A GENETIC APPROACH TO EVOLVE FINITE STATE AUTOMATA

A Dissertation

Submitted in Partial Fulfillment of the

Requirement for the Award of the Degree of

MASTER OF ENGINEERING

IN

COMPUTER TECHNOLOGY AND APPLICATIONS

By

SHRADDHA SINGHAI

(Roll No: 3012)

Under the Guidance

of

Mrs. RAJNI JINDAL

DEPARTMENT OF COMPUTER ENGINEERING

DELHI COLLEGE OF ENGINEERING

DELHI 110042

Department of Computer Engineering
Delhi College of Engineering, Delhi

CERTIFICATE

This is to certify that the project report entitled

"A GENETIC APPROACH TO EVOLVE FINITE STATE AUTOMATA"

being submitted by **Ms SHRADDHA SINGHAI (Roll No: 3012)**

is a bonafide record of her own work carried under our guidance and supervision in partial fulfillment for the award of the degree of Master of Engineering in Computer Technology and Applications from Delhi College of Engineering, Delhi.

Mrs. Rajni Jindal
Lecturer, Project Guide
Delhi College of Engg, Delhi

Dr. D. Roy Choudhury
Professor & HOD
Delhi College of Engg, Delhi

ACKNOWLEDGEMENT

I wish to express my deep sense of gratitude and veneration to my project guide **Mrs. Rajni Jindal**, Lecturer, Department of Computer Engineering, Delhi College of Engineering, Delhi, for her perpetual encouragement, constant guidance, valuable suggestions and continuous motivation which has enabled me to complete this work.

I would like to express my sincere thanks to **Dr. P. B. Sharma**, Principal, Delhi College of Engineering, to allow me to perform this study and for providing all the necessary facilities to carry out this work.

I am deeply indebted to **Dr. D. Roy Choudhury**, HOD, Department of Computer Engineering, Delhi College of Engineering, for his constant encouragement, valuable guidance, resourceful suggestions and alignment evaluations throughout the course of this project. I also thank all the teachers of Computer Engineering department of Delhi college of Engineering for their kind co-operation and enormous support.

I am also grateful to my parents, who have been a constant source of inspiration for me throughout my life and enabled me to reach at this stage. A special appreciation also goes to all my friends for their love and constant support.

(SHRADDHA SINGHAI)

ABSTRACT

Finite-state automata are one of the most pervasive models of computation, not only theoretically, but also in all of its applications to real-life problems such as natural and formal language processing, pattern recognition, control, etc. Automatically inferring finite automata from sets of positive and negative data samples has been an important problem in computer science and many schemes have been proposed for its solution. The previous works in the evolution of finite state automata were limited to the evolution of strictly non-modular FSA. In this dissertation, a modular architecture to develop FSA accepting a particular regular language is proposed and a genetic programming procedure for evolving such structures is presented. The results on the Tomita Language benchmark indicate that the proposed procedure is able to evolve an NFA with less number of generations explored and lesser amount of time taken than the earlier non-modular evolution.

CONTENTS

Page No

CERTIFICATE **i**

ACKNOWLEDGEMENT **ii**

ABSTRACT **iii**

TABLE OF CONTENTS **iv**

LIST OF FIGURES **viii**

LIST OF TABLES **ix**

CHAPTER-1 INTRODUCTION **1**

1.1 Objective 1

1.2 Contribution 1

1.3 Organization of the Dissertation 2

CHAPTER-2 BACKGROUND **4**

2.1 Overview of Finite State Machines 4

2.1.1 Deterministic Finite Automata 4

2.1.2 Non-deterministic Finite Automata 5

2.1.3 Language of the Automata 5

2.2 Overview of Genetic Programming 5

2.2.1 Genetic programming Process 6

2.3 Problem Statement 7

2.3.1 Problem Overview and Historical Background 7

2.3.2 Proposed Problem Solving Approach 8

CHAPTER-3 GENETIC PROGRAMMING **9**

3.1 Introduction 9

3.2 Preparatory Steps of Genetic Programming 9

3.2.1 Function Set and Terminal Set 10
3.2.2 Fitness Measure 10
3.2.3 Control Parameters 11
3.2.4 Termination 11
3.3 Executional Steps of Genetic Programming 11
3.4 The Genetic Programming Algorithm in Action 14
3.4.1 Generate Random Initial Population 14
3.4.1.1 Grow Method 14
3.4.1.2 Full Method 15
3.4.1.3 Ramped-half-and-half Method 15
3.4.2 Measure Fitness of Population 16
3.4.2.1 Raw Fitness 16
3.4.2.2 Standardized Fitness 16
3.4.2.3 Adjusted Fitness 17
3.4.4.4 Normalized Fitness 17
3.4.2.5 Probability of Selection 17
3.4.3 Select Better Individuals From The Population 18
3.4.3.1 Fitness-Proportional Selection 18
3.4.3.2 Ranked Selection 18
3.4.3.3 Tournament Selection 19
3.4.3.4 Truncation Selection 19
3.4.4 Apply Genetic Operators to Generate New Population 19
3.4.4.1 Reproduction 19
3.4.4.2 Crossover 20
3.4.4.3 Mutation 22
3.4.4.4 Permutation 23
3.4.4.5 Editing 24
3.4.4.6 Encapsulation 24
3.4.4.7 Decimation 24
3.4.5 Control Parameters 24

3.4.6 Repeat Until Program Solves Problem or Time Runs Out 25

CHAPTER-4 EDGE ENCODING FOR FSA INDIVIDUALS 26

4.1 Introduction 26
4.2 Encoding an NFA 27

CHAPTER-5 PROBLEM AND SCHEME SPECIFICATIONS 30

5.1 Chromosomal Encoding 30
5.2 Generating the Initial Population 32
5.3 Tomita Languages 32
5.4 Fitness Assessment 34
5.5 Selections and Breeding 34
5.5.1 Selection Scheme 34
5.5.2 Crossover Scheme 35
5.5.3 Mutation Scheme 35
5.6 Control Parameters 37

CHAPTER-6 MODULAR ARCHITECTURE TO EVOLVE FINITE STATE AUTOMATA 38

6.1 Proposed Evolution Model 38
6.2 Design Of Modular Architecture 39
6.3 Computational Algorithms 42
6.3.1 Basic Algorithm for NFA Evolution 42
6.3.2 Proposed Modular Algorithm 44
6.4 Example Evolution 44
6.5 Summary 45

CHAPTER-7 EXPERIMENTS AND RESULTS **46**

7.1 Experimental Setup 46

7.1.1 Implementation 46

7.1.2 Test Data 46

7.1.3 Tomita Decomposition For The Modular Evolution 46

7.1.4 Fitness Metric 47

7.2 Experimental Results 48

7.2.1 Population-based Analysis 48

7.2.2 Timing Analysis 52

7.2.3 Performance Evaluation 52

CHAPTER-8 CONCLUSIONS AND FUTURE DIRECTIONS **54**

8.1 Conclusions 54

8.2 Future Work 54

REFERENCES **56**

APPENDIX A: OUTPUT **59**

APPENDIX B: SOURCE CODE **72**

LIST OF FIGURES

		Page No
Figure 3.1:	Preparatory Steps	9
Figure 3.2:	Flowchart of Genetic Programming	13
Figure 3.3:	Crossover	21
Figure 3.4:	Mutation	23
Figure 4.1:	The Double Function	27
Figure 4.2:	An Edge Encoding Genome, which Describes an NFA that Reads the Regular Expression ((0|1)*101)	29
Figure 4.3:	The Growth of the NFA from the Encoding in Figure 4.2	29
Figure 5.1:	An edge encoding genome and its equivalent s-expression	31
Figure 5.2:	Subtree Crossover	36
Figure 5.3:	Subtree Mutation	36
Figure 6.1:	Edge Encoding for 1* and (10)*	44
Figure 6.2:	Encoding Tree and NFA for 1* and (10)*	45
Figure 6.3:	Edge Encoding Tree and the NFA for 1*(10)*	45
Figure 7.1:	Number of Generations Explored in the Best case for the Tomita Languages	50
Figure 7.2:	Number of Generations Explored in the Average case for the Tomita Languages	50
Figure 7.3:	Number of Nodes Evaluated in the Best case for the Tomita Languages	51
Figure 7.4:	Number of Nodes Evaluated in the Average case for the Tomita Languages	51
Figure 7.5:	Time Taken for finding the Solution in the Best case for the Tomita Languages	53
Figure 7.6:	Time Taken for finding the Solution in the Average case for the Tomita Languages	53

LIST OF TABLES

Page No

Table 4.1:	Simple Topological Functions for Edge Encoding	28
Table 4.2:	NFA Semantic Functions for Edge Encoding	28
Table 5.1:	Description of function set and terminal set for edge encoding an NFA	31
Table 5.2:	The Tomita Language Set	32
Table 5.3:	Positive and Negative Training Examples	33
Table 7.1:	Tomita Decomposition for the Modular Evolution	47
Table 7.2:	Number of generations explored and the number of nodes evaluated for the evolution of each Tomita language using non-modular approach	49
Table 7.3:	Number of generations explored and the number of nodes evaluated for the evolution of each Tomita language using proposed modular approach	49
Table 7.4:	Average elapsed time in milliseconds to learn the Tomita languages for both non-modular and modular architecture	52
Table A.1:	Positive and Negative Training Sets for Tomita 7	59
Table A.2:	Positive and Negative Training Sets for Sub Expression 1 of Tomita 7	67

CHAPTER-1
INTRODUCTION

Automatic programming has been the goal of computer scientists for a number of decades. Scientists would like to be able to give the computer a problem and ask the computer to build a program to solve it. Genetic Programming, a technique pioneered by John Koza [15], shows the most potential way to automatically write computer programs, via the core, but highly abstracted principles of natural selection.

GP begins with a population of randomly generated individuals (candidate solutions). It then tests these individuals and assesses their quality. The better ones are then selected to breed and create new individuals, which in turn are tested, selected, and bred. This cycle continues until a sufficiently good solution is found for the problem, or until time or other resources are exhausted. In a sentence, it is the compounded breeding of (initially random) computer programs, where only the relatively more successful individuals pass on genetic material to the next generation [3].

1.1 Objective

This dissertation focuses on the problem of automatic creation of finite state automata accepting a particular regular language using the genetic programming paradigm. The basic problem is, given a set of positive and negative example strings, automatically infer corresponding automata, which generates or recognizes those examples. The Finite Automata is evolved to induce the Tomita language Set [24], a popular and nontrivial language induction benchmark.

1.2 Contribution

In this dissertation a modular architecture to evolve finite state automata is proposed and a genetic programming procedure for evolving such structures is presented. In the proposed modular architecture, the given regular expression is

decomposed into few smaller sub-expressions, the finite automata for each of these sub-expressions are evolved using the genetic programming paradigm, and then the evolved sub automata are combined to get the complete automata describing the given regular expression as a whole.

Tomita Set results indicate that the proposed procedure is indeed capable of successfully evolving modular finite state machines and that such modularity can result in a significantly increased rate of optimization. It is also supported by the fact, that, a difficult task when decomposed into simpler subtasks can be solved with lower computational effort, and their solutions can be combined to give the overall solution to the task. Further, already discovered solutions to subtasks may also be reused to repeatedly solve similar sub problems.

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows:

Chapter 2 starts with the brief background of the finite state machines and the genetic programming process, and is followed by a brief overview to the problem, previous work and to the proposed problem solving approach.

Chapter 3 comprises of learning about, understanding, and implementing the concept of Genetic Programming.

Chapter 4 investigates the issue of chromosomal representation of an FSA individual. The chapter describes Edge Encoding, a technique for evolving graph and network structures via genetic programming. The FSA chromosomes are represented in the form of Edge Encoding tree, which can develop into a directed graph when evaluated.

Chapter 5 provides the problem and scheme specifications for evolving finite state automata for the Tomita Language Set using the Genetic Programming Process.

Chapter 6 presents the proposed modular architecture to evolve finite state automata. The design of architecture as well as the computational algorithm is described. The chapter then concludes by explaining an example evolution using the proposed architecture.

Chapter 7 examines the experimental setup and compares the results of evolving NFA's for the Tomita Languages using both the modular as well as the non-modular approach.

Chapter 8 summarizes the main results of the research and presents some conclusions. Some promising future research topics are described as a natural extension of this work.

CHAPTER - 2

BACKGROUND

2.1 Overview of Finite State Automata

Finite state automata are one of the most important mathematical constructs used in the construction of practical computer programs. They have applicability in virtually every area of computer science, especially in language translators. They involve states and transitions among states in response to inputs. They are useful for developing several kinds of software components, including the lexical analysis component of compilers and systems for verifying the correctness of circuits and protocols.

Finite State automata consists of a set of states, a start state, set of final states, an input alphabet, and a transition function that maps an input symbol and current state to next state [1].

2.1.1 Deterministic Finite Automata

A Deterministic Finite Automata (DFA) is a finite automaton having a finite set of states and a finite set of input symbols. One state is designated the start state, and one or more states are accepting states. A transition function determines how the state change each time an input symbol is processed [14].

A DFA can be represented by a 5-tuple $(Q, \sum, \delta, q_0, F)$

where, Q is finite nonempty set of states

$\sum$ is a finite nonempty set of inputs called input alphabets

δ is a function which maps $Q \times \sum$ into Q

$q_0 \in Q$ is the initial state

$F \subseteq Q$ is the set of final states.

2.1.2 Non-deterministic Finite Automata

Non-deterministic Finite state automata differ from DFAs in that they allow for an input bit to specify multiple possible next moves. Namely, in an NFA we may move from state *p* to any of states $q_1, q_2, \ldots, q_k$, by seeing the same input. Hence we will not have the same type of transition function as a DFA. Instead, our transition function will take as input a state and an element of the alphabet but return some subset of states. In an NFA, we also allow for multiple start states [14].

A Non-deterministic Finite State Automata is a model with 5-tuple $(Q, \sum, \delta, q_0, F)$

where, Q is finite nonempty set of states

$\sum$ is a finite nonempty set of inputs called input alphabets

δ is a function which maps $Q \times \sum$ into 2^Q

$q_0 \in Q$ is the initial state

$F \subseteq Q$ is the set of final states.

2.1.3 Language of the Automata

All automata accept strings. A string is accepted if, starting in the start state, the transitions caused by processing the symbols of that string one at a time leads to an accepting state. An algebraic notation called regular expressions can also describe the language of a finite automaton [1]. We can convert any definition involving regular expressions into an implement able finite automaton in two steps:

Regular expression ⟷ NFA ⟷ DFA

2.2 Overview of Genetic Programming

One of the central challenges of computer science is to get a computer to do what needs to be done, without telling it how to do it. Genetic Programming addresses this challenge by providing a method for automatically creating a working computer program from a high-level program statement of the problem. It achieves this goal of

automatic programming by genetically breeding a population of computer programs, using the principles of Darwinian natural selection and biologically inspired operations.

Genetic Programming (GP) searches for good solutions to problems by trying large numbers of candidate solutions, selecting the "better" ones, modifying them, and producing new candidate solutions to test [15].

Because it is inspired by natural selection and genetics, genetic programming borrows much of its vernacular from genetics, cellular biology, and evolutionary theory. In GP, a candidate solution is known as an individual. The pool of current individuals in the system is collectively known as the population. This population may, depending on the nature of the problem being solved, be broken into several subpopulations. The actual encoding of an individual's solution is known as its genome (occasionally chromosome). The solution's representation when undergoing modification is known as the individual's genotype. The way the solution operates when tested in the problem environment is known as the individual's phenotype. When individuals are modified to produce new individuals, they are said to be breeding. During testing an individual receives a grade, known as its fitness, which indicates how good a solution it is. The period in which the individual is evaluated and assigned fitness is known as fitness assessment. When a population has been entirely replaced by children, the new population is known as the next generation. The whole process of finding an optimal solution is known as evolving a solution [3].

2.2.1 Genetic programming Process

Initially a population is generated randomly. Each individual is evaluated using the fitness measure given at the start. After this is done, a few individuals are selected to perform mutation, crossover and reproduction. The selected individuals after applying the genetic operations are copied to the next generation. The individuals in the next generation undergo the same process of fitness evaluation, selection, and modification. This is repeated until the termination condition is not satisfied. Termination condition,

generally, is either the maximum number of generations or discovery of an ideal individual.

2.3 Problem Statement

This dissertation focuses on the problem of automatic creation of finite automata accepting a particular regular language using the genetic programming paradigm. The basic problem is, given a set of positive and negative data samples, automatically infer a corresponding automaton, which generates or recognizes those samples.

2.3.1 Problem Overview and Historical Background

We evolve FSMs to recognize regular expressions. Regular expressions are used to represent a subset of strings. FSM evolution can be beneficial because it requires no human interaction. It can be used for data analysis, pattern matching or profiling. It can also be used in an adaptive system. The FSM would adapt to its environment to improve its fitness level.

The automatic creation of finite automata has long been a goal of the evolutionary computation community. Fogel [8] was the first to propose the generation of deterministic finite automata (DFAs) by means of an evolutionary process, and the possibility of inferring languages from examples was initially established by Gold. Since then, much work has been done in the induction of DFAs for language recognition.

Gruau [10] has proposed a method called Cellular Encoding, where the GP tree is a program, which builds a graph, often for use as a neural network and for developing directed graph.

Scott Brave [4] presented a method for the evolution of deterministic finite automata that combines genetic programming and cellular encoding. Luke and Spector [18] have published a preliminary report on a significant variation to cellular encoding

called edge encoding. Their main change is that forests of trees are used to represent the entire network, with the trees having the ability to recurse and to call other trees. Therefore, the same structure may be represented once and used multiple times.

.

Chongstitvatana et al. [6] used genetic inferencing to synthesize FSMs using multiple input/output sequences. Nippaman et al. [19] provided another FSM inference method, one where only the state transitions were evolved. State outputs were determined post-evolution.

Jason et al [13] presented a method to reduce the total number of generations needed to evolve a finite state machine using genetic inferencing. The time required to evolve a design is reduced, by only evolving, a small partition of the input-output relationship.

2.3.2 Proposed Problem Solving Approach

The previous works in the evolution of finite state machines were limited to the evolution of strictly non-modular FSA In this dissertation, a modular FSA architecture is proposed and a genetic programming procedure for evolving such structures is presented.

In this approach a regular expression is evolved, by decomposing the expression into simpler subparts. These subparts may then be solved with lower computational effort and their solutions can be combined to give the overall solution to the problem. Further, already discovered solutions to subtasks may be reused to repeatedly solve similar sub problems. Thus the total number of generations and the time required to evolve a finite state automata is reduced.

Results on the Tomita Language Set indicate that the proposed procedure is indeed capable of successfully evolving modular FSA and that such modularity can result in a statistically significantly increased rate of optimization.

CHAPTER-3
GENETIC PROGRAMMING

3.1 Introduction

Genetic programming is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done. It is a collection of methods for the automatic generation of computer programs that solve carefully specified problems, via the core, but highly abstracted principles of natural selection. In a sentence, it is the compounded breeding of (initially random) computer programs, where only the relatively more successful individuals pass on genetic material (programs and program fragments) to the next generation [15].

3.2 Preparatory Steps of Genetic Programming

Genetic programming starts from a high-level statement of the requirements of a problem and attempts to produce a computer program that solves the problem. The human user communicates the high-level statement of the problem to the genetic programming system by performing certain well-defined preparatory steps [23].

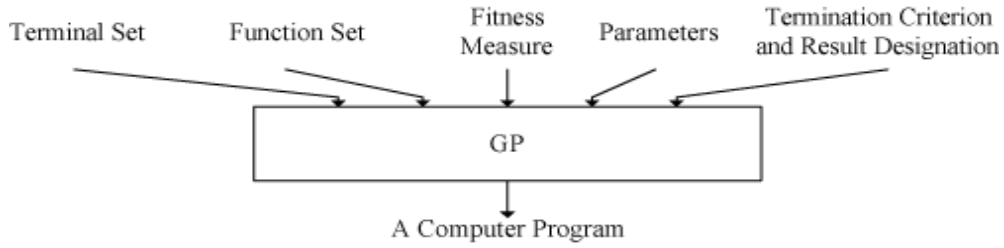


Figure 3.1: Preparatory Steps

The five major preparatory steps for the basic version of genetic programming require the human user to specify

(1) the set of terminals (e.g., the independent variables of the problem, zero argument functions, and random constants) for each branch of the to-be-evolved program

(2) the set of primitive functions for each branch of the to-be-evolved program

(3) the fitness measure (for explicitly or implicitly measuring the fitness of individuals)

(4) certain parameters for controlling the run, and

(5) the termination criterion and method for designating the result of the run.

3.2.1 Function Set and Terminal Set

The individuals in the population are compositions of functions and terminals appropriate to the particular problem domain. The set of functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. The set of terminals used typically includes inputs appropriate to the problem domain and various constants [3].

The compositions of functions and terminals described above correspond directly to the parse tree that is internally created by most compilers and to the programs found in programming languages such as LISP (where they are called symbolic expressions or S-expressions) [23]. In genetic programming, we view the search for a solution to the problem as a search in the space of all possible compositions of functions that can be recursively composed of the available functions and terminals.

3.2.2 Fitness Measure

The fitness measure specifies what needs to be done. The fitness measure is the primary mechanism for communicating the high-level statement of the problem's requirements to the genetic programming system. The first two preparatory steps define the search space whereas the fitness measure implicitly specifies the search's desired goal [15].

3.2.3 Control Parameters

The fourth preparatory step entails specifying the control parameters for the run. The most important control parameter is the population size. In practice, the user may choose a population size that will produce a reasonably large number of generations in the amount of computer time we are willing to devote to a problem. Other control parameters include the probabilities of performing the genetic operations, the maximum size for programs, and other details of the run [23].

3.2.4 Termination

The fifth preparatory step consists of specifying the termination criterion and the method of designating the result of the run. The termination criterion may include a maximum number of generations to be run as well as a problem-specific success predicate. In practice, one may manually monitor and manually terminate the run when the values of fitness for numerous successive best-of-generation individuals appear to have reached a plateau. The single best-so-far individual is then harvested and designated as the result of the run [23].

3.3 Executional Steps of Genetic Programming

Genetic programming typically starts with a population of randomly generated computer programs composed of the available programmatic ingredients. Genetic programming iteratively transforms a population of computer programs into a new generation of the population by applying analogs of naturally occurring genetic operations. These operations are applied to individual(s) selected from the population. The individuals are probabilistically selected to participate in the genetic operations based on their fitness [23].

The executional steps of genetic programming [23] are as follows:

1. Randomly create an initial population (generation 0) of individual computer programs composed of the available functions and terminals.

2. Iteratively perform the following sub-steps (called a generation) on the population until the termination criterion is satisfied:
   a) Execute each program in the population and ascertain its fitness (explicitly or implicitly) using the problem's fitness measure.

   b) Select one or two individual program(s) from the population with a probability based on fitness (with reselection allowed) to participate in the genetic operations in (c).

   c) Create new individual program(s) for the population by applying the following genetic operations with specified probabilities:
      - Reproduction: Copy the selected individual program to the new population.
      - Crossover: Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.
      - Mutation: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.
      - Architecture-altering operations: Choose an architecture altering operation from the available repertoire of such operations and create one new offspring program for the new population by applying the chosen architecture-altering operation to one selected program.

3. After the termination criterion is satisfied, the single best program in the population produced during the run (the best-so-far individual) is harvested and designated as the result of the run. If the run is successful, the result may be a solution (or approximate solution) to the problem.

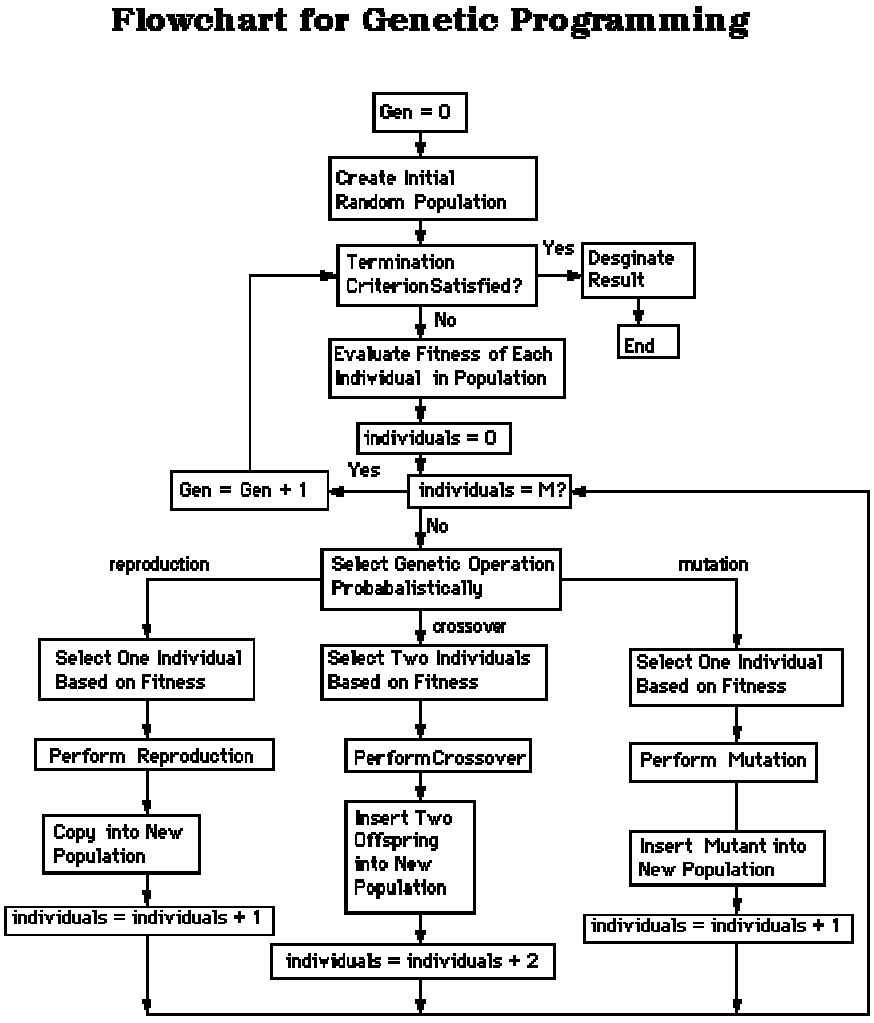


Figure 3.2: Flowchart of Genetic Programming

3.4 The Genetic Programming Algorithm in Action

3.4.1 Generate Random Initial Population

Though genetic programming prints its individuals as s-expressions, it represents them internally as trees of named nodes. In genetic programming parlance, leaf nodes in the tree are known as terminals and non-leaf nodes are known as non-terminals. Depending on the problem being solved, an individual may be a single tree, or a forest of trees. The experimenter must provide the genetic programming system with a primordial soup of basic tree nodes from which to build its program trees [15].

At the beginning of the evolution process, initial individuals must be generated at random. Genetic programming creates these individuals' trees by applying a tree generation algorithm to each tree's function set. Tree generation algorithms work by selecting and copying nodes from the templates in the function set, then hanging the copied nodes together to form the tree. The three traditional tree-generation algorithms are FULL, GROW and RAMPED HALF-AND-HALF [15].

The full method selects nodes from F until the tree reaches a pre-determined depth then it selects from T. This results in trees with uniform depth. The grow method differs in that a node is selected from C if the depth is less than a predetermined maximum; else a node is selected from T. A third method combining the full and grow is called ramped half and half. Ramped half and half operates by creating an equal number of trees with a depth between 2 and a pre-determined maximum. Then for each depth, 50% of the trees are created using the full method and 50% using the grow method.

3.4.1.1 Grow Method

With this first technique the entire population is created by using the grow method which creates one individual at a time. An individual created with this method may be a tree of any depth up to a specified maximum, *m* [15].

1. Starting from the root of the tree every node is randomly chosen as either a function or terminal.
2. If the node is a terminal, a random terminal is chosen.
3. If the node is a function, a random function is chosen, and that node is given a number of children equal to the arity (number of arguments) of the function. For every one of the function's children the algorithm starts again, unless the child is at depth *m*, in which case the child is made a randomly selected terminal.

3.4.1.2 Full Method

The full method is very similar to the grow method except the terminals are guaranteed to be a certain depth. This method requires a final depth, *d* [15].

1. Every node, starting from the root, with a depth less than *d*, is made a randomly selected function. If the node has a depth equal to *d*, the node is made a randomly selected terminal.
2. All functions have a number (equal to the arity of the function) of child nodes appended, and the algorithm starts again. Thus, only if *d* is specified as one, could this method produce a one-node tree.

3.4.1.3 Ramped-half-and-half Method

To increase the variation in structure both grow and full methods can be used in creating the population. Only a maximum depth, *md*, is specified but the method generates a population with a good range of randomly sized and randomly structured individuals [15].

1. The population is evenly divided into parts: a total of *md-1*.
2. Half of each part of the population is produced by the grow method. The other half is produced using the full method. For the first part, the argument for the grow method, *m*, and the argument for the full method, *d*, is 2. For the second part 3 is used. This continues to part *md-1*, where the number *md* is used. Thus a population is created with good variation, utilizing both grow and full methods.

3.4.2 Measure Fitness of Population

Once the initial random population has been created, the individuals need to be assessed for their fitness. In GP, the user-provided AssessFitness(pi) function usually assesses the fitness of pi by directly executing it in some problem domain as if it were an actual Lisp s-expression program, and then examining the result [11].

3.4.2.1 Raw Fitness

The definition of raw fitness depends on the problem. For many problems, raw fitness can be defined as the sum of the distances (i.e. errors), taken over all the fitness cases, between the point in the range space returned by the S-expression for the set of arguments for the particular fitness case and the correct point in the range space for the particular fitness case [11].

When raw fitness is error, the raw fitness *raw(i)* of an individual S-expression *i* in the population of size *M* is

$$raw(i) = \sum_{j=1}^{Ne} S(i, j) - C(j)$$

where *S(i,j)* is the value returned by S-expression *i* for fitness case *j* (of *Ne* cases) and *C(j)* is the correct value for fitness case *j*.

3.4.2.2 Standardized Fitness

The standardized fitness *std(i)* restates the raw fitness so that a lower numerical value is better. If a lower value of raw fitness is better (e.g. when raw fitness represents error), then standardized fitness

$$std(i) = raw(i)$$

If a higher value of raw fitness is better (e.g. when food is being eaten), standardized fitness equals the maximum possible value of raw fitness raw_{max} minus the observed raw fitness [11].

$$std(i) = raw_{max} - raw(i)$$

3.4.2.3 Adjusted Fitness

The adjusted fitness measure *adj(i)* is computed from the standardized fitness *std(i).* The adjusted fitness *adj(i*) is

$$adj(i) = \frac{1}{(1 + std(i))}$$

where *std(i)* is the standardized fitness for individual *i* at time *t*. The adjusted fitness lies between 0 and 1. The use of this adjustment is beneficial for separation of individuals with standardized fitness values that approach zero [11].

3.4.2.4 Normalized Fitness

The normalized fitness *norm(i)* is computed from the adjusted fitness value *adj(i).* The normalized fitness *norm(i)* is [11]

$$norm(i) = \frac{adj(i)}{\sum_{k=1}^{M} adj(k)}$$

Normalized fitness has three desirable characteristics.

• It ranges between 0 and 1.

• It is larger for better individuals in the population.

• The sum of the normalized fitness values is one.

3.4.2.5 Probability of Selection

The probability of selection (*sp*) is:

$$sp(i) = \frac{norm(i)}{\sum_{K=1}^{M} norm(k)}$$

(a) Order the individuals in a population by their normalized fitness

(b) Chose a random number, *r*, from zero to one.

(c) From the top of the list, loop through every individual keeping a total of there normalized fitness values. As soon as this total exceeds *r*, stop the loop and select the current individual.

3.4.3 Select the Better Individuals from the Population

Once individuals have had their fitness's assessed, they may be selected and bred to form the next generation in the evolution cycle. This is done by selecting one or two individuals from the old population, copying them, modifying them, and returning the modified copies for addition to the new population. There are several common selection strategies in use:

3.4.3.1 Fitness-Proportional Selection

This selection method, due to Holland, normalizes all the fitnesses in the population. These normalized fitnesses then become the probabilities that their respective individuals will be selected. Fitnesses may be transformed in some way prior to normalization; for example, Koza [15] normalizes the adjusted fitness rather than the standardized fitness.

3.4.3.2 Ranked Selection

One of the problems with fitness-proportional selection is that it is based directly on the fitness. Assessed fitnesses are rarely an accurate measure of how "good" an individual really is. Another approach, which addresses this issue, is to rank individuals by their fitness, and use that ranking to determine selection probability. In linear ranking individuals are first sorted according to their fitness values, with the first individual being the worst and the last individual being the best. Each individual is then selected with a probability based on some linear function of its sorted rank. This is usually done by assigning to the individual at rank *i* a probability of selection [11]

$$Pi = \frac{1}{\| P \|}\left(2 - c + (2c - 2)\frac{i-1}{\| P \| - 1}\right)$$

where ||P|| is the size of the population P, and $1 \leq c \leq 2$ is the selection bias: higher values of c cause the system to focus more on selecting only the better individuals. The

best individual in the population is thus selected with the probability $\frac{c}{\| P \|}$; the worst individual is selected with the probability $\frac{2-c}{\| P \|}$.

.

3.4.3.3 Tournament Selection

In tournament selection [23], a pool of *n* individuals is picked at random from the population. These are independent choices: an individual may be chosen more than once. Then tournament selection selects the individual with the highest fitness in this pool. Clearly, the larger the value n, the more directed this method is at picking highly fit individuals. On the other hand, if n = 1, then the method selects individuals totally at random. Popular values for n include 2 and 7.

3.4.3.4 Truncation Selection

In truncation selection [3], the next generation is formed from breeding only the best individuals in the population. One form of truncation selection, (μ, λ) selection, works as follows. Let the population size λ = kμ, where k and μ are positive integers. The μ best individuals in the population are "selected". Each individual in this group is then used to produce k new individuals in the next generation. In a variant form, (μ + λ) selection, μ individuals are "selected" from the union of the population and the μ parents, which had created that population previously.

3.4.4 Apply Genetic Operators to Generate New Population

Once parents are selected, they are used as input into the child producing algorithms known as genetic operators. There are many ways to produce children; the three most common are reproduction, crossover and mutation.

3.4.4.1 Reproduction

Reproduction is where a selected individual copies itself into the new population. It is effectively the same as one individual surviving into the next generation. Koza [15] allowed 10% of the population to reproduce. If the fitness test does not change, reproduction can have a significant effect on the total time required for GP because a reproduced individual will have an identical fitness score to that of its parent. Thus a reproduced individual does not need to be tested, as the result is already known. For Koza [15], this represented a 10% reduction in the required time to fitness test a population. However, a fitness test that has a random component, which is effectively a test that does not initialize to exactly the same starting scenario, would not apply for this increase in efficiency. The selection of an individual to undergo reproduction is the responsibility of the selection function.

3.4.4.2 Crossover

Crossover takes two parents and replaces a randomly chosen part of one parent with another. This is often very destructive to the structure and functionality of the child program. It is, however, the means by which valuable code can be transferred between programs and is also the theoretical reason why genetic programming is an efficient and successful search strategy [23].

Two parents are selected based on the fitness measure. Then, the algorithm chooses a random point in each individual (this point can be either a function or a terminal), and swaps the sub trees rooted at this point. Crossover can yield great diversity in the resulting expressions, and therefore helps prevent premature convergence of a population. It can swap functions with functions, functions with terminals, terminals with terminals or entire individuals. If crossover will yield an individual of unacceptable size then the algorithm will choose one of the parents for reproduction. An interesting fact about crossover is that if the two parents are the same individual, then it is likely that the resulting offspring will not be the same as the parents.

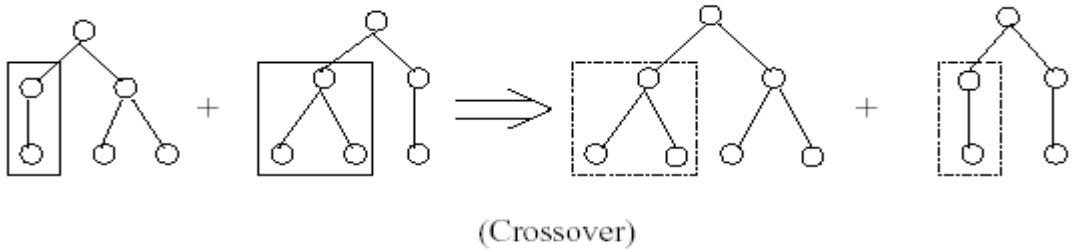


Figure 3.3: Crossover

The various ways to do crossover are given below

(i) Subtree Crossover

Subtree Crossover [23] selects two genetic programs from the population and selects one point on each. Each sub-tree from this point is swapped from the other. The closure property of the genetic program ensures that these new genetic programs are still `legal' possibilities within the domain.

(ii) Hoist

The hoist operator [25] creates a new individual entirely from a randomly chosen subtree of an existing individual. The operator is useful for promoting parsimony.

(iii) Create

The create operator [25] is unique in that it does not require any existing individuals. It generates an entirely new individual in the same way that an individual in the initial population is generated. This operator is similar to Hoist in that it helps to reduce the size of program trees.

(iv) Self-Crossover

The self-crossover [23] operator uses a single individual to represent both parents. The single individual is itself can be selected using the standard fitness proportional selection or tournament selection methods.

(v) Modular/Cassette-Crossover

One of the restrictions of the standard crossover operator is that is not possible to swap blocks that occur in the middle of a tree path. The standard crossover operator

only allows entire subtrees to be swapped. One of the reasons for the success of ADFs might be due to an effective relaxation of this restriction. This possibility lead Kinnear and Altenberg to develop the Modular or Cassette crossover operator. One can best view the operator as a module swap between two individuals. A module is defined in the program tree of the first individual, and another in the program tree of the second individual. The module in the first individual is then replaced by the one in the second individual. The new module is then expanded. Clearly there is problem created by the possible mismatch of the arguments passed to the module (actual parameters) and the formal parameters defined by the module. There are two such possibilities, either the number of formal parameters are greater than the number of actual parameters or vice-versa. The former is resolved by extending the existing actual parameters with random chosen copies of themselves. The latter is resolved by choosing at random a subset of the existing actual parameters [25].

(vi) Context Preserving Crossover (SCPC/WCPC)

D'Haeseleer also inspired by the success of ADFs has suggested alternative genetic operators called Strong Context Preserving Crossover (SCPC) and Weak Context Preserving crossover (WCPC). Both of these operators require a system of coordinates for identifying node positions in a tree. SCPC allows two subtrees to be exchanged during crossover between two parents only if the points of crossover have the same coordinates. WCPC relaxes this rule slightly by allowing crossover of any subtree of the equivalent node in the other parent [25].

3.4.4.3 Mutation

Mutation takes one parent and replaces a randomly selected chunk of that parent with a randomly generated sequence of code. One of the advantages of this operator is it maintains diversity in the population, since any of the function/terminal set can be inserted into the program, whereas crossover can only insert code present in the current generation’s population. Mutation occurs by selecting a point at random, generating a

new expression, and inserting it into the individual at the specified point. Mutation also helps prevent premature convergence [23].

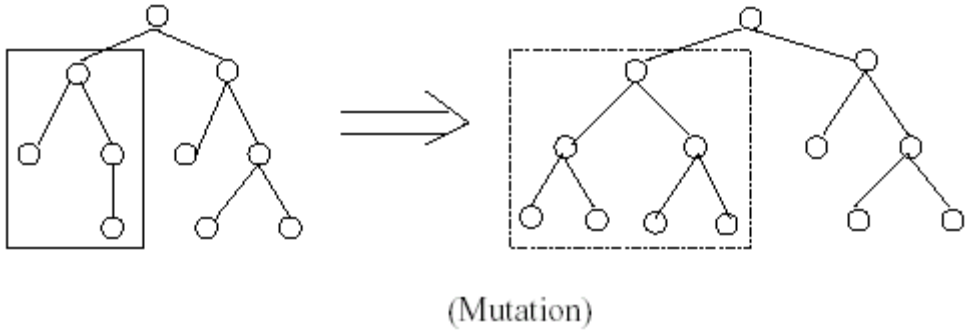


Figure 3.4: Mutation

The various ways to do mutation are given below

(i) Allele Mutation

This comprises of genes within the genetic program being swapped with other genes with certain constraints. Any terminal can be swapped with any other terminal but functions can only be swapped with other functions with the same number of arguments. This means that the mutation does not have to create new branches when different function types are swapped which would probably slow any form of convergence if the mutation rate has been set relatively high [3].

(ii) Shrink Mutation

Shrink mutation takes the child of a particular gene and moves that child into the position of the parent. This means that genetic programs will 'shrink'. [3] This is a particularly useful property when considering how long, some genetic programs get as the evolutionary process continues.

3.4.4.4 Permutation

Permutation is another operation in which a parent is selected, and then the algorithm chooses a function that composes part of the parent. The algorithm then

permutes each of the function's arguments and passes the new expression to the next generation [15].

3.4.4.5 Editing

Editing does not create an individual that evaluates differently than the parent, but forms expressions that are structurally different. Basically, editing simplifies expressions. An editing procedure recursively examines an individual and evaluates all functions that only contain terminals as arguments [15].

3.4.4.6 Encapsulation

Encapsulation [23] takes a valuable sub expression and adds it to the function list. This increases the number of times the expression occurs in a given population and makes the expression atomic, so it cannot be mutated or changed during a crossover.

3.4.4.7 Decimation

Decimation is a servicing operator that is applied to the initial generation. It removes all the programs that have least fitness. Upon creation of the initial generation it can be noted that a majority of the population have a very poor fitness. Therefore, if the desired population size is 1000 individuals then upon application of decimation of 10%, an initial population of 10,000 individuals (ten times the desired value) is created. From this population the best 1000 individuals are selected.

3.4.5 Control Parameters

The user must specify a number of control parameters before the GP system may begin. The control parameters that need to be set are:

1. **Population size**: A larger population allows for a greater exploration of the problem space at each generation and increases the chance of evolving a

solution. In general, the more complex a problem the greater the population size needed [15].

2. **Maximum number of generations**: The evolutionary process needs to be given time; the greater the maximum number of generations the greater the chance of evolving a solution. However, further evolution of a population does not guarantee a solution will be found-it may be better to start again with a different initial population [15]. So if, after a user-defined number of generations, a sufficiently successful individual has not evolved then the process should halt.

3. **Probability of crossover:** What proportion of the population will undergo crossover before entering the new population? Koza [15] does not change this value from 0.90—90% of the population undergoes crossover.

4. **Probability of reproduction:** The proportion of individuals in a population that will undergo reproduction. Throughout Koza's [15] work this value stays constant, at 0.10—10% of the population undergoes reproduction.

3.4.6 Repeat Until a Program Solves the Problem or Time Runs Out

At this point a new population is available to be evaluated for fitness. The cycle will continue, until ether a single member of the population is found which satisfies the problem within the level of error designated as acceptable by the success criteria, or the number of generations exceeds the limit specified [23].

As discussed earlier, if a run does not succeed after a large number of generations, it has probably converged onto a semi-fit solution and the lack of diversity in the population is drastically slowing evolutionary progress. Statistically, the likelihood of finding a successful individual is, at that point, most increased by starting the entire run over again with a wholly new random population.

CHAPTER-4
EDGE ENCODING FOR FSA INDIVIDUALS

Edge encoding is a technique for evolving graph and network structures via genetic programming. It was introduced by Luke and Spector [18], and aimed to allow more flexible control of the edge growth in the topology evolution. It is a genetic programming tree, which produces a directed graph when evaluated. This graph is then used in the problem domain as appropriate-as an electrical circuit, FSA, etc.

4.1 Introduction

Edge encoding [18] uses a tree-structured chromosome, which can develop into a directed graph when executed. Each node of the chromosome tree is an operator that can act on the edges of a graph. An edge operator can accept from its parent node a single edge in the graph, sometimes with additional data such as a stack of nodes. Edge encoding operators are executed in a pre-ordered way, modifying a graph edge and passing that edge (and any new edges) to its children for further modification. The starting point of the development can be a graph with a single edge, which is fed to the root node of the chromosome tree as the starting point for development.

For example, consider the double function shown in Figure 6.1. This function has two children in the encoding tree. It receives from its parent a single edge $E(a,b)$ in the graph (where a is the tail of the edge E, and b is E's head). From $E(a,b)$, double "grows" an additional edge $F(a,b)$. These two edges are each passed to child functions for additional modifications; E is passed to the double's left child, and F is passed to its right child.

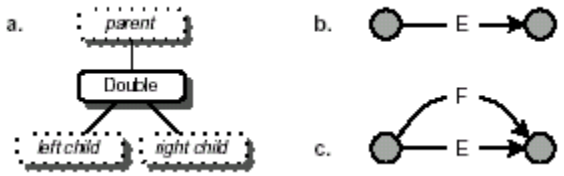


Figure 4.1: The Double Function

Edge encoding's graph-generation process begins with a graph consisting of a single edge. This edge is passed to the root node in the edge-encoding tree, which modifies the edge and passes resultant edges to its children, and so on. Terminals in the edge-encoding tree have no children, and so stop the modification process for a particular edge. After all nodes in the tree have made their modifications, the resultant graph is returned.

The functions in a particular edge encoding are commonly of two forms. First, there are functions, which change the topology of the graph, by adding or deleting edges or vertices. Second, there are functions, which add semantics to the edges or vertices: labeling edges, assigning transfer functions to vertices, etc.

4.2 Encoding an NFA

To encode an NFA we have simple set of basic functions, which are sufficient to build all non-deterministic finite-state automata (NFA). These functions each take an edge and no optional data from their parents, and pass on to children at most two resultant edges. A nice property of these functions is that although they have side effects (in the way of modifying the graph), these side effects are localized in such a way that the functions are referentially transparent. Thus nodes can be executed in any order, so long as parent functions are executed prior to child functions [18].

In this encoding, each individual consists of a single tree of functions. Assume that each function is passed some edge $E(a,b)$, which after processing is passed to the left child. The functions, which describe the topology of the graph, are shown in Table 4.1. These functions are sufficient to develop the topology of an NFA, which recognizes any regular expression. To develop the full NFA, some custom semantic functions are necessary to define the starting and accepting states of the NFA and label the edges with tokens, which are shown in Table 4.2.

Function Syntax	Arity	Description
double	2	Create an edge $F(a,b)$.
bud	2	Create a vertex c. Create an edge $F(b,c)$.
split	2	Create a vertex c. Modify E to be $E(a,c)$. Create an edge $F(c,b)$.
loop	2	Create a self-loop edge $F(b,b)$.
reverse	1	Reverse E to be $E(b,a)$.

Table 4.1: Simple Topological Functions for Edge Encoding

Function Syntax	Arity	Description
start	1	Assign the head of $E(a,b)$ (vertex b) to be a starting state.
accept	1	Assign the head of $E(a,b)$ (vertex b) to be an accepting state. It's valid for a state to be a starting state and an accepting state at the same time.
1	0	Label an edge with a "1", that is, define it to be an edge which can be traversed only on reading a 1.
0	0	Label an edge with a "0", that is, define it to be an edge which can be traversed only on reading a 0.
ϵ	0	Label an edge with an "ϵ", that is, define it to be an edge which may be traversed without reading any token.

Table 4.2: NFA Semantic Functions for Edge Encoding

Figure 4.2 shows an edge encoding genome using these functions whose phenotype is an NFA that reads the regular expression ((0|1)*101). Figure 4.3 shows the development of the NFA from this genome. These basic sets of functions and terminals, alone are sufficient to build all non-deterministic finite-state automata (NFA) of interest.

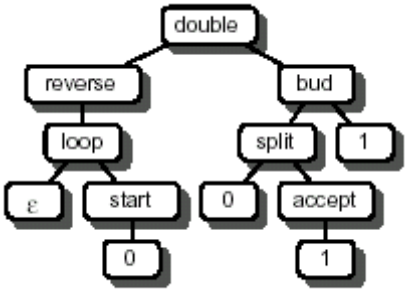
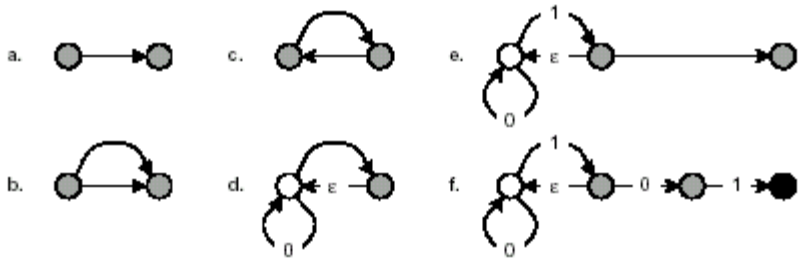


Figure 4.2: An Edge Encoding Genome, which Describes an NFA that Reads the Regular Expression ((0|1)*101)



Legend. (a) The initial edge. (b) After applying double. (c) After applying reverse. (d) After applying loop, S, ϵ, and 0. The white circle is a starting state. (e) After applying bud and 1. (f) After applying split, 0, A, and 1. The black circle is an accepting state.

Figure 4.3: The Growth of the NFA from the Encoding in Figure 4.2

CHAPTER 5

PROBLEM AND SCHEME SPECIFICATIONS

Finite state automata are well understood, and inherently efficient models of simple languages. The basic problem is, given a set of positive and negative example strings, automatically infer corresponding automata, which generates or recognizes those examples. Genetic programming has been used for evolving such an automata for the benchmark problem of Tomita Language Set [24]. The acceptable solution automata should correctly recognize all positive test cases, and reject all negative ones.

The first step in building an NFA from a regular expression is to determine the set of functions and terminals, which is used for the population initialization.

5.1 Chromosomal Encoding

Since edge encoding [18] is more suitable for evolving graphs with low connectivity. So, for representing the FSA chromosomes we use edge encoding, as explained in the previous chapter. Edge encoding uses a tree-structured chromosome, which can develop into a directed graph when executed. Each node of the chromosome tree is an operator that can act on the edges of a graph. An edge operator can accept from its parent node a single edge in the graph, sometimes with additional data such as a stack of nodes. Edge encoding operators are executed in a pre-ordered way, modifying a graph edge and passing that edge (and any new edges) to its children for further modification. The starting point of the development can be a graph with a single edge or an embryo with many edges, but only one of them is fed to the root node of the chromosome tree as the starting point for development.

To encode an NFA, the basic set of functions and terminals, which are sufficient to build all non-deterministic finite-state automata (NFA) are provided here again in

Table 5.1, and Figure 5.1 shows an edge encoding genome tree created randomly using those functions and terminals and its equivalent s-expression [18].

Function Syntax	**Arity**	**Description**
Double	2	Create an edge F(a,b).
Bud	2	Create a vertex c. Create an edge F(b,c).
Split	2	Create a vertex c. Modify E to be E(a,c). Create an edge F(c,b).
Loop	2	Create a self-loop edge F(b,b).
Reverse	1	Reverse E to be E(b,a).
Start	1	Assign the head of E(a,b) (vertex b) to be a starting state.
Accept	1	Assign the head of E(a,b) (vertex b) to be an accepting state.
1	0	Label an edge with a "1", that is, define it to be an edge which can be traversed only on reading a 1.
0	0	Label an edge with a "0", that is, define it to be an edge which can be traversed only on reading a 0.
∈	0	Label an edge with an "∈", that is, define it to be an edge which may be traversed without reading any token.

Table 5.1: Description of function set and terminal set for edge encoding an NFA

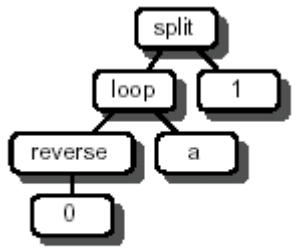


Figure 5.1: An edge encoding genome and its equivalent s-expression

5.2 Generating the Initial Population

Initialize(*n*) function forms a population by generating individuals at random until it has collected *n* unique individuals. In this thesis, individuals' trees are initially generated with the Ramped Half and Half Tree Creation Algorithm, with a depth bound range from 2 to 6 inclusive.

5.3 Tomita Languages

In our problem domain, the NFA is evolved to induce the Tomita language set, shown in Table 5.2, a popular and nontrivial language induction benchmark (Tomita [26]).

Language	Description
1	1*
2	(10)*
3	(0|11)*(1*|(100(00|1)*)) Any string without an odd number of consecutive 0's after an odd number of consecutive 1's
4	1*((0|00)11*)*(0|00|1*) Any string without more than 3 consecutive 0's
5	(((1|0)(1|0))*(1|0))|((11|00)*((01|10)(00|11)*(01|10)(00|11)*)*(11|00)*) Any string of even length which, making pairs, has an even number of (01)'s or (10)'s
6	((0(01)*(1|00))|(1(10)*(0|11)))* Any string such that the difference between the number of 1's and 0's is a multiple of 3
7	0*1*0*1*

Table 5.2: The Tomita Language Set

The basic problem is, given a set of positive and negative example strings, from the Tomita Language set, automatically infer corresponding automata, which generates or recognizes those examples. The acceptable solution automata should correctly recognize all positive test cases, and reject all negative ones. Afterwards, it is tested for generality on the full population of binary strings of that expression.

The example positive and negative training sets used for the Tomita Languages are shown in Table 5.3 below.

Tomita Lang.	Positive Examples	Negative Examples
1	ϵ, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111	0, 10, 01, 00, 011, 110, 11111110, 10111111
2	ϵ, 10, 1010, 101010, 10101010, 10101010101010	1, 0, 11, 00, 01, 101, 100, 1001010, 10110, 110101010
3	ϵ, 1, 0, 01, 11, 00, 100, 110, 111, 000, 100100, 110000011100001, 111101100010011100	10, 101, 010, 1010, 1110, 10001, 111010, 1001000, 11111000, 0111001101, 1011, 11011100110
4	ϵ, 1, 0, 10, 01, 00, 100100, 001111110100, 0100100100, 11100, 010	000, 11000, 0001, 000000000, 00000, 11111000011, 1101010000010111, 1010010001, 0000
5	ϵ, 11, 00, 1001, 0101, 1010, 1000111101, 1001100001111010, 111111, 0000	1, 0, 111, 010, 000000000, 1000, 01, 10, 011, 1110010100, 010111111110, 0001
6	ϵ, 10, 01, 1100, 101010, 111, 000000, 10111, 0111101111, 100100100	1, 0, 11, 00, 101, 011, 00000000, 010111, 10111101111, 11001, 1001001001, 1111
7	ϵ, 1, 0, 10, 01, 11111, 000, 00110011, 0101, 0000100001111, 00, 00100, 011111011111	1010, 00110011000, 0101010101, 1011010, 10101, 010100, 101001, 100100110101

Table 5.3: Positive and Negative Training Examples.

5.4 Fitness Assessment

The fitness function is used to rank the individual solutions present in each generation of the GP. The fitness criteria used here is the number of strings that the particular finite state machine is able to correctly classify.

In this dissertation, the Hit Count gives the sum of the number of positive strings accepted and the number of negative strings rejected.

The raw fitness F_{raw}, [4] is defined as a fitness parameter in the range [0,1), where 0 represents the optimum and 1 worst possible fitness. In our case it is given by

$$F_{raw} = 1 - \frac{correct\ \ positive\ examples + correct\ \ negative\ examples}{total\ \ positive\ examples + total\ \ negative\ examples}$$

The individual's adjusted fitness F_{adj}, defined as $F_{adj} = \frac{1}{1+F_{raw}}$, maps the fitness into the interval (0,1], where 0 is worse than the worst fitness and 1 is the optimum.

5.5 Selection and Breeding

Once individuals have had their fitness's assessed, they may be selected and bred to form the next generation in the evolution cycle, through repeated application of Breed(...). This function usually selects one or two individuals from the old population, copies them, modifies them, and returns the modified copies for addition to the new population.

5.5.1 Selection Scheme

The selection mechanism used here is *tournament selection* [15], as it is simple, fast, and has well-understood statistical properties. In tournament selection, a pool of *n*

individuals is picked at random from the population. Then tournament selection selects the individual with the highest fitness in this pool. Clearly, the larger the value *n*, the more directed this method is at picking highly fit individuals. Tournament size of 7 is used here in this dissertation, as it is the standard in the genetic programming literature, and is highly selective.

5.5.2 Crossover Scheme

The crossover scheme used here is *subtree crossover* [15], which starts with two individuals selected and copied from the old population. A random point is selected within one tree of each copied individual with nonterminals selected 90% of the time and terminals selected 10% of the time. Then crossover swaps the subtrees rooted at these two nodes and returns the modified copies. If the crossover process results in a tree greater than a maximum depth bound (17), then the modified child is discarded and its parent (the tree into which a subtree was inserted to form the child) is simply copied through reproduction. Crossover may only occur if two trees share the same function set. This process is illustrated in Figure 5.2.

5.5.3 Mutation Scheme

The mutation scheme used is *subtree mutation* [15]. Subtree mutation starts with a single individual selected and copied from the old population. One node is selected from among of the copied individual's trees, using the same node-selection technique as described for subtree crossover. The subtree rooted at this node is removed and replaced with a randomly generated subtree, using the GROW algorithm and the appropriate function set for the tree. If mutation results in a tree greater than the maximum depth (17), then the copy is discarded and its parent is reproduced instead. Subtree mutation is illustrated in Figure 5.3.

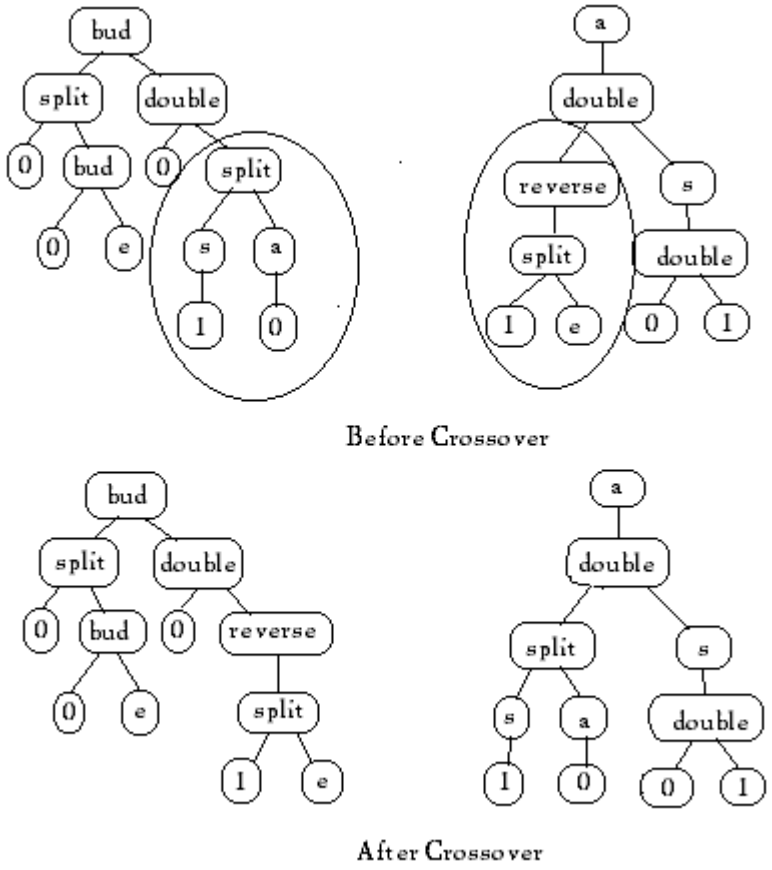


Figure 5.2: Subtree Crossover

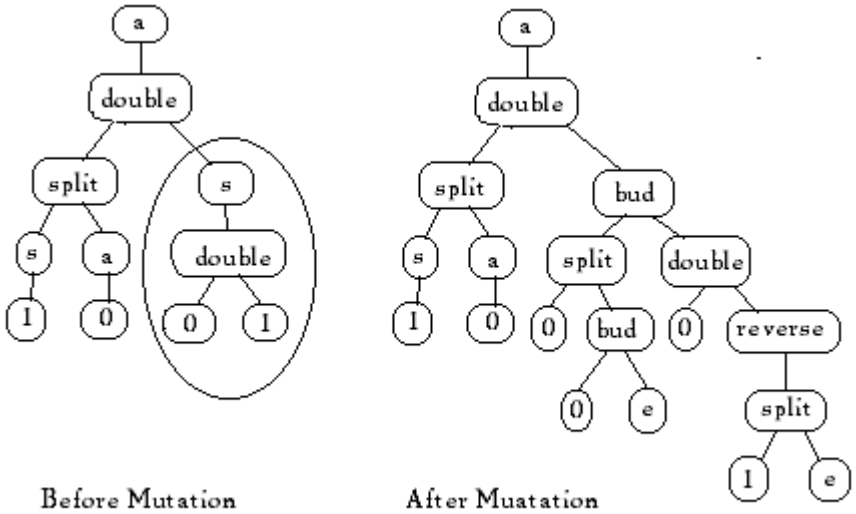


Figure 5.3 :Subtree Mutation

5.6 Control Parameters

The fourth preparatory step entails specifying the control parameters for the run. The decisions are critically important as they have a limiting effect on the search space of possible programs. Too great a limit may remove all chance of evolving an acceptable individual.

The control parameters that need to be set are:

1. Population Size: Population size represents the number of chromosomes that constitute the population at any given time. If there are too few chromosomes, the GP has few possibilities to perform crossovers and only a small part of search space is explored. On the other hand, if there are too many chromosomes, the GP slows down. Here, the population size is taken to be 500.

2. Maximum number of generations: The evolutionary process needs to be given time; the greater the maximum number of generations the greater the chance of evolving a solution. However, further evolution of a population does not guarantee a solution will be found - it may be better to start again with a different initial population. So if, after a user-defined number of generations, a sufficiently successful individual has not evolved then the process should halt. The maximum number of generations to be evolved is chosen to be 50.

3. Probability of crossover: This specifies the proportion of the population that will undergo crossover before entering the new population. It is taken to be 0.9.

4. Probability of reproduction: The proportion of individuals in a population that will undergo reproduction. Following the Koza's standard, 0.1 of the population will undergo reproduction.

CHAPTER-6
MODULAR ARCHITECTURE TO EVOLVE FINITE STATE AUTOMATA

As we have seen in the previous chapters that the automatic creation of finite automata has long been a goal of the evolutionary computation community. The previous works in the evolution of finite state machines were limited to the evolution of strictly non-modular FSA. Here, a modular FSA architecture is proposed and a genetic programming procedure for evolving such structures is presented. Preliminary results indicate that the proposed procedure is indeed capable of successfully evolving modular FSA and that such modularity can result in a statistically significantly increased rate of optimization.

Perhaps the most important attractive property of finite state automata is that they can be combined in various interesting ways, with the guarantee that the result again is a finite state automaton. This property is perhaps most clearly exploited here in modular FSA architecture.

6.1 Proposed Evolution Model

The previous works in the evolution of finite automata accepting a particular regular language comprises of, first generating the sample positive strings described by the given regular expression and sample negative strings which are not described by the given regular expression. And then, the finite state automaton is inferred, from these sets of positive and negative data samples, thereby accepting a particular regular language.

In the proposed Modular Architecture, we will first breakdown the given regular expression into few smaller sub-expressions, and evolve the finite automata for each of

these sub-expressions, and then further combine each of the evolved automata to get the complete automata describing the given regular expression as a whole.

The proposed architecture is also supported by the fact, that, a difficult task when decomposed into simpler subtasks can be solved with lower computational effort, with their solutions combined to give the overall solution. Further, already discovered solutions to subtasks may also be reused to repeatedly solve similar sub problems.

Thus, the method reduces the total number of generations needed to evolve finite state automata for a complex regular expression, as breaking it down into simpler smaller sub-expressions reduces the complexity of the search space. Also, the time required and the size of the edge encoding tree as well as the size of the evolved finite automata is reduced.

6.2 Design Of Modular Architecture

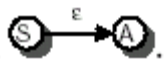
The design of modular FSA roughly follows Thompson's construction as described in Aho, Sethi, and Ullman [1] and Thompson [22]. Thompson's construction first parses a regular expression into its subexpressions, and then builds an NFA bottom-up by grouping smaller NFAs that represent those subexpressions.

By definition, for any regular expression r, one of the following is true.

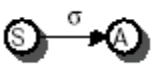
- **r** is **ε**.
- **r** is a symbol $\sigma \in \sum$.
- **r** can be broken down into **s t** for some regular expressions s and t.
- **r** can be broken down into **s|t** for some regular expressions s and t.
- **r** can be broken down into **s*** for some regular expression s.
- **r** can be broken down into **(s)** for some regular expression s.

For each of these cases, the edge encoding which produces the appropriate NFA, and also the NFA itself is given below. Each constructed NFA will have one start-state and one accepting-state, indicated with an "S" and "A" respectively.

Case 1. r is **ε**. In this case, the edge encoding for **r** is simply **ε**. This produces the NFA

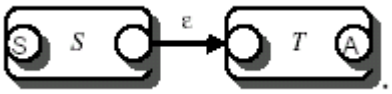


Case 2. r is a symbol $\sigma \in \sum$**.** In this case, the edge encoding for **r** is simply $\mathbf{t_\sigma}$, where t_σ is the terminal that corresponds with r as described above. This produces the NFA



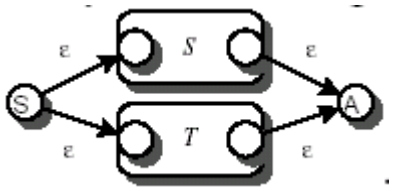
Case 3. r can be broken down into **s t**. Let *S* and *T* be the edge encodings for s and t, respectively, with NFAs

Then the edge encoding for **r** is **(split *S* (split ε *T*)),** which produces the NFA

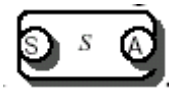


Case 4. r can be broken down into **s | t**. Let *S* and *T* be the edge encodings for s and t, respectively, with NFAs

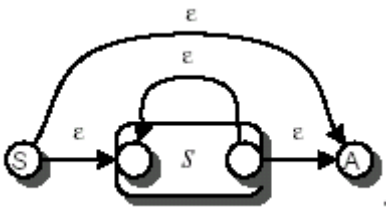
Then the edge encoding for **r** is **(double (split (split ε *S*) ε) (split (split ε *T*) ε)),** which produces the NFA



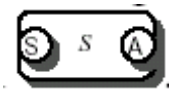
Case 5. r can be broken down into **s*.** Let *S* be the edge encoding for s, encoding the NFA



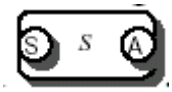
Then the edge encoding for **r** is **(double (split (split ε (double *S* (reverse ε))) ε) ε),** which produces the NFA



Case 6. r can be broken down into **(s).** Let *S* be the edge encoding for s, respectively, encoding the NFA



Then the edge encoding for **r** is the same as *S*, producing the NFA



Labeling Start and Accepting States- The start-state vertex will have one or more labeled edges leaving it, and the accepting-state vertex will have one or more labeled edges entering it. When we have finished constructing our final NFA, we can indicate its start state by finding the terminal function (call it *A*) responsible for labeling some outgoing edge of the start-state vertex. We convert this terminal function into (reverse (start (reverse *A*))), which labels the start-state vertex. To indicate the NFA's accepting

state, we find the terminal function (*B*) responsible for labeling some incoming edge of the accepting-state vertex. We convert this terminal function into (accept *B*), which labels the accepting-state vertex.

6.3 Computational Algorithms

The procedure to evolve a finite state automaton for a given regular expression using the modular architecture is divided into two algorithms. The first one is the Basic NFA Evolution Algorithm [18], which evolves an automaton for the whole expression, by creating the positive and negative test samples of the regular expression and representing the chromosomes as edge encoding tree.

The second one, Modular Algorithm, proposed in this dissertation, first, decomposes the regular expression into simple smaller sub-expressions, and then, uses the Basic NFA Evolution Algorithm for each of the sub-expressions. After this, the sub-automata evolved for each of the sub-expressions is joined together using the modular design described in Section 6.2 resulting in the complete automata for the initial whole expression. The procedural steps for both the algorithms are given below.

6.3.1 Basic Algorithm for NFA Evolution

1. First of all create around 10-15 sample positive and negative test strings from the regular expression for which the automata is to be evolved. Also the set of all-positive strings described by the expression and the set of all-negative strings not recognized by the expression is created to check the generalization score of the final evolved automata.

2. Randomly create an initial population (generation 0), of 500 individual edge encoding tree, composed of the available functions and terminals using ramped half-and-half algorithm.

3. Iteratively perform the following sub-steps (called a generation) on the population until the termination criterion is satisfied:

    a. Execute each program in the population and ascertain its fitness by testing the NFA against the positive and negative test examples.
    b. Select one or two individual program(s) using tournament selection with the tournament size 7, from the population with a probability based on fitness (with reselection allowed) to participate in the genetic operations in (c).
    c. Create new individual program(s) for the population by applying the following genetic operations with specified probabilities:
        - Reproduction: Copy the selected individual program to the new population.
        - Crossover: Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.
        - Mutation: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.

4. After the termination criterion is satisfied, the single best program in the population produced during the run (the best-so-far individual) is harvested and designated as the result of the run.

5. The Best Individual is checked for its generalization score by testing it against all the positive and negative training strings of the given regular expression. The generalization score of 1 results when the solution found accepts all the positive strings and rejects all the negative ones.

6. If the run is successful and the generalization score is one, the result may be a solution to the problem.

6.3.2 Proposed Modular Algorithm

1. Given a regular expression, first break it down into 2-4 sub-expressions.
2. For each of the sub-expression, define the positive and negative test strings files.
3. Call the Basic NFA evolution Algorithm for each of the sub-expressions with the test strings files as the input.
4. Recompose the final NFA, by grouping each of the evolved sub-automata from step (3) using the procedure described in section 6.2.
5. The final NFA is checked for the generalization score, so that it may be a correct solution to the expression.

6.4 Example Evolution

To gain a better understanding of the modular evolution, we will here step through the growth of a simple automaton that recognizes the language 1* (10)*. In this, the regular expression is divided into two sub-expression, as 1* and (10)* and for each of the expressions the NFA is evolved using the Basic NFA Evolution Algorithm. The evolved edge encoding tree and the NFA are shown in the Figure 6.1 and 6.2.

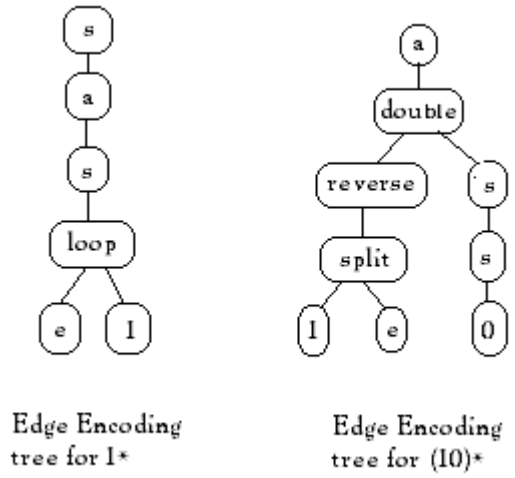


Figure 6.1: Edge Encoding for 1* and (10)*

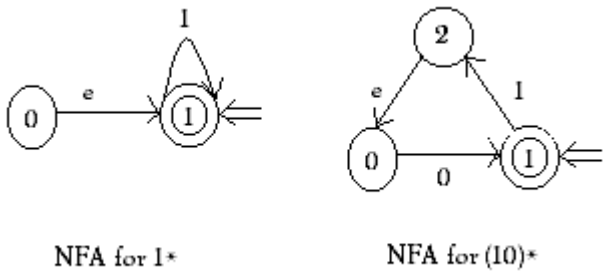


Figure 6.2: Encoding Tree and NFA for 1* and (10)*

As we have seen earlier, the edge encoding for **r** when broken down into **s t**, where *S* and *T* be the edge encoding for s and t respectively, is (split *S* (split ε *T*)). Thus, the expression 1*(10)* after combining have the encoding tree and the NFA as shown in Figure 6.3.

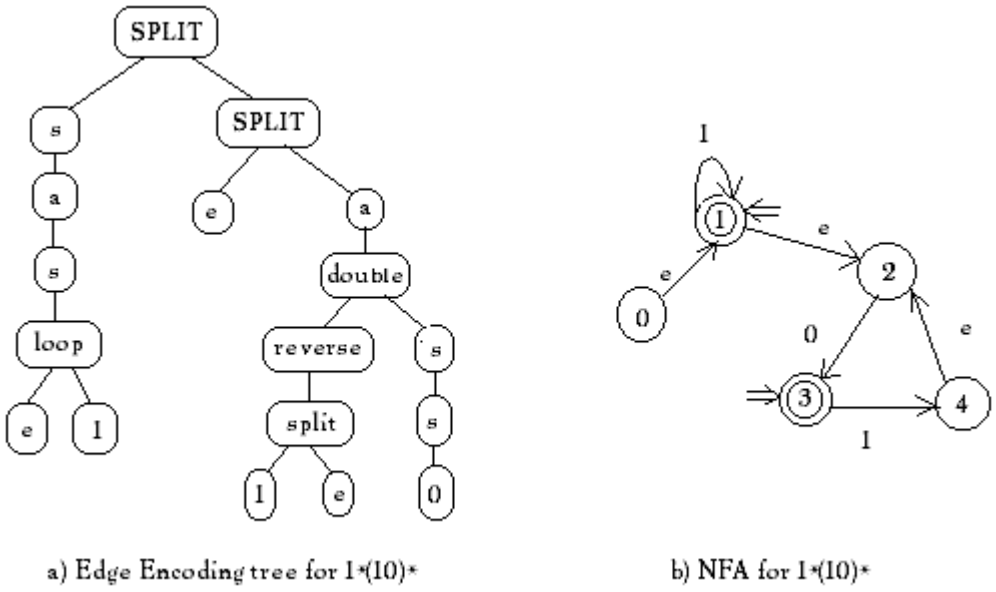


Figure 6.3: Edge Encoding Tree and the NFA for 1*(10)* using modular evolution

6.5 Summary

The modular architecture is developed to provide for some general mechanism whereby a problem can be decomposed into distinct subtasks and to allow for the preservation of elements of the representation that may be useful in solving the task at hand.

CHAPTER-7
EXPERIMENTS AND RESULTS

7.1 Experimental Setup

7.1.1 Implementation

The GP system, for the evolution of NFA using both modular and non-modular architecture has been implemented in Java Programming Language. Java has been extensively used for the implementation of GP system as it is portable and can be easily replicated across platforms. It also provides distributed computation and is easily extensible.

7.1.2 Test Data

To assess the performance of the proposed procedure it was tested on the Tomita Language Set [24], a popular and nontrivial language induction benchmark. The Tomita set results indicate that using modular architecture we can induce an automaton with far fewer fitness evaluations than the previous non-modular methods.

7.1.3 Tomita Decomposition For The Modular Evolution

Table 7.1 shows the decomposition of the Tomita languages into sub-expressions for evolution using the modular architecture. For the Tomita 1 and Tomita 2, no further breakdown is possible, so the expression is evolved as a whole similar to the non-modular approach. Tomita 3, Tomita 4 and Tomita 5 are decomposed further into two sub-expressions respectively, and the NFA for the sub-expressions are evolved separately, later the sub NFA's are joined using the modular design. For the Tomita 6 and Tomita 7 only one NFA for the sub-expression 1 is evolved and the resulting NFA can be used for the sub-expression 2, as they are similar. In the case of sub-expression 2 of Tomita 6, only the need is to change 1's into 0's and 0's into 1's in the NFA of sub-

expression 1, whereas in the case of Tomita 7 there's no such need as the sub-expressions are exactly same.

Non-modular Evolution	**Modular Evolution**	
Tomita Languages as a Whole	**Tomita Languages Sub expression 1**	**Tomita Languages Sub expression 2**
1*	1*	-
(10)*	(10)*	-
(0 | 11)* (1* | (100 (00 | 1)*))	(0 | 11)*	(1* | (100 (00 | 1)*))
1*((0 | 00) 11*)*(0 | 00 | 1*)	1*((0 | 00) 11*)*	(0 | 00 | 1*)
(((1|0)(1|0))*(1|0)) | ((11|00)*((01|10)(00|11)*(01|10)(00|11)*)*(11|00)*)	(((1|0)(1|0))*(1|0))	((11|00)*((01|10)(00|11)*(01|10)(00|11)*)*(11|00)*)
((0(01)*(1|00)) | (1(10)*(0|11)))*	((0(01)*(1|00))	(1(10)*(0|11)))*
0*1*0*1*	0*1*	0*1*

Table 7.1: Tomita Decomposition for the Modular Evolution

7.1.4 Fitness Metric

The standard experimental methodology for most Tomita language induction experiments in the literature is to attempt to induce a mechanism, which properly classifies all positive and negative examples in a limited training set. Afterwards, this mechanism is tested for generality on the full population of binary strings of that length. The same accuracy measurement, which was used by the other experiments for the raw fitness metric is used here, namely:

$$F_{raw} = 1 - \frac{correct\ \ positive\ examples + correct\ negative\ examples}{total\ \ positive\ examples + total\ \ negative\ examples}$$

After an evolutionary run is completed, the generalization accuracy of its highest-fitness individual is measured. Generalization accuracy uses all possible strings

up to 15 symbols in length. The resultant score estimates how closely the NFA was able to properly generalize to that particular Tomita language. Generalization score metric is:

$$Gen.\ Accuracy = \frac{correct\ \ positive\ examples + correct\ \ negative\ examples}{total\ \ positive\ examples + total\ \ negative\ examples}$$

7.2 Experimental Results

7.2.1 Population-based Analysis

In this dissertation a procedure for evolving Finite Automata using modular architecture is proposed. Two separate sets of experiments were performed to compare the effectiveness of this approach against the existing non-modular approach, evolving FSA for the Tomita Language Set. A total of 30 runs were carried out for each of the Tomita Languages in both the experiment set. Each set of experiments used a population size of 500 machines and evolution lasted for 50 generations or until a solution was found which correctly classified all its training examples.

Table 7.2 summarizes the results, showing the number of generations needed, as well as the number of nodes evaluated, in the best run and the average of the 30 runs, for the Tomita Languages evolved using the non-modular approach. Similarly, Table 7.3 shows the results for each sub-expression evolution using the modular approach. Figure 7.1 and Figure 7.2 compares the modular and non-modular approach on the basis of number of generations explored for the Tomita Languages. Similarly, Figure 7.3 and Figure 7.4 compares both the approaches on the basis of Number of Nodes Evaluated.

From the results it can be concluded that the proposed procedure is capable of successfully evolving modular Finite Automata with significantly increased rate of optimization, by reducing the number of generations evolved and the number of nodes evaluated.

Tomita No.	Best		Average	
	No. of Generations Needed	No. of Nodes Evaluated	No. of Generations Needed	No. of Nodes Evaluated
1	1	10714	1.133	11068.2
2	1	10856	2.833	24174.4
3	7	179622	15.466	780046.7
4	8	223316	14.533	491532.6
5	14	1228148	38.866	5944380.2
6	17	942558	36.633	3564287.6
7	7	242900	13.57	611230.3

Table 7.2: Number of generations explored and the number of nodes evaluated for the evolution of each Tomita language using non-modular approach

Tomita No.	Sub-Expressions	Best		Average	
		No. of Generations Needed	No. of Nodes Evaluated	No. of Generations Needed	No. of Nodes Evaluated
1	1	1	10714	1.133	11068.2
2	1	1	10856	2.833	24174.4
3	1	1	10624	3.166	46020.2
	2	2	27368	5.2	111626.1
4	1	2	73296	3.86	90664.3
	2	1	3405.55	3.2	52238.4
5	1	3	45580	6.5	159616.3
	2	5	109414	9.06	344836.4
6	1	8	557904	16.43	1050733.1
7	1	1	11090	2.16	28544.6

Table 7.3: Number of generations explored and the number of nodes evaluated for the evolution of each Tomita language using proposed modular approach

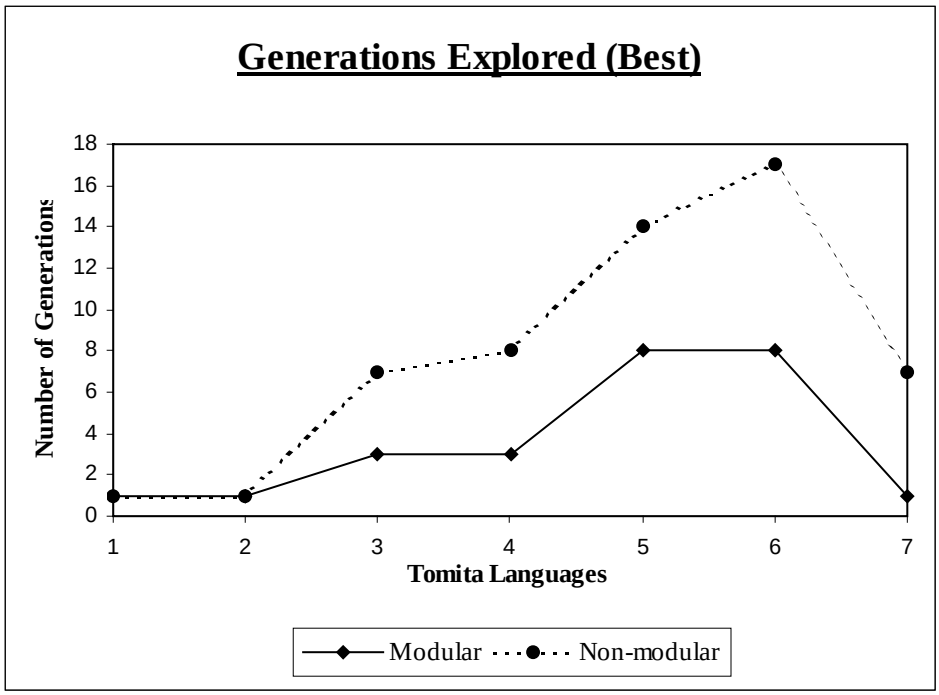


Figure 7.1: Number of Generations Explored in the Best case for the Tomita Languages

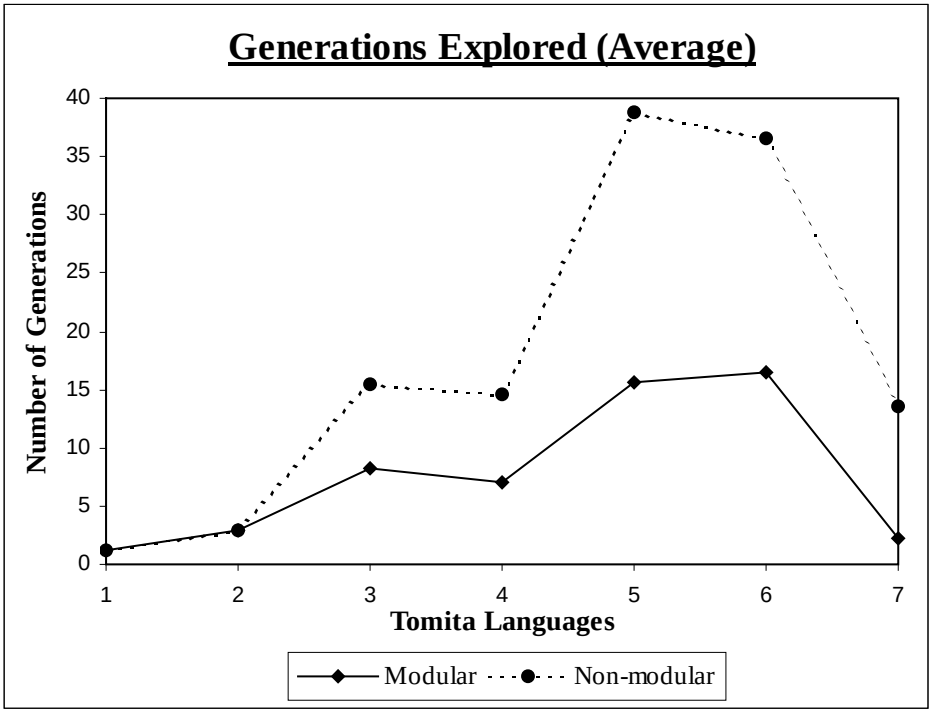


Figure 7.2: Number of Generations Explored in the Average case for the Tomita Languages

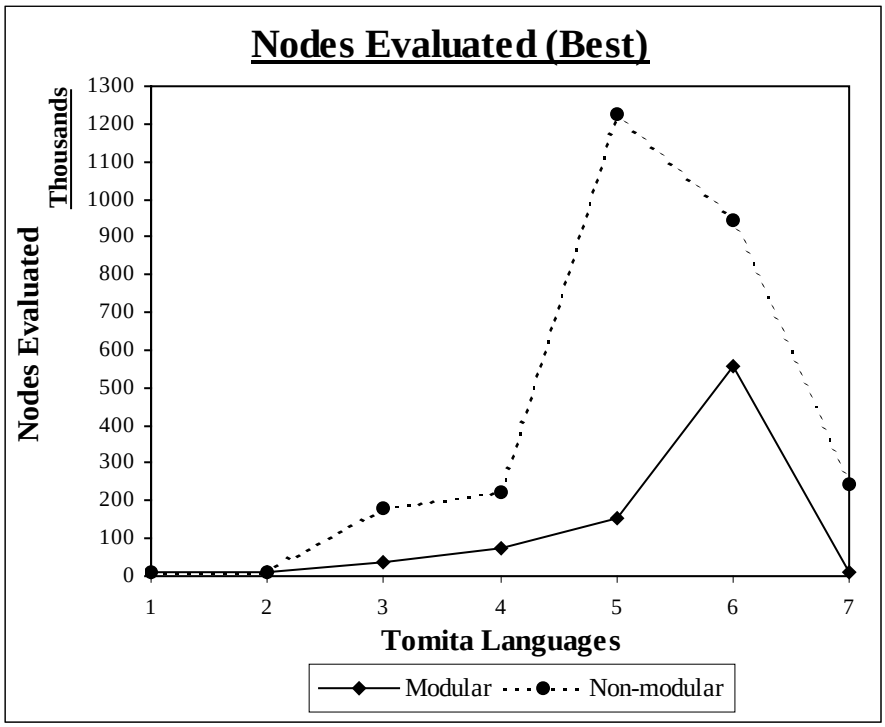


Figure 7.3: Number of Nodes Evaluated in the Best case for the Tomita Languages

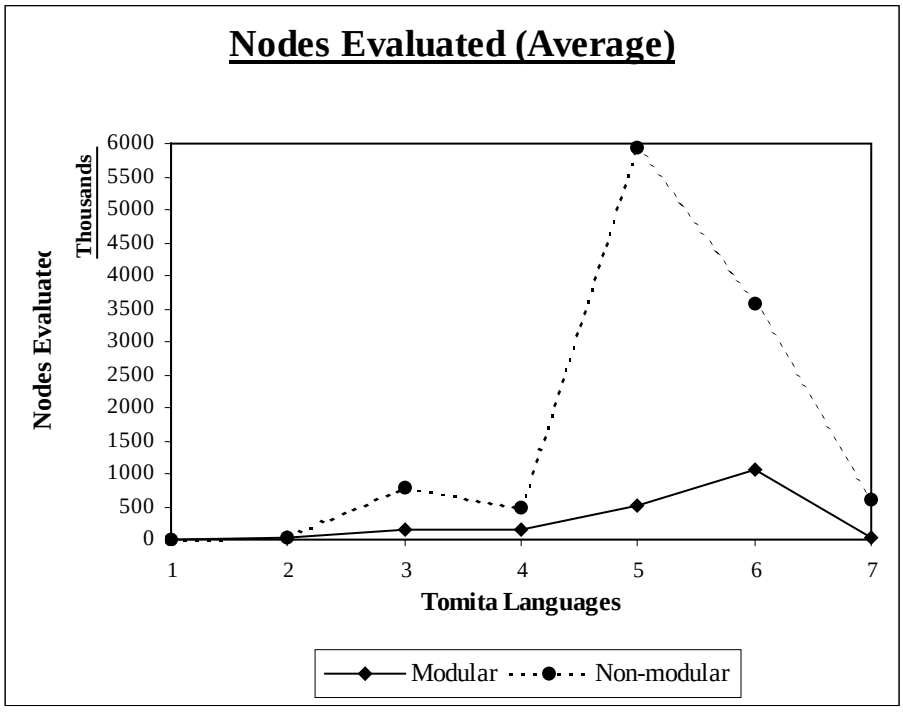


Figure 7.4: Number of Nodes Evaluated in the Average case for the Tomita Languages

7.2.2 Timing Analysis

Table 7.4 shows the average elapsed time for the evolution of Tomita Language Sets both using modular and non-modular approach. Also Figure 5.5 and Figure 5.6 compares both the approaches on the basis of Time Taken for finding the solution for the Tomita Languages in the best and the average case. All timings are based on Java implementations running on a 706 MHz Pentium processor.

Tomita Number	**Non-modular Evolution**		**Modular Evolution**					
	Total Time taken (ms)		**Time taken by sub expression 1 (ms)**		**Time taken by sub expression 2 (ms)**		**Total Time taken (ms)**	
	Best	**Average**	**Best**	**Average**	**Best**	**Average**	**Best**	**Average**
1	**1650**	**1808.16**	1650	1808.16	-	-	**1650**	**1808.16**
2	**1700**	**2297.76**	1700	2297.76	-	-	**1700**	**2297.76**
3	**8010**	**25665.2**	1760	3240.13	2470	5667.16	**4230**	**8907.29**
4	**9170**	**16974.5**	3720	5088.57	1980	3405.55	**5700**	**8494.12**
5	**38950**	**182420**	3830	6920.4	5160	10450.2	**8990**	**17370.6**
6	**31970**	**116053**	8140	23041.4	-	-	**8140**	**23041.4**
7	**8900**	**20920.7**	1700	2520.2	-	-	**1700**	**2520.2**

Table 7.4: Average elapsed time in milliseconds to learn the Tomita languages for both non-modular and modular architecture

7.2.3 Performance Evaluation

The performance over the seven Tomita targets indicates that the proposed modular approach evolves an automaton with statistically increased rate of optimization as compared to the non-modular approach. Furthermore, inspection of the evolved automata showed that all solutions were fully generalizable. Thus, modular approach is able to evolve an NFA with lesser number of generations explored and fewer number of nodes evaluated in a significantly reduced amount of time.

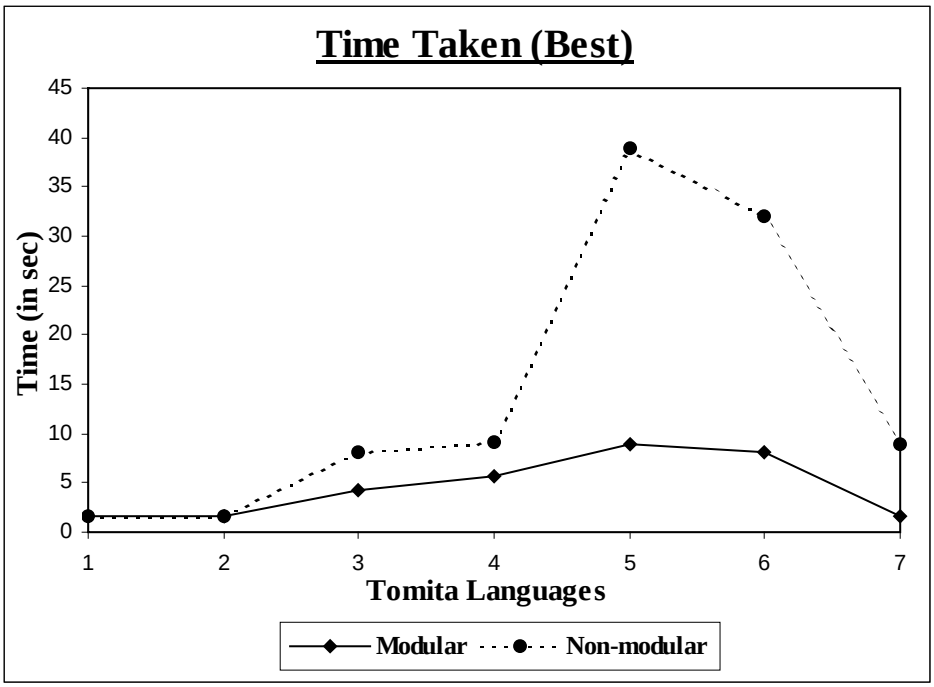


Figure 7.5: Time Taken for finding the Solution in the Best case for the Tomita Languages

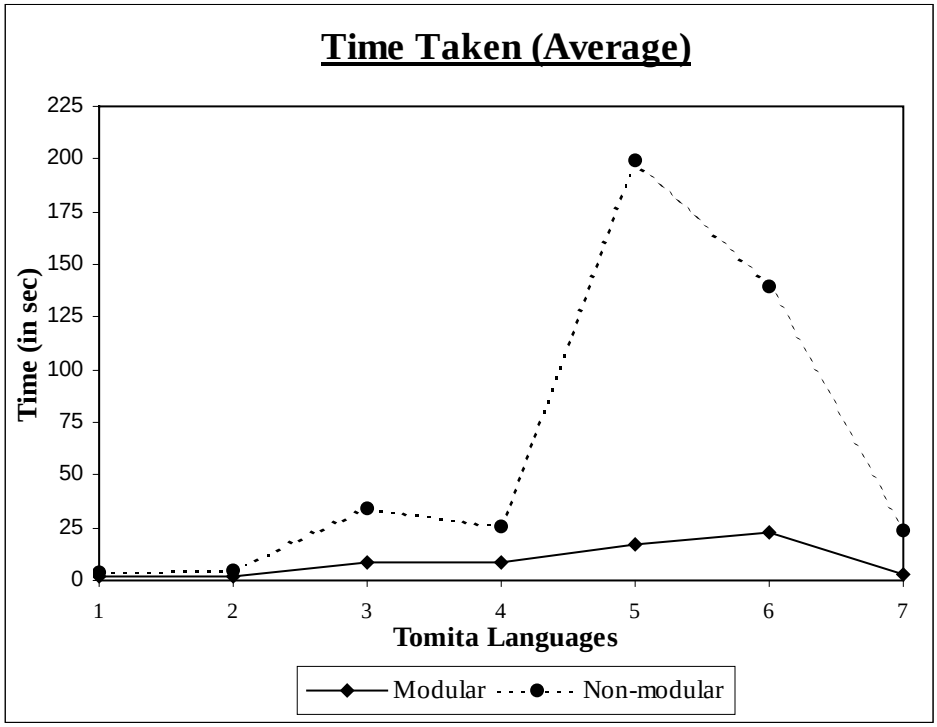


Figure 7.6: Time Taken for finding the Solution in the Average case for the Tomita Languages

CHAPTER-8
CONCLUSIONS AND FUTURE DIRECTIONS

8.1 Conclusions

In this dissertation, finite state automata for the Tomita Languages are evolved using the genetic programming paradigm. A modular architecture to evolve finite state automata is proposed. The method is simple to understand, easy to implement, and is potentially a powerful tool for evolving complex automata. The procedure is evaluated on the Tomita languages, which evolved an NFA consistent with the given training set.

The experimental results obtained indicate that the modular architecture is able to evolve finite state machines typically in lesser number of generations and many fewer nodes evaluations than previous non-modular approach. Also, the average time taken to learn a Tomita language with modular method is 9205.64 ms, which compares very favorably with the non-modular method where it is 52305.6 ms. Thus, it can be concluded that the proposed procedure is capable of successfully evolving modular finite state automata and that such modularity can result in a significantly increased rate of optimization.

8.2 Future Work

The present work can be extended in several directions. One possible improvement of the fitness function might be to rate smaller automata higher than larger ones, to stimulate the search towards a minimal, parsimonious solution. The next step consists of refining the design by using Automatically Defined Functions and Modular/Cassette Crossover to swap blocks that occur in the middle of a tree.

In the experiments, the wide degree of qualitative variations between runs indicates that, some times evolution quickly gets stuck at sub optimal solutions. Parallel subpopulations may help in this regard. Also, during evolution the Tomita languages can be further modularized depending upon the requirement.

The evolution of finite state machines using both the basic architecture and the modular architecture can be extended in several areas like, they can be used in the field of grammatical inference (GI), or can be used to encode computations, recognize events, or can be used to solve more real world applications like developing several kinds of software components, including the lexical analysis component of compilers and systems for verifying the correctness of circuits and protocols.

REFERENCES

[1] Aho, A. V., Sethi, R. and Ullman, J. D. "Compilers: Principles, Techniques, and Tools". Addison-Wesley, 1988.

[2] Andre and Teller. "A Study In Program Response And The Negative Effects Of Introns In Genetic Programming". Genetic Programming: Proceedings of the First Annual Conference, 12–20. Stanford University, CA, USA: MIT Press.1996

[3] Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D. "Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications". Morgan Kaufmann.1998.

[4] Brave. S. "Evolving Deterministic Finite Automata Using Cellular Encoding". In Genetic Programming, pages 39-44 1996.

[5] Chellapilla. K. and Czarnecki. D. "A Preliminary Investigation into Evolving Modular Finite State Machines". In Congress on Evolutionary Computation, pages 1349-1356, 1999.

[6] Chongstitvatana. P. and Aporntewan. C. "Improving Correctness of Finite-State Machine Synthesis from Multiple Partial Input/Output Sequences". In Workshop on Evolvable Hardware, 1999.

[7] Dupont. P. "Regular Grammatical Inference from positive and negative samples by genetic search: the GIG method". Proceedings of the International Colloquium on Grammatical Inference ICGI-94. Springer-Verlag Series in Artificial Intelligence, 1994

[8] Fogel, L.J., A.J. Owens, and M.J. Walsh. "Artificial Intelligence through Simulated Evolution". New York: John Wiley.1966

[9] Gruau, F., Whitley D., and Pyeatt, L. "A Comparison Between Cellular Encoding And Direct Encoding For Genetic Neural Networks". Proceedings of the First Genetic Programming Conference, 1996.

[10] Gruau, F. "Automatic Definition Of Modular Neural Networks". Adaptive Behaviour, vol. 3, 1994.

[11] Goldberg, D. "Genetic Algorithms in Search, Optimization, and Machine Learning". Reading:Addison-Wesley. 1989.

[12] Hornby, G. S. "Shortcomings with Tree-structured Edge Encodings for Neural Networks. Genetic and Evolutionary Computation Conference". Springer-Verlag, 2004.

[13] Jason W.H., and Lu Y.H. "Improving FSM evolution with progressive fitness functions". Proceedings of the 14th ACM Great Lakes symposium on VLSI. 2004

[14] John E. Hopcroft, Rajeev Motwani and J.D. Ullman: "Introduction to Automata Theory, Language, and Computation". Pearson Education. 2001.

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

[16] Lankhorst.M. "A Genetic Algorithm for Induction of Nondeterministic Pushdown Automata". Technical report, University of Groningen, Computer Science, CS-R 9502.

[17] Lucas. S. M. and Reynolds. T. J. "Learning DFA: Evolution Versus Evidence Driven State Merging". In Proceedings of the Congress on Evolutionary Computation, 2003.

[18] Luke, S. & Spector, L. "Evolving Graphs And Networks With Edge Encoding: Preliminary Report". Late-Breaking Papers of the Genetic Programming 1996 Conference. Available at www.cs.umd.edu/~seanl/papers/graph-paper.ps

[19] Nipaman. N. and Chongstivatana. P. "An Improved Genetic Algorithm for The Inference of Finite State Machine". In IEEE International Conference on Systems, Man and Cybernetics, pages 1-5, 2002.

[20] Pitt. L. "Inductive Inference, DFAs and Computational Complexity". Proceedings of the International Workshop on Analogical and Inductive Inference. Lecture Notes in Artificial Intelligence 397, Springer-Verlag. 1989.

[21] Svingen. B. "Learning Regular Languages Using Genetic Programming". In Proc. Genetic Programming. 1998.

[22] Thompson, K. "Regular Expression Search Algorithm". Communications of the ACM 11(6):419–422. 1968.

[23] "The GP Tutorial" http://www.geneticprogramming.com/Tutorial/index.html

[24] Tomita, M. "Dynamic Construction Of Finite Automata From Examples Using Hill-Climbing". In Proceedings of the Fourth Annual Conference of the Cognitive Science Society, 105–108. 1982.

[25] William M. Spears and Vic Anand "A Study Of Crossover Operators In Genetic Programming".

[26] Zomorodian, A. "Context-Free Language Induction By Evolution Of Deterministic Push-Down Automata Using Genetic Programming". In Koza, J. R., ed., Genetic Algorithms at Stanford, California, 1994.

APPENDIX A: OUTPUT

As this dissertation evolves the finite state automata for the Tomita Language Sets using both modular and non-modular architecture, a brief implementation overview for evolving Tomita 7, is provided here in this section.

A.1 Non-modular Evolution

The positive and negative training sets used for Tomita 7 are shown in Table A.1.

Tomita 7	Positive Set	Negative Set
0*1*0*1*	e, 1, 0, 10, 01, 11111, 000, 0101, 00110011, 0000100001111, 00, 00100, 011111011111	1010, 00110011000, 0101010101, 1011010, 10101, 010100, 101001, 100100110101

Table A.1: Positive and Negative Training Sets for Tomita 7

A.1.1 Example Run

An experimental run with the training set of Tomita 7, evolved the ideal individual in the 8th generation. The population size used was 500. Below, are the few sample chromosomes generated during the run and the best individual of the run. Also the NFA generated for the best individual has been shown.

GENERATION 0

==========

Individual : 0

Evaluated: true

Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8

TREE

(s e)

Individual : 1
Evaluated: true
Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8
TREE
(reverse (split e 0))

Individual : 2
Evaluated: true
Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8
TREE
(reverse (s (reverse 1)))

Individual : 3
Evaluated: true
Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8
TREE
(s (bud e 1))

Individual : 4
Evaluated: true
Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8
TREE
1

Individual : 5
Evaluated: true
Fitness: Raw=0.47619048 Adjusted=0.67741936 Hits=11
TREE
(a (a (loop (split (reverse 1) (double 0 e)) (s (split 1 e)))))

Individual : 6
Evaluated: true
Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8
TREE

(a 1)

Individual : 7
Evaluated: true
Fitness: Raw=0.3809524 Adjusted=0.72413796 Hits=13
TREE
(double (split (split (a 1) (a 0)) (loop (loop 1 1) (a 0))) (double (s (loop e 1))
(s (double 0 0))))

Individual : 8
Evaluated: true
Fitness: Raw=0.5714286 Adjusted=0.6363636 Hits=9
TREE
(split (split (a (s 0)) (bud (a e) (split 0 0))) (reverse (loop (split e 1)
(reverse 0))))

Individual : 9
Evaluated: true
Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8
TREE
(bud (bud (double (bud (split e e) (loop 1 0)) (loop 1 1)) (split (split (a 1) (reverse e)) (bud 0 (double 0 e)))) (split (split (split (double 1 e) (s e)) (loop (double 1 e) (bud 1 e))) 0))

…
…
…
…
…

GENERATION 7
==========

Individual : 0
Evaluated: true

Fitness: Raw=0.14285715 Adjusted=0.87499994 Hits=18

TREE

(bud (loop (s (bud (double 1 1) (loop (bud (a e) (a 1)) (bud 0 1)))) 0) (reverse (reverse (a (split (s (loop 0 1)) (loop (double (double (bud (double e 1) (a 1)) (loop (loop 1 0) (s (a e)))) (double (double (a 1) (split 0 0)) (bud (bud 1 1) (a 0)))) (loop 0 0)))))))

Individual : 1

Evaluated: true

Fitness: Raw=0.2857143 Adjusted=0.7777778 Hits=15

TREE

(split (double (s (split (split e 1) (double (double (bud e 1) (reverse e)) 1))) (loop 1 e)) (loop (double (s (split 1 (s (double (double (a 1) (split 0 0)) (bud (bud 1 1) (a 0)))))) (bud (s 1) (s (a (loop (a 1) (s 0)))))) (a (reverse (reverse 1)))))

Individual : 2

Evaluated: true

Fitness: Raw=0.3809524 Adjusted=0.72413796 Hits=13

TREE

(split (double (s (bud (reverse (double (split 0 1) (a (split (s (loop 0 1)) (loop (double (double (bud (double e 1) (a 1)) (loop (loop 1 0) (s (a e)))) (loop (bud (loop 0 0) (a 1)) (double (loop 0 e) (reverse (reverse 0)))))) (loop 0 0)))))) (reverse (reverse (double (split 0 1) (a (split (s (loop 0 1)) (loop (double (double (bud (double e 1) (a 1)) (loop (loop 1 0) (s (a e)))) (loop (s (double (a 1) (split 0 0))) (double (loop 0 e) (reverse (s e)))) (loop 0 0)))))))) (loop (bud (a e) (a 1)) (bud 0 1))) (loop (double (s (split (split (double (s 0) (double 0 e)) (loop (loop 0 0) (a 1))) (s (double (double (a 1) (split 0 0)) (bud (double 1 1) (loop 1 e)))))) (bud (s 1) (s 0))) (a (bud (s 1) (reverse 1)))))

Individual : 3

Evaluated: true

Fitness: Raw=0.61904764 Adjusted=0.61764705 Hits=8

TREE

(bud (loop (s (bud (double 1 1) (loop 1 e))) 0) (reverse (loop (s (bud (double 1 1) (loop 1 e))) 0)))

Individual : 4

Evaluated: true

Fitness: Raw=0.33333334 Adjusted=0.75 Hits=14

TREE

(bud (loop (s (bud (double 1 1) (loop (bud (a (loop (reverse 0) (s 0))) (a 1)) (bud 0 1)))) 0) (reverse (bud (bud 1 1) (a 0))))

…

…

…

…

…

Individual : 495

Evaluated: true

Fitness: Raw=0.04761905 Adjusted=0.9545454 Hits=20

TREE

(split (double (s (split (split (a (split (s (bud (loop (s (double (loop 0 e) (reverse (reverse 0)))) 0) (reverse (reverse (double (split 0 1) (a (split (s (loop 0 1)) (loop (s e) (loop 0 0)))))))) (loop (s e) (loop 0 0)))) 1) (double e 1))) (loop (bud (a e) (a 1)) (bud 0 1))) (loop (double (s (split e 1)) (bud (s 1) (s 0))) (a (bud (s 1) (reverse 1)))))

Individual : 496

Evaluated: true

Fitness: Raw=0.1904762 Adjusted=0.84000003 Hits=17

TREE

(split (s (a (loop (a 1) (s 0)))) (loop (split (bud (double e 1) (a 1)) (s 0)) (a 1)))

Individual : 497

Evaluated: true

Fitness: Raw=0.14285715 Adjusted=0.87499994 Hits=18

TREE

(bud (loop (s (bud (double 1 1) (bud (double e 1) (a 1)))) 0) (reverse (reverse (double (split 0 1) (a (split (s (loop 0 1)) (loop (double (double (s (s (a (loop (bud (bud 1 1) (a 0)) (s 0))))) (loop (loop 1 0) (s (a e)))) (loop (bud (loop 0 0) (a 1)) (double (loop 0 e) (reverse (s e))))) (loop 0 0))))))))

Individual : 498
Evaluated: true
Fitness: Raw=0.0952381 Adjusted=0.9130435 Hits=19
TREE
(split (s (a (loop (a 1) (s 0)))) (loop (split (s (s (a (loop (s (a (loop (a 1) (s 0)))) (s 0))))) (s 0)) (a 1)))

Individual : 499
Evaluated: true
Fitness: Raw=0.1904762 Adjusted=0.84000003 Hits=17
TREE
(bud (a (split (split (s (bud (bud 1 1) (a 0))) (a (a (split (s (loop 0 1)) (loop (s e) (loop 0 0)))))) (loop (s e) (loop (split 0 0) 0)))) (a (loop 1 e)))

Best Individual of Generation 7:
==========
Evaluated: true
Fitness: Raw=0.0 Adjusted=1.0 Hits=21
TREE
(double (loop (a e) (s e)) (double (a (bud (a (reverse e)) (loop 1 1))) (reverse (bud (s 0) (bud (a (split (s (loop 0 1)) (loop (s (double 1 0)) (a (loop (a e) (s 0)))))) (a (double (loop 1 1) (split 0 (s 0))))))))

Final Statistics
==========

Total Individuals Evaluated: 4000

Best Individual of Run:

Evaluated: true

Fitness: Raw=0.0 Adjusted=1.0 Hits=21

TREE

(double (loop (a e) (s e)) (double (a (bud (a (reverse e)) (loop 1 1))) (reverse (bud (s 0) (bud (a (split (s (loop 0 1)) (loop (s (double 1 0)) (a (loop (a e) (s 0)))))) (a (double (loop 1 1) (split 0 (s 0)))))))))

Best Individual's Generalization Score...

Pos: 1940/1940 Neg: 30827/30827

(pos+neg)/(allpos+allneg): 1.0

Best Individual's NFA

==========

States			**Transitions**
0	S	=>	(0:5) (e:1)
1	SA	=>	(0:0) (1:2) (e:0,1)
2		=>	(1:2)
3	SA	=>	(0:3,6) (1:4) (e:3)
4	SA	=>	(1:4)
5	S	=>	(0:3) (1:3,5)
6		=>	(0:4)

A.2 Modular Evolution

The modular evolution of Tomita 7 involves the decomposition of the expression into two smaller sub-expressions, and then evolving the automata for both the sub-expressions. The sub automata generated are further combined to give the resultant automata for the language.

The decomposition of Tomita 7 (0*1*0*1*) into two sub expressions can be done as (0*1*) and (0*1*), where, only the automata for sub expression 1 is evolved and the same results can be used for the sub expression 2, thereby reducing the time as well as the resources for evolving sub expression 2. The encoding tree for the sub expression 1 is concatenated with itself using the property (split X (split ε X)) (where X is the encoding tree for the sub expression 1), to get the complete automata for the Tomita 7 language.

The positive and negative training sets used for the sub expression 1 of the Tomita 7 are shown in Table A.2.

Tomita 7	Positive Set	Negative Set
0*1*	e, 0, 00, 1, 11, 01, 0011, 0001, 0111, 000000011111, 000011111111	10, 010, 100, 101, 00110 111100000, 10000 0001111000

Table A.2: Positive and Negative Training Sets for Sub Expression 1 of Tomita 7

A.2.1 Example Run

An experimental run with the training set for the sub expression 1 of the Tomita 7, evolved the ideal individual in the 2nd generation. The population size used was 500. Below, are the few sample chromosomes generated during the run and the best individual of the run. Also the NFA generated for the best individual has been shown.

GENERATION 0

==========

Individual : 0

Evaluated: true

Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7

TREE

(a (bud (a (double (double 1 e) (reverse 0))) (double (loop (double 0 1) (reverse e)) (double (bud 0 e) (loop 1 e)))))

Individual : 1
Evaluated: true
Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7
TREE
1

Individual : 2
Evaluated: true
Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7
TREE
(a (double e (a (split (a 0) e))))

Individual : 3
Evaluated: true
Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7
TREE
(bud (a (reverse e)) (a (split 1 0)))

Individual : 4
Evaluated: true
Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7
TREE
(bud e e)

Individual : 5
Evaluated: true
Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7
TREE
(loop (bud (double 1 (split 1 1)) e) e)

…

…

…

…

…

GENERATION 1

==========

Individual : 0

Evaluated: true

Fitness: Raw=0.44444445 Adjusted=0.6923077 Hits=10

TREE

(s (bud (double (bud e (double 1 0)) 1) (s

(double (a 1) (loop 1 1)))))

Individual : 1

Evaluated: true

Fitness: Raw=0.3888889 Adjusted=0.72 Hits=11

TREE

(a (s (loop (reverse (reverse e)) (a (s (split 0 (loop 1 0)))))))

Individual : 2

Evaluated: true

Fitness: Raw=0.5555556 Adjusted=0.64285713 Hits=8

TREE

(s (a 0))

Individual : 3

Evaluated: true

Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7

TREE

(s (s (loop e e)))

…

…

…

…

…

Individual : 497

Evaluated: true

Fitness: Raw=0.44444445 Adjusted=0.6923077 Hits=10

TREE

(s (split (a (s (double 1 1))) (bud (bud (bud (loop (bud (reverse (s (double 1 1))) (double (a (split 1 e)) (a (loop 1 e)))) (a (s (a (loop e e))))) 0) (a 0)) (a (reverse e)))))

Individual : 498

Evaluated: true

Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7

TREE

e

Individual : 499

Evaluated: true

Fitness: Raw=0.6111111 Adjusted=0.62068963 Hits=7

TREE

(s 1)

Best Individual of Generation:

==========

Evaluated: true

Fitness: Raw=0.0 Adjusted=1.0 Hits=18

TREE

(loop (bud 0 (a (loop 1 1))) (s (double (reverse (a 0)) (loop (a 0) (double e e)))))

Final Statistics

==========

Total Individuals Evaluated: 1000

Best Individual of Run:

Evaluated: true

Fitness: Raw=0.0 Adjusted=1.0 Hits=18

TREE

(loop (bud 0 (a (loop 1 1))) (s (double (reverse (a 0)) (loop (a 0) (double e e)))))

Best Individual's Generalization Score...

Pos: 15/15 Neg: 32752/32752

(pos+neg)/(allpos+allneg): 1.0

Best Individual's NFA

==========

States			**Transitions**
0		=>	(0: 1)
1	SA	=>	(0: 1,1) (1: 2) (e: 1,1)
2	A	=>	(1: 2)

A.2.2 Combining Sub Automata

After the automata for each of the sub expressions are evolved, the encoding tree of the sub automata's are joined using the design as explained in Section 6.2, depending upon the joining conditions. Here, sub automata 1 is concatenated with itself using the condition (split X (split ε X)).

The resulting encoding tree for the Tomita 7 using the above procedure is:

(split (loop (bud 0 (a (loop 1 1))) (s (double (reverse (a 0)) (loop (a 0) (double e e))))) (split e (loop (bud 0 (a (loop 1 1))) (s (double (reverse (a 0)) (loop (a 0) (double e e)))))))

And, the NFA generated for the above encoding is:

Complete NFA for Tomita 7 using modular approach

==========

States			**Transitions**
0		=>	(0: 1)
1	SA	=>	(0: 1,1) (1: 2) (e: 1,1)
2	A	=>	(1: 2)
3		=>	(0: 4)
4	SA	=>	(0: 4,4) (1: 5) (e: 4,4)
5	A	=>	(1: 5)

APPENDIX B: SOURCE CODE

package fsa

Breeder.java

```
package fsa;

public  class Breeder
   {

   public abstract Population breedPopulation(final EvolutionState state) throws
CloneNotSupportedException;

   }
```

BreedingSource.java

```
package fsa;
import fsa.util.*;


public abstract class BreedingSource implements Prototype, RandomChoiceChooser
   {
   public static final String P_PROB = "prob";
   public static final float NO_PROBABILITY = -1.0f;
   public static final int UNUSED = -1;
   public static final int CHECKBOUNDARY = 8;
   public static final int DEFAULT_PRODUCED = 1;

   public float probability;

   public void setup(final EvolutionState state, final Parameter base)
         {

         Parameter def = defaultBase();

         if (!state.parameters.exists(base.push(P_PROB),def.push(P_PROB)))
            probability = NO_PROBABILITY;
         else
            {
            probability = state.parameters.getFloat(base.push(P_PROB),def.push(P_PROB),0.0);
```

```
            if (probability<0.0) state.output.error("Breeding Source's probability must be a
            floating point value >= 0.0, or empty, which represents NO_PROBABILITY.",
            base.push(P_PROB),def.push(P_PROB));
             }
        }

    public final float getProbability(final Object obj)
        {
        return ((BreedingSource)obj).probability;
        }

    public final void setProbability(final Object obj, final float prob)
        {
        ((BreedingSource)obj).probability = prob;
        }

    public static int pickRandom(final BreedingSource[] sources,final float prob)
        {
        return RandomChoice.pickFromDistribution(sources,sources[0],
                                                 prob,CHECKBOUNDARY);
        }

    public static void setupProbabilities(final BreedingSource[] sources)
        {
        RandomChoice.organizeDistribution(sources,sources[0],true);
        }

    public abstract int typicalIndsProduced();

    public abstract boolean produces(final EvolutionState state,  final Population newpop,
                                     final int subpopulation,  int thread);

    public abstract void prepareToProduce(final EvolutionState state,  final int subpopulation,
                                          final int thread);

    public abstract void finishProducing(final EvolutionState s,  final int subpopulation,
                                         final int thread);

    public abstract int produce(final int min, final int max, final int start, final int subpopulation,
                                final Individual[] inds, final EvolutionState state,
                                final int thread) throws CloneNotSupportedException;

    public Object protoClone() throws CloneNotSupportedException
        {
        return super.clone();
        }

    public final Object protoCloneSimple()
        {
        try { return protoClone(); }
```

```
        catch (CloneNotSupportedException e)
            { throw new InternalError(); }
        }

    public abstract void preparePipeline(final Object hook);
    }
```

Fitness.java

```
package fsa;
import java.io.*;

public interface Fitness extends Prototype
    {
    public static final String P_FITNESS = "fit";

    public abstract float fitness();

    public abstract boolean isIdealFitness();

    public abstract boolean equivalentTo(Fitness _fitness);

    public abstract boolean betterThan(Fitness _fitness);

    public abstract void printFitness(EvolutionState state, int log,
                                      int verbosity);

    public abstract void printFitness(final EvolutionState state,
                                      final PrintWriter writer);

    public abstract void readFitness(final EvolutionState state,
                                     final LineNumberReader reader)
        throws IOException, CloneNotSupportedException;
    }
```

FsaEvolution.java

```
package fsa;
import java.io.IOException;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.OptionalDataException;
import java.io.PrintWriter;
```

```
public class FsaEvolution
   {

  public  static long inittime;
  public  static long exittime;
  public  static long tottime;

   public static final String A_FILE = "-file";

   public static final String P_EVALTHREADS = "evalthreads";

   public static final String P_BREEDTHREADS = "breedthreads";

   public static final String P_SEED = "seed";

   public static final String V_SEED_TIME = "time";

   public static final String P_STATE = "state";

   public static void main(String[ ] args)
          {

          EvolutionState state=null;
          ParameterDatabase parameters=null;
          Output output;
          MersenneTwisterFast[] random;
          int[] seeds;
          int breedthreads = 1;
          int evalthreads = 1;
          int x;


          if (state==null)
             {

             for(x=0;x<args.length-1;x++)
                    if (args[x].equals(A_FILE))
                       {
                       try
                              {
                              parameters=new ParameterDatabase(
                                 new File(new File(args[x+1]).getAbsolutePath()),
                                 args);
                              break;
                              }
                       catch(FileNotFoundException e)
                              { Output.initialError(
                                 "A File Not Found Exception was generated upon" +
                                 "reading the parameter file \"" + args[x+1] +
                                 "\".\nHere it is:\n" + e); }
```

```
            catch(IOException e)
                  { Output.initialError(
                     "An IO Exception was generated upon reading the" +
                     "parameter file \"" + args[x+1] +
                     "\".\nHere it is:\n" + e); }
            }
if (parameters==null)
      Output.initialError(
         "No parameter file was specified." );

breedthreads = parameters.getInt(
      new Parameter(P_BREEDTHREADS),null,1);

if (breedthreads < 1)
      Output.initialError("Number of breeding threads should be an integer >0.",
                          new Parameter(P_BREEDTHREADS));

evalthreads = parameters.getInt(
      new Parameter(P_EVALTHREADS),null,1);

if (evalthreads < 1)
      Output.initialError("Number of eval threads should be an integer >0.",
                          new Parameter(P_EVALTHREADS));

random = new MersenneTwisterFast[breedthreads > evalthreads ?
                                 breedthreads : evalthreads];
seeds = new int[breedthreads > evalthreads ?
                breedthreads : evalthreads];

int time = (int)System.currentTimeMillis();
String seed_message = "Seed: ";

for (x=0;x<random.length;x++)
      {
      int seed = 1;
      String tmp_s = parameters.getString(
         new Parameter(P_SEED).push(""+x),null);
      if (tmp_s==null)
         {
         Output.initialError("Seed should be an integer.",
                             new Parameter(P_SEED).push(""+x));

         }
      else if (tmp_s.equalsIgnoreCase(V_SEED_TIME))
         {
         seed = time++;
         }
      else
          {
         try
```

```
                {
                seed = parameters.getInt(new Parameter(P_SEED).push(""+x),null);
                }
            catch (NumberFormatException e)
                {
                Output.initialError("Invalid Seed Value (must be an integer):\n" + e);
                }
        seed_message = seed_message + seed + " ";
        }

        seeds[x] = seed;
        }

for (x=0;x<random.length;x++)
        {
        for (int y=x+1;y<random.length;y++)
        if (seeds[x]==seeds[y])
                {
                Output.initialError(P_SEED+"."+x+" ("+seeds[x]+") and
                "+P_SEED+"."+y+" ("+seeds[y]+") should not be the same seed.");
                }
        random[x] = new MersenneTwisterFast(seeds[x]);
        }


state = (EvolutionState)
        parameters.getInstanceForParameter(new Parameter(P_STATE),null,
                                                EvolutionState.class);
state.parameters = parameters;
state.random = random;
state.output = output;
state.evalthreads = evalthreads;
state.breedthreads = breedthreads;

output.systemMessage("Threads:  breed/" + breedthreads + " eval/" + evalthreads);
output.systemMessage(seed_message);

try
        {
        inittime= System.currentTimeMillis();

        state.run(EvolutionState.C_STARTED_FRESH);

        System.out.println("\n time=" + inittime);
        extime=System.currentTimeMillis();
        System.out.println("\n time=" + extime);
        tottime= System.currentTimeMillis()-inittime;
        System.out.println("\n time=" + tottime);
        }
catch (IOException e)
```

```
                {
                Output.initialError(
                   "An IO Exception  generated " + e);
                }

            output.flush();

            PrintWriter pw = new PrintWriter(System.err);

            pw.flush();

            System.err.flush();
            System.out.flush();

         output.close();
         }
      }

}
```

Group.java

```
package fsa;
import fsa.util.Parameter;

public interface Group extends Setup, Cloneable
   {
   public Group emptyClone() throws CloneNotSupportedException;
   }
```

Individual.java

```
package fsa;
import java.io.*;

public abstract class Individual implements Prototype
   {
   public Fitness fitness;

   public Species species;

   public boolean evaluated;

   public Individual deepClone()
```

```
    {
    return (Individual) protoCloneSimple();
    }


  public Object protoClone() throws CloneNotSupportedException
        {
        Individual myobj = (Individual) (super.clone());

        if (myobj.fitness!=null) myobj.fitness = (Fitness)(fitness.protoClone());
        return myobj;
        }

  public final Object protoCloneSimple()
        {
        try { return protoClone(); }
        catch (CloneNotSupportedException e)
           { throw new InternalError(); }
        }

  public abstract void setup(final EvolutionState state, final Parameter base);


  public abstract void printIndividual(final EvolutionState state,
                                       final int log,
                                       final int verbosity);

  public abstract void printIndividual(final EvolutionState state,
                                       final PrintWriter writer);

  public abstract void readIndividual(final EvolutionState state, final LineNumberReader
reader)
        throws IOException, CloneNotSupportedException;


  public long size() { return 0; }
  }
```

Initializer.java

```
package fsa;
import fsa.util.Parameter;


public abstract class Initializer implements Singleton
  {

  public static final String P_POP = "pop";
```

```
   public abstract Population initialPopulation(final EvolutionState state);
   }
```

Population.java

```
package fsa;
import fsa.util.Parameter;


public class Population implements Group
   {
   public Subpopulation[] subpops;
   public static final String P_SIZE = "subpops";
   public static final String P_SUBPOP = "subpop";


   public Group emptyClone()
         {
         try
            {
            Population p = (Population)clone();
            p.subpops = new Subpopulation[subpops.length];
            for(int x=0;x<subpops.length;x++)
                  p.subpops[x] = (Subpopulation)(subpops[x].emptyClone());
            return p;
            }
         catch (CloneNotSupportedException e) { throw new InternalError(); }
         }

   public void setup(final EvolutionState state, final Parameter base)
         {

         Parameter p;

         p = base.push(P_SIZE);
         int size = state.parameters.getInt(p,null,1);
         if (size==0)
            state.output.fatal("Population size must be >0.\n",base.push(P_SIZE));
         subpops = new Subpopulation[size];

         for (int x=0;x<size;x++)
            {
            p = base.push(P_SUBPOP).push(""+x);
           subpops[x] = (Subpopulation)(state.parameters.getInstanceForParameterEq
           (p,null,Subpopulation.class));
            subpops[x].setup(state,p);
            }
```

```
        }

    public void populate(EvolutionState state)
        {
        // let's populate!
        for(int x=0;x<subpops.length;x++)
            subpops[x].populate(state);
        }

    }
```

Problem.java

```
package fsa;

public abstract class Problem implements Prototype
    {
    public static final String P_PROBLEM = "problem";

    public Parameter defaultBase()
        {
        return new Parameter(P_PROBLEM);
        }

    public void setup(final EvolutionState state, final Parameter base)
        { }

    public Object protoClone() throws CloneNotSupportedException
        {
        return clone();
        }

    public Object protoCloneSimple()
        {
        try { return protoClone(); }
        catch (CloneNotSupportedException e) {  }
        return null;
        }
    }
```

Prototype.java

```
package fsa;

public interface Prototype extends Cloneable
        {
```

```
    public Object protoClone() throws CloneNotSupportedException;

    public Object protoCloneSimple();

    public void setup(final EvolutionState state, final Parameter base);

    public Parameter defaultBase();

   }
```

SelectionMethod.java

```
package fsa


public abstract class SelectionMethod
   {
   public static final int INDS_PRODUCED = 1;

   public int typicalIndsProduced() { return INDS_PRODUCED; }

   public abstract int produce(final int subpopulation, final EvolutionState state,   final int
thread);

public boolean produces(final EvolutionState state, final Population newpop,  final int
subpopulation,
                        final int thread)
        {
        return true;
        }


public void prepareToProduce(final EvolutionState s, final int subpopulation,  final int thread)
        { return; }


public void finishProducing(final EvolutionState s,         final int subpopulation, final int
thread)
        { return; }

public int produce(final int min, final int max, final int start, final int subpopulation, final
Individual[ ]  inds,  final EvolutionState state,  final int thread) throws
CloneNotSupportedException
        {
        int n=INDS_PRODUCED;
        if (n<min) n = min;
        if (n>max) n = max;
```

```
        for(int q=0;q<n;q++)
           inds[start+q] = state.population.subpops[subpopulation].
                 individuals[produce(subpopulation,state,thread)];
        return n;
        }

   public void preparePipeline(Object hook)
        {

        }
   }
```

Setup.java

```
package fsa;
import java.io.Serializable;


public interface Setup extends Serializable
   {

   public void setup(final EvolutionState state, final Parameter base);
   }
```

package fsa.edge.func

Accept.java

```
package fsa.edge.func;
import fsa.*;
import fsa.edge.*;

public class Accept extends GPNode
   {
   public String toString() { return "a"; }

  public void checkConstraints(final EvolutionState state, final int tree,     final GPIndividual
  typicalIndividual,  final Parameter individualBase)
        {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=1)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
        }
```

```
   public void eval(final EvolutionState state, final int thread,  final GPData input,
                    final ADFStack stack, final GPIndividual individual,  final Problem
                    problem)
        {
        int edge = ((EdgeData)(input)).edge;
        Edge prob = (Edge)problem;

        prob.accept[prob.to[edge]] = true;

        children[0].eval(state,thread,input,stack,individual,problem);
        }
   }
```

Bud.java

```
package fsa.edge.func;
import fsa.*;
import fsa.edge.*;


public class Bud extends GPNode
   {
   public String toString() { return "bud"; }

   public void checkConstraints(final EvolutionState state,  final int tree,
                                final GPIndividual typicalIndividual,  final Parameter
individualBase)
        {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=2)
        state.output.error("Incorrect number of children for node " + toStringForError() + " at "
        +  individualBase);
        }

   public void eval (final EvolutionState state,  final int thread,  final GPData input,
                    final ADFStack stack,  final GPIndividual individual,  final Problem
                  problem)
        {
        int edge = ((EdgeData)(input)).edge;
        Edge prob = (Edge)problem;

        if (prob.from.length==prob.numEdges)
           {
           int[] from_ = new int[prob.numEdges*2];
           int[] to_ = new int[prob.numEdges*2];
           int[] reading_ = new int[prob.numEdges*2];
           System.arraycopy(prob.from,0,from_,0,prob.from.length);
           System.arraycopy(prob.to,0,to_,0,prob.to.length);
```

```
            System.arraycopy(prob.reading,0,reading_,0,prob.reading.length);
            prob.from = from_;
            prob.to = to_;
            prob.reading = reading_;
            }

        if (prob.start.length==prob.numNodes)
            {
            boolean[] start_ = new boolean[prob.numNodes*2];
            boolean[] accept_ = new boolean[prob.numNodes*2];
            System.arraycopy(prob.start,0,start_,0,prob.start.length);
            System.arraycopy(prob.accept,0,accept_,0,prob.accept.length);
            prob.start = start_;
            prob.accept = accept_;
            }

        int newedge = prob.numEdges;
        prob.numEdges++;
        int newnode = prob.numNodes;
        prob.numNodes++;

        prob.accept[newnode] = false;
        prob.start[newnode] = false;

        prob.from[newedge] = prob.to[edge];
        prob.to[newedge] = newnode;
        prob.reading[newedge] = prob.reading[edge];


        children[0].eval(state,thread,input,stack,individual,problem);

        ((EdgeData)(input)).edge = newedge;

        children[1].eval(state,thread,input,stack,individual,problem);
        }
    }
```

Double.java

```
package fsa.edge.func;
import fsa.*;
import fsa.edge.*;

public class Double extends GPNode
    {
    public String toString() { return "double"; }

    public void checkConstraints(final EvolutionState state,  final int tree,
            final GPIndividual typicalIndividual,  final Parameter individualBase)
```

```
        {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=2)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
        }

public void eval(final EvolutionState state,  final int thread,  final GPData input,  final
ADFStack stack,   final GPIndividual individual,  final Problem problem)
        {
        int edge = ((EdgeData)(input)).edge;
        Edge prob = (Edge)problem;

        if (prob.from.length==prob.numEdges)
           {
           int[] from_ = new int[prob.numEdges*2];
           int[] to_ = new int[prob.numEdges*2];
           int[] reading_ = new int[prob.numEdges*2];
           System.arraycopy(prob.from,0,from_,0,prob.from.length);
           System.arraycopy(prob.to,0,to_,0,prob.to.length);
           System.arraycopy(prob.reading,0,reading_,0,prob.reading.length);
           prob.from = from_;
           prob.to = to_;
           prob.reading = reading_;
           }

        int newedge = prob.numEdges;
        prob.numEdges++;
        prob.from[newedge] = prob.from[edge];
        prob.to[newedge] = prob.to[edge];
        prob.reading[newedge] = prob.reading[edge];

        children[0].eval(state,thread,input,stack,individual,problem);

        ((EdgeData)(input)).edge = newedge;

        children[1].eval(state,thread,input,stack,individual,problem);
        }
   }
```

Epsilon.java

```
package fsa.edge.func;
import fsa.*;
import fsa.edge.*;

public class Epsilon extends GPNode
   {
   public String toString() { return "e"; }
```

```
   public void checkConstraints(final EvolutionState state,  final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
         {
         super.checkConstraints(state,tree,typicalIndividual,individualBase);
         if (children.length!=0)
            state.output.error("Incorrect number of children for node " +
                               toStringForError() + " at " +
                               individualBase);
         }

   public void eval(final EvolutionState state,  final int thread,  final GPData input,
                    final ADFStack stack,  final GPIndividual individual,
                    final Problem problem)
         {
         int edge = ((EdgeData)(input)).edge;
         Edge prob = (Edge)problem;

         prob.reading[edge] = Edge.EPSILON;
         }
   }
```

Loop.java

```
package fsa.edge.func;
import fsa.*;
import fsa.app.edge.*;

public class Loop extends GPNode
   {
   public String toString() { return "loop"; }

   public void checkConstraints(final EvolutionState state,  final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
         {
         super.checkConstraints(state,tree,typicalIndividual,individualBase);
         if (children.length!=2)
            state.output.error("Incorrect number of children for node " +
                               toStringForError() + " at " +
                               individualBase);
         }

   public void eval(final EvolutionState state,  final int thread,  final GPData input,
                    final ADFStack stack,  final GPIndividual individual,
                    final Problem problem)
         {
         int edge = ((EdgeData)(input)).edge;
```

```
        Edge prob = (Edge)problem;

        if (prob.from.length==prob.numEdges)
            {
            int[] from_ = new int[prob.numEdges*2];
            int[] to_ = new int[prob.numEdges*2];
            int[] reading_ = new int[prob.numEdges*2];
            System.arraycopy(prob.from,0,from_,0,prob.from.length);
            System.arraycopy(prob.to,0,to_,0,prob.to.length);
            System.arraycopy(prob.reading,0,reading_,0,prob.reading.length);
            prob.from = from_;
            prob.to = to_;
            prob.reading = reading_;
            }

        int newedge = prob.numEdges;
        prob.numEdges++;
        prob.from[newedge] = prob.to[edge];
        prob.to[newedge] = prob.to[edge];  // same
        prob.reading[newedge] = prob.reading[edge];

        children[0].eval(state,thread,input,stack,individual,problem);

        ((EdgeData)(input)).edge = newedge;

        children[1].eval(state,thread,input,stack,individual,problem);
        }
    }
```

One.java

```
package fsa.edge.func;
import fsa.*;
import fsa.edge.*;

public class One extends GPNode
    {
    public String toString() { return "1"; }

    public void checkConstraints(final EvolutionState state,  final int tree,
                                 final GPIndividual typicalIndividual,
                                 final Parameter individualBase)
        {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=0)
            state.output.error("Incorrect number of children for node " +
                               toStringForError() + " at " +
                               individualBase);
        }
```

```
   public void eval(final EvolutionState state, final int thread,  final GPData input,
                    final ADFStack stack,  final GPIndividual individual,
                    final Problem problem)
        {
        int edge = ((EdgeData)(input)).edge;
        Edge prob = (Edge)problem;
        prob.reading[edge] = Edge.READING1;
        }
   }
```

Reverse.java

```
package fsa.edge.func;
import fsa.*;
import fsa.edge.*;

public class Reverse extends GPNode
   {
   public String toString() { return "reverse"; }

   public void checkConstraints(final EvolutionState state,  final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
        {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=1)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
        }

   public void eval(final EvolutionState state,  final int thread,  final GPData input,
                    final ADFStack stack,  final GPIndividual individual,
                    final Problem problem)
        {
        int edge = ((EdgeData)(input)).edge;
        Edge prob = (Edge)problem;

        int swap = prob.from[edge];
        prob.from[edge] = prob.to[edge];
        prob.to[edge] = swap;

        children[0].eval(state,thread,input,stack,individual,problem);
        }
   }
```

Split.java

```
package fsa.edge.func;
import fsa.*;
import fsa.app.edge.*;

public class Split extends GPNode
   {
   public String toString() { return "split"; }

   public void checkConstraints(final EvolutionState state,  final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
         {
         super.checkConstraints(state,tree,typicalIndividual,individualBase);
         if (children.length!=2)
            state.output.error("Incorrect number of children for node " +
                               toStringForError() + " at " +
                               individualBase);
         }

   public void eval(final EvolutionState state,  final int thread,  final GPData input,
                    final ADFStack stack,  final GPIndividual individual,
                    final Problem problem)
         {
         int edge = ((EdgeData)(input)).edge;
         Edge prob = (Edge)problem;

         if (prob.from.length==prob.numEdges)
            {
            int[] from_ = new int[prob.numEdges*2];
            int[] to_ = new int[prob.numEdges*2];
            int[] reading_ = new int[prob.numEdges*2];
            System.arraycopy(prob.from,0,from_,0,prob.from.length);
            System.arraycopy(prob.to,0,to_,0,prob.to.length);
            System.arraycopy(prob.reading,0,reading_,0,prob.reading.length);
            prob.from = from_;
            prob.to = to_;
            prob.reading = reading_;
            }

         if (prob.start.length==prob.numNodes)
            {
            boolean[] start_ = new boolean[prob.numNodes*2];
            boolean[] accept_ = new boolean[prob.numNodes*2];
            System.arraycopy(prob.start,0,start_,0,prob.start.length);
            System.arraycopy(prob.accept,0,accept_,0,prob.accept.length);
            prob.start = start_;
            prob.accept = accept_;
            }
```

```
        int newedge = prob.numEdges;
        prob.numEdges++;
        int newnode = prob.numNodes;
        prob.numNodes++;

        // set up new node
        prob.accept[newnode] = false;
        prob.start[newnode] = false;

        // set up new edge
        prob.from[newedge] = newnode;
        prob.to[newedge] = prob.to[edge];
        prob.reading[newedge] = prob.reading[edge];
        // modify old edge
        prob.to[edge] = newnode;

        // pass the original edge down the left child

        children[0].eval(state,thread,input,stack,individual,problem);

        // reset input for right child
        ((EdgeData)(input)).edge = newedge;

        // pass the new edge down the right child

        children[1].eval(state,thread,input,stack,individual,problem);
        }
    }
```

Start.java

```
package fsa.edge.func;
import fsa.*;
import fsa.app.edge.*;

public class Start extends GPNode
    {
    public String toString() { return "s"; }

    public void checkConstraints(final EvolutionState state,  final int tree,
                                 final GPIndividual typicalIndividual,
                                 final Parameter individualBase)
        {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=1)
            state.output.error("Incorrect number of children for node " +
                               toStringForError() + " at " +
                               individualBase);
        }
```

```
   public void eval(final EvolutionState state, final int thread, final GPData input,
                    final ADFStack stack, final GPIndividual individual,
                    final Problem problem)
        {
        int edge = ((EdgeData)(input)).edge;
        Edge prob = (Edge)problem;

        prob.start[prob.to[edge]] = true;

        // pass the edge down

        children[0].eval(state,thread,input,stack,individual,problem);
        }
   }
```

Zero.java

```
package fsa.edge.func;
import fsa.*;
import fsa.app.edge.*;

public class Zero extends GPNode
   {
   public String toString() { return "0"; }

   public void checkConstraints(final EvolutionState state,  final int tree,
                                final GPIndividual typicalIndividual, final Parameter
individualBase)
        {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=0)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
        }

   public void eval(final EvolutionState state, final int thread, final GPData input,
                    final ADFStack stack, final GPIndividual individual,
                    final Problem problem)
        {
        int edge = ((EdgeData)(input)).edge;
        Edge prob = (Edge)problem;

        prob.reading[edge] = Edge.READING0;
        }
   }
```

package fsa.edge

EdgeData.java

```
package fsa.edge;
import fsa.util.*;
import fsa.*;

public class EdgeData extends GPData
   {

   public int edge;

   public GPData copyTo(final GPData gpd)
          {
          ((EdgeData)gpd).edge = edge; return gpd;
          }
   }
```

EdgeStatistics.java

```
package fsa.app.edge;
import fsa.*;
import fsa.util.*;

public class EdgeStatistics extends KozaStatistics
   {
   public void finalStatistics(final EvolutionState state, final int result)
          {
          super.finalStatistics(state,result);


          ((SimpleProblemForm)(state.evaluator.p_problem.protoCloneSimple())).describe(
             best_of_run[0], state, 0, statisticslog,Output.V_NO_GENERAL);
          }

   }
```

Edge.java

```
package fsa.edge;
import java.io.*;
import java.util.*;
import fsa.*;
import fsa.gp.*;
```

```
public class Edge extends GPProblem implements SimpleProblemForm
    {
    public static final String P_DATA = "data";
    public static final String P_GENERALIZE = "generalize";
    public static final String P_ALLPOS = "allpos";
    public static final String P_ALLNEG = "allneg";
    public static final String P_TESTPOS = "testpos";
    public static final String P_TESTNEG = "testneg";
    public static final String P_MAXTEST = "maxtest";

    public static final int MIN_ARRAY_SIZE = 64;

    public static final int BAD = 0;
    public static final int READING0 = 1;
    public static final int READING1 = 2;
    public static final int EPSILON = 3;

    public EdgeData input;

    public boolean[] start;
    public boolean[] accept;
    public int numNodes;
    public int[] from;
    public int[] to;
    public int[] reading;
    public int numEdges;

    public int[][] reading1;
    public int[] reading1_l;
    public int[][] reading0;
    public int[] reading0_l;
    public int[][] epsilon;
    public int[] epsilon_l;

    public boolean[][] posT;
    public boolean[][] negT;
    public boolean[][] posA;
    public boolean[][] negA;

    public boolean[] state1;
    public boolean[] state2;

    public boolean generalize;

    public Object protoClone() throws CloneNotSupportedException
        {

        Edge myobj = (Edge) (super.protoClone());
        myobj.input = (EdgeData)(input.protoClone());
```

```
        return myobj;
        }

    public static String fill(int num, char c)
        {
        char[] buf = new char[num];
        for(int x=0;x<num;x++) buf[x]=c;
        return new String(buf);
        }


    public String printCurrentNFA()
        {
        int strsize = String.valueOf(numNodes).length();
        String str = "";
        for(int x=0;x<numNodes;x++)
           {
           str += justify(String.valueOf(x),strsize,J_RIGHT) + " " +
                  (start[x] ? "S" : " ") + (accept[x] ? "A" : " ") +
                  "=> ";

           if (reading0_l[x]>0)
                  {
                  str += "(0:";
                  for(int y=0;y<reading0_l[x];y++)
                     str += ((y>0 ? "," : "") + String.valueOf(reading0[x][y]));
                  str += ") ";
                  }

           if (reading1_l[x]>0)
                  {
                  str += "(1:";
                  for(int y=0;y<reading1_l[x];y++)
                     str += ((y>0 ? "," : "") + String.valueOf(reading1[x][y]));
                  str += ") ";
                  }

           if (epsilon_l[x]>0)
                  {
                  str += "(e:";
                  for(int y=0;y<epsilon_l[x];y++)
                     str += ((y>0 ? "," : "") + String.valueOf(epsilon[x][y]));
                  str += ")";
                  }
           str += "\n";
           }
        return str;
        }

    public boolean[][] restrictToSize(int size, boolean[][]cases, EvolutionState state, int thread)
```

```
{
int csize = cases.length;
if (csize < size) return cases;

Hashtable hash = new Hashtable();
for(int x=0;x<size;x++)
   {
   while(true)
         {
         boolean[] b = cases[state.random[thread].nextInt(csize)];
         if (!hash.contains(b)) { hash.put(b,b); break; }
         }
   }

boolean[ ][ ] newcases = new boolean[size][ ];
Enumeration e = hash.keys();
for(int x=0;x<size;x++)
   {
   newcases[x] = (boolean[ ])(e.nextElement());
   }


QuickSort.qsort(newcases,
                  new SortComparator()
                     {
                     public boolean lt(Object a, Object b)
                           {
                           boolean[] aa = (boolean[])a;
                           boolean[] bb = (boolean[])b;
                           for(int x=0;x<Math.min(aa.length,bb.length);x++)
                              if (!aa[x] && bb[x]) return true;
                              else if (aa[x] && !bb[x]) return false;
                           if (aa.length<bb.length) return true;
                           return false;
                           }

                     public boolean gt(Object a, Object b)
                           {
                           boolean[] aa = (boolean[])a;
                           boolean[] bb = (boolean[])b;
                           for(int x=0;x<Math.min(aa.length,bb.length);x++)
                              if (!aa[x] && bb[x]) return false;
                              else if (aa[x] && !bb[x]) return true;
                           if (aa.length>bb.length) return true;
                           return false;
                           }
                     });
return newcases;
}
```

```
public boolean[][] slurp(final File f)
        throws IOException
        {
        LineNumberReader r = new LineNumberReader(new InputStreamReader(new
        FileInputStream(f)));
        String bits;

        Vector v = new Vector();
        while((bits=r.readLine())!=null)
           {
           bits = bits.trim();
           int len = bits.length();
           if (len==0) continue; // empty line
           if (bits.charAt(0)=='#') continue;  // comment
           if (bits.equalsIgnoreCase("e"))
                   v.addElement(new boolean[0]);
           else
                   {
                   boolean[] b = new boolean[len];
                   for(int x=0;x<len;x++)
                      b[x] = (bits.charAt(x)=='1');
                   v.addElement(b);
                   }
           }
        r.close();
        boolean[][] result = new boolean[v.size()][];
        v.copyInto(result);
        return result;
        }


public void printBits(final EvolutionState state, final boolean[][] bits)
        {
        StringBuffer s;
        for(int x=0;x<bits.length;x++)
           {
           s = new StringBuffer();
           for(int y=0;y<bits[x].length;y++)
                   if (bits[x][y]) s.append('1');
                   else s.append('0');
           if (s.length()==0) state.output.message("(empty)");
           else state.output.message(s.toString());
           }
        }


public void setup(final EvolutionState state,  final Parameter base)
        {
        super.setup(state,base);
```

```
File ap = null;
File an = null;
File tp = null;
File tn = null;
int restriction;

if (generalize)
   {
   ap = state.parameters.getFile(base.push(P_ALLPOS),null);
   an = state.parameters.getFile(base.push(P_ALLNEG),null);
   }

tp = state.parameters.getFile(base.push(P_TESTPOS),null);
tn = state.parameters.getFile(base.push(P_TESTNEG),null);

if (generalize)
   {
   if (ap==null) state.output.error("File doesn't exist", base.push(P_ALLPOS));
   if (an==null) state.output.error("File doesn't exist", base.push(P_ALLNEG));
   }

if (tp==null) state.output.error("File doesn't exist", base.push(P_TESTPOS));
if (tn==null) state.output.error("File doesn't exist", base.push(P_TESTNEG));
state.output.exitIfErrors();

if (generalize)
   {
   if (!ap.canRead()) state.output.error("File cannot be read", base.push(P_ALLPOS));
   if (!an.canRead()) state.output.error("File cannot be read", base.push(P_ALLNEG));
   }

if (!tp.canRead()) state.output.error("File cannot be read", base.push(P_TESTPOS));
if (!tn.canRead()) state.output.error("File cannot be read", base.push(P_TESTNEG));
state.output.exitIfErrors();

if (generalize)
   {
   state.output.message("Reading Positive Examples");
   try { posA = slurp(ap); }
   catch(IOException e)
         { state.output.error("IOException reading file (here it is)\n" + e,
         base.push(P_ALLPOS)); }
   state.output.message("Reading Negative Examples");
   try { negA = slurp(an); }
   catch(IOException e)
         { state.output.error( "IOException reading file (here it is)\n" + e,
         base.push(P_ALLNEG)); }
   }

state.output.message("Reading Positive Training Examples");
```

```
        try { posT = slurp(tp); }

        catch(IOException e)
                { state.output.error( "IOException reading file (here it is)\n" + e,
                base.push(P_TESTPOS)); }
        if ((restriction = state.parameters.getInt(
                 base.push(P_MAXTEST),null,1))>0)
           {
           state.output.message("Restricting to <= " + restriction + " Unique Examples");
           posT = restrictToSize(restriction,posT,state,0);
           }

        state.output.message("");
        printBits(state,posT);
        state.output.message("");

        state.output.message("Reading Negative Training Examples");
        try { negT = slurp(tn); }
        catch(IOException e)
        { state.output.error(  "IOException reading file (here it is)\n" + e,
base.push(P_TESTNEG)); }
        if ((restriction = state.parameters.getInt(
                 base.push(P_MAXTEST),null,1))>0)
           {
           state.output.message("Restricting to <= " + restriction + " Unique Examples");
           negT = restrictToSize(restriction,negT,state,0);
           }

        state.output.message("");
        printBits(state,negT);
        state.output.message("");

        state.output.exitIfErrors();


        input = (EdgeData) state.parameters.getInstanceForParameterEq(
           base.push(P_DATA), null, EdgeData.class);
        input.setup(state,base.push(P_DATA));
        }


   public boolean test(final boolean[] sample)
        {
        final boolean STATE_1 = false;
        final boolean STATE_2 = true;
        boolean st = STATE_1;

        for(int x=0;x<numNodes;x++)
           state1[x]=start[x];
```

```
for(int x=0;x<sample.length;x++)
   {
   if (st==STATE_1)
         {
         for(int y=0;y<numNodes;y++)
            state2[y]=false;
         for(int y=0;y<numNodes;y++)
            if (state1[y])
                  {
                  // advance edges
                  if (sample[x]) // reading a 1
                     for(int z=0;z<reading1_l[y];z++)
                           state2[reading1[y][z]] = true;
                  else  // reading a 0
                     for(int z=0;z<reading0_l[y];z++)
                           state2[reading0[y][z]] = true;
                  }


         // advance along epsilon boundary
         boolean moreEpsilons = true;
         while(moreEpsilons)
            {
            moreEpsilons = false;
            for(int y=0;y<numNodes;y++)
                  if (state2[y])
                     for(int z=0;z<epsilon_l[y];z++)
                           {
                           if (!state2[epsilon[y][z]]) moreEpsilons = true;
                           state2[epsilon[y][z]] = true;
                           }
            }
         }


   else //if (st==STATE_2)
         {
         for(int y=0;y<numNodes;y++)
            state1[y]=false;
         for(int y=0;y<numNodes;y++)
            if (state2[y])
                  {
                  // advance edges
                  if (sample[x]) // reading a 1
                     for(int z=0;z<reading1_l[y];z++)
                           state1[reading1[y][z]] = true;
                  else  // reading a 0
                     for(int z=0;z<reading0_l[y];z++)
                           state1[reading0[y][z]] = true;
```

```
                        }

                // advance along epsilon boundary
                boolean moreEpsilons = true;
                while(moreEpsilons)
                    {
                    moreEpsilons = false;
                    for(int y=0;y<numNodes;y++)
                            if (state1[y])
                                for(int z=0;z<epsilon_l[y];z++)
                                        {
                                        if (!state1[epsilon[y][z]]) moreEpsilons = true;
                                        state1[epsilon[y][z]] = true;
                                        }
                    }
                }

            st = !st;
            }

        if (st==STATE_1)
            {
            for(int x=0;x<numNodes;x++)
                    if (accept[x] && state1[x]) return true;
            }
        else // (st==STATE_2)
            {
            for(int x=0;x<numNodes;x++)
                    if (accept[x] && state2[x]) return true;
            }
        return false;
        }



    int totpos;
    int totneg;

// Tests an individual, returning its successful positives in totpos and its successful negatives in
totneg.

 public void fullTest(final EvolutionState state, final Individual ind, final int threadnum,
                            boolean[ ][ ] pos,  boolean[ ][ ] neg)
        {
        numNodes = 2;
        numEdges = 1; from[0]=0; to[0]=1;
        start[0]=start[1]=accept[0]=accept[1]=false;
        ((EdgeData)input).edge = 0;

        ((GPIndividual)ind).trees[0].child.eval(
```

```
        state,threadnum,input,stack,((GPIndividual)ind),this);

    if (reading1.length < numNodes ||
        reading1[0].length < numEdges)
        {
        reading1 = new int[numNodes*2][numEdges*2];
        reading0 = new int[numNodes*2][numEdges*2];
        epsilon = new int[numNodes*2][numEdges*2];
        reading1_l = new int[numNodes*2];
        reading0_l = new int[numNodes*2];
        epsilon_l = new int[numNodes*2];
        }

    for(int y=0;y<numNodes;y++)
        {
        reading1_l[y]=0;
        reading0_l[y]=0;
        epsilon_l[y]=0;
        }

    for(int y=0;y<numEdges;y++)
        switch(reading[y])
                {
                case READING0:
                    reading0[from[y]][reading0_l[from[y]]++]=to[y];
                    break;
                case READING1:
                    reading1[from[y]][reading1_l[from[y]]++]=to[y];
                    break;
                case EPSILON:
                    epsilon[from[y]][epsilon_l[from[y]]++]=to[y];
                    break;
                }

    if (state1.length < numNodes)
        {
        state1 = new boolean[numNodes*2];
        state2 = new boolean[numNodes*2];
        }


totpos=0;
totneg=0;
    for(int y=0;y<pos.length;y++)
        if (test(pos[y])) totpos++;
    for(int y=0;y<neg.length;y++)
        if (!test(neg[y])) totneg++;
    }
```

```
public void evaluate(final EvolutionState state,  final Individual ind,  final int threadnum)
        {
        if (start==null)
           {
           start = new boolean[MIN_ARRAY_SIZE];
           accept = new boolean[MIN_ARRAY_SIZE];
           reading = new int[MIN_ARRAY_SIZE];
           from = new int[MIN_ARRAY_SIZE];
           to = new int[MIN_ARRAY_SIZE];
           state1 = new boolean[MIN_ARRAY_SIZE];
           state2 = new boolean[MIN_ARRAY_SIZE];
           reading1 = new int[MIN_ARRAY_SIZE][MIN_ARRAY_SIZE];
           reading0 = new int[MIN_ARRAY_SIZE][MIN_ARRAY_SIZE];
           epsilon = new int[MIN_ARRAY_SIZE][MIN_ARRAY_SIZE];
           reading1_l = new int[MIN_ARRAY_SIZE];
           reading0_l = new int[MIN_ARRAY_SIZE];
           epsilon_l = new int[MIN_ARRAY_SIZE];
           }

        if (!ind.evaluated)
           {
           fullTest(state,ind,threadnum,posT,negT);

           KozaFitness f = ((KozaFitness)ind.fitness);

           f.setFitness(state,(float)
                        (1.0 - ((double)(totpos + totneg)) /
                         (posT.length + negT.length)));

           f.hits = totpos + totneg;
           ind.evaluated = true;
           }
        }


public void describe(final Individual ind,  final EvolutionState state,  final int threadnum,
final int log,
                        final int verbosity)
        {
        if (start==null)
           {
           start = new boolean[MIN_ARRAY_SIZE];
           accept = new boolean[MIN_ARRAY_SIZE];
           reading = new int[MIN_ARRAY_SIZE];
           from = new int[MIN_ARRAY_SIZE];
           to = new int[MIN_ARRAY_SIZE];
           state1 = new boolean[MIN_ARRAY_SIZE];
           state2 = new boolean[MIN_ARRAY_SIZE];
           reading1 = new int[MIN_ARRAY_SIZE][MIN_ARRAY_SIZE];
           reading0 = new int[MIN_ARRAY_SIZE][MIN_ARRAY_SIZE];
```

```
            epsilon = new int[MIN_ARRAY_SIZE][MIN_ARRAY_SIZE];
            reading1_l = new int[MIN_ARRAY_SIZE];
            reading0_l = new int[MIN_ARRAY_SIZE];
            epsilon_l = new int[MIN_ARRAY_SIZE];
            }

        if (generalize)
            fullTest(state,ind,threadnum,posA,negA);
        else
            fullTest(state,ind,threadnum,posT,negT);

        if (generalize)
            state.output.println("\n\nBest Individual's Generalization Score...\n" +
                                 "Pos: " + totpos + "/" + posA.length +
                                 " Neg: " + totneg + "/" + negA.length +
                                 "\n(pos+neg)/(allpos+allneg):     " +
                                 (float)
                                 (((double)(totpos+totneg))/(posA.length+negA.length)),
                                       verbosity,log);

        state.output.println("\nBest Individual's NFA\n====================\n",
                             verbosity,log);

        state.output.println(printCurrentNFA(),verbosity,log);
        }

}
```