma, \Delta, \delta, \lambda_0, q_0)$ and the other one is an evolving machine $M' = (Q, \Sigma, \Delta, \delta, \lambda_1, q_0)$. Let $\lambda_1 = \neg\lambda_0$. When M' is evaluated by a sequences set ζ generated from M, the result from the evaluation is always 0 regardless of what ζ is. This is because the output of M' always differs from the output of the sequences. The evaluation value of 0 indicates that the evolving machine M' is very bad. However, the transition function of M and M' are the same, the only difference between M and M' is the output function which is totally different. Usually a description of an output function is much shorter than a description of a transition function, therefore the encoding of M and M' are not much different. In this case, the evaluation value indicates that M' is very bad but in fact, M' is

very close to the target machine M.

The idea behind the new evaluation function is to make the evaluation value to be output-value-independent. So that evaluation value is not affected when the output function is wrong. the evaluation value depends entirely on transition function. The method can be thought of as an implicit local search for the best output function, i.e., the new evaluation function implicitly tries to find an output function such that it maximizes the score under the reference evaluation function.

Now, let us get back to a calculation of an OC table. An OC table is *not affected* by an output function of an evaluating machine. No matter what output alphabet is assign to each transition, the way that each transition is associated with the given output alphabet is still the same. So, what exactly is OC table indicates? The $OC[q, i, b]$ indicates that, if the transition of state q on input i of the evaluating machine is b, the score it would get under *reference evaluation* is $OC[q, i, b]$. For example, let us consider the previous example of evaluation again and concentrate on the transition of state B on input 1. $OC[B, 1, 0]$ is 2 and $OC[B, 1, 1]$ is 1. It indicates that the transition of state B on input 1 is used 3 times, two of them is associated with 0 and one of them is associated with 1. If the output value of this transition is assigned as 0 and evaluate it with reference evaluation, the evaluation value that this transition will contribute is 2 because this transition will produce correct input (0) two time, and will produce incorrect input (1) one time. Since $OC[q, i, b]$ indicates the score it would get, the $\max_{b \in \Delta}(OC[q, i, b])$ will indicate the *maximum* score it would get under reference evaluation.

The new evaluation function, which sums $\max_{b \in \Delta}(OC[q, i, b])$ on all transition, indicates the maximum score that the evaluating value would get under reference evaluation. This is the main idea of the new evaluation value, the implicit local search for the best output function.

Now, it is made clear what is the idea of the new evaluation function. Next, the other question will be addressed. The question is "why the output function of the evolving machine play no role in evaluation." One might also notice that although the new evaluation value is the maximum value that the evolving machine would get under reference evaluation, the evolving machine *is not exactly* the machine that can produce maximum score because its output value might be wrong. This problem is addressed by two step. First, the output function of the machine is entirely discarded at the beginning of the algorithm.

Next, whenever the evolving machine is really required to produce an output, its output function will be defined according to the given input output sequence set. Be noted that the entire process of the new evaluation value does not require *any* output to be produced at all. The redefinition of an output function can be postponed until it is required. For this new genetic algorithm, the output function of the machine is required at the final stage of the algorithm, when the algorithm has to answer the result as a machine. So, the redefinition is done only once, when the algorithm is stopped and the machine is evolved.

The redefinition of the output function is done according to the OC table. Simply, the output of the transition of state q on input i is b such that $\forall x[OC[q, i, b] \geq OC[q, i, x]]$. Verbally, the redefined output value is the most frequently associated output alphabet. However, it is useful to explain the redefinition more closely. For simplicity, the symbol "$\delta(q, i)$" is use in place of "the transition of state q on input i."

Three cases of output redefinition for $\delta(q, i)$ is possible. The first case is the case that no output is associated to $\delta(q, i)$ at all. This case happens when state q is unreachable from the starting state or when the input/output sequence does not exercise $\delta(q, i)$. The second case is the case that there is only one value of output is associated to $\delta(q, i)$, e.g., $\delta(q, i)$ is associated with the value 0 four times and no other value of output is associated to $\delta(q, i)$. The last case is the case when more than one distinct value of output is associated to $\delta(q, i)$. For example, $\delta(q, i)$ might be mapped with 0 two times and mapped with 1 one time. This case is called as a "conflict" case. The three cases mentioned earlier indicate different situations of the output redefinition.

First, the no-mapping case means that the input/output sequences have nothing to do with $\delta(q, i)$. When M' is evaluated by the reference method, $\delta(q, i)$ will not effect the evaluation value. In this case, any arbitrary output alphabet can be assignd to $\delta(q, i)$ because it will not affect the consistency of the machine with the given input output sequences. Second, the one value mapping case means that $\delta(q, i)$ is associated by only one output alphabet and the output of $\delta(q, i)$ should be defined as that alphabet. Finally, the "conflict" case which has special meaning. It indicates that M' can not be consistent for the given sequence for any possible output alphabet. Since there are more than one value of output is mapped to $\delta(q, i)$, suppose that the output value of $\delta(q, i)$ is b, it always has some other output value b' of the given sequences such that it is not equal to b and it is also associated to $\delta(q, i)$. However, if this machine is really have to be put in to use, the output value of this transition is redefined to the value that is most frequently associated to

$\delta(q, i)$ since redefining in this way yields the maximum score. Choosing the other value, the score it would get is less than the maximum which means that it is less consistency.

In conclusion, the new evaluation function redefines the output value for each transition such that it can maximize the score. The evaluation function counts the number that each output is mapped to each transition and uses this number as a guide for output redefinition. The principle of redefinition is to choose the most frequently mapped output value in a transition as the actual output value of that transition. The maximum frequency itself also denotes the evaluation value contributed by each transition. It is also possible that some transition is not mapped by any output. In such case, the output of that transition can be defined by any arbitrary value because it does not effect the consistency of the machine.

5.2.2 New Encoding Scheme

From the evaluation method in the Section 5.2.1, the output function is no longer needed to be evolved because the output function is implicitly defined by the transition function and the given sequences. So, it is not required to encode an output function into an individual. The output part is removed from the encoding, leaving only a transition function in the new encoding scheme. The new encoding scheme is organized the same way as the reference encoding except that the output value is not included in the encoding. Figure 5.7 illustrates the new encoding scheme.

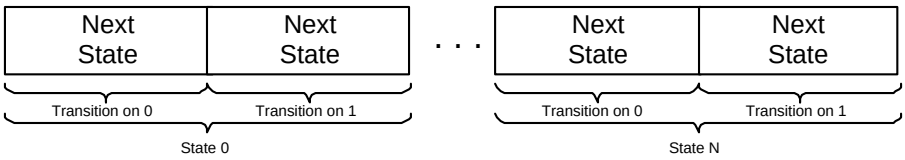

Figure 5.7: New Encoding Scheme

Since an output function is no longer encoded into a chromosome, the length of chromosome is shorter. The $\lceil \log_2(|\Delta|) \rceil$ bits for encoding Δ is removed. Thus, the chromosome length is given by

$$\lceil \log_2(|Q|) \rceil \times |\Sigma| \times |Q|$$

Table 5.3 shows the length of chromosome for the new encoding scheme at different size.

Table 5.3: Chromosome Length of the New Method

Target Size	Evolving machine size	Bits for encoding Q	Chromosome length
2–3	4	2	16
4–7	8	3	48
8–15	16	4	128
16–31	32	5	320

5.2.3 New Crossover Operator

Genetic Algorithms work by mixing small highly fit sub-solutions called *building block* into larger highly fit sub-solutions and finally into the solution itself. However, the simple GA can not effectively mix building block in a hard problem unless a proper linkage information is known and put into use (Thierens, 1999). A hard problem, in building block mixing points of view, is a problem whose building block are not compactly represented in its encoding, i.e., genes that form the building block does not locate close to each other in a chromosome. When genes, or schema, that constitute a building block are not tightly located, there is a high chance that they will be destroyed by a crossover operator. Some mechanism has to be used to cope with this issue.

For the problem in this thesis, it is possible that building blocks are not compactly encoded. The ways that a machine is encoded into a bit string implicitly introduces possibility of non-compact representation. The machine is encoded in a chromosome according to the label of each state. Under different labeling, a machine can be encoded into a different chromosome. Since an evaluation value of a machine is computed according to the topology of the transition function, the labeling does not effect the evaluation value. Nonetheless, state labeling affects how a machine is encoded into a chromosome. Since the crossover operator is performed at chromosome level, the state labeling effect how individual are recombined. For an example, consider the case when the walk by the given input sequence set passes exactly three states of the machine. The evaluation value of this machine is affected only by these three states (or more precisely, these three states with their transition function). These three states are the building block of this problem instance. These three states can be labelled by any value according to the initial population. If that three states are labelled close together, they will be compactly located in the

bit string. However, if they are not closely labelled, they will be separated. Also, by the effect of mutation, these three states might get loss or reappear under different labelling. The fact that building blocks of this problem are not fixed to pre-defined gene positions, i.e., they could move to any position according to the state labeling, makes it possible that the encoding might be non-compact at any time. Non-compactness makes standard recombination operator performs poorly. A new crossover operator is proposed in this section to cure the problem.

The main idea of the new operator comes from the fact that the new evaluation function evaluates a machine according to its transition function. Since the new evaluation function is done by feeding input output sequences into a machine, parts of the machine that make contribution to an evaluation value are the states that are in the walk by the input output sequences. If, somehow, the walk by the given input/output sequence is detected, that walk can be considered as a building block.

When looking into a machine and concentrates on a transition function and its states, a machine can be regarded as a directed graph where its states represent nodes in the graph while its transition function represents edges. Usually, the given input/output sequence is quite long (more than 30 bits) thus genes that constitute building blocks are ones that lie in a state correlating with depth first order of vertices in the graph. The new crossover is designed to capture building blocks that consist of vertices of a graph that are arranged in depth first fashion. The new crossover is described as follows. The new crossover operator takes two parents and produces one offspring. It starts by selecting the better parent and then traverses the graph of that parent in depth first search manner starting from the state q_0. It creates a list of states according to their order of visit in the traversal. The list represents the states of the graph which are arranged in depth first fashion. After that, the list is randomly cut. The cut indicates the states of the first parent that should be propagated to the offspring. The other missing states of the offspring will be taken from the other parent. Figure 5.8 illustrates a sample execution of the new crossover operator.

The process can be thought of as a single point crossover applied to the list of state rather than on the chromosome directly. The process can be divided into two sub processes, the first one is the state list generating and the second is the crossover on that list. Figure 5.9 shows the pseudo-code of the state list generation. The new crossover is shown in Figure 5.10 where δ_1 and δ_2 are parents and δ_o is an offspring.

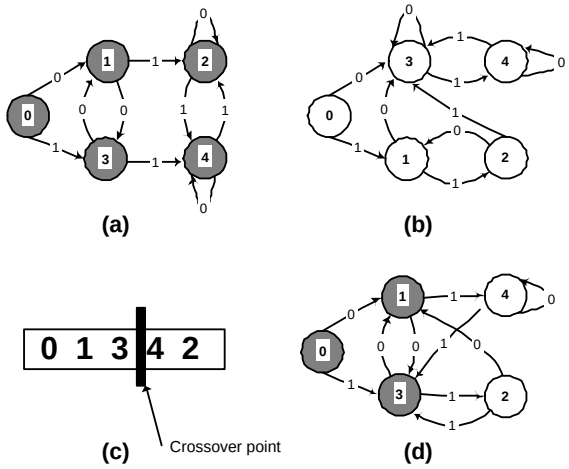

Figure 5.8: Example of crossover operation. **(a)**: First parent. **(b):** Second parent, we assume that (a) is fitter than (b). **(c):** The list of vertices of (a), sorted in an order of visiting by depth first traversal. It is assumed that the crossover point lies between the third and the forth vertex. **(d):** Offspring of the operation.

This new crossover operator produces one offspring that is composed of the head (states which their labels are in the first part of the list) of the fitter parent and the tail (states which their labels are not in the first part of the list) of the other parent. The reason that this crossover produces only one offspring is that the depth first search of two parents is unlikely to be the same. If another offspring is created from the tail of the fitter parent and the head of the other parent, it is unlikely that the head of this new offspring that comes from the lesser fit parent captures the building block of that parent because the list is generated from the fitter parent, not the other parent. The head of the machine is very important since it is usually used by evaluation more often than the tail. If the second offspring is really necessary, it is preferable to use the list of states generated from the lesser fit parent than the first parent. The process of generating another list from the lesser fit parent and crossing again takes times as much as doing another round of crossover except for the time used to select the fitter parent which is very small. In that case, it is more reasonable to produce one offspring and let another offspring be produced by another round of crossover since new parents can be selected instead of using the lesser fit parent.

5.3 Summary

This chapter described two Genetic Algorithms that are going to be compared and analyzed in the next few chapters. Both methods, the reference method and the new

```
1  funct DepthFirstOrder(M)
2     Let S be a stack
3     Let L be an empty array
4     begin
5        Make S empty
6        Push(S, q_0)
7        idx ← 0
8        Marks q_0
9        while S is not empty do
10             q ← Pop(S)
11             L[idx] ← q
12             idx ← idx + 1
13             for i := 0 to |Σ| − 1 do
14                if q is not marked
16                   Mark q
17                   Push(S, δ(q, i))
18                fi
19             od
20       od
21       return L
22    end
```

Figure 5.9: New Crossover: State List Generation Pseudo-code

method, are described into their implementation details. The reference method is a straightforward method. It searches for an appropriate transition function and the output function. The new method is based on the reference method with some modifications that address some problem faced by the reference method. Essentially, the new method is different from the reference method in encoding scheme, the evaluation function and the recombination operator. The key part of the new method is the new way of evaluation that acts as a local search and the new crossover operator that is aware of the underlying structure of an individual.

The next chapter will concentrate on the empirical comparison between the methods presented here. It describes the experimental setup and the results of the experiment.

```
1  funct CrossOver(δ1, δ2, Q, Σ)
2    begin
3      score ← 0
4      L ← DepthFirstOrder(M1')
5      CrossOverPoint ←$ {0, 1, ..., |Q| − 1}
6      δo ← δ2
7      for i := 0 to CrossOverPoint do
8        for j := 0 to |Σ| − 1 do
9          δo(i, j) ← δ1(i, j)
10       od
11     od
12     return δo
13   end
```

Figure 5.10: New Crossover: Offspring Generation Pseudo-code

CHAPTER VI

Experiment and Results

This chapter describes the framework of experiments and compares the proposed method with other methods. The experiments are carried out to provide evidences that justify the claim made in this thesis. The experiments emphasize on comparing the performance of the new method with the existing methods. Two sets of experiments are presented. The first one is the experiment that compares the performance of the new method with the reference method in Section 5.1. The result of the first experiment can be used as the evidence which supports or disproof the primary claim of this thesis, "A new Genetic Algorithm method proposed in this thesis is a better way to solve the problem of finite state machine inference than the former genetic algorithm." The other experiment is a supplementary experiment. It compares the correctness of the solution produced by the proposed method with the solution produced by the heuristic approach. For each experiment, the section starts by describing its design and its measurement. It finished by providing the detail of the experiment.

6.1 Experiment A: Performance Comparison between GA-Based Methods

This work aims to develop a better genetic algorithm for the problem of finite state machine inference. However, due to stochastic nature of Genetic Algorithms, it is hard to proof the superiority of one method over another method formally. Hence, this work uses empirical results to demonstrate the improvement of the new method over the reference method. To compare the new method with other methods, this experiment runs them on the same test set and measures some quality of the result. Two measurements are used to compare these methods, one is the number of problems which are solved and the other one is the time required to solve each problem. 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.

This experiment is designed to compare the reference method with the new method presented in Section 5.2. The new method has three techniques that are different from the reference method. Two of them, which involve in encoding and evaluation, can not be easily decomposed, so they are combined together and is called *reduced encoding* technique. The last technique involves a recombination operator and has been given the

name of *graph preserved crossover*. The detail of these techniques can be found in Section 5.2.

In this experiment, three different methods are compared together. The first one is the reference method. The second is the reference method with reduced encoding technique turned on, this method is called as `NEW1`. The last method, referred as `NEW2`, is the `NEW1` method with graph preserved crossover feature turned on.

The experiments are designed to justify the hypothesis in Section 1.1, which is "The new genetic algorithm method proposed in this thesis is a better way to solve the problem of finite state machine inference than the former genetic algorithm." As described in previous paragraph, a better way means that it could find an answer faster and it could solve more problems. Hence, the experiments are designed to measure the time used to solve the problems and the number of problems solved by each method.

6.1.1 Measurement

As stated before, this experiment measures two things, the first is the how fast that a method could find an answer and the second is how often that a method could solve the problem. There are two sub-measurements that are used to measure the time. Totally, there are three measurements in this experiment. They are described as follows.

Two measurements are used to compare on how fast each method is. They are the number of generations used before a method could find an answer and the real world time. The number of generation used reflects the number of function evaluation that is called by the method, which indicates how fast an algorithm searches for the answer. However, different methods mean different implementations, thus the wall clock time used by each algorithm to perform each sub-task in genetic algorithm process is different. In this work, the new method uses more complicated encoding, evaluation and recombination which require more CPU time than the reference method. These differences in CPU time do not exhibit in the number of generations used but it affects the real world time used.

The number of generations used is a subject of interest by theorist since it represents, in the theoretical view, the efficiency of an algorithm to reach the desired solution. However, for the experimentalist, the performance is the real world time used to run the algorithm. In this case, the property of concern is rather the real world time than the number of generations used.

By the stochastic nature of the method, each run of the method might take different time. This experiment uses the average value over 10 runs of execution as a measure. This applies to both the number of generations and the real world time used.

Formally, the number of generations used to solve a particular problem instance is the average number of generations, over 10 runs, which are produced by a method until that method stops. If no answer could be found for any single run, the generation limit, which is used as a criterion for stopping, is then counted as the number of generations for that run. The real world time measurement is defined in the same way. The third and the last measurement is the number of runs that yield solution. It measures how often a method yields an answer. As stated in the previous chapter, the Genetic Algorithms for the problem can stop according to two criteria. The first one is that it finds an answer and it stops. The second one is that it reaches generation limitation and still could not find an answer. So, it is possible that on some run, a method could not find an answer. Each problem instance is run repeatedly 10 times. The number of runs that yield solution is the number of run, out of these 10 runs, that has the answer.

6.1.2 Experiment

These experiments were run using all three methods, the reference method, `NEW1` method, and `NEW2` method, on the same problem set. The problem set consists of 80 problem instances. Each problem instance is an input/output sequence set. Each sequence set is randomly generated and each set contains 20 strings of 50-bit sequence which totals as 1,000 input/output pairs per one problem instance.

There are many ways to create an input/output sequence set. A target machine is created first and then it is used to create a problem instance. Creating a test set in this way has a benefit that the target machine is known in advance and it can be used to provide some information such as a target size. A target Mealy machine of size m is created by constructing a random directed graph of $\frac{5}{4}m$ nodes where each node has exactly an out-degree of 2. Every edges of the graph are labeled by either `0` or `1` by flipping a fair coin. After that, a node is randomly chosen to act as a root node and then all unreachable nodes are removed . Following that, the machine is reduced by the implication chart method as described in (Katz, 1994). This process results in a machine that has the number of states near m. If the size of the reduced machine is not m, the procedure is restarted again until the machine of exactly m states is found. Figure 6.1 shows the pseudo-code of the

```
1  funct RandomMachine(m)
2    begin
3      do
4          Q ← {0, 1, 2, ..., m − 1}
5          Σ, Δ = {0, 1}
6          q0 ← 0
7          for i := 0 to M − 1 do
8              a, b ←$ {0, 1, 2, ..., m − 1}
9              δ(i, 0) ← a
10             δ(i, 1) ← b
11             c, d ←$ {0, 1}
12             λ(i, 0) ← c
13             λ(i, 1) ← d
14         od
15         M ← (Q, Σ, Δ, δ, λ, q0)
16         M′ ← Reduce(M)
18     while SizeOf(M′) ≠ m
20   end
```

Figure 6.1: Random Machine Generation Pseudo-code

algorithm.

Currently the procedure has a machine of size m in hand. The next step is to use this machine to create an input/output sequence set. The procedure creates an input sequence of size n by flipping a fair coin and then feeds this input sequence into the machine. The output generated is collected. The output generated from the machine together with the corresponding input sequence made a complete input/output sequence. The process of creating an input/output sequence is repeated until the desired number of sequences is generated. This procedure resembles a random draw of input/output sequence with replacement. Figure 6.2 gives the pseudo-code of generating the input/output sequence set consisting of p sequence of length q from the target machine M.

This experiment uses 40 target machines of size ranging from 2 to 21 states, with distribution of exactly 2 machines per one size of state, i.e., there are exactly two machines that have 2 states and there are exactly two machines that have 3 states and so forth. Two problem instances are generated from each machine. Thus, there are four generated problem instances per one size of a target machine. Figure 6.3 illustrates the generation of problem instances and Figure 6.4 gives the pseudo-code.

For each problem instance, each genetic algorithm method is applied 10 times. The

```
1  funct IOSEQGenerate(M, p, q)
2    begin
3      for j := 1 to p do
4          reset M
5          for k := 1 to q do
6              x ←$ {0, 1}
7              i_k ← x
8              feed x to M, let y be the output of M
9              o_k ← y
10         od
11         S_j = (i_1 i_2 ... i_q, o_1 o_2 ... o_q)
12     od
13     ζ = ⋃ S_i
14     return ζ
15   end
```

Figure 6.2: Input/output Sequence Generation

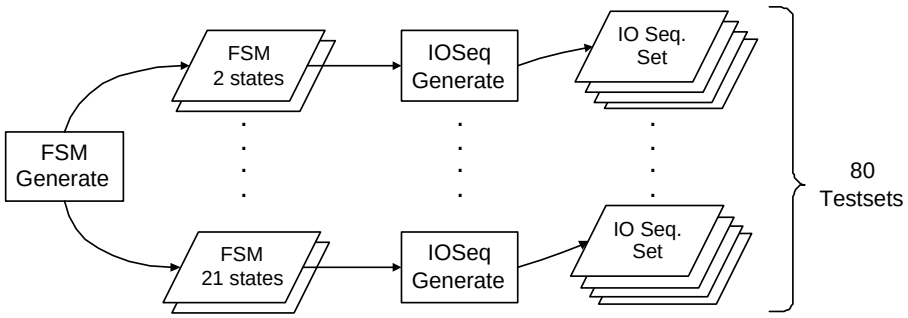

Figure 6.3: Test Set Generation

method has a generation limitation, which means that the method will stop when the number of produced generations exceeds the limit, regardless of the solution. A problem instance is considered solved when there is at least 1 out of 10 runs that yields a consistent machine. The parameters for each genetic algorithm is exactly the same for all method. Table 6.1 summarizes parameters of the experiment. Each algorithm was implemented in C++ programming language and was run on Pentium-III 1GHz with 128MB memories running Linux operating system.

6.1.3 Results

This section presents the results of the experiment. Figure 6.5 shows the average number of generations used by each method. The data are sorted according to the value of the reference method. It can be though that problems are sorted by their hardness

```
1  proc TestSetGenerate
2    begin
3      cnt ← 0
4      for i := 2 to 21 do
5        for j := 1 to 2 do
6          M ← RandomMachine(i)
7          for k := 1 to 2 do
8            ProblemInstance_cnt ← IOSEQGenerate(M, 20, 50)
9            cnt ← cnt + 1
10         od
11       od
12     od
13   end
```

Figure 6.4: Problem Instance Generation Pseudo-code for Experiment A

Table 6.1: Parameters for Every Method

Parameter	Values
Population Size	500
Crossover Rate	0.5
Mutation Rate	0.01
Tournament Size	3
Generation Limit	20000

measured by the reference method. This graph does not provide much of information since lines in the graph have no mathematical meaning, i.e., the data should be plotted as dots instead of lines, but lines provide more visual intuition of the result. Later on, the detailed data are presented. The reference method is labeled as `REF` in the graph. The X-axis is a problem index and Y-axis is an average number of generations. A higher generation used means that it has less performance.

Figure 6.6 displays a fraction of problem solved versus the size of the target machine. The X-axis is the size of the target machine and Y-axis is fraction of number of problem solved. A higher fraction of problem solved means that it has better performance.

It can be seen clearly that the `NEW1` and `NEW2` perform much better than the reference method. However, the distinction between these two can not be easily noticed. When looking more closely in Figure 6.5, especially at the problem number 55 and upward, the `NEW2`, which has new crossover, performs better than `NEW1`. However, in the easy problem instances, the problem number 54 and lesser, `NEW1` turns out to be faster

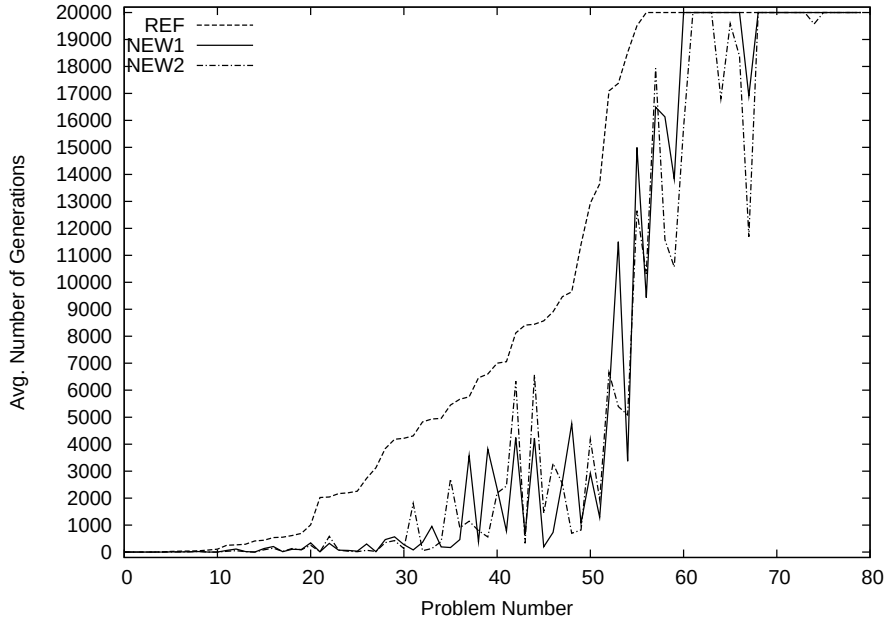

Figure 6.5: Result of Experiment A: Average Number of Generations

than NEW2. Anyway, NEW2 could solve more problems than NEW1 as indicated in Figure 6.6.

Table 6.2 shows the summary result of the experiment. The “Total” value is the total number of generation that the method produced while “Relative” value is the percentage of the “Total” value by the value of the reference method. The “Best Among Other” value indicates number that the method outperforms other method on same problem instance. Please note that when all methods perform the same, which mostly happens when the problem is too hard to solve for all methods, no method is counted as the best among other methods.

Table 6.2: Summary Result in Average Number of Generation Used for Experiment A

	REF	New1	New2
Total (Generation)	749246	527488	505780
Relative	100.00%	70.40%	67.51%
Best among others	0	31	35
Successful runs	461	561	585

The results indicate that, when considering only the number of generations used, the method NEW2 outperforms the method NEW1 since its number of generations used to

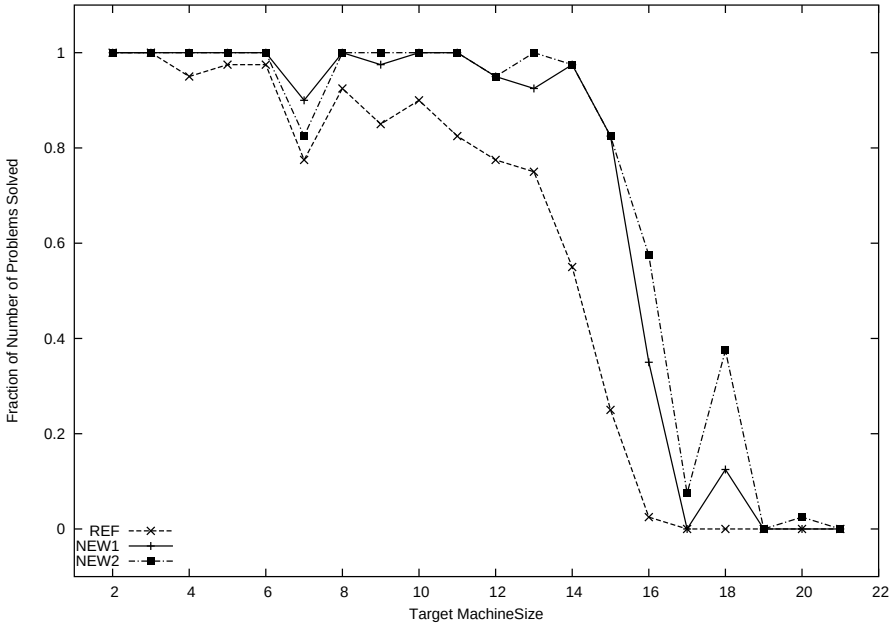

Figure 6.6: Result of Experiment A: Fraction of Problem Solved

Table 6.3: Summary Result in Real World Time Used for Experiment A

	REF	New1	New2
Total (sec.)	64350	46393	51801
Relative	100.00%	72.10%	80.50%
Best among others	2	52	26

solve all problems is less than other methods. The number of runs that `NEW2` could yield a result is also higher than other methods. However, since the new crossover operator is quite complicate and takes much CPU time, when looking into the real world time used, the method `NEW2` took more time than `NEW1`.

6.2 Experiment B: Solution Quality Comparison between GA Method and Heuristic Method

This experiment is a supplementary experiment. It is done to justify the claim that, when the size of the training set is getting smaller, the algorithm that has size restriction will have better accuracy in identifying unknown data than the heuristic algorithm that does not have size restriction. This experiment compares the quality of solutions from the new genetic algorithm method with the quality of solutions from the heuristic-based

algorithm. This comparison pays no attention on how fast or how often it could find the solution since the nature of these two algorithms are quite different such that the comparison has no interest, e.g., on a time-used measurement, the heuristic algorithm mostly run faster than the Genetic Algorithms based method and it is always yields a consistent solution.

The experiment compares methods by running them on the same training set. It calculates the quality of the answer in identifying the unseen data. The size of the problem varies from sufficiently large enough to correctly identify the target machine downward to a very sparse size. The results of the experiment can be used as a supporting evidence to emphasize the claim of the accuracy between both algorithms, empirically.

This experiment emphasizes on the error rate of the solution in identifying the unknown input/output sequence generated from the same machine from which the training input/output sequence is generated. The experiment measures an error rate by using the Cross Validation method (Mitchell, 1997). The next section describes the Cross Validation method.

6.2.1 Cross Validation

Cross Validation (Mitchell, 1997) is a method to compare two learning algorithms when the sample data at hand is limited. Assume that the data at hand is D_0 drawing from a distribution of all possible data D of size n. Cross Validation divides the available data D_0 into k disjoint subsets, namely $T_1, T_2, \ldots, T_k$, of equal size each of which is at least 30. The process then iterates k times, for each time, it uses T_i as an unseen test set and the remaining subsets $\{D_0 - T_i\}$ as a training set. The error rate for each iteration is calculated based on how the result of the algorithm that takes $\{D_0 - T_i\}$ as a training data can correctly identify T_i. The final error rate is calculated from the average of all iterations.

The average value can be viewed as an estimate of the expected value of the difference of error between two algorithms, which is

$$E_{S \subset D_0}[error_D(L_A(S)) - error_D(L_B(S))]$$

where S is a random training data of size $\frac{k-1}{k}|D_0|$, $L_x(S)$ is the hypothesis resulted from the learning algorithm x on training data S, and $error_D(h)$ is an error rate of the hypoth-

esis h on test data D.

The pseudo-code of the Cross Validation method is presented in Figure 6.7

```
1  funct CrossValidation(D_0, L_a, L_b)
2     Let T_1, T_2, ..., T_k be k disjoint subset of D_0 of equal size.
3     begin
4        for i := 1 to k do
5           S_i ← {D_0 − T_i}
6           h_A ← L_a(S_i)
7           h_b ← L_b(S_i)
8           δ_i ← error_{T_i}(h_a) − error_{T_i}(h_a)
9        od
10       return δ̄ ← (1/k) Σ_{i=1}^{k} δ_i
11    end
```

Figure 6.7: Cross Validation Algorithm

6.2.2 Measurement

For this experiment, h_a and h_b in Figure 6.7 are the finite state machines generated from different algorithms. The $error_{T_i}(h_a)$ is defined as follows. Let M be a target machine which generates D_0, let T be a test set and let h be a conjectured machine, the $error_T(h)$ is the number of different bits between the output of h and the output of M when using T as an input sequence for both machines divided by the total number of bits of output sequence. At heart, the error rate is the ratio of the number of correct output bits and the number of total output bits. However, this experiment is set up to measure *accuracy*, not *inaccuracy*. It is more appropriate to use correctness instead of error. Since the error is a ratio of correct output which means that its value ranges from 0 to 1, the correctness is defined as $1 - e$ where e is the error rate. A machine with 0 error means that it has 1 correctness. On the other hand, a machine with 1 error means that it has 0 correctness.

6.2.3 The Experiment

The environment of the experiment is the same as the Experiment A. The code of the Blue-Fringe algorithm can be found in a dfa-learning suite available for download at http://abbadingo.cs.unm.edu/ (Lang, 1997). There are four variations of state merging algorithm in the suite. The first one is the original state merging algorithm introduced by Trakhtenbrot and Barzdin (Trakhtenbrot and Barzdin, 1973) which produces the least accurate result but runs fastest. The second one, called as `red-blue`, is a blue-fringe algo-

rithm of Juille and Pollack who won the Abbadingo One competition. Their algorithm are described in (Juille and Pollack, 1998). The other two programs are EDSM-based methods with no restriction in selecting candidate which result in much slower algorithms. This comparison uses `red-blue` as a representative for state merging algorithms because `red-blue` is accurate and runs in a reasonable amount of time. The GA methods in this experiment is the `NEW2` algorithms used in Experiment A. The parameter of `NEW2` is the same as ones used in Experiment A which is shown in Table 6.1.

There is one issue about the function $error_T(L_A(D)))$ when the learning algorithm L_A is `NEW2`. Since the method `NEW2` is a genetic algorithm, it is possible that `NEW2` could not find the required answer in some runs. However, when `NEW2` stops according to the generation limit, it will have an *evolving* answer in hand even though it is not the one that is consistent with the given training set. In this experiment, when `NEW2` could not find a consistency answer for any run, the *evolving* answer at hand is considered as a result from the algorithm.

The experiment consists of many problem instances of various sizes. Every problem instance contains an input/output sequence set that is generated from the same machine. The size of problem instance varies from the upper bound size downward to a very small size. The upper bound size is the size which the heuristic algorithm could almost perfectly captures the structure of target machine. The upper bound size is determined by running the `red-blue` algorithm on different sample sizes and tests the conjectured machine with a very large set of data. The test uses the same measurement in 6.2.2. The test data consists of 200 sequence of length 200 bits. The upper bound size is the size that the error rate of conjectured machine from `red-blue` is less than 0.01.

The target machines in this experiment are newly generated with the same method as in Experiment A. The size of the machine is 15 states which means that the size of an evolving machine of `NEW2` algorithm is 16 states. The upper bound size of training data for this machine is 36 sequences, each of them has a length of 35 bits. The problem instances are generated from the size of 42 sequences downward to 6 sequences at the interval of 6 sequences and the length of 35 bits downward to 5 bits at interval of 2 bits. The reason behinds these numbers comes from the use of cross validation.

The cross validation used in the experiment divides sample data into 6 distinct subsets of equal size. It divides the given examples along their sequences. For examples, a

problem instance with m sequence of length n are divided into 6 sub-problem instance, each of which has $m - m/6$ sequence of length n. The interval of 6 sequence in the problem instance suite makes the number of sequences in every problem instances be divisible by 6. It makes the sub-problem instance which is created by dividing the set of sequences be in the equal size. Since the number of subsets in cross validation is 6, each of sub-instance has $m/6$ sequences less than the original problem instance. The largest problem instance with 42 sequences results in sub-instance of 35 sequences which is very close to the upper bound size.

In conclusion, there are 112 original problem instances of which the size ranges from 6 to 42 sequences at 6 sequences interval and their length vary from 5 to 35 bits at 2 bits interval. Each original problem instance is redistributed into 6 sub-problem instance for cross validating. Totally, each algorithm is run for 672 times and produces 112 data points. Figure 6.8 shows the pseudo-code of problem instance generation.

```
1  proc TestSetGenerate(M)
2    begin
3      cnt ← 0
4      for i := 5 to 35 step 2 do
5        for j := 6 to 42 step 6 do
6          for k := 1 to 2 do
7            ProblemInstance_cnt ← IOSEQGenerate(M, j, i)
8            cnt ← cnt + 1
9          od
10       od
11     od
12   end
```

Figure 6.8: Problem Instance Generation Pseudo-code for Experiment B

6.2.4 Results

The results are plotted as a graph of the number of examples versus correctness as shown in Figure 6.9.

The results indicate that the answers produced from the both methods do not show any significant difference in accuracy, under Cross Validation method. However, this is coarse conclusion from the results. The analysis of this experiment is presented in the next chapter.

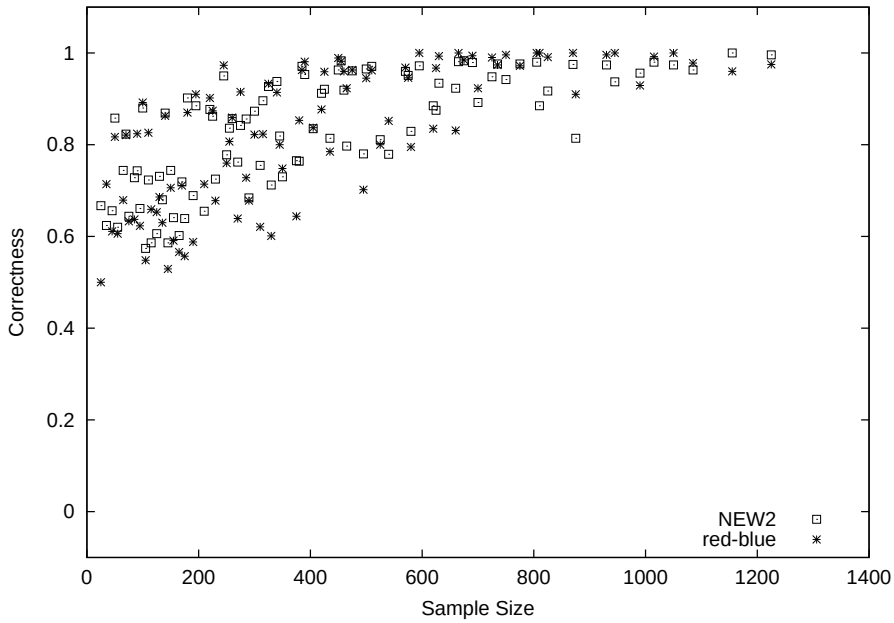

Figure 6.9: Result of Experiment B

6.3 Summary

This chapter described two experiments along with their setup and measurement. The first one is the comparison between the reference method and the new method presented in previous chapter. The result from the experiment indicates that the new method outperforms the former method.

The second experiment is the comparison between the Genetic Algorithms approach and the heuristic approach. It is set up to demonstrate the difference in a correctness of identification of unseen data when the restriction of the size of the result is used and is not used at various sample size. The results from the experiment indicates that GA-based method which has a size restriction does not show any significant difference in accuracy over the heuristic method which has no size restriction.

The next chapter will analyze the result of this chapter and make some conjectures on how these effects are happened.

CHAPTER VII

Analysis of Results

This chapter gives the discussion of the experiment made in the previous chapter. It analyzes the result of the experiment and makes some conjecture regarding what happens in the experiment.

7.1 Discussion of the Experiment A

The result of the experiment in the previous chapter indicates that the method NEW1 performs better than the reference method. The method NEW2 also performs better than NEW1. The method NEW2 can produce more results with less function evaluation count. However, NEW2 takes longer wall clock time in many cases. The results that show the improvement of NEW1 and NEW2 is important but more importantly is how could these improvements happen. This section discusses the mechanism that causes the effects. The better performance of NEW1 and NEW2 over REF is a result of reduction of search space, schema preservation. The difference in time used of NEW2 and NEW1 comes from the difference in time complexity of the process.

7.1.1 Search Space Reduction

Essentially, the effect of the local search, which is implicitly performed by the new evaluation and the new encoding, is the reduction of the search space. Since an output function is not encoded in a chromosome and the evaluation function evaluates a machine in the way such that the output function is always optimized, it can be though that an output function is removed from the search. Naturally, the new search space has different fitness landscape. Whether this new fitness landscape is beneficial to the search is uncertain. However, a reduction of search space usually results in a faster search process for most problem.

Supplementary experiments are set up to check whether the reduction of search space does have an effect to the performance or not. The area of the search space that was reduced is the output function. To illustrate the effect, two experiments are set up. The new experiments are the same as the Experiment A, except that the target machines are generated with two-bit output (four possible output values) for one experiment and three-bit output (eight possible output values) for the other experiment. The machines has

the same size as the Experiment A, two machines per size. The size is ranging from 2 to 21 states. The problem instances are generated in the same way by using these new machines. The two-bit and three-bit output experiments are named Experiment A1 and Experiment A2 respectively. The method that are compared are the `REF` method and the `NEW1` method. The `NEW1` method uses only the new evaluation and encoding but not the new crossover operator. The new crossover operator does not effect the reduction of the search space so it is inappropriate to use it in this experiment.

The result of the experiment is shown in a graph format for better understanding. Two graphs are shown for each experiment, one is graph of the number of generations used for each problem instance. The problem instances are sorted by the value of reference method. This graph is similar to the results graph in the previous chapter that is shown in Figure 6.5. The other graph is the fraction of runs achieving the required solution to the total runs for different sizes of the target machine. For each size, the data are taken from four problem instances. The interesting point in this graph is the area under the line of reference method and over the line of the new method. It roughly indicates the difference between two method, though it does not have real mathematical meaning since the X-Axis is discrete. For this graph, the X-axis is the size of the target machine and the Y-axis is the fraction of problems that are solved. Figure 7.1 and Figure 7.2 show the result of experiment A, Figure 7.3 and Figure 7.4 show the result of experiment A1 and Figure 7.5 and Figure 7.6 show the result of experiment A2, respectively.

When the number of output alphabets is increased, the difference between the size of the search space of the reference method and the new method is getting bigger and bigger. The results indicate that the advantages in term of number of generations of the new method over the reference method is getting more and more noticeable when the output size is increased.

Table 7.1 shows relative values of `NEW1` based on `REF`. A value in each entry is a percentage of the raw value of `NEW1` based on the raw value of `REF`. The raw value are measured in the same way as in Section 6.1.

Under coarse estimation, the size of the reduced space is the number of possible output values encoded in a chromosome which is $2^{\lceil log_2(|\Delta|)\rceil \times |\Sigma| \times |Q|}$. However, when looking into the detail, the previous value is just the upper bound. There is a case where a machine that is encoded in a chromosome having capability of encoding $|Q|$ states does not really

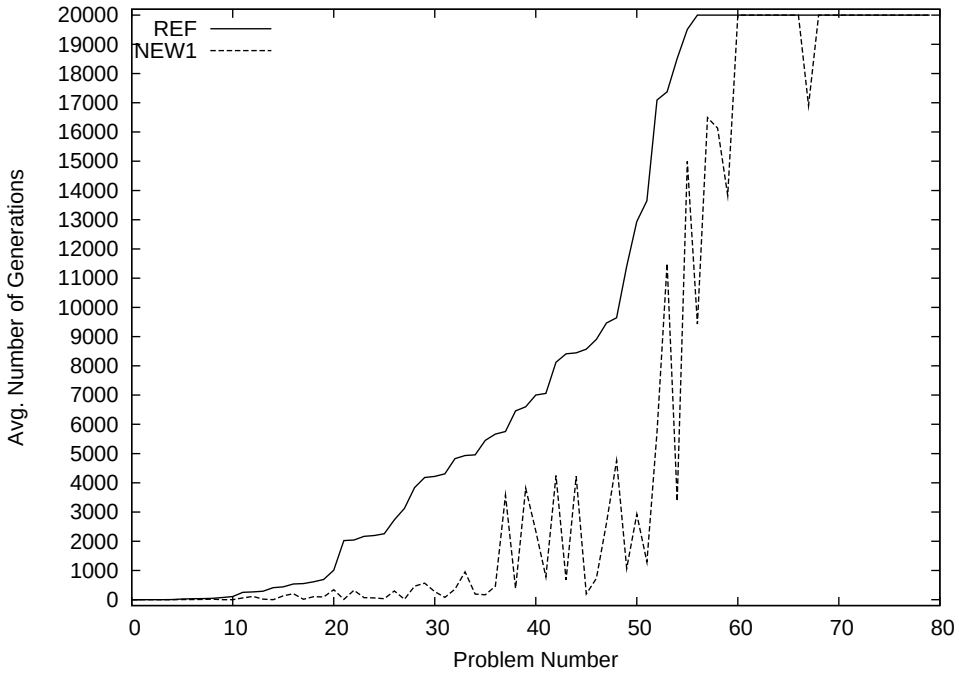

Figure 7.1: Number of Generation Used of Experiment A

Table 7.1: Relative Comparison of Experiment A, A1 and A2

	Experiment A	Experiment A1	Experiment A2
Avg. Generation Used	70.40%	61.60%	54.12%
Avg. Time Used	72.09%	55.16%	49.43%
# of Problem Solved	121.69%	135.62%	150.65 %

have $|Q|$ *reachable* states. It is possible that some of its states is inaccessible. Moreover, the given sequences might not exercise some transitions even if those transitions is accessible from the starting state. In that case, the output value and the next state value of those transitions do not effect the evaluation value for both the new encoding and the reference encoding. Those inaccessible states have effect on the process of Genetic Algorithm, though. They act as *introns*. Intron or *non-coding segment* is a term borrowed from biological system. It refers to parts of an individual that have no effect on its behavior which implies that they do not effect evaluation value. Introns are very common in the field of Genetic Programming since recombination in Genetic Programmings usually introduces non-coding segment. Their functionality and effect are not completely elucidated. There are many theories which describe introns (Luke, 2000). However, there are many evidence indicating that when introns are in the appropriate position, it can improve

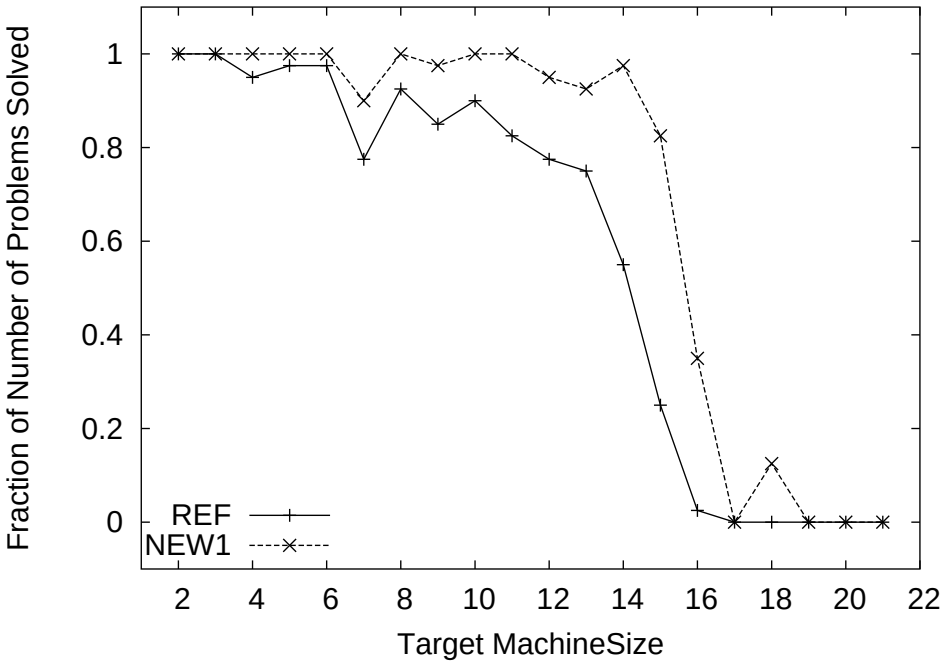

Figure 7.2: Fraction of Successful Runs of Experiment A

the performance of Genetic Algorithm.

Introns are also emerged in the methods in this work. All of `NEW1`, `NEW2` and `REF` have possibilities that introns might emerge to the chromosome. For the reference encoding, the output value might be an intron if its corresponding transition is not exercised or it belongs to an unreachable state. Under new encoding where the output function is omitted, introns of output values are no longer exist. The next state values can be introns for the same case. All three methods have introns of the next state value. The effect of introns is discussed again in Section 7.1.2.

In summary, new encoding reduces size of the search space. At the same time, it takes out introns caused by the output value of inaccessible and unexercised states.

7.1.2 Schema Preservation

The Schema Theorem states that Genetic Algorithms search for the required solution by combining building blocks or highly fit low-order schemata together (Lobo, 2000). Selection and recombination of individual are the mechanism that do the combining. It is crucial that Genetic Algorithms know which schema is good and which one is bad so

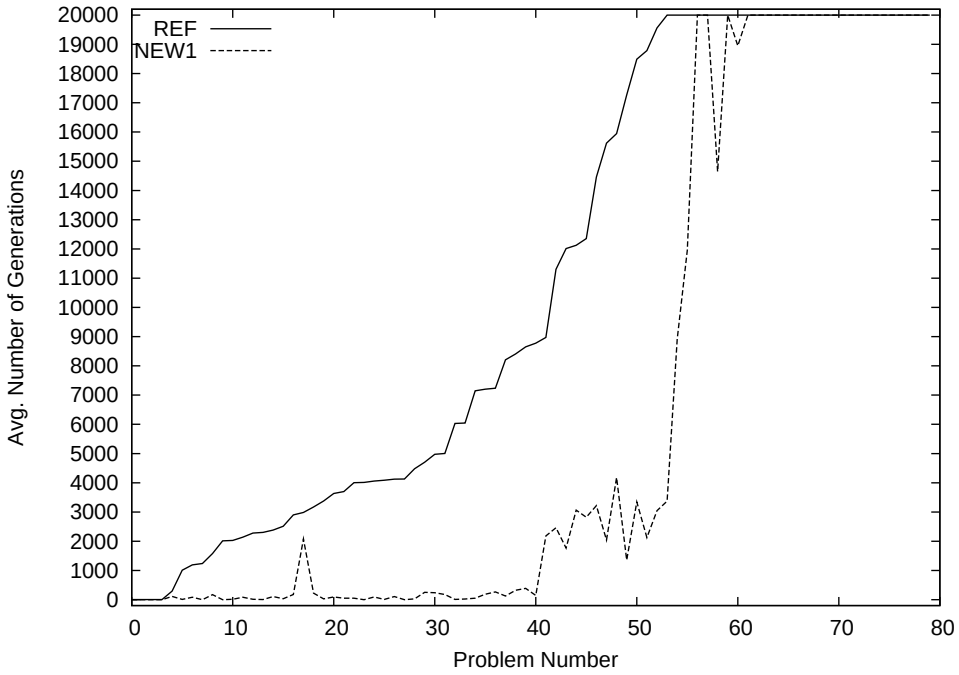

Figure 7.3: Number of Generation Used of Experiment A1

that it can propagate. The propagation of good schemata is a duty of selection. However, when recombination operator comes into play, it will introduces what is known as "destructive effect of crossover". While recombination operator recombines solution, it implicitly destroys some schemata at the same time. The possibility that a particular schema is destroyed by recombination operator is determined by the encoding of that schema and the particular recombination operator that is used. Under standard crossover operator, the chance that a schema H is disrupted is proportion of its defining length. The longer defining length that schema is, the higher chance that it would be disrupted. This disruption effect should be minimized.

There is also the issue of building block mixing (Goldberg et al., 1992). Building Block mixing tells how building blocks are combined together into an individual. The work confirms that Simple Genetic Algorithm without proper linkages takes unreasonable long time to solve problems that do not have compact encoding. The abilities that Genetic Algorithms can detect which genes constitute building blocks and propagate those genes as an unbreakable unit are very crucial for solving difficult problems (Lobo, 2000).

The new method presented in this work has a special characteristic. The interpretation of each particular gene highly depends on other genes. The way that each gene

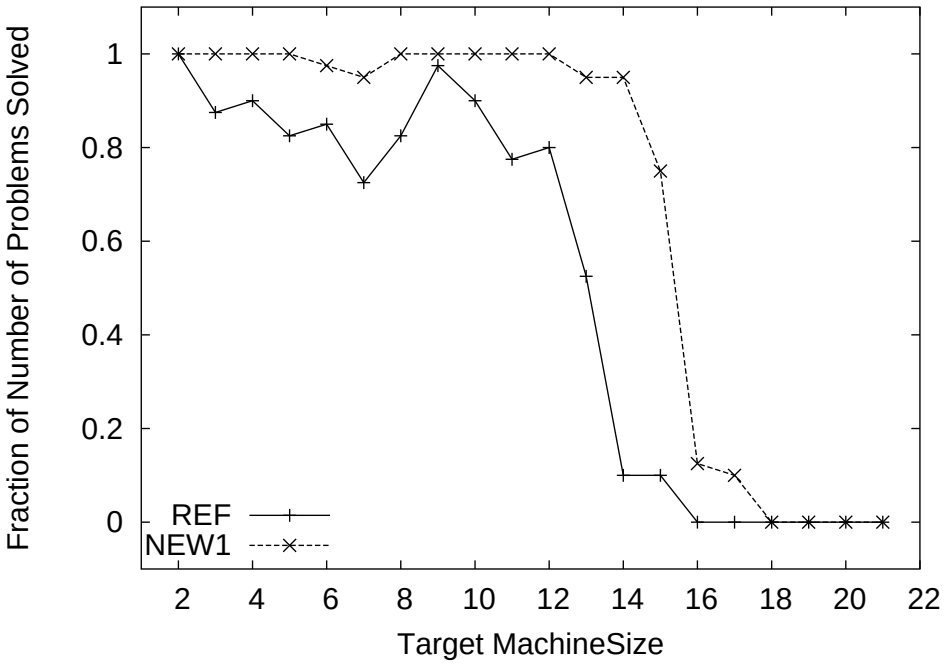

Figure 7.4: Fraction of Successful Runs of Experiment A1

interacts to each others is also positioning non-static, i.e., sometime a particular gene at position x interacts with genes located at position y, at other time, it interacts with one that located at position z. Linkages, a group of genes that forms building block which should be propagated together, is position-independent. The new crossover operator presented in Section 5.2.3 is designed to cope with the issue of encoding.

The new crossover can be thought of as a bounded crossover on re-arranged genes. The new crossover exploits the fact that the underlying structure of an individual is a graph. The new crossover creates a list of states in a predefined deterministic way (using depth first order) and then applies single point crossover on that list. The list is a chromosome which is rearranged in a way that (hopefully) has a better encoding compactness. Moreover, the new crossover restricts that the crossover point never to cut in the middle of gene that represents single next state value or output value. Unlike the standard crossover which regards every bits equally, the new crossover which knows that some bits are parts of the same gene does not permit cutting in the middle of those bits. Usually, Genetic Algorithms that handle individuals in a way that is consistent with their underlying structure are more accurate and more efficient (Angeline, 1994). Cutting in the middle of genes might results in randomly different next state values and destroys the characteristic of

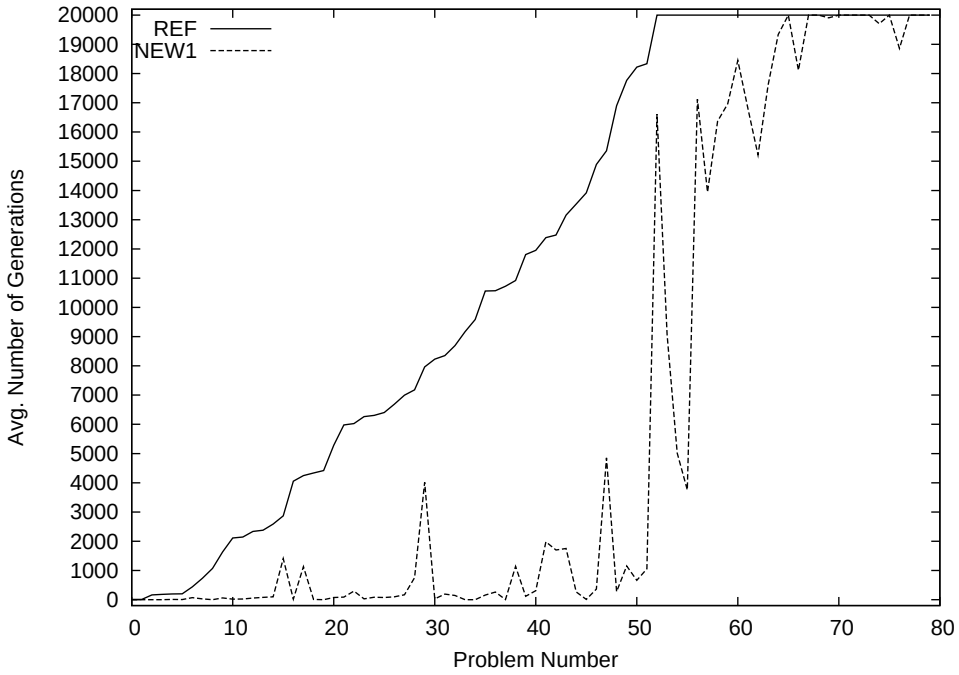

Figure 7.5: Number of Generation Used of Experiment A2

that transition completely. This effect is not entirely negative because it can generate a new value of the gene but when it is incorporated into a crossover which executed at very high probability, one-half in the cases of the methods used in this work, it mostly destroys information which should be propagated.

The new crossover also effects introns. Usually, the effect of introns are positive. Many studies shows that introns can improve success rate of Genetic Algorithms such as the work in (Levenick, 1991). Many studies also suggest that introns or non-coding segment increase chances of building block mixing and reduce the destruction of existing building block (Wu and Lindsay, 1996). When a crossover point falls in introns segments, the building block is usually preserved. A larger segment of intron yields higher chance of this “defense against crossover.” This theory of defense against crossover is accepted by many researchers as stated in (Luke, 2000).

Under the new crossover operator, the effect of introns, that reduces the chance that the crossover will destroy schemata, is nullified because parts of a machine that is inaccessible and acts as introns is filtered out. When the new crossover is used in combination with the new encoding, introns caused by the output value are taken out. However, they are not entirely eliminated because parts of a machine that is not exercised by the given

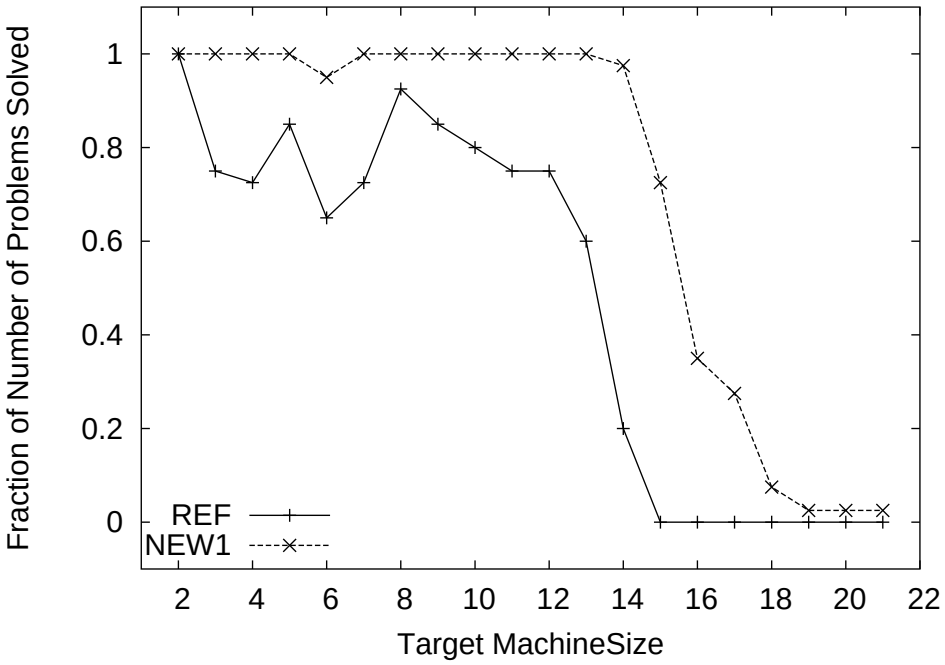

Figure 7.6: Fraction of Successful Runs of Experiment A2

sequence but is reachable from the starting state are still included. Though it is possible that introns are still present (from unexercised transition), it is unlikely to happen because the given sequence used in the experiments are usually large and they should exercise most transitions, if not all.

In conclusion, the new crossover, along with the new encoding, almost entirely eliminate introns. The benefits of introns that happen in reference encoding and crossover might be lost but the benefits of the new crossover are occurring. The empirical results from the experiments exhibit evidences showing that the gain of benefit caused by the new crossover can overcome the loss of benefits caused by introns.

7.1.3 Time Complexity of New Method

The result shown in the previous chapter indicates that when the measurement of interest is the number of generations, the new method outperforms the former method. However, each operator of the new method requires more time to process than the reference method. Mainly, two processes cause the different in processing time. They are the evaluation function and the crossover operator. The other process is either the same for both methods or has very little impact on processing time. For example, both methods

use the same selection operator (tournament selector.) Other examples are encoding and decoding functions of both methods. They are different only in that the output function is either present or not present. This difference has very little effect in running time. The main difference in process time is largely depended on evaluation functions and crossover operators. This section analyzes the time complexity of this process.

7.1.3.1 Evaluation Function

The code of the reference evaluation function shown in Figure 5.2. It has a nested loop. The outer loop iterates through all elements of the input/output sequence set while the inner loop runs for all alphabets in the input string of the sequence. Totally, the loop iterates over all input/output pairs of the given sequence. Hence, the time complexity of the reference evaluation function F is

$$F \in \Theta\Big(\parallel \zeta \parallel\Big)$$

The new evaluation function has two parts. The first part is the output frequency counting which is very similar to the reference evaluation function. The first part contributes $\Theta(\parallel \zeta \parallel)$ to the time complexity of the function. The second part iterates through all transitions and scans for the maximum value of mapping. Since no special data structure is used to help retrieving the maximum value, the process always checks all elements in the output count table. The second part contributes $\Theta(|Q| \times |\Sigma| \times |\Delta|)$ and it is the part that makes the new evaluation function takes more time than the reference evaluation function. The time complexity of the new evaluation function F' is

$$F' \in \Theta\Big(\parallel \zeta \parallel +(|Q| \times |\Sigma| \times |\Delta|)\Big)$$

7.1.3.2 Crossover Operator

The reference method uses standard single point crossover. The running time of the operator is the time used to copy every value of one chromosome to other two chromosomes. The running time for this process is $\Theta(n)$ where n is the size of a chromosome. The length of a chromosome can be calculated from M' as stated in Section 5.1.1, so the running time of reference crossover is

$$\Theta\Big(\big(\lceil\log_2(|Q|)\rceil + \lceil\log_2(|\Delta|)\rceil\big) \times |\Sigma| \times |Q|\Big)$$

The new crossover operator is more complicated than the single point crossover. It can be divided into three steps which are: the decoding of a bit string to a graph, the generation of depth first order list and the copying of selected transitions. Decoding and copying are both linear on the size of the chromosome which is given in Section 5.2.2. Making a depth first order list has a running time of $\Theta(|Q| \times |\Sigma|)$. This value is dominated by the running time of decoding and copying. In total, the running time of the new crossover operator is

$$\Theta\Big(\lceil\log_2(|Q|)\rceil \times |\Sigma| \times |Q|\Big)$$

Thought analysis in the view point of asymptotic notation tells that the new crossover operator is better than the reference method, the real processing time of the new operator is much higher. This is because the extra time of the new operator which comes from the decoding and graph traversal does not effect the asymptotic notation and $|\Delta|$ is very small, e.g., it is 1 in Experiment A. This makes the actual running time of the new crossover operator higher than the standard operator.

7.2 Discussion of the Experiment B

The result from the Experiment B does not show any significant difference between the genetic algorithm approach (`NEW2`) and the inexact heuristic approach (`red-blue`.) Does this mean that the conjecture of accuracy related to size limitation is wrong? The answer is no, it is not entirely wrong. First, let us make clear that the goal of this experiment is to compare the difference of correctness of result of two approaches. One is inexact heuristic approach that infers a machine by using the stage merging algorithm. The result of this approach is always consistent. The other one is Genetic Algorithm approach that infers a machine by finding a consistent machine with the result of minimal consistency approach.

Each data point in the result is calculated from Cross Validation method. In this experiment, Cross Validation divides each problem into 6 sub-problems and averages the correctness of the result of all sub-problems. The averaged value is the result of Cross

Validation. However, for each sub-problem, NEW2 is run repeatedly 10 times since NEW2 is non-deterministic while red-blue is run only once. It is possible that NEW2 might not be able to find a consistent result for all 10 runs. Some run might yield inconsistent result which is also included into the average value. However, this inconsistent result performs bad in identifying unseen data. So, for a problem instance that NEW2 yields inconsistent result for some runs, the average score is dragged down. Since the main goal of this experiment is to compare the inexact heuristic approach with the small consistency approach, it is unfair to include inconsistent results into the calculation. The experiment is modified and run again to rectify this problem.

The experiment is re-run again with the modified version of NEW2. The modified version of NEW2 is exactly the same as the normal NEW2, except that instead of returning 10 hypothesis machines for 10 runs, the modified version selects the best machine out of these 10 runs and returns it as a single result instead. The best machine is the machine that has the highest correctness according to the *training set*. In other words, it can be thought as that the Cross Validation takes the best answer out of 10 runs of Genetic Algorithms, instead of taking the average of all runs. This eliminates the inconsistent machine from the calculation of correctness. Still, there is a case that NEW2 can not find any consistent answer in all 10 runs. In such a case, the best consistent machine is selected as the best one. This procedure seems fair if it is to compare the inexact heuristic approach with the minimal consistency approach. This is because the consistency is the essential property that must be compared, not the inconsistency. Also it is made clear in the beginning of the experiment that the running time is not the subject of interest. The result of this experiment is shown in Figure 7.7.

The result indicates that the hypothesis machine of NEW2, using only the best answer of 10 runs (according to training data), has better accuracy in identifying unseen data than the hypothesis machine of the red-blue when the size of training set is getting smaller and smaller. The difference is decreasing when the training size is near zero.

It is very essential to put a strong emphasize that, although the result indicates that NEW2 can produce a better accurate hypothesis than red-blue's, it is true for only target machine of small size. Inexact heuristic approach, such as red-blue, can solve much larger problem than NEW2 and it can solve it in very short amount of time. For a large problem, such as one whose target machine's size is larger than 100 states, NEW2 can not find the consistent answer in a reasonable time. In such a case, red-blue is much more

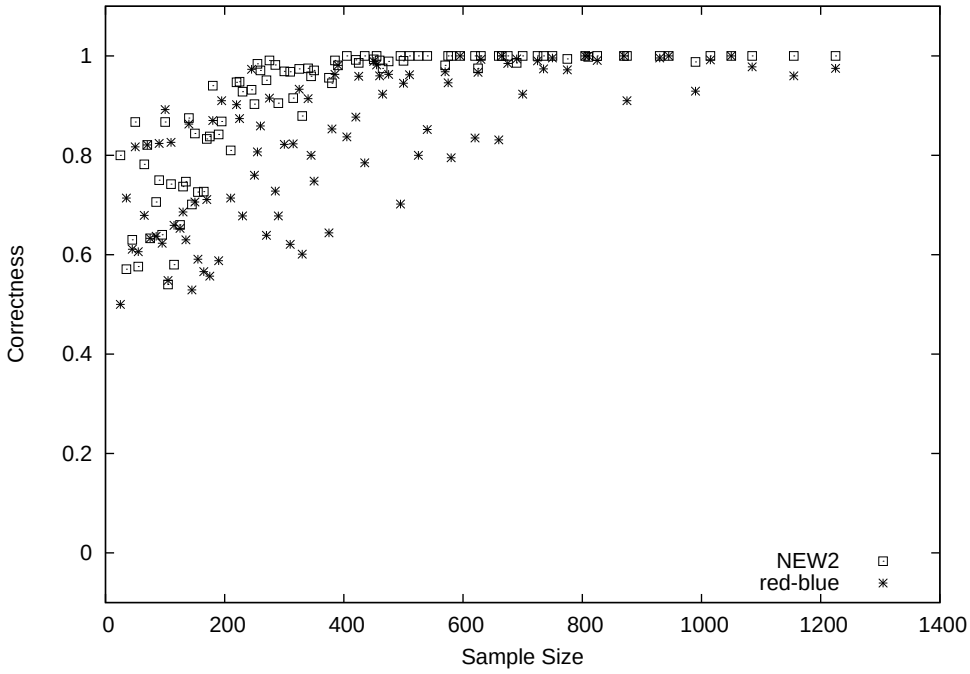

Figure 7.7: Result of Experiment B using Best Solutions

practical.

It is clear that modified `NEW2` which represents the minimal consistency approach really performs better than `red-blue` which represents the inexact heuristic approach. It is interesting to analyze this effect. The main difference between the result of both methods is the size of hypothesis. Size of `NEW2` is restricted while `red-blue`'s is not. There is a principle in logic that says about the complexity of entities to explain a hypothesis. That principle is called Occam's Razor.

Occam's Razor is a logical principle that was used extensively and made famous by medieval philosopher named *William of Ockham*. The principle states that one should not increase, beyond what is necessary, the number of entities required to explain anything. For example, let us consider a curve fitting problem. Suppose that there is two data points in a plane. A straight line could fit this two data points. Lots of other complicated higher degree curves could also fit this points. However, by the Occam's Razor principle, the straight line is preferable. It is possible that the straight line is wrong if more sampling points is observed but that is another matter. Occam's Razor states that to choose among theories or models that can explain the same thing, the simplest one is preferable. This makes developing a model easier and there is less chance that inconsistency might occur.

Under this problem, when NEW2 and red-blue are applied to a particular problem instance, they will yield a result with different size. Both hypothesis machines could explain the training data. However, Occam's Razor suggests that the smaller one is preferable and should have higher chance of being consistent. If Occam's Razor is applied to this problem, the smaller hypothesis machine should perform better in identifying unseen data. To see this, the number of states of the hypothesis machines of Experiment B is measured. Figure 7.8 shows the size of hypothesis machines of both methods for each problem instance size. Please note that the machine is reduced before the measurement is taken.

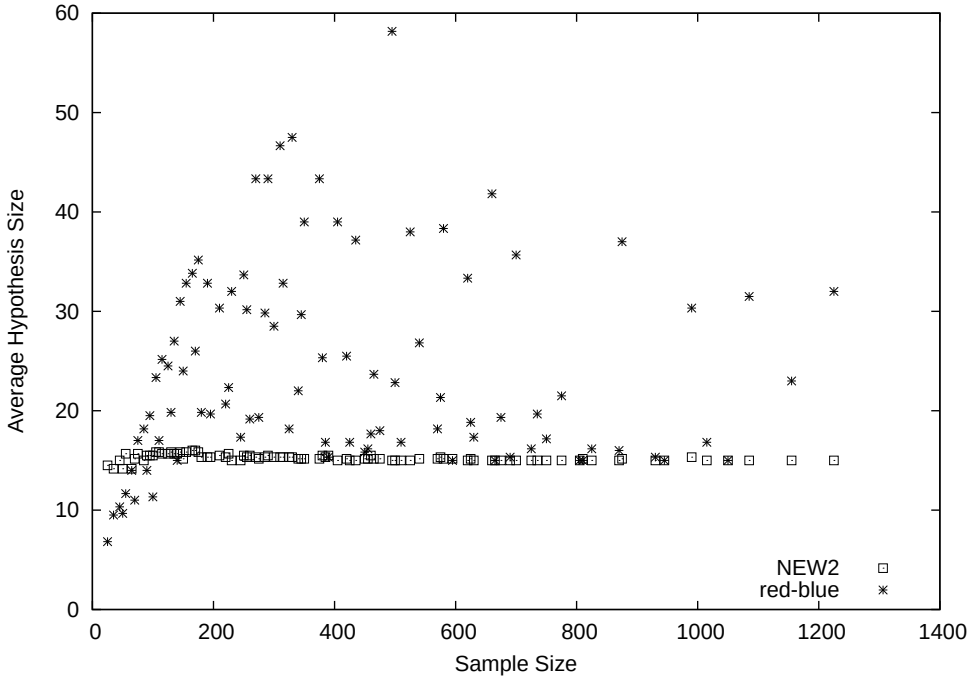

Figure 7.8: Size of Hypothesis Machines

The graph indicates that results from NEW2 are smaller than the results from red-blue except for small area that the problem instance has very sparse training data. The area that data are very sparse is the same area that the difference in accuracy of both method is unnoticeable. This evidence supports the principle of Occam's Razor. In fact, there is an finite automata inference algorithm that uses Occam's Razor approach such as Occam algorithms (Blumer et al., 1989). There are also a lot of works that show strong correlation between the complexity of the hypothesis and the desired behavior as noted in (Oliveira and Silva, 2001).

7.3 Summary

This chapter provided an analysis of the experiments. In Experiment A, the results indicate that the method `NEW2` is better than `NEW1` where `NEW1`, itself, is better than the reference method, `REF`. However, `NEW2` takes slightly more running time than `NEW1`. The improvement over the reference method comes from the reduction of the search space. Two supplementary experiments are carried out and their results show the evidence that, when the size of reduced search space is increased, the improvement is more noticeable. The other cause is the preservation of schema which is the effect of the new crossover operator. However, the new operator has a side effect that it nullifies the effect of introns which is implicitly happen in the reference method. The effect of intron is still unclear.

The result of Experiment B indicates that the GA-based method with size restriction has a better generalization than the heuristic method, though the latter method could find an answer in a very short amount of time. The effect can be argued to be a result from Occam's Razor which states that a simpler hypothesis that could explain the same thing is better than more complicated hypothesis.

CHAPTER VIII

Conclusion

This chapter summarizes the work in this thesis. It starts by summing up the entire work. It gives recommendations on the future research for this work. Finally, it gives one paragraph conclusion of the work.

8.1 Summary

This work attacks the problem of finite state machine inference. It starts by describing the problem and the importance of the problem. It discusses that the problem is useful in many context. Further, the problem is considered to be hard in theoretical point of view but practical in an average case. There are many approaches that solve the problem. Each approach has its own advantage. A recommendation about how to choose a method to solve the problem is given. This work emphasizes on a genetic algorithm approach which is suitable when the training size is relatively small and the size of hypothesis machine is restricted.

This work proposes a new genetic algorithm for the problem. It claims that the new method is better than the former genetic algorithms for the problem. It shows the evidence to support the claim by empirically comparing the proposed method with the former method. The results of comparison show that the proposed method performs better than the former method. In addition, it also compares the proposed method with the method from another approach to show the advantage of the genetic algorithm approach. The result shows that, under a particular constraint, the genetic algorithm approach can beat the other approach.

Moreover, it gives some analysis of the difference of the compared methods. It analyzes why the proposed method is better than the former method. The analysis deals with many theoretical aspect of genetic algorithms. It also informally discusses the difference between the genetic algorithm approach and other approach.

8.2 Future Research

Many topics in this work deserve future research. Mostly, they are about Genetic Algorithms for the problem of finite state machine inference. The topics can be divided

into two classes. The first is the practical issue of the proposed method and the second is the theoretical issue.

8.2.1 Practical Issue

First, the performance of the proposed genetic algorithm can be further improved. For example, the proposed method uses specialized crossover that is designed for graph based representation, in hoping that it could preserve building block better than a standard crossover. However, there are a lot of work that deal with this problem. Specifically, there are many algorithms proposed for linkage learning that can learn where the linkage is. The application of such method to this problem might improve the performance of the method.

Second, there is an issue on chromosome representation. The encoding of the proposed method and the reference method have redundancy, finite state machines that are behavioral equivalent can be encoded into different chromosomes. Usually, redundant encoding should be avoided. New encoding that addresses this problem should be developed. The method is also limits that the number of genes must be a power of two. The main reason for this encoding is to remove the inadmissible representation. This partly limits the application of the method. Encoding that can represent machines at any size should be developed.

8.2.2 Theoretical Issue

It is noted in the analysis that the proposed method introduces introns in the encoding. The effect of intron for this particular problem is not extensively studied. This work notes that there are introns in the reference method and introns are mostly reduced in the proposed method. Future research in this topic should provide more understanding about introns on this particular problem.

The analysis also notes the application of Occam's Razor. Though it is discussed, the formal analysis has not been studied. Future formal comparison between the minimal consistency approach and the inexact heuristic approach should be carried out by focusing on the correlation between the size of the hypothesis and the accuracy in identifying unseen data.

8.3 Conclusion

In conclusion, the contribution of this work is the new genetic algorithm for the problem of finite state machine inference. The method is shown to be better than the former genetic algorithm for the problem. This method is suitable for the case that the size of given sequence is relatively small and the size of hypothesis machine is restricted.

References

Angeline, P. J. 1994. Genetic programming: A current snapshot. In Fogel, D. B. and Atmar, W. (eds.), Proceedings of the Third Annual Conference on Evolutionary Programming. Evolutionary Programming Society.

Angluin, D. and Smith, C. H. 1983. Inductive inference: Theory and methods. Computing Surveys, 15(3):237–269.

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

Aporntewan, C. and Chongstitvatana, P. 2000. An on-line evolvable hardware for learning finite-state machine. In Proceedings of International Conference on Intelligent Technologies, pp. 125–134.

Biermann, A. W., Baum, R. I., and Petry, F. E. 1975. Speeding up the synthesis of programs from traces. IEEE Transactions on Computers, 24(C):122–136.

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

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. 1989. Learnability and the vapnik-chervonenkis dimension. Journal of the ACM (JACM), 36(4):929–965.

Chongstitvatana, P. and Aporntewan, C. 1999. Improving correctness of finite-state machine synthesis from multiple partial input/output sequences. In Proceedings of First NASA/DoD Workshop of Evolvable Hardware.

Dupont, P. 1994. Regular grammatical inference from positive and negative samples by genetic search: the GIG method. In Proceedings of the International Colloquium on Grammatical Inference, pp. 236–245.

Dupont, P., Miclet, L., and Vidal, E. 1994. What is the search space of the regular inference? In Carrasco, R. C. and Oncina, J. (eds.), Proceedings of the Second International Colloquium on Grammatical Inference (ICGI-94): Grammatical Inference and Applications, volume 862, pp. 25–37, Berlin: Springer.

Fogel, L. J., Owens, A. J., and Walsh, M. J. 1965. Artificial intelligence through a simulation of evolution. In Maxfield, M., Callahan, A., and Fogel, L. (eds.), Biophycisc and Cybernetic System: Proceedings of the 2nd Cybernetic Sciences Symposium, pp. 131–155, Washington, DC: Spartan Books.

Fogel, L. J., Owens, A. J., and Walsh, M. J. 1998. Artificial intelligence through a simulation of evolution. In Fogel, D. B. (ed.), Evolutionary Computation : the Fossil Record, pp. 230–254. Piscataway, NJ: IEEE Press.

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

Goldberg, D. E. 1989. Genetic algorithm in search, optimization and machine learning. Reading, MA: Addison–Wesley.

Goldberg, D. E., Deb, K., and Thierens, D. 1992. Toward a better understanding of mixing in genetic algorithms. Technical Report 92009, Illinois Genetic Algorithms Laboratory, University of Illinoise at Urbana-Champaign.

Holland, J. H. 1992. Adaptation in Natural and Artificial Systems. Cambridge, MA: MIT Press.

Hopcroft, J. E. and Ullman, J. D. 1979. Introduction to Automata Theory, Languages, and Computation. Reading, MA: Addison–Wesley.

Juille, H. and Pollack, J. B. 1998. A sampling-based heuristic for tree search applied to grammar induction. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98) Tenth Conference on Innovative Applications of Artificial Intelligence (IAAI-98), Madison, Wisconsin, USA: AAAI Press Books.

Katz, R. H. 1994. Contemporary Logic Design. Reading, MA: Addison–Wesley.

Koza, J. R. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: MIT Press.

Lang, K. J. 1992. Random DFA's can be approximately learned from sparse uniform examples. In Proceedings of the Fifth ACM Workshop on Computational Learning Theory, pp. 45–52, New York, NY: ACM.

Lang, K. J. 1997. Abbadingo One: DFA Learning Competition[Online]. Available from: http://abbadingo.cs.unm.edu/abbadingo/[2002, Mar 26]

Lang, K. J., Pearlmutter, B. A., and Price, R. A. 1998. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. Fourth International Colloquium on Grammatical Inference, Lecture Notes in Computer Science 1433.

Lankhorst, M. M. 1995. A genetic algorithm for the induction of pushdown automata. In Proceedings of the Second IEEE Conference on Evolutionary Computation, pp. 741–746.

Levenick, J. R. 1991. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In Belew, R. and Booker, L. (eds.), Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 123–127, San Mateo, CA: Morgan Kaufman.

Lobo, F. G. 2000. The parameter-less genetic algorithm: Rational and automated parameter selection for simplified genetic algorithm operation. Technical Report 2000030, Illinois Genetic Algorithms Laboratory, University of Illinoise at Urbana-Champaign.

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

Luke, S. 2000. Code growth is not caused by introns. In Whitley, D. (ed.), Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, pp. 228–235, Las Vegas, Nevada, USA.

Machado, P., Pereira, F. B., Cardoso, A., and Costa, E. 1999. Busy beaver – the influence of representation. In Poli, R., Nordin, P., Langdon, W. B., and Fogarty, T. C. (eds.), Genetic Programming, Proceedings of EuroGP'99, volume 1598, pp. 29–38, Goteborg, Sweden: Springer–Verlag.

Manovit, C., Aporntewan, C., and Chongstitvatana, P. 1998. Synthesis of synchronous sequential logic circuits from partial input/output sequences. In Proceedings of 2nd International Conference on Evolvable Systems, pp. 98–105.

Mitchell, T. M. 1997. Machine Learning. Singapore: McGraw–Hill.

Oliveira, A. L. and Silva, J. P. M. 2001. Efficient algorithms for the inference of minimum size DFAs. Machine Learning, 44(1/2):93–119.

Pereira, F. B., Machado, P., Costa, E., and Cardoso, A. 1999. Busy beaver: An Evolutionary approach. In Proceedings of the second Symposium on Artificial Intelligence.

Pereira, F. B., Machado, P., Costa, E., and Cardoso, A. 1999. Graph based crossover-A case study with the busy beaver problem. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E. (eds.), Proceedings of the Genetic and Evolutionary Computation Conference, volume 2, pp. 1149–1155, Orlando, Florida, USA: Morgan Kaufmann.

Pereira, F. B., Machado, P., Costa, E., Cardoso, A., Ochoa-Rodriguez, A., Santana, R., and Soto, M. 2000. Too busy to learn. In Proceedings of the 2000 Congress on

Evolutionary Computation CEC00, pp. 720–727. IEEE Press.

Pitt, L. and Warmuth, M. K. 1993. The minimum consistent dfa cannon be approximated within any polynomial. Journal of ACM, 40(1):95–142.

Trakhtenbrot, B. A. and Barzdin, Y. M. 1973. Finite Automata. Amsterdam: North-Holland.

Thierens, D. 1999. Scalability problems of simple genetic algorithms. Evolutionary Computation, 7(4):331–352.

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

Wolpert, D. H. and Macready, W. G. 1995. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM.

Wolpert, D. H. and Macready, W. G. 1997. No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation, 1(1):67–82.

Wu, A. S. and Lindsay, R. K. 1996. A survey of intron research in genetics. In Voigt, H.-M., Ebeling, W., Rechenberg, I., and Schwefel, H.-P., editors, Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation, volume 1141, pp. 101–110, Berlin, Germany: Springer–Verlag.

Appendix

APPENDIX A

Experimental Results in Details

This appendix provides results of each experiment in detailed version. The data presented here are raw data before being summarized.

A.1 Experiment A

This section provides results of Experiment A which compares method `REF`, `NEW1` and `NEW2`. Bold value represents the best value among all methods.

Table A.1: Raw Results of Experiment A

	Measurement					
	Number of Generations			Time (s)		
	REF	NEW1	NEW2	REF	NEW1	NEW2
Problem 0	1.4	**0.0**	0.0	**0.1**	0.1	0.1
Problem 1	1.5	**0.0**	0.0	0.1	**0.0**	0.0
Problem 2	1.5	**0.0**	0.0	**0.1**	0.1	0.1
Problem 3	2.5	**0.0**	0.0	0.2	**0.0**	0.0
Problem 4	7.3	**0.6**	1.0	0.4	**0.1**	0.1
Problem 5	25.3	3.6	**3.4**	1.3	**0.2**	0.3
Problem 6	31.4	4.5	**3.3**	1.6	**0.3**	0.3
Problem 7	40.5	**11.4**	12.0	2.1	**0.7**	0.7
Problem 8	47.2	**18.4**	24.8	2.3	**1.1**	1.5
Problem 9	75.0	**0.6**	0.8	3.3	**0.1**	0.1
Problem 10	108.8	**1.2**	1.5	5.1	**0.1**	0.1
Problem 11	251.6	60.5	**26.8**	12.4	3.3	**1.7**
Problem 12	265.2	114.5	**54.1**	13.1	6.3	**3.3**
Problem 13	291.1	21.7	**15.3**	14.4	1.2	**1.0**
Problem 14	412.0	**0.2**	0.9	18.6	**0.0**	0.1
Problem 15	440.3	133.0	**85.2**	27.6	9.0	**6.2**
Problem 16	536.0	207.3	**146.1**	33.3	13.9	**10.5**
Problem 17	551.1	**15.3**	25.2	27.2	**0.8**	1.5
Problem 18	613.9	**102.3**	132.2	38.3	**6.9**	9.5

Table A.1: Raw Results of Experiment A

	Measurement					
	Number of Generations			Time (s)		
	REF	NEW1	NEW2	REF	NEW1	NEW2
Problem 19	692.5	95.2	**85.3**	43.2	6.4	**6.1**
Problem 20	1008.0	342.8	**254.2**	62.7	22.2	**18.2**
Problem 21	2025.3	9.6	**9.0**	99.3	**0.6**	0.6
Problem 22	2040.2	**323.7**	584.7	127.1	**21.7**	41.8
Problem 23	2167.5	**68.6**	69.8	134.3	**4.6**	5.0
Problem 24	2197.5	61.0	**33.6**	107.7	3.4	**2.0**
Problem 25	2257.3	34.2	**16.9**	112.1	2.0	**1.0**
Problem 26	2732.8	299.5	**64.2**	137.0	16.4	**3.8**
Problem 27	3127.1	**13.0**	24.2	153.7	**0.8**	1.5
Problem 28	3833.8	462.3	**351.8**	238.6	29.7	**25.2**
Problem 29	4181.1	567.1	**425.2**	259.2	37.9	**30.3**
Problem 30	4216.3	276.0	**131.1**	266.4	18.5	**9.5**
Problem 31	4306.0	**79.3**	1803.6	267.5	**5.3**	129.0
Problem 32	4825.7	358.5	**52.5**	300.5	24.0	**3.8**
Problem 33	4934.4	952.7	**129.6**	305.9	63.8	**9.2**
Problem 34	4955.2	**192.1**	394.1	307.5	**12.9**	28.1
Problem 35	5450.6	**167.7**	2711.1	268.9	**9.3**	158.4
Problem 36	5664.9	**465.3**	899.9	355.5	**30.0**	64.4
Problem 37	5752.6	3572.0	**1146.9**	357.2	238.8	**82.3**
Problem 38	6460.3	**382.7**	793.6	405.8	**24.7**	56.7
Problem 39	6599.7	3817.0	**561.5**	412.6	245.9	**40.0**
Problem 40	7001.5	2351.6	**2185.2**	438.6	**151.7**	157.0
Problem 41	7055.6	**782.7**	2449.0	442.1	**49.4**	196.3
Problem 42	8124.6	**4245.8**	6339.1	401.9	**232.2**	370.2
Problem 43	8410.2	676.0	**326.7**	537.8	43.7	**23.3**
Problem 44	8445.0	**4221.9**	6570.5	419.9	**230.7**	384.6
Problem 45	8572.7	**196.8**	1442.3	532.7	**13.2**	102.7
Problem 46	8906.8	**730.4**	3290.4	561.6	**47.1**	234.7
Problem 47	9466.2	2625.1	**2526.5**	586.4	**165.0**	180.2

Table A.1: Raw Results of Experiment A

	Measurement					
	Number of Generations			Time (s)		
	REF	NEW1	NEW2	REF	NEW1	NEW2
Problem 48	9645.7	4757.9	**698.3**	598.5	301.8	**49.9**
Problem 49	11406.8	1070.9	**817.3**	710.1	67.4	**59.3**
Problem 50	12938.5	**2924.5**	4179.0	808.7	**183.6**	306.3
Problem 51	13645.2	**1288.3**	1933.3	864.0	**80.9**	137.7
Problem 52	17096.1	**5728.4**	6654.1	1073.0	**361.3**	475.0
Problem 53	17373.1	11504.4	**5395.9**	1086.9	722.7	**416.7**
Problem 54	18516.4	**3367.9**	5082.4	1159.7	**211.4**	362.0
Problem 55	19509.5	14995.5	**12653.7**	1945.3	**1385.3**	1472.1
Problem 56	20000.0	**9428.1**	10308.9	1254.7	**594.7**	736.6
Problem 57	20000.0	**16490.0**	17927.6	2002.5	**1514.5**	1902.7
Problem 58	20000.0	16130.8	**11578.1**	2038.5	1479.8	**1219.6**
Problem 59	20000.0	13831.4	**10564.2**	1997.2	1272.1	**1130.8**
Problem 60	20000.0	20000.0	**15742.7**	1999.1	1834.3	**1684.2**
Problem 61	20000.0	20000.0	20000.0	2000.7	**1836.5**	2135.9
Problem 62	20000.0	20000.0	20000.0	1998.6	**1835.4**	2130.3
Problem 63	20000.0	20000.0	20000.0	1996.2	**1834.3**	2113.1
Problem 64	20000.0	20000.0	**16804.6**	1995.4	1846.6	**1782.1**
Problem 65	20000.0	20000.0	**19589.9**	1999.9	**1836.5**	2168.0
Problem 66	20000.0	20000.0	**18379.9**	1967.6	**1827.9**	1955.9
Problem 67	20000.0	16903.9	**11689.3**	1965.6	1543.7	**1232.4**
Problem 68	20000.0	20000.0	20000.0	2000.0	**1831.5**	2100.9
Problem 69	20000.0	20000.0	20000.0	1999.3	**1825.8**	2117.5
Problem 70	20000.0	20000.0	20000.0	2002.4	**1830.5**	2175.6
Problem 71	20000.0	20000.0	20000.0	2007.4	**1840.3**	2121.0
Problem 72	20000.0	20000.0	20000.0	2008.5	**1835.5**	2339.6
Problem 73	20000.0	20000.0	20000.0	2004.7	**1837.1**	2116.7
Problem 74	20000.0	20000.0	**19576.1**	2004.2	**1834.0**	2151.5
Problem 75	20000.0	20000.0	20000.0	2004.0	**1840.5**	2145.9
Problem 76	20000.0	20000.0	20000.0	2001.1	**1834.1**	2276.1

Table A.1: Raw Results of Experiment A

	Measurement					
	Number of Generations			Time (s)		
	REF	NEW1	NEW2	REF	NEW1	NEW2
Problem 77	20000.0	20000.0	20000.0	2000.8	**1838.8**	2133.1
Problem 78	20000.0	20000.0	20000.0	2004.0	**1837.9**	2105.0
Problem 79	20000.0	20000.0	20000.0	2006.1	**1840.6**	2144.2
Total	749246	527488	505780	64350	46393	51801
Relative	100.00%	70.40%	67.51%	100.00%	72.10%	80.50%
Best among others	0	31	35	2	52	26

A.2 Experiment A1 and A2

This section provides results of Experiment A, A1 and A2. These set of experiments are carried out to compare the method `REF` and `NEW1`, so the data of `NEW2` of the Experiment A are omitted. Bold value indicates the best value among all methods.

Table A.2: Raw Results of Experiment A, A1 and A2

	Number of Generations					
	Experiment A		Experiment A1		Experiment A2	
	REF	NEW1	REF	NEW1	REF	NEW1
Problem 0	1.4	**0.0**	5.8	**0.0**	9.7	**0.0**
Problem 1	1.5	**0.0**	6.3	**0.0**	10.3	**0.0**
Problem 2	1.5	**0.0**	6.5	**0.0**	166	**0.0**
Problem 3	2.5	**0.0**	6.9	**0.0**	184.2	**0.0**
Problem 4	7.3	**0.6**	296.6	**111.3**	196.3	**8.2**
Problem 5	25.3	**3.6**	1011.7	**7.9**	200.4	**7.4**
Problem 6	31.4	**4.5**	1191.5	**86.4**	439.4	**71.1**
Problem 7	40.5	**11.4**	1237.7	**0.8**	735.3	**29.2**
Problem 8	47.2	**18.4**	1577.7	**175.9**	1068.6	**0.5**
Problem 9	75.0	**0.6**	2011.8	**0.7**	1635.6	**62.5**
Problem 10	108.8	**1.2**	2028.3	**16.0**	2110.7	**18.9**
Problem 11	251.6	**60.5**	2138.9	**84.1**	2142.9	**24.3**

Table A.2: Raw Results of Experiment A, A1 and A2

	Number of Generations					
	Experiment A		Experiment A1		Experiment A2	
	REF	NEW1	REF	NEW1	REF	NEW1
Problem 12	265.2	**114.5**	2278.1	**13.5**	2333.9	**56.1**
Problem 13	291.1	**21.7**	2304	**7.6**	2379.8	**80.2**
Problem 14	412.0	**0.2**	2381.6	**106.1**	2586.4	**99.5**
Problem 15	440.3	**133.0**	2512.4	**34.9**	2868.6	**1407.0**
Problem 16	536.0	**207.3**	2900.4	**178.4**	4056.2	**20.8**
Problem 17	551.1	**15.3**	2984.1	**2078.6**	4247.3	**1130.4**
Problem 18	613.9	**102.3**	3170.5	**224.1**	4332.9	**11.0**
Problem 19	692.5	**95.2**	3377.4	**27.7**	4420.3	**0.5**
Problem 20	1008.0	**342.8**	3633.4	**92.8**	5282.9	**73.7**
Problem 21	2025.3	**9.6**	3698.2	**50.7**	5980	**89.6**
Problem 22	2040.2	**323.7**	4003	**53.3**	6027.6	**296.3**
Problem 23	2167.5	**68.6**	4012.4	**0.9**	6264.2	**27.9**
Problem 24	2197.5	**61.0**	4058.4	**93.1**	6307.5	**86.1**
Problem 25	2257.3	**34.2**	4087.5	**9.4**	6405	**75.1**
Problem 26	2732.8	**299.5**	4124	**109.3**	6689.5	**98.1**
Problem 27	3127.1	**13.0**	4127.1	**0.9**	6997.4	**168.6**
Problem 28	3833.8	**462.3**	4481.1	**26.9**	7179.1	**750.8**
Problem 29	4181.1	**567.1**	4705.4	**250.8**	7962.3	**4018.6**
Problem 30	4216.3	**276.0**	4976.8	**239.7**	8228.9	**28.2**
Problem 31	4306.0	**79.3**	5000.6	**180.2**	8352.1	**192.9**
Problem 32	4825.7	**358.5**	6031.5	**12.5**	8691.1	**147.5**
Problem 33	4934.4	**952.7**	6045.3	**25.4**	9166.9	**0.5**
Problem 34	4955.2	**192.1**	7148.6	**52.6**	9586.3	**1.8**
Problem 35	5450.6	**167.7**	7200.3	**188.2**	10561.6	**157.5**
Problem 36	5664.9	**465.3**	7237.2	**264.8**	10570.7	**264.1**
Problem 37	5752.6	**3572.0**	8207.4	**127.8**	10723.4	**7.1**
Problem 38	6460.3	**382.7**	8408.5	**322.8**	10919.9	**1139.8**
Problem 39	6599.7	**3817.0**	8648	**388.5**	11806.4	**116.3**
Problem 40	7001.5	**2351.6**	8777.8	**147.7**	11953.2	**307.7**

Table A.2: Raw Results of Experiment A, A1 and A2

	Number of Generations					
	Experiment A		Experiment A1		Experiment A2	
	REF	NEW1	REF	NEW1	REF	NEW1
Problem 41	7055.6	**782.7**	8972.2	**2182.8**	12385.8	**1982.1**
Problem 42	8124.6	**4245.8**	11301.7	**2459.9**	12474.6	**1699.0**
Problem 43	8410.2	**676.0**	12014.1	**1767.4**	13155.5	**1753.0**
Problem 44	8445.0	**4221.9**	12126.4	**3064.4**	13536.5	**266.0**
Problem 45	8572.7	**196.8**	12354.4	**2823.7**	13916.6	**8.2**
Problem 46	8906.8	**730.4**	14459.9	**3220.8**	14890.7	**365.4**
Problem 47	9466.2	**2625.1**	15616.5	**2043.0**	15356.8	**4854.7**
Problem 48	9645.7	**4757.9**	15943.8	**4180.8**	16898.1	**278.0**
Problem 49	11406.8	**1070.9**	17269.2	**1356.0**	17766.1	**1152.4**
Problem 50	12938.5	**2924.5**	18491.5	**3344.6**	18220.2	**665.4**
Problem 51	13645.2	**1288.3**	18785.8	**2126.9**	18334.3	**1056.5**
Problem 52	17096.1	**5728.4**	19559.5	**3043.7**	20000	**16615.0**
Problem 53	17373.1	**11504.4**	20000	**3374.5**	20000	**9065.2**
Problem 54	18516.4	**3367.9**	20000	**8910.9**	20000	**5016.2**
Problem 55	19509.5	**14995.5**	20000	**11957.4**	20000	**3757.5**
Problem 56	20000.0	**9428.1**	20000	20000	20000	**17122.0**
Problem 57	20000.0	**16490.0**	20000	20000	20000	**13940.7**
Problem 58	20000.0	**16130.8**	20000	**14649.1**	20000	**16371.7**
Problem 59	20000.0	**13831.4**	20000	20000	20000	**16964.8**
Problem 60	20000.0	20000	20000	**18949.6**	20000	**18449.5**
Problem 61	20000.0	20000	20000	20000	20000	**16805.8**
Problem 62	20000.0	20000	20000	20000	20000	**15212.9**
Problem 63	20000.0	20000	20000	20000	20000	**17607.9**
Problem 64	20000.0	20000	20000	20000	20000	**19334.0**
Problem 65	20000.0	20000	20000	20000	20000	20000
Problem 66	20000.0	20000	20000	20000	20000	**18110.6**
Problem 67	20000.0	**16903.9**	20000	20000	20000	20000
Problem 68	20000.0	20000	20000	20000	20000	20000
Problem 69	20000.0	20000	20000	20000	20000	**19907.3**

Table A.2: Raw Results of Experiment A, A1 and A2

	Number of Generations					
	Experiment A		Experiment A1		Experiment A2	
	REF	NEW1	REF	NEW1	REF	NEW1
Problem 70	20000.0	20000	20000	20000	20000	20000
Problem 71	20000.0	20000	20000	20000	20000	20000
Problem 72	20000.0	20000	20000	20000	20000	20000
Problem 73	20000.0	20000	20000	20000	20000	20000
Problem 74	20000.0	20000	20000	20000	20000	**19705.4**
Problem 75	20000.0	20000	20000	20000	20000	20000
Problem 76	20000.0	20000	20000	20000	20000	**18858.1**
Problem 77	20000.0	20000	20000	20000	20000	20000
Problem 78	20000.0	20000	20000	20000	20000	20000
Problem 79	20000.0	20000	20000	20000	20000	20000
Total	749246	527488	868936	535248	938716	508001
Relative	100.00%	70.40%	100.00%	61.60%	100.00%	54.12%
Successful Runs	461	561	408	552	383	577

A.3 Experiment B

This section provides results of the *modified version* of Experiment B where the values of `NEW2` method are the best value of 10 runs rather than the average value. In other words, this is the raw data of Figure 7.7. Each data point is the result of Cross Validation algorithm. The first column is the length (in bit) of the input/output sequences and the number of the sequences. The other columns comprise of the correctness and the size of the hypothesis. The calculation of correctness is presented in Section 6.2.2. The size is the average of the size of hypothesis machines of all sub-problems.

Table A.3: Raw Results of Experiment B

	NEW2		red-blue	
Length x Num of Seq.	Correctness	Size	Correctness	Size
5 x 5	**0.800**	14.500	0.500	6.833
5 x 10	**0.867**	14.167	0.817	9.667

Table A.3: Raw Results of Experiment B

Length x Num of Seq.	NEW2		red-blue	
	Correctness	Size	Correctness	Size
5 x 15	**0.933**	15.000	0.889	10.000
5 x 20	0.867	15.500	**0.892**	11.333
5 x 25	0.953	15.167	0.953	11.500
5 x 30	**0.967**	15.500	0.950	13.167
5 x 35	**0.957**	15.667	0.952	13.333
7 x 5	0.571	14.167	**0.714**	9.500
7 x 10	0.821	15.167	0.821	11.000
7 x 15	0.873	15.333	0.873	14.333
7 x 20	**0.875**	15.667	0.863	15.000
7 x 25	0.914	15.500	0.914	17.500
7 x 30	**0.913**	15.167	0.905	16.500
7 x 35	0.932	15.000	**0.973**	17.333
9 x 5	**0.630**	15.000	0.611	10.333
9 x 10	0.750	15.500	**0.824**	14.000
9 x 15	0.809	16.000	**0.833**	18.167
9 x 20	**0.940**	15.333	0.870	19.833
9 x 25	**0.941**	15.667	0.893	18.833
9 x 30	**0.963**	14.667	0.932	16.167
9 x 35	**0.992**	15.667	0.971	16.000
11 x 5	0.576	15.667	**0.606**	11.667
11 x 10	0.742	15.833	**0.826**	17.000
11 x 15	**0.823**	15.333	0.753	22.667
11 x 20	**0.947**	15.333	0.902	20.667
11 x 25	**0.991**	15.167	0.915	19.333
11 x 30	**0.977**	15.500	0.902	19.500
11 x 35	**0.991**	15.333	0.963	16.833
13 x 5	**0.782**	14.167	0.679	14.000
13 x 10	**0.737**	15.833	0.686	19.833
13 x 15	0.868	15.333	**0.910**	19.667
13 x 20	**0.971**	15.500	0.859	19.167

Table A.3: Raw Results of Experiment B

Length x Num of Seq.	NEW2		red-blue	
	Correctness	Size	Correctness	Size
13 x 25	**0.974**	15.333	0.933	18.167
13 x 30	0.981	15.500	0.981	15.333
13 x 35	**1.000**	15.167	0.982	16.167
15 x 5	0.633	15.667	0.633	17.000
15 x 10	**0.844**	15.167	0.706	24.000
15 x 15	**0.948**	15.667	0.874	22.333
15 x 20	**0.969**	15.333	0.822	28.500
15 x 25	**0.982**	15.333	0.951	15.167
15 x 30	**0.993**	15.167	0.989	15.833
15 x 35	**0.992**	15.167	0.986	15.500
17 x 5	**0.706**	15.000	0.637	18.167
17 x 10	**0.833**	16.000	0.711	26.000
17 x 15	**0.984**	15.333	0.807	30.167
17 x 20	**0.975**	15.167	0.914	22.000
17 x 25	**0.986**	15.000	0.959	16.833
17 x 30	**1.000**	15.000	0.962	16.833
17 x 35	1.000	15.167	1.000	15.000
19 x 5	**0.640**	15.500	0.623	19.500
19 x 10	**0.842**	15.333	0.588	32.833
19 x 15	**0.982**	15.333	0.728	29.833
19 x 20	**0.945**	15.500	0.853	25.333
19 x 25	**0.989**	15.167	0.963	18.000
19 x 30	**0.981**	15.167	0.968	18.167
19 x 35	1.000	15.000	1.000	15.000
21 x 5	0.540	15.833	**0.548**	23.333
21 x 10	**0.810**	15.500	0.714	30.333
21 x 15	**0.915**	15.333	0.823	32.833
21 x 20	**0.992**	15.167	0.877	25.500
21 x 25	**0.992**	15.167	0.973	17.000
21 x 30	**1.000**	15.000	0.993	17.333

Table A.3: Raw Results of Experiment B

Length x Num of Seq.	NEW2		red-blue	
	Correctness	Size	Correctness	Size
21 x 35	**1.000**	15.000	0.974	19.667
23 x 5	0.580	15.667	**0.659**	25.167
23 x 10	**0.928**	15.000	0.678	32.000
23 x 15	**0.959**	15.167	0.800	29.667
23 x 20	**0.991**	15.500	0.960	17.667
23 x 25	**1.000**	15.333	0.946	21.333
23 x 30	0.986	15.000	**0.994**	15.333
23 x 35	1.000	15.000	1.000	15.000
25 x 5	**0.660**	15.667	0.653	24.500
25 x 10	**0.903**	15.500	0.760	33.667
25 x 15	**0.956**	15.167	0.644	43.333
25 x 20	**0.990**	15.000	0.945	22.833
25 x 25	**0.975**	15.167	0.967	18.833
25 x 30	**1.000**	15.000	0.996	17.167
25 x 35	1.000	15.000	1.000	15.000
27 x 5	**0.747**	15.667	0.630	27.000
27 x 10	**0.951**	15.333	0.639	43.333
27 x 15	**1.000**	15.000	0.837	39.000
27 x 20	**1.000**	15.167	0.852	26.833
27 x 25	**1.000**	15.000	0.985	19.333
27 x 30	0.998	15.167	**1.000**	15.000
27 x 35	1.000	15.000	1.000	15.000
29 x 5	**0.701**	15.833	0.529	31.000
29 x 10	**0.905**	15.500	0.678	43.333
29 x 15	**1.000**	15.000	0.785	37.167
29 x 20	**1.000**	15.167	0.795	38.333
29 x 25	**1.000**	15.000	0.990	16.167
29 x 30	1.000	15.000	1.000	16.000
29 x 35	**1.000**	15.000	0.992	16.833
31 x 5	**0.726**	15.833	0.591	32.833

Table A.3: Raw Results of Experiment B

Length x Num of Seq.	NEW2		red-blue	
	Correctness	Size	Correctness	Size
31 x 10	**0.968**	15.333	0.621	46.667
31 x 15	**0.975**	15.167	0.923	23.667
31 x 20	**1.000**	15.000	0.835	33.333
31 x 25	**0.994**	15.000	0.972	21.500
31 x 30	**1.000**	15.000	0.996	15.333
31 x 35	**1.000**	15.000	0.978	31.500
33 x 5	**0.727**	16.000	0.566	33.833
33 x 10	**0.879**	15.333	0.601	47.500
33 x 15	**1.000**	15.000	0.702	58.167
33 x 20	**1.000**	15.000	0.831	41.833
33 x 25	**1.000**	15.000	0.991	16.167
33 x 30	**0.988**	15.333	0.929	30.333
33 x 35	**1.000**	15.000	0.960	23.000
35 x 5	**0.838**	15.833	0.557	35.167
35 x 10	**0.971**	15.167	0.748	39.000
35 x 15	**1.000**	15.000	0.800	38.000
35 x 20	**1.000**	15.000	0.923	35.667
35 x 25	**1.000**	15.167	0.910	37.000
35 x 30	1.000	15.000	1.000	15.000
35 x 35	**1.000**	15.000	0.975	32.000

Biography

Nattee Niparnan was born in Bangkok, Thailand, on May 11, 1979. He received Bachelor Degree of Engineering from Chulalongkorn University since April, 2001. His field of study is Computer Engineering. His main interest is in the area of data structure and algorithm analysis. His current research is the novel applications of evolutionary computation and the theory of Genetic Algorithms.