Tallinn University of Technology
Institute of Cybernetics

Genetic Inference of Finite State Machines
Master thesis
Margarita Spichakova

Supervisor:
Professor Jaan Penjam

Tallinn 2007
Declaration

Hereewith I declare that this thesis is based on my own work. All ideas, major views and data from different sources by other authors are used only with a reference to the source. The thesis has not been submitted for any degree or examination in any other university.

.........................                .........................
(Date)                                             (Author)
Abstract

This thesis proposes a methodology based on genetic algorithms for inferring finite state machines. It defines basic parameters like fitness functions and chromosomal representation that can be used with canonical genetic algorithm. Different types of finite state machines can be inferred. Several improvements were introduced: inner decimal representation has been used to improve the speed of initialization stage, post-processing stage has been used to remove unaccessible states.

The experimental implementation has also been presented, together with some experiments.
Contents

1 Introduction 8
  1.1 Motivation and goal 8
  1.2 Historical background 9
  1.3 Organization of the work 10

2 Finite State Machines 11
  2.1 Alphabets, strings, language 12
  2.2 Finite acceptor 13
    2.2.1 Basic definitions 13
    2.2.2 Minimal automata 15
  2.3 Mealy and Moore machines 16
    2.3.1 Basic definitions 16
    2.3.2 Equivalence of Moore and Mealy machines 17
    2.3.3 FSM minimization 17
    2.3.4 FA as FSM with output 19

3 Genetic Algorithms 21
  3.1 Evolutionary algorithm 21
  3.2 Canonical genetic algorithm 22
    3.2.1 Characteristics of canonical GA 22
    3.2.2 Genetic representation 22
    3.2.3 Formal description of canonical GA 23
  3.3 Genetic operators 23
    3.3.1 Mutation 23
    3.3.2 Crossover 24
    3.3.3 Selection 25

4 Genetic Identification of Finite State Machines 27
  4.1 Fitness functions 27
    4.1.1 Distance between strings 27
    4.1.2 Fitness value evaluation 28
4.2 Chromosomal encoding. Genetic operators .......................... 29
  4.2.1 Restrictions .................................................................. 30
  4.2.2 Moore machines .......................................................... 30
  4.2.3 Mealy machines ........................................................... 35
4.3 Initialization. Reduction of invalid individuals ..................... 37

5 Implementation .................................................................. 39
  5.1 Experimental software description ..................................... 39
    5.1.1 FSM package ............................................................. 40
    5.1.2 Beings package .......................................................... 41
    5.1.3 Evolution package ....................................................... 42
  5.2 Evolutionary trace ............................................................ 44
  5.3 Post processing ................................................................. 44

6 Experiments ................................................................... 46
  6.1 Learning Random Finite State Machines ............................... 46
    6.1.1 Getting the training data .............................................. 46
    6.1.2 Equivalence of Mealy and Moore machines .................... 46
    6.1.3 Adjusting the parameter ’number of states’ .................... 48
    6.1.4 Moore machine with unknown number of states ............ 52
  6.2 Learning Mealy machine. Parity checker ............................... 54
  6.3 Learning Mealy machine. Two unit delay ............................ 54
  6.4 Learning Moore machine. Pattern recognizer .................... 56
  6.5 Learning Moore machine. Up–down counter ........................ 58

7 Conclusions and future work .............................................. 61
  7.1 Conclusions ................................................................. 61
  7.2 Future work ................................................................. 62
## List of Tables

<table>
<thead>
<tr>
<th>No.</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The example of transition table for FA</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Moore machine minimization. Step 1</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Moore machine minimization. Step 2</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Roulette–wheel selection</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>String distance functions</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Measuring fitness value</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>One section of chromosome for storing the Moore machine with fixed number of states</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Chromosomal representation of the Moore machine with fixed number of states</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Chromosomal representation of a Moore machine with fixed number of states</td>
<td>32</td>
</tr>
<tr>
<td>4.6</td>
<td>One section of chromosome for storing the Moore machine with unknown number of states</td>
<td>33</td>
</tr>
<tr>
<td>4.7</td>
<td>Chromosomal representation of the Moore machine. Adopting the number of states</td>
<td>33</td>
</tr>
<tr>
<td>4.8</td>
<td>Example. Chromosomal representation of the Moore machine. Adopting the number of states</td>
<td>34</td>
</tr>
<tr>
<td>4.9</td>
<td>One section of chromosome for storing the Mealy machine with fixed number of states</td>
<td>35</td>
</tr>
<tr>
<td>4.10</td>
<td>Chromosomal representation of the Mealy machine with fixed number of states</td>
<td>36</td>
</tr>
<tr>
<td>4.11</td>
<td>Example. Chromosomal representation of the Mealy machine with fixed number of states</td>
<td>36</td>
</tr>
<tr>
<td>4.12</td>
<td>Error in decoding process</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Removing unreachable states. Example</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Training data. Equivalence of Mealy and Moore machines</td>
<td>47</td>
</tr>
<tr>
<td>6.2</td>
<td>Equivalence of Mealy and Moore machine. Results</td>
<td>48</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Problem of system identification .......................... 9
2.1 FSM classification ........................................ 11
2.2 Finite Acceptor ........................................... 13
2.3 FA. Transition diagram ................................... 14
2.4 FSM .......................................................... 16
2.5 Moore machine minimization .............................. 19
2.6 Converting FA to FSM. FA ............................... 20
2.7 Converting FA to FSM. FSM ............................. 20
3.1 Mutation ...................................................... 24
3.2 Inversion ..................................................... 24
3.3 One point crossover ....................................... 24
3.4 Roulette-wheel selection ................................ 26
4.1 A Moore machine represented as transition diagram .... 31
4.2 A Moore machine with an unreachable state ............ 33
4.3 A Mealy machine represented as transition diagram .... 36
4.4 Coding/decoding process from binary genotype to FSM . 37
4.5 Coding/decoding process from binary genotype to FSM using decimal representation ...................... 38
5.1 Packages ..................................................... 40
5.2 FSM package ................................................ 41
5.3 Beings package ............................................. 42
5.4 Evolution package ......................................... 43
5.5 Evolutionary trace ......................................... 44
6.1 Choice of number of states. Frequencies ................. 52
6.2 Parity checker. Transition diagram ....................... 55
6.3 Two unit delay. Evolutionary trace ....................... 56
6.4 Two unit delay. Transition diagram ....................... 56
6.5 Pattern recognizer for "abb". Evolutionary trace ........ 57
6.6 Pattern recognizer for "abb", incorrect. Transition diagram . . 57
6.7 Pattern recognizer for "abb". Transition diagram . . . . . . 58
6.8 Up–down counter. Original machine. Transition diagram . . 58
6.9 Up–down counter. Generated machine. Transition diagram . . 59
6.10 Up–down counter. Evolutionary trace . . . . . . . . . . . . . . 60
Chapter 1

Introduction

1.1 Motivation and goal

Identification has been defined as an inference process, which deduces an internal representation of the system (named internal model, usually a state machine) from samples of its functioning (named external model, usually given by several input/output sequences) [18]. There are many possibilities to represent a system, one of them is to use a finite state machine (FSM).

There are many applications for identification in different fields such as logical design, verification or test, sequential learning. For example finite automata (FA) are useful models for many important kind of hardware and software systems (lexical analysis, system for verifying the correctness).

The modeling problem consists of researching the 'best' FSM (in practice, one of those that best describe the process behavior), which respects the dynamics of the external model. The process is indeed considered a black box.

Finding a FSM consistent with the given input/output sequences is a problem in the field of grammatical inference [18].

Finding a minimum size deterministic FA consistent with the set of given samples is NP–complete [7]. The genetic simulation approach is an alternative that reduces the complexity of the preceding methods.

Our goal is to create methods based on Genetic Algorithm (GA), which allow to find FSM using only external behavior.
The task is to find a FSM $M$ that is consistent with all elements in the set given input/output sequence set $\zeta$, that is composed of a number of input/output sequences (external behavior).

There are two types of solutions that can be found during the process of identification:

- **Complete solution** is a solution that operated correctly for all given input/output sequences
- **Incomplete solution** is a solution that operated correctly for some input/output sequences.

1.2 Historical background

In the early 1960s Fogel introduced Evolutionary Programming (EP) [5]. The simulated evolution was performed by modifying a population of FSM. Other authors also used EP for solving the problem of FSM identification. Kumar Chellapilla and David Czarnecki proposed the variation of EP to solve the problem of modular FSM synthesis [4]. Karl Benson presented a model comprising FSM with embedded genetic programs which co-evolve to perform the task of Automatic Target Detection [3].

Another approach to solve the problem of FSM identification is based on GA. This method has been researched by several authors. Tongchim and Chongistitvatana investigated parallel implementation of GA to solve the problem of FSM synthesis [23]. Chongistitvatana and Niparnan improved GA by evolving only the state transition function [20]. Chongistitvatana also presented a
method of FSM synthesis from multiple partial input/output sequences [21]. Jason W. Horihan and Yung–Hsiang Lu paid more attention to improving the FSM evolution by using progressive fitness functions [10].

Different types of machines can be inferred using GA: Lamine Ngome used genetic simulation for Moore machine identification [19], Pushmeet Kohli used GA to synthesize FA accepting particular language using accept/reject data [11], Philip Hingston showed in [8] how GA can be used for the inference of regular language from a set of positive examples, Xiaojun Geng applied GA for solving identification problem for asynchronous FSM [6]. The algorithm for Automated Negotiations presented by Tu, Wolff and Lamersdorf is based on GA synthesis of FSM [24]. Simon M. Lucas paid more attention to finite state transducers [16] and compares his method to ”Heuristic State Merging” [17].

GA has also been used for solving other similar problems: for solving State Assignment Problem [1], for identification of nondeterministic pushdown automata [12], for inferring regular and context-free grammars [15], for protecting resources [22].

1.3 Organization of the work

The thesis is organized in the following way: in chapter 2 we give theoretical overview of FSM. Chapter 3 contains the basic theory of GA. In chapter 4 we present a method for genetic inference of FSM. The implementation of that method is presented in chapter 5. Experiments required to show the efficiency of the presented method are described in chapter 6.
Chapter 2

Finite State Machines

The term *finite state machine* describes a class of models that are characterized by having a finite number of states (figure 2.1). The class of FSM can be subdivided into several subclasses, the most important are *finite acceptor* (finite state machine without output, or finite automaton), discussed in section 2.2 and *finite transducer* (finite state machine with output), described in section 2.3.

In this thesis, the term *finite state machine* will denote only finite transducers, if not specified otherwise.

![Figure 2.1: FSM classification](image-url)
2.1 Alphabets, strings, language

Information is encoded by means of sequences of symbols. A symbol is a basic component of strings and alphabets. Alphabet is a set of symbols. A sequence of symbols from alphabet Σ is called string. The set of all strings over alphabet Σ is denoted by Σ*. The empty string denoted as ϵ.

If w is a string then |w| denotes the number of symbols in w and is called the length of w, |ε| = 0. Two strings w and u are equal if they contain the same number of symbols in the same order.

Given two strings w, u ∈ Σ*. We can form a new string w · u (w · u = wu), called the concatenation of w and u. Concatenation of w and u is adjoining the symbols in u to symbols in w. The order in which strings are concatenated is important. The concatenation with empty string ϵ has the following property:

\[ \epsilon w = w \epsilon = w \]

**Example 2.1.** Let’s define \( x = a_1a_2\ldots a_n \) and \( y = b_1b_2\ldots b_m \), then \( x \cdot y = a_1a_2\ldots a_n b_1b_2\ldots b_m \)

Let \( w, u \in \Sigma^* \). If \( x = wu \) then w is called prefix of x and u is called suffix of x.

For any alphabet Σ, a subset of Σ* is called language.

There are several Boolean operations defined on languages: if \( L \) and \( M \) are languages over alphabet Σ then \( L \cup M \) is the union of \( L \) and \( M \), \( L \cap M \) is the intersection of languages and \( \overline{L} \) is complement of \( L \).

Let’s define two more operations on languages: the product and Kleene star. Let \( L \) and \( M \) be languages over alphabet Σ then:

\[ L \cdot M = \{ wu : w \in L \text{ and } u \in M \} \]

is called the product of \( L \) and \( M \). A string belongs to \( LM \) if it can be written as a string in \( L \) concatenated with a string in \( M \).

For a language \( L \), we can define \( L^0 = \{ \epsilon \} \) and \( L^{n+1} = L^n \cdot L \). For \( n > 0 \), the language \( L^n \) consists of all strings \( w \) of form \( w = u_1u_2\ldots u_n \) where \( u_i \in L \). The Kleene Star of language \( L \) is denoted as \( L^* \) and defined to be

\[ L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \]
### 2.2 Finite acceptor

An information-proceeding device transforms inputs to outputs. There are two alphabets associated with this device: input alphabet $\Sigma$ and output alphabet $\Delta$. Input alphabet is for communicating with the device and output alphabet for receiving the answers.

![Finite Acceptor](image)

**Figure 2.2: Finite Acceptor**

In this section we will describe a special case: there is an input alphabet $\Sigma$, but device can output only *yes* or *no* (figure 2.2). So all input strings can be divided into two subsets: strings *accepted* by the machine (answer *yes*) and strings *rejected* by the machine (answer *no*). Let’s describe the mathematical model of finite acceptor.

#### 2.2.1 Basic definitions

**Definition 2.1.** A finite acceptor is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where $Q$ is a finite set of states, $\Sigma$ is a finite input alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, $\delta$ is a transition function: $\delta : Q \times \Sigma \rightarrow Q$.

If, for each state $q \in Q$ and each symbol $a \in \Sigma$, there exists at most a single transition, i.e., $|\delta(q, a)| \leq 1$, we call the acceptor a *deterministic* finite acceptor (DFA), otherwise it is called a *nondeterministic* finite acceptor (NDFA).

**Theorem 2.1.** Let $L$ be a language accepted by nondeterministic finite automaton, then where exists a deterministic finite automaton that accepts $L$ [9].

There are two popular ways to represent FA: transition diagram (graph) and transition table.

A *transition graph* is a special case of directed labeled graph where vertices are labeled by states $Q$; there is an arrow labeled $a$ from vertex labeled $s$ to vertex labeled $t$ exactly when $t \in \sigma(s, a)$. The initial state is marked by inward-pointing arrow and final state by double circles.
Example 2.2. Here is an example of simple FA represented as transition diagram (figure 2.3) and as transition table 2.1.

![Transition Diagram](image)

Figure 2.3: FA. Transition diagram

Five ingredients of FA:
- Set of states \(\{q_0, f_1\}\)
- The input alphabet \(\Sigma = \{a, b\}\)
- The initial state \(q_0\)
- The set of final states \(F = \{f_1\}\)
- The transition function \(\delta : Q \times \Sigma \to Q\) is defined as

\[
\delta(q_0, a) = q_0, \quad \delta(q_0, b) = f_1, \quad \delta(f_1, a) = q_0, \quad \delta(f_1, b) = f_1
\]

The transition table for this acceptor is represented in table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>q₀</td>
<td>f₁</td>
</tr>
<tr>
<td>←</td>
<td>f₁</td>
<td>q₀</td>
</tr>
</tbody>
</table>

Table 2.1: The example of transition table for FA

Let’s define the language accepted by FA.

**Definition 2.2.** If \(M = (Q, \Sigma, \delta, q_0, F)\) is FA, \(a \in \Sigma \cup \epsilon\), we say \((q, aw) \vdash_M (p, w)\), iff \(p \in \delta(q, a)\). \(\vdash_M\) is called yield operator.

**Definition 2.3.** A string \(w\) is said to be accepted by an acceptor \(M\) iff \((q_0, w) \vdash_M^* (p, \epsilon)\) for some \(p \in F\), i.e., there exists a finite sequence of transitions, corresponding to the input string \(w\), from the initial state to some final state.

The language accepted by \(M\), is denoted as \(L(M)\) and defines as:

\[
L(M) = \{w \mid \exists p \in F : (q_0, w) \vdash_M^* (p, \epsilon)\}.
\]
2.2.2 Minimal automata

An acceptor can be reduced in size without changing the language recognized. There are two methods of doing this:

- removing the states that cannot play any role in deciding whether a string is accepted;
- merging the states that 'do the same work'.

Let $A = (Q, \Sigma, i, q_0, F)$ be a FA.

**Definition 2.4.** We can say that state $q \in Q$ is accessible if there is a string $x \in \Sigma^*$ such that $q_0 \cdot x = q$, where $q_0 \cdot x = q$ means that state $q$ can be reached from state $q_0$ by making transitions according to corresponding symbols of $x$.

$$q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} \ldots \xrightarrow{x_n} q$$

A state that is not accessible is inaccessible. An acceptor is accessible if it’s every state is accessible. The inaccessible states of an automaton play no role in accepting strings, so they can be removed without the language being changed.

**Definition 2.5.** Two states $q_i \in Q$ an $q_j \in Q$ are distinguishable if there exists $X \in \Sigma^*$ such that

$$(q_i \cdot x, q_j \cdot x) \in (F \times F') \cup (F' \times F),$$

where $F' \in Q \setminus F$.

In other words, for the same string $x$, one of states $q_i \cdot x$, $q_j \cdot x$ is final and the other is not final.

If states $q_i$ an $q_j$ are indistinguishable thus means that for each $x \in \Sigma^*$

$$q_i \cdot x \in F \Leftrightarrow q_j \cdot x \in F.$$  

Define the relation $\simeq_A$ on the set of states $Q$ by

$$q_i \simeq_A q_j \Leftrightarrow q_i \text{ and } q_j \text{ are indistinguishable}.$$

**Definition 2.6.** An acceptor is reduced if each pair of states in an acceptor is distinguishable.

**Theorem 2.2.** Let $A$ be a FA. There is an algorithm that constructs an accessible acceptor, $A^a$, such that $L(A) = L(A^a)$ [14].
**Theorem 2.3.** Let $A$ be a FA. There is an algorithm that constructs a reduced acceptor $A'$, such that $L(A) = L(A')$ [14].

Those two theorems can be applied to any acceptor $A$ in turn yielding an acceptor $A^{ar} = (A^a)^r$ that is both accessible and reduced.

**Definition 2.7.** Let $L$ be a language accepted by FA. A finite acceptor $A$ is said to be minimal (for $L$) if $L(A) = L$, and if $B$ is any FA such that $L(B) = L$, then the number of states of $A$ is less than or equal to number of states of $B$. If $A$ is minimal acceptor then it is reduced and accessible. An algorithm for FA minimization can be found in [13].

### 2.3 Mealy and Moore machines

#### 2.3.1 Basic definitions

By generalizing FA (subsection 2.3.4) we can get the machine with output (figure 2.4). It can produce an output sequence according to the input sequence. There are several types of FSM, we will discuss only two of them: the Moore machine and the Mealy Machine.

**Definition 2.8.** A Moore Machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where $Q$ is a finite set of states, where $q_0$ denotes the start state; $\Sigma$ is the input alphabet; $\Delta$ is the output alphabet; $\delta : Q \times \Sigma \rightarrow Q$ is the transition function; $\lambda : Q \times \Sigma \rightarrow \Delta$ is the output function represented by the output table that shows what character from $\Delta$ will be printed by each state that is entered.

**Definition 2.9.** A Mealy machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where $Q$ is a set of states, where $q_0$ denotes the start state; $\Sigma$ is the input alphabet; $\Delta$ is the output alphabet; $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, and $\lambda : Q \times \Sigma \rightarrow \Delta$ the output function represented by the output table that shows what character from $\Delta$ will be printed by each transition that is processed [9].
The key difference between Moore and Mealy machines:

- Moore machines print a character when entering the state;
- Mealy machines print a character when traversing an arc.

**Definition 2.10.** An input/output sequence $S$ of length $n$ is a set of pairs $\{(i_0, o_0), (i_1, o_1), \ldots, (i_n, o_n)\}$ where $(i_i, o_i) \in \Sigma \times \Delta$. An input/output sequence set $\zeta$ is a set of $S$.

A FSM $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ is said to be consistent with an input/output sequence $S$ iff $o_j = \lambda(q_j, i_j)$ for all $0 \leq j \leq n$.

### 2.3.2 Equivalence of Moore and Mealy machines

Given a Mealy machine $Me$ and a Moore machine $Mo$, which automatically prints the character $x$ in the start state, we say that these two machines are equivalent if for every input string the output string from $Mo$ is exactly $x$ concatenated with output from $Me$.

**Theorem 2.4.** If $Mo$ is a Moore machine, then there is a Mealy machine $Me$ that is equivalent to it.

**Theorem 2.5.** For every Mealy machine $Me$, there is an equivalent Moore machine $Mo$.

### 2.3.3 FSM minimization

Both Moore and Mealy machines can be minimized in a way that directly generalizes the minimization of FA described in subsection 2.2.2.

**Definition 2.11.** Two FSMs are equivalent iff for any given input sequence, identical output sequences are produced.

**Definition 2.12.** Two states $q_0$ and $q_1$ in a given FSM are equivalent iff each input sequence beginning from $q_0$ yields an output sequence identical to that obtained by starting from $q_1$.

To minimize Moore machine we need to find and remove equivalent states. First, we need to find all pairs of states which have the same output. Second, we need to check all pairs of states with the same output: if target states of transitions marked by each input symbol are equal for two states, whether this is a pair of equivalent states (algorithm 2.1).
Algorithm 2.1. Minimization of Moore machine

for (each pair of states (qi, qj)) {
    if (qi and qj have different output) {
        mark qi and qj as not equivalent;
    }
}
for (each unmarked pair) {
    for (each input symbol, qi and qj are transferred to states which are not equivalent) {
        Mark qi and qj as not equivalent;
    }
}
} while (marking is possible);
Unmarked pairs are equivalent;

Example 2.3. The Moore machine Mo represented in figure 2.5 (left) can be minimized to machine Mo’ (right) using algorithm 2.1. The process of Moore machine minimization is described in tables 2.2 and 2.3. During step 1 (table 2.2) we mark all states as not equivalent if they have different output symbols.

Table 2.2: Moore machine minimization. Step 1

<table>
<thead>
<tr>
<th></th>
<th>(0, 1)</th>
<th>(0, 2)</th>
<th>(0, 3)</th>
<th>(0, 5)</th>
<th>(1, 2)</th>
<th>(1, 3)</th>
<th>(1, 5)</th>
<th>(2, 3)</th>
<th>(2, 5)</th>
<th>(3, 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

During step 2 (table 2.3) we analyze the states that were not marked during the previous step. In our case we analyze states '3' and '5'. As a result of analysis we can say that states '3' and '5' are equivalent.

Table 2.3: Moore machine minimization. Step 2

<table>
<thead>
<tr>
<th>Input</th>
<th>State '3' leads to</th>
<th>State '5' leads to</th>
<th>Equivalent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>State '1'</td>
<td>State '1'</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>State '2'</td>
<td>State '2'</td>
<td>+</td>
</tr>
</tbody>
</table>
To minimize a Mealy machine we need to find and remove equivalent states. The idea is similar to previous algorithm. For each pair of states we need to check if all target states of transitions (checking all input and output symbols) are the same (algorithm 2.2).

Algorithm 2.2. Minimization of Mealy machine

for (each pair of states (qi, qj)){
  for (each input symbol){
    if (transition from qi and transition from qj labeled with same input symbol have different output){
      mark qi and qj as not equivalent;
    } else {
      if (transition from qi and transition from qj labeled with same input symbol have different target state){
        mark qi and qj as not equivalent;
      }
    }
  }
}
Unmarked pairs are equivalent;

2.3.4 FA as FSM with output

FA can be modified to produce output. Suppose that language $L$ is recognized by FA $A = \{Q, \Sigma, \delta, q_0, F\}$. We can make several alternations to $A$. 
Instead of labeling a state as 'final', we shall add label 1 to the state, so if state $q$ is final it will be relabeled as $q/1$. If the state is not final, then we should relabel it as $q/0$. The set $F$ can be recovered as set of all states labeled '1'. So we define the output function $\lambda \rightarrow 2$. Now our modified FA $A$ can be described as $B = \{Q, \Sigma, \delta, q_0, F, \lambda\}$.

Let $x = x_0 \ldots x_n$ be a string over $\Sigma$. $q_0 \ldots q_n$ - states that $B$ passes when processing $x$.

$$\epsilon \rightarrow q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} \ldots \xrightarrow{x_n} q$$

We can define string $w_B(x) = \lambda(x_0) \ldots \lambda(x_n)$. $w_B(x)$ can be deemed as output function of $B$.

**Example 2.4.** Consider the FA $A$ (figure 2.6).

![Figure 2.6: Converting FA to FSM. FA](image)

The language $L(A)$ consists of all strings in $(a + b)^*$ of even length. Now we convert it into FSM $B$ (figure 2.7)

![Figure 2.7: Converting FA to FSM. FSM](image)

Let’s $x = aba$, then FSM $B$ will pass through states $q, r, q, r$. Thus $w(aba) = \lambda(q)\lambda(r)\lambda(q)\lambda(r) = 1010$. 

20
Chapter 3

Genetic Algorithms

3.1 Evolutionary algorithm

Evolutionary Algorithm (EA) is a search algorithm based on simulation of biological evolution. In EA some mechanisms inspired by evolution are used: reproduction, mutation, recombination, selection. Candidate solution presents the individuals of the evolution.

All basic instances of EA share a number of common properties which can characterize a general EA:

- EA implements the process of collective learning of population of individuals. Each individual presents a search point in the space of potential solutions to the problem.

- New individuals are randomly generated from previous population using processes that model mutation and recombination. Mutation usually represents a self–replication, but reproduction require two or more individuals.

- A measure of the quality can be assigned to all individuals (fitness function).

Some differences in implementation of those principles characterize the instances of EA (Thomas Bäck [2]):

- Genetic algorithms (originally described by Holland (1962, 1975)) — the solution represented as an array of numbers (usually binary numbers). A recombination operator is also used to produce new solutions.
• *Evolutionary strategies* (developed by Rechenberg (1965, 1973) and Schwefel (1965, 1977)) — uses the real number vectors for representing solution. Mutation and crossover are essential operators for searching in search space.

• *Evolutionary programming* (developed by Lawrence J Fogel (1962)) — the algorithm was originally developed to evolve FSM, but most applications are for search spaces involving real-valued vectors. Does not incorporate the recombination of individuals and emphasizes the mutation.

In this work we will discuss only GA.

### 3.2 Canonical genetic algorithm

GAs are a class of evolutionary algorithms first proposed and analyzed by John Holland (1975).

#### 3.2.1 Characteristics of canonical GA

According to Larry J Eshelman [2] there are three features which distinguish GAs, as first proposed by Holland from other evolutionary algorithms:

- the representation used — *bi strings*;
- the method of selection — *proportional selection*;
- the primary method of producing variations — *crossover*;

Many subsequent alternations of GA have adopted different kinds of selections, and also some of them use different representations (not binary) of solutions. But all those methods are inspired by the original algorithm.

#### 3.2.2 Genetic representation

Genetic representation is a way of representing solutions/individuals. In GA the binary representation of fixed size is used (*chromosome*). The main property that makes this representation convenient is that their parts are easily aligned due to their fixed size.

The *genotype* of the individual is manipulated by the GA. To evaluate the individual we need to decode the genotype to *phenotype* and assign the fitness.
value. The process of mapping from genotype to phenotype is called *decoding*. The process of mapping from phenotype to genotype is called *coding*.

### 3.2.3 Formal description of canonical GA

First step is to generate (randomly) the initial population $P(0)$. Each individual must be evaluated by the fitness function. Then some individuals will be selected for reproduction and copied to $C(t)$. Next the genetic operators (mutation and crossover) are applied to $C(t)$ producing $C'(t)$. After new offsprings are generated a new population must be created from $C'(t)$ and $P(t - 1)$. If the size of population is $M$, then $M$ individuals must be selected from $C'(t)$ and $P(t - 1)$ to produce $P(t)$.

**Algorithm 3.1. (Canonical GA)**

```plaintext
t=0;
initialize P(t);
evaluate structures in P(t);

while (termination condition is not satisfied){
    t++;
    selectReproduction C(t) from P(t-1);
    recombine and mutate structures in C(t) forming C'(t);
    evaluate structures in C'(t);
    selectReplace P(t) from C'(t) and P(t-1);
}
```

### 3.3 Genetic operators

#### 3.3.1 Mutation

One of the best known mechanisms for producing variations is *mutation*, where new trial solutions are created by making small, random changes in representation of prior trial solutions. If binary representation is used, the mutation is achieved by randomly flipping one bit (figure 3.1).

Another option is called *inversion*. In this case the segment of the chromosome after random mutation point (suffix) is reversed (figure 3.2).
There are many crossover techniques: one–point crossover, two-point crossover, uniform crossover, half uniform crossover etc. In this thesis we will use only one–point crossover.

First of all a crossover point is selected. Secondly, parts of parent individuals are swapped. The resulting individuals are called children (figure 3.3).
3.3.3 Selection

Roulette–wheel selection

First the fitness method assigns fitness values to possible solutions. This value is used to associate probability of selection with each individual. The better the chromosomes are the better are the chances to be selected for reproduction.

The candidate solution with a higher fitness has less probability to be eliminated during the selection process, although the possibility still exists. Also there is a chance that some weaker solution may survive the selection. This is good, because in future populations, the weaker solutions may introduce components which could provide useful information.

Let’s assume we have a roulette–wheel where there is a pocket on the wheel for each candidate solution. The size of the pocket is proportional to the probability of selection. So to select a population of size $M$ means playing $M$ games. Each candidate is selected independently.

First step, for each individual in the population we must compute the fitness value. After that we can compute the total fitness $F$. To select one individual for reproduction we require: to generate a random number $R$ which belongs to segment $[0, F]$, after that we start passing through the population while summing up fitness value $s$, if this sum $s$ gets greater than the random number $R$, then we will return the individual currently pointed to. The formal description is given in algorithm 3.2.

Algorithm 3.2. (Roulette–wheel selection)

```plaintext
for (i=0; i< size of population M; i++){ 
    evaluate chromosome[i]; 
    F+=fitness value of chromosome[i]; 
} 
for (i=0; i<size of population M; i++){ 
    R=random value from interval [0,F]; 
    s=go through the population and sum fitnesses from 0; 
    if (s>R){
        return the chromosome where you are;
    }
}
```

25
Example 3.1. Let’s have a situation described in table 3.1: 10 individuals, for each individual we have computed fitness value and assigned selection probability. We want to select 6 random individuals, for this we generate 6 random numbers $0.81, 0.32, 0.96, 0.01, 0.65, 0.42 \in [0, 1]$.

Table 3.1: Roulette–wheel selection

<table>
<thead>
<tr>
<th>Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness</td>
<td>2.0</td>
<td>1.8</td>
<td>1.6</td>
<td>1.4</td>
<td>1.2</td>
<td>1.0</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
</tr>
<tr>
<td>Probability</td>
<td>0.18</td>
<td>0.16</td>
<td>0.15</td>
<td>0.13</td>
<td>0.11</td>
<td>0.09</td>
<td>0.07</td>
<td>0.06</td>
<td>0.03</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Figure 3.4 shows the selection process of the individuals. Selected individuals are: 6, 2, 9, 1, 5, 3.

Figure 3.4: Roulette–wheel selection

We shall not discuss other types of the genetic operators. More information about them can be found in [25] or [2].
Chapter 4

Genetic Identification of Finite State Machines

In this chapter we will apply the GA to the problem of FSM identification. The major difficulty lies in the efficient coding of the information and choice of adaptation function of the members of population.

4.1 Fitness functions

We choose a fitness function defined as a calculus performed on all I/O sequences (couples \{input, output\}). The idea is to estimate the proximity between the evaluated individual and searched FSM.

4.1.1 Distance between strings

We can specify several functions for computing distance between strings. The simplest one is strict distance. It can be measured as

\[
d_{\text{strict}}(x, y) = \begin{cases} 
0 & : x = y \\
1 & : x \neq y 
\end{cases}
\]  \hspace{1cm} (4.1)

Strict distance returns 0 if strings are equal, otherwise it returns 1.

Given two strings \(x, y\). The computing of strict distance has the worst case time complexity \(O(|x|)\), when \(|x| = |y|\), otherwise it is constant in time (if
$|x| \neq |y|$, then function will return 1).

We can specify a function $\Delta(a, b)$, where $a, b$ are symbols in some alphabet:

$$\Delta(a, b) = \begin{cases} 0 & : a = b \\ 1 & : a \neq b \end{cases} \quad (4.2)$$

that returns 1 if chars are not equal.

To compute Hamming distance $d_{Ham}$ between two strings we need to count the number of different bits in the same positions.

$$d_{Ham}(x, y) = \Sigma_{i=1}^{\text{Min}(|x|, |y|)} \Delta(x_i, y_i) \quad (4.3)$$

Computing Hamming distance has a complexity of $O(\text{Min}(|x|, |y|))$.

The distance between two strings can be also evaluated by the length of maximal equal prefix $d_{LP}$ (the analysis will be stopped at the first difference between strings).

$$d_{LP}(x, y) = \Sigma_{i=1}^{x=y} \Delta(x_i, y_i) \quad (4.4)$$

The comparison between those functions is shown in table 4.1. It shows string distances between ”qwerty” and the strings in each column, computed using distance measures.

<table>
<thead>
<tr>
<th>Function</th>
<th>”qwerty”</th>
<th>”qeerty”</th>
<th>”qweryt”</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_{\text{strict}}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$d_{Ham}$</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$d_{LP}$</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

### 4.1.2 Fitness value evaluation

Using functions described in section 4.1 we can specify several fitness functions for evaluating an individual.

Assume we have our training data represented as a collection of input/output sequences (the size of a collection is $n$). We also have output strings produced by the individual (table 4.2).
Table 4.2: Measuring fitness value

<table>
<thead>
<tr>
<th>Input</th>
<th>Expected output</th>
<th>Produced output</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_{n_0}$</td>
<td>$O_{n_0}^{expected}$</td>
<td>$O_{n_0}^{produced}$</td>
<td>$distance_{n_0}$</td>
</tr>
<tr>
<td>$I_{n_1}$</td>
<td>$O_{n_1}^{expected}$</td>
<td>$O_{n_1}^{produced}$</td>
<td>$distance_{n_1}$</td>
</tr>
<tr>
<td>$I_{n_...}$</td>
<td>$O_{n...}^{expected}$</td>
<td>$O_{n...}^{produced}$</td>
<td>$distance_{n...}$</td>
</tr>
<tr>
<td>$I_{n_n}$</td>
<td>$O_{n_n}^{expected}$</td>
<td>$O_{n_n}^{produced}$</td>
<td>$distance_{n_n}$</td>
</tr>
</tbody>
</table>

Our task is to measure how ”far” are the strings generated by the individual from the expected strings.

We can compose a fitness function by using the Hamming distance ($d_{Ham}$). In this case fitness value can be computed as

$$Fv = \sum_{i=1}^{n} (l_i - d_{Ham}(O_{n_i}^{expected}, O_{n_i}^{produced}))$$

(4.5)

where $n$ is a number of given data and $l$ is a length of $O_{n_i}^{expected}$.

In case we use $d_{LP}$ for measuring the distance the fitness value can be defined as sum of the lengths for all sequences.

$$Fv = \sum_{i=1}^{n} (d_{LP}(O_{n_i}^{expected}, O_{n_i}^{produced}))$$

(4.6)

Those two functions were implemented (see chapter 5).

4.2 Chromosomal encoding. Genetic operators

To code FSM we will use a chromosomal structure, which memorizes all states of the graph and transitions between them (applied according to input symbols). In our case chromosome is coded in binary alphabet.

Different individual representations can totally change the way of evolution.
4.2.1 Restrictions

To be able to represent FSM as a binary chromosome several restrictions are required.

No final state. According to definition 2.8 and definition 2.9 FSM does not have a final state. FSM finishes it’s work then proceeds the input string to the end.

Initial state. FSM must have only one initial state and this state is always labeled as '0'.

Deterministic. FSM has only one initial state and only one possible transition for each input value. (For example, a situation, where there exist both transitions 'a/1' and 'a/0' is not allowed).

Complete. For each state and each input symbol, there must be one edge.

4.2.2 Moore machines

Moore machine with fixed number of states

Let’s have a target machine $M_o$ with the number of states in the target machine known in advance ($n$ states).

Input alphabet: $\Sigma = \{i_0, \ldots, i_{k-1}\}$.

Output alphabet: $\Delta = \{o_0, \ldots, o_{m-1}\}$. To code one symbol of output alphabet requires $\lceil \log_2 m \rceil$ bits.

Set of states: $Q = \{q_0, \ldots, q_{n-1} \}$. To code one state requires $\lceil \log_2 n \rceil$ bits.

To store information about one state requires one section of the chromosome (presented in table 4.3).

To store the information about state $q_j$ we need to store the value associated with that state (output value) $o_j$ and for all transitions by reading symbol $i_0$ we save the target state $q_k$.

The number of bits required for storing one section can be counted using equation 4.7.
Table 4.3: One section of chromosome for storing the Moore machine with fixed number of states

<table>
<thead>
<tr>
<th>State $q_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q^0$</td>
</tr>
</tbody>
</table>

Length$_{section} = \lceil \log_2 m \rceil + k \lceil \log_2 n \rceil$; \hspace{1cm} (4.7)

The structure required to store the whole Mo FSM is represented in table 4.4.

Table 4.4: Chromosomal representation of the Moore machine with fixed number of states

<table>
<thead>
<tr>
<th>State $q_0$</th>
<th>...</th>
<th>State $q_{n-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q^0_0$</td>
<td>$q^0_1$</td>
<td>$q^{k-1}_0$</td>
</tr>
</tbody>
</table>

The number of bits required for storing the whole chromosome can be counted using equation 4.8.

Length$_{chromosome} = n \cdot (\lceil \log_2 m \rceil + k \lceil \log_2 n \rceil)$; \hspace{1cm} (4.8)

**Example 4.1.** Let’s take a look at a Moore machine with transition diagram represented on figure 4.1.

![Figure 4.1: A Moore machine represented as transition diagram](image-url)

We have 4 states $Q = \{0, 1, 2, 3\}$, the input alphabet contains 2 symbols $\Sigma = \{a, b\}$ and the output alphabet 2 symbols $\Delta = \{0, 1\}$. 
Thus we need 20 bits to store this FSM (equation 4.8): $4 \cdot (\lceil \log_2 2 \rceil + \lceil \log_2 4 \rceil \cdot 2) = 20$ bits.
Table 4.5: Chromosomal representation of a Moore machine with fixed number of states

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Dec. representation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Bin. representation

Mutation/Crossover. During the mutation/crossover process every part of the chromosome may be affected. This means that if we change one random bit of the chromosome or swap parts of chromosomes

- the output value associated with state (if one of the bits that represent the state value is changed),
- the target state of the concrete transition (if one of the bits that represent transition value is changed)

may be modified.

Moore machine. Adopting the number of states

Let’s have a target machine Mo with unknown number of states in the target machine, but with known maximal possible number of states $n$.

Input alphabet: $\Sigma = \{i_0, \ldots, i_{k-1}\}$.

Output alphabet: $\Delta = \{o_0, \ldots, o_{m-1}\}$. To code one symbol of the output alphabet requires $\lceil \log_2 m \rceil$ bits.

Set of states: $Q = \{q_0, \ldots, q_{n-1}\}$. To code one state requires $\lceil \log_2 n \rceil$ bits.

To store information about one state requires one section of the chromosome (presented in table 4.6). It is similar to the section represented in table 4.3, but here the one extra tag bit $a^j$ is added, to store information if state $q^j$ is active or not. If state is ‘not active’ FSM does not use it while running. The number of bits required for storing one section can be counted using equation 4.9.

$$
\text{Length}_{section} = \lceil \log_2 m \rceil + k \lceil \log_2 n \rceil + 1; \quad (4.9)
$$

The structure required to store the whole Mo FSM is represented in table 4.7.
Table 4.6: One section of chromosome for storing the Moore machine with unknown number of states

<table>
<thead>
<tr>
<th>State $q_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q^0$</td>
</tr>
</tbody>
</table>

Table 4.7: Chromosomal representation of the Moore machine. Adopting the number of states

<table>
<thead>
<tr>
<th>State $q_0$</th>
<th>...</th>
<th>State $q_{n-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q^0$</td>
<td>$q^1$</td>
<td>$q^2$</td>
</tr>
</tbody>
</table>

The number of bits required for storing the whole chromosome can be counted using equation 4.10.

$$Length_{chromosome} = n \cdot (\lceil \log_2 m \rceil + k\lceil \log_2 n \rceil + 1);$$  \hspace{1cm} (4.10)

**Example 4.2.** Let's take a look to Moore machine with transition diagram represented on figure 4.1: $Q = \{0, 1, 2, 3\}$, $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$.

$$4 \cdot (\lceil \log_2 2 \rceil + \lceil \log_2 4 \rceil \cdot 2 + 1) = 24 \text{ bits (equation 4.10) is required to store the FSM.}$$

It is possible to "turn off" some states, by setting the tag bit to 0. See, for example, state '2' on figure 4.2.

![Moore machine diagram](image)

Figure 4.2: A Moore machine with an unreachable state
Table 4.8: Example. Chromosomal representation of the Moore machine. Adopting the number of states

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
<th>a</th>
<th>b</th>
<th>Dec. repres.</th>
<th></th>
<th>a</th>
<th>b</th>
<th>Bin. repres.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

In that case there appear some problems like

- **Non-complete transitions** (if state '2' is turned off then transition from state '1' labeled 'b' has no target state);
- **Unaccessible states** (state '3');

The problem of unaccessible states can be solved by using algorithm 5.1.

**Partial output.** To solve the problem of non–complete transitions we will use the method based on partial solution. FSM will stop working when it reaches the state it cannot leave and will produce partial output (during ‘original’ work-flow FSM can stop only if an input string is processed).

**Example 4.3.** FSM Mo as defined by transition diagram presented in figure 4.2. For example, input string \( w = bbaabaab \). The process of work:

\[ \rightarrow 0/1 \xrightarrow{b} 0/1 \xrightarrow{b} 0/1 \xrightarrow{a} 1/1 \xrightarrow{a} 1/1 \xrightarrow{b} ?? \]

This FSM is able to process only 'bbaa'. So the partial output will be '11111'.

**Mutation/Crossover.** During mutation/crossover process every part of chromosome may be affected. Mutation/crossover operator can

- change a value associated with a state (if one of the bits that represent the state value is changed),
- change a state transition (if one of the bits that represent the transition value is changed),
- turn off a state (if the tag bit is affected),
- turn on a state (if the tag bit is affected).
Finite Acceptor

In section 2.3.4 is shown how it is possible to represent FA as a FSM with output (Moore machine). This gives a possibility to use methods described above to infer FA.

4.2.3 Mealy machines

Mealy machine with fixed number of states

Let’s have a target machine $Me$ with exactly $n$ states in the target machine. Input and output alphabets are observed from input/output sequences.

Input alphabet: $\Sigma = \{i_0, \ldots, i_{k-1}\}$.

Output alphabet: $\Delta = \{o_0, \ldots, o_{m-1}\}$. To code one symbol of the output alphabet requires $\lceil \log_2 m \rceil$ bits.

Set of states: $Q = \{q_0, \ldots, q_{n-1}\}$. To code one state requires $\lceil \log_2 n \rceil$ bits.

To store information about one state requires one section of the chromosome (presented in table 4.9). For the state $q_j$ we store information about transitions (output value $o^0$ and the target state $q^0$) for all input symbols $i$.

Table 4.9: One section of chromosome for storing the Mealy machine with fixed number of states

<table>
<thead>
<tr>
<th>State $q_j$</th>
<th>$q^0$</th>
<th>$q^1$</th>
<th>...</th>
<th>$q^{k-1}$</th>
<th>$q^{k-1}$</th>
</tr>
</thead>
</table>

The number of bits required for storing one section can be counted using equation 4.11.

$$Length_{section} = k \cdot (\lceil \log_2 m \rceil + \lceil \log_2 n \rceil); \quad (4.11)$$

The structure required to store the whole $Me$ FSM is represented in table 4.10.

The number of bits required for storing whole chromosome can be counted using equation 4.12.

$$Length_{chromosome} = n \cdot k \cdot (\lceil \log_2 m \rceil + \lceil \log_2 n \rceil); \quad (4.12)$$
Table 4.10: Chromosomal representation of the Mealy machine with fixed number of states

<table>
<thead>
<tr>
<th>State 0</th>
<th>...</th>
<th>State n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_0^0$</td>
<td>...</td>
<td>$o_{n-1}^k$</td>
</tr>
<tr>
<td>$q_0^0$</td>
<td>...</td>
<td>$q_{n-1}^k$</td>
</tr>
</tbody>
</table>

Example 4.4. Let’s take a look at a Mealy machine with transition diagram represented on figure (figure 4.3). $Q = \{0, 1, 2, 3\}$, $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$.

![Transition diagram](image)

Figure 4.3: A Mealy machine represented as transition diagram

We require $4 \cdot 2 \cdot (\lceil \log_2 2 \rceil + \lceil \log_2 4 \rceil) = 24$ bits (equation 4.12) to store the FSM.

Table 4.11: Example. Chromosomal representation of the Mealy machine with fixed number of states

<table>
<thead>
<tr>
<th>State 0</th>
<th>State 1</th>
<th>State 2</th>
<th>State 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>
**Mutation/Crossover.** During mutation/crossover process every part of the chromosome may be affected. This means that if we change one random bit of the chromosome or swap parts of chromosomes

- the value associated with transition (if one of the bits that represent the output value is changed),
- the target state of the concrete transition (if one of the bits that represent the transition value is changed)

may be modified.

### 4.3 Initialization. Reduction of invalid individuals

With the term *invalid individuals* we will denote those individuals where process of coding or process of decoding is incorrect. Several reasons can cause this problem: one of them is described in example 4.5, another — if rules described in subsection 4.2.1 are not fulfilled.

During the population initialization stage, many invalid individuals are created due to random generation of chromosomes. According to equation 4.7, equation 4.9 and equation 4.11 it is possible that some genotypes cannot be correctly decoded to FSM (figure 4.4).

![Figure 4.4: Coding/decoding process from binary genotype to FSM](image)

**Example 4.5.** Assume FSM $M_0$ has 3 states $Q = \{0, 1, 2\}$. 4 bits are required to store this information. During the process of decoding a pair of bits is interpreted as shown in table 4.12. The machine represented by such chromosome will be invalid.
Table 4.12: Error in decoding process

<table>
<thead>
<tr>
<th>bits</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>$q_0$</td>
</tr>
<tr>
<td>01</td>
<td>$q_1$</td>
</tr>
<tr>
<td>10</td>
<td>$q_2$</td>
</tr>
<tr>
<td>11</td>
<td>error</td>
</tr>
</tbody>
</table>

To solve the problem described in (example 4.5) we will use intermediate decimal coding (figure 4.5). In this case it is possible to generate correct decimal representations of FSM (we know the exact upper value of each decimal gene) and after that will get correct binary representation.

The decimal intermediate coding is also used in decoding during evaluation process.

Invalid individuals also appear as a result of mutation and crossover processes (if a gene is destroyed by operation). In such case the fitness value of that individual will be 0.

Figure 4.5: Coding/decoding process from binary genotype to FSM using decimal representation
Chapter 5

Implementation

5.1 Experimental software description

The methods described in chapter 4 were implemented in a tool GeSM, which is written in Java SE 2.0. This tool allows to generate form input/output data:

- a Moore machine with fixed number of states;
- a Moore machine with unknown number of states (the maximal number of states is given);
- a Mealy machine with fixed number of states;

It also allows to:

- change parameters of evolution (like size of population, number of generations, mutation probability etc);
- rerun the program for statistical purposes;
- generate random FSM and random input data (required for some tests);

The program contains several packages (figure 5.1):

- `fsm` describes the structure of FSM (subsection 5.1.1).
- `world` contains two packages: `evolution` and `beings`. `evolution` contains the implementation of evolution process (subsection 5.1.3). `beings` describes the process of coding/decoding of FSM (subsection 5.1.2).
- `ui` contains user interface.
- `logging` is required for logging and output formatting.
5.1.1 FSM package

To represent FSM we will use several classes. The whole structure of classes you can see on the class diagram (figure 5.2).

Class Alphabet is required for storing information about input and output alphabets of the machine. Class State is used for storing data of one concrete state, class Transition is for storing data of one transition.

Abstract class FiniteStateMachine describes the general structure of FSM. The implementation depends on machine type.

Finite automaton

For representing FA we need to use such fields as State.initial, State.final. Fields State.stateValue (we don’t have any output) and State.active are not required.
Moore machine

In the Moore machine the output depends on state value, so we will use fields State.stateValue, but the field Transition.outputSymbol is not required. In case we want to turn on/off some states we will use field State.active. The Moore machine does not have the final state, so the field State.final is never used.

Mealy machine

In the Mealy machine the output function depends on the current state and transition, so here we require the field Transition.outputSymbol to produce output. The field State.stateValue is not used.

This kind of structure can represent several types of FSM.

5.1.2 Beings package

The package Beings (figure 5.3) is required to transform different types of FSM to individuals. The class Individual is an abstract class, which con-
contains descriptions of several methods like `Individual.evaluate`, `Individual.decode`, `Individual.generate`. Implementation of those methods is unique for each type of individual.

In this implementation we have a class `MoConst` for representing the Moore machine with constant number of states, `MoMax` for representing the Moore machine with undefined number of states and class `MeConst` for presenting the Mealy machine with constant number of states.

![Diagram of class relationships]

Figure 5.3: Beings package

### 5.1.3 Evolution package

Package `Evolution` contains classes required for implementing the evolution process.

Class `FitnessFunctions` implements two types of fitness functions: based on hamming distance and on length of maximal prefix (subsection 4.1.2).

Class `Mutation` implements the methods of mutation (section 3.3.1): simple one point mutation and inversion.

One point crossover (section 3.3.2) is implemented in class `Crossover`.

Class `RouletteWheelSelection` contains the realization of selection process (section 3.3.3).
Class **Parameter** is required for storing several parameters like: mutation probability, type of the FSM, number of states in the machine, type of fitness function, type of mutation, number of generations, size of population, names of input and output files, number of runs (if program is running several times for statistical purpose), size of input/output data (for generating random data).

- **Crossover**
  - `onePointCrossover(parent1 : Vector, parent2 : Vector) : void`

- **Mutation**
  - `mutate(individual : String) : String`
  - `invert(individual : String) : String`

- **Environment**
  - `inputAlphabet : Alphabet`
  - `outputAlphabet : Alphabet`
  - `bestFitness : float`
  - `inputData : Vector`
  - `outputData : Vector`
  - `inDataR : BufferedReader`
  - `outDataR : BufferedReader`
  - `inDataW : PrintWriter`
  - `outDataW : PrintWriter`
  - `logger : Logger`

- **Evolution**
  - `population : int`
  - `newPopulation : Vector`
  - `solvedIn : int`
  - `solution : Individual`
  - `fillEvolutionTrace() : void`
  - `printEvolutionTrace() : void`
  - `evolute() : void`
  - `printSolution() : void`
  - `selection() : void`
  - `breeding() : void`
  - `mutate() : void`
  - `evaluatePop() : void`
  - `generateInitialPop() : void`
  - `population : int`
  - `newPopulation : Vector`
  - `solvedIn : int`
  - `solution : Individual`

- **FitnessFunctions**
  - `hammingDistance(string1 : String, string2 : String) : int`
  - `maxLengthCorrect(string1 : String, string2 : String) : int`

- **RouletteWheelSelection**
  - `RouletteWheelSelection() : void`
  - `searchInd() : void`
  - `getIndividualForBreeding() : void`

- **Parameter**
  - `defineLog() : void`
  - `readData() : void`
  - `printData() : void`
  - `create(arg2 : int) : void`
  - `generateTestData() : void`

- **Evolution**
  - `printSolution() : void`

**Figure 5.4: Evolution package**

Class **Environment** contains information about input/output data and the log file. It is also required to create random FSM and random input/output pairs.

Class **Evolution** describes the whole process of evolution.
5.2 Evolutionary trace

To illustrate the process of evolution we will use a figure that we call *evolutionary trace* (for example see figure 5.5).

Each row corresponds to the best individual of the current population. Each column corresponds to one input/output pair. Color of cell shows the error between the given output string and the string generated by the machine. White color denotes no error (error is 0). Black — that error is more than 50% of length.

![Evolutionary Trace](image)

**Figure 5.5: Evolutionary trace**

5.3 Post processing

One of the most important parameters of evolution is the *number of states*. The algorithm generates a FSM with exact number of states. But it is possible that some of the states will be unreachable. So we need to remove the unreachable states before printing out the result (definition 2.4).
Algorithm 5.1. *Removing unreachable states*

Create array reachableStates;
Add initial state to array reachableStates;
while(there are some unchecked states in array reachableStates) {
    check all transitions from current state;
    if (end state of a transition is not in reachableStates) {
        add end point of a transition to array reachableStates;
    }
    next state;
}

Example 5.1. *Let us have a Moore machine with 6 states represented in table 5.1.*

<table>
<thead>
<tr>
<th>number: value: i?: f?</th>
<th>Out symbol/next state</th>
<th>Out symbol/next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0: 0: i</td>
<td>/0</td>
<td>/2</td>
</tr>
<tr>
<td>#1: 0</td>
<td>/2</td>
<td>/5</td>
</tr>
<tr>
<td>#2: 0</td>
<td>/2</td>
<td>/5</td>
</tr>
<tr>
<td>#3: 1</td>
<td>/2</td>
<td>/1</td>
</tr>
<tr>
<td>#4: 0</td>
<td>/1</td>
<td>/0</td>
</tr>
<tr>
<td>#5: 0</td>
<td>/3</td>
<td>/5</td>
</tr>
</tbody>
</table>

*As you can see the state number 4 is unreachable from the initial state 0. To remove it we can use algorithm 5.1.*

reachableStates[]
reachableStates[0]
checking 0 reachableStates[0, 2]
checking 2 reachableStates[0, 2, 5]
checking 5 reachableStates[0, 2, 5, 3]
checking 3 reachableStates[0, 2, 5, 3, 1]
checking 1 reachableStates[0, 2, 5, 3, 1]
Chapter 6

Experiments

6.1 Learning Random Finite State Machines

6.1.1 Getting the training data

To show how this approach works we need training data. One way would be to generate random data for experiments.

Algorithm 6.1. (The process of getting training data)

Generate random FSM of \( t \) type with \( n \) states;
Reset the FSM to start state;
Produce a random input sequence;
Feed the input sequence to the FSM and collect the corresponding output sequence;
Store input/output sequences;
Store number of states \( n \);
Store type of the machine \( t \);

The stored input/output sequences, number of states and type of the machine are used for experiments.

6.1.2 Equivalence of Mealy and Moore machines

The equivalence of Mealy and Moore machines (subsection 2.3.2) allows us to use both methods described in subsection 4.2.2 and subsection 4.2.3.

Task Generate Moore and Mealy machines for given training data.
Training data  To get training data let us use algorithm 6.1. The training data is presented in table 6.1. To generate the Moore machine we shall use the data as it is represented in table, while to generate Mealy machine we need to remove the prefix (in this case the first symbol ’0’).

Table 6.1: Training data. Equivalence of Mealy and Moore machines

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbabbba</td>
<td>00011110</td>
</tr>
<tr>
<td>abaaaaa</td>
<td>01101010</td>
</tr>
<tr>
<td>babbbaa</td>
<td>00100010</td>
</tr>
<tr>
<td>baaabaa</td>
<td>00111010</td>
</tr>
<tr>
<td>bbabbab</td>
<td>00011100</td>
</tr>
<tr>
<td>bbabbbb</td>
<td>00011111</td>
</tr>
<tr>
<td>bababaa</td>
<td>00101011</td>
</tr>
<tr>
<td>baaabab</td>
<td>00111011</td>
</tr>
<tr>
<td>bababaa</td>
<td>00101101</td>
</tr>
<tr>
<td>abbbaba</td>
<td>01111001</td>
</tr>
</tbody>
</table>

Parameters  Mutation: One point mutation; Mutation probability: 0.04; Size of population: 50; Number of generations: 20; Fitness function type: Hamming distance; Selection type: Roulette Wheel Selection; Crossover type: One point crossover; Generating: Smart generation; 5 states for Moore Machine; 4 states for Mealy machine;

Results  The results of these runs are presented in table 6.2.

This experiment shows that both methods described in subsection 4.2.2 and subsection 4.2.3 can be used for the same data.
### 6.1.3 Adjusting the parameter 'number of states'

One of the most important parameters for the evolution process is the number of states. This parameter affects the speed and quality of the process. During this experiment we will show how this parameter affect to the quality of solution.

**Task**  We are searching for a Moore machine with 5 states. All parameters of evolutions will be the same for all runs, only the 'number of states' (from 3 to 8) will be changing. Program will run 30 times for each value of 'number of states'.

<table>
<thead>
<tr>
<th>Run</th>
<th>Mealy machine</th>
<th>Moore machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generation</td>
<td>Fitness</td>
</tr>
<tr>
<td>0</td>
<td>182</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>38</td>
<td>100.0</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>100.0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>82.85715</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>100.0</td>
</tr>
<tr>
<td>5</td>
<td>144</td>
<td>100.0</td>
</tr>
<tr>
<td>6</td>
<td>197</td>
<td>100.0</td>
</tr>
<tr>
<td>7</td>
<td>83</td>
<td>100.0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>80.0</td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>100.0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>77.14286</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>100.0</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>71.42857</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>74.28571</td>
</tr>
<tr>
<td>14</td>
<td>198</td>
<td>100.0</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>72.85714</td>
</tr>
<tr>
<td>16</td>
<td>129</td>
<td>100.0</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>72.85714</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>87.14285</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>72.85714</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>70.0</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>87.14285</td>
</tr>
</tbody>
</table>
Training data   Let us use algorithm 6.1 to get *training data*. Training data is presented in table 6.3.

Table 6.3: Training data. Number of states

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbabbabbbb</td>
<td>101010111111</td>
</tr>
<tr>
<td>baaabaaaba</td>
<td>101001100111</td>
</tr>
<tr>
<td>aaaaaaabbb</td>
<td>110000000010</td>
</tr>
<tr>
<td>bbabbbbba</td>
<td>101010111110</td>
</tr>
<tr>
<td>bbaabaabba</td>
<td>101001101010</td>
</tr>
<tr>
<td>baaaaabaa</td>
<td>101000001100</td>
</tr>
<tr>
<td>abbababaab</td>
<td>111111111000</td>
</tr>
<tr>
<td>abbbabbaab</td>
<td>111110101010</td>
</tr>
<tr>
<td>baaabaabbb</td>
<td>101001101010</td>
</tr>
<tr>
<td>babbabaaa</td>
<td>101110101000</td>
</tr>
<tr>
<td>abbbbbbbaa</td>
<td>111111111110</td>
</tr>
<tr>
<td>baabbbabba</td>
<td>101010101010</td>
</tr>
<tr>
<td>babbbaaab</td>
<td>101010110101</td>
</tr>
<tr>
<td>babbaaaaa</td>
<td>101011100000</td>
</tr>
<tr>
<td>ababbaabbb</td>
<td>111010101010</td>
</tr>
</tbody>
</table>

Parameters   Mutation: One point mutation; Mutation probability: 0.04; Size of population: 90; Number of generations: 201; Fitness function type: Hamming distance; Selection type: Roulette Wheel Selection; Crossover type: One point crossover; Generating: Smart generation;

Results   The results of these runs are presented in table 6.4 and table 6.5.
Table 6.4: Choice of number of states. Results

<table>
<thead>
<tr>
<th>States</th>
<th>3 states</th>
<th>4 states</th>
<th>5 states</th>
<th>6 states</th>
<th>7 states</th>
<th>8 states</th>
</tr>
</thead>
<tbody>
<tr>
<td>82,4242</td>
<td>94,5455</td>
<td>97,5758</td>
<td>100,000</td>
<td>100,000</td>
<td>90,9091</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>100,000</td>
<td>92,1212</td>
<td>96,9697</td>
<td>100,000</td>
<td>97,5758</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>90,9091</td>
<td>92,1212</td>
<td>92,1212</td>
<td>100,000</td>
<td>90,9091</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>100,000</td>
<td>92,1212</td>
<td>100,000</td>
<td>85,4545</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>92,1212</td>
<td>95,7576</td>
<td>90,9091</td>
<td>94,5455</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>87,2727</td>
<td>94,5455</td>
<td>100,000</td>
<td>100,000</td>
<td>90,9091</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>90,9091</td>
<td>92,1212</td>
<td>95,7576</td>
<td>91,5152</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>94,5455</td>
<td>90,9091</td>
<td>94,5455</td>
<td>90,9091</td>
<td>92,1212</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>89,6970</td>
<td>92,1212</td>
<td>96,3636</td>
<td>87,2727</td>
<td>92,7273</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>94,5455</td>
<td>100,000</td>
<td>100,000</td>
<td>99,3939</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>89,6970</td>
<td>86,6667</td>
<td>100,000</td>
<td>100,000</td>
<td>93,9394</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>92,1212</td>
<td>92,7273</td>
<td>92,1212</td>
<td>94,5455</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>94,5455</td>
<td>92,1212</td>
<td>90,9091</td>
<td>93,9394</td>
<td>99,3939</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>94,5455</td>
<td>94,5455</td>
<td>100,000</td>
<td>94,5455</td>
<td>94,5455</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>100,000</td>
<td>100,000</td>
<td>92,1212</td>
<td>91,5152</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>87,2727</td>
<td>90,9091</td>
<td>93,9394</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>90,9091</td>
<td>89,6970</td>
<td>100,000</td>
<td>100,000</td>
<td>89,6970</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>89,6970</td>
<td>94,5455</td>
<td>100,000</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>94,5455</td>
<td>100,000</td>
<td>88,4849</td>
<td>90,9091</td>
<td>96,9697</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>90,9091</td>
<td>92,1212</td>
<td>94,5455</td>
<td>93,3333</td>
<td>92,1212</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>90,9091</td>
<td>92,1212</td>
<td>92,1212</td>
<td>93,9394</td>
<td>90,9091</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>90,3030</td>
<td>90,3030</td>
<td>90,9091</td>
<td>95,1515</td>
<td>89,6970</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>90,9091</td>
<td>92,7273</td>
<td>92,1212</td>
<td>89,6970</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>100,000</td>
<td>100,000</td>
<td>100,000</td>
<td>100,000</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>100,000</td>
<td>95,1515</td>
<td>90,9091</td>
<td>92,7273</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>92,1212</td>
<td>87,2727</td>
<td>100,000</td>
<td>100,000</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>100,000</td>
<td>92,1212</td>
<td>100,000</td>
<td>96,3636</td>
<td>90,9091</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>89,6970</td>
<td>94,5455</td>
<td>100,000</td>
<td>94,5455</td>
<td>90,9091</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>100,000</td>
<td>100,000</td>
<td>100,000</td>
<td>100,000</td>
<td>89,0909</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>100,000</td>
<td>92,1212</td>
<td>100,000</td>
<td>92,7273</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>82,4242</td>
<td>94,5455</td>
<td>92,1212</td>
<td>86,6667</td>
<td>95,1515</td>
<td>91,5152</td>
<td></td>
</tr>
</tbody>
</table>
In table 6.4 each cell shows the fitness value (in %) of the found solution for that run of program.

In table 6.5 each cell shows how many FSM with this were generated.

Table 6.5: Choice of number of states. Frequencies

<table>
<thead>
<tr>
<th>Fitness value %</th>
<th>3 states</th>
<th>4 states</th>
<th>5 states</th>
<th>6 states</th>
<th>7 states</th>
<th>8 states</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>83</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>85</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>86</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>87</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>88</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>89</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>91</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>92</td>
<td>0</td>
<td>8</td>
<td>12</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>93</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>94</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>95</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>96</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>98</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>99</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>7</td>
</tr>
</tbody>
</table>

Using table 6.5 we can draw graph (figure 6.1).

This experiment shows that:

- Choosing the number of states less than required gives bad results;
- An ideal solution is to choose 1–2 states more than required (gives more freedom);
- Adding too many extra states increases the run time;
6.1.4 Moore machine with unknown number of states

To continue the previous experiment we will use the method described in subsection 4.2.2.

Task  We want to build the Moore machine with unknown number of states (knowing the maximal number). let us assume that the maximal number of states is 8 (or in the second run — 7). The training data is in table 6.3.

Parameters  Mutation: One point mutation; Mutation probability: 0.04; Size of population: 90; Number of generations: 201; Fitness function type: Hamming distance; Selection type: Roulette Wheel Selection; Crossover type: One point crossover; Generating: Smart generation;

Results  The results of those runs are presented in table 6.6. If during the process of evolution the best solution was found (with fitness value = 100%), then the number in corresponding ‘Gen’ column shows the number of generations.
Table 6.6: Generating a Moore machine with unknown number of states.

Results

<table>
<thead>
<tr>
<th>Run</th>
<th>Gen</th>
<th>Fitness</th>
<th>number of states</th>
<th>Gen</th>
<th>Fitness</th>
<th>number of states</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>201</td>
<td>87,2727</td>
<td>6</td>
<td>74</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>201</td>
<td>95,1515</td>
<td>6</td>
<td>84</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>201</td>
<td>90,9091</td>
<td>6</td>
<td>201</td>
<td>88,4849</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>201</td>
<td>87,8788</td>
<td>6</td>
<td>201</td>
<td>88,4849</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>201</td>
<td>89,0909</td>
<td>7</td>
<td>201</td>
<td>97,5758</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>201</td>
<td>92,1212</td>
<td>7</td>
<td>55</td>
<td>100,0000</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>201</td>
<td>87,2727</td>
<td>5</td>
<td>87</td>
<td>100,0000</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>201</td>
<td>94,5455</td>
<td>6</td>
<td>201</td>
<td>88,4849</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>201</td>
<td>95,7576</td>
<td>7</td>
<td>55</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>201</td>
<td>92,1212</td>
<td>7</td>
<td>81</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>201</td>
<td>94,5455</td>
<td>7</td>
<td>142</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>181</td>
<td>100,0000</td>
<td>6</td>
<td>201</td>
<td>90,9091</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>201</td>
<td>89,6970</td>
<td>7</td>
<td>201</td>
<td>90,9091</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>201</td>
<td>95,7576</td>
<td>6</td>
<td>120</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>201</td>
<td>83,6364</td>
<td>7</td>
<td>133</td>
<td>100,0000</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>201</td>
<td>90,9091</td>
<td>6</td>
<td>201</td>
<td>90,9091</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>201</td>
<td>92,1212</td>
<td>7</td>
<td>201</td>
<td>91,5152</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>201</td>
<td>87,2727</td>
<td>7</td>
<td>201</td>
<td>89,6970</td>
<td>7</td>
</tr>
<tr>
<td>18</td>
<td>201</td>
<td>87,2727</td>
<td>6</td>
<td>102</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>19</td>
<td>201</td>
<td>86,0606</td>
<td>6</td>
<td>201</td>
<td>87,2727</td>
<td>8</td>
</tr>
<tr>
<td>20</td>
<td>201</td>
<td>87,8788</td>
<td>6</td>
<td>201</td>
<td>99,3939</td>
<td>7</td>
</tr>
<tr>
<td>21</td>
<td>201</td>
<td>88,4849</td>
<td>5</td>
<td>201</td>
<td>87,2727</td>
<td>7</td>
</tr>
<tr>
<td>22</td>
<td>201</td>
<td>89,6970</td>
<td>7</td>
<td>201</td>
<td>96,9697</td>
<td>8</td>
</tr>
<tr>
<td>23</td>
<td>69</td>
<td>100,0000</td>
<td>7</td>
<td>201</td>
<td>87,8788</td>
<td>6</td>
</tr>
<tr>
<td>24</td>
<td>201</td>
<td>89,0909</td>
<td>7</td>
<td>201</td>
<td>87,2727</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>201</td>
<td>94,5455</td>
<td>7</td>
<td>17</td>
<td>100,0000</td>
<td>7</td>
</tr>
<tr>
<td>26</td>
<td>53</td>
<td>100,0000</td>
<td>6</td>
<td>201</td>
<td>85,4545</td>
<td>7</td>
</tr>
<tr>
<td>27</td>
<td>201</td>
<td>86,6667</td>
<td>7</td>
<td>201</td>
<td>92,1212</td>
<td>6</td>
</tr>
<tr>
<td>28</td>
<td>201</td>
<td>89,0909</td>
<td>7</td>
<td>201</td>
<td>84,8485</td>
<td>8</td>
</tr>
<tr>
<td>29</td>
<td>201</td>
<td>86,0606</td>
<td>6</td>
<td>57</td>
<td>100,0000</td>
<td>8</td>
</tr>
</tbody>
</table>
During the experiment where maximal possible number of states was 7 only 3 maximal solutions were found: 2 solutions with 6 states and 1 with 7 states.

During the experiment where maximal possible number of states was 8, 12 maximal solutions were found: 8 solutions with 7 states and 4 with 8 states.

### 6.2 Learning Mealy machine. Parity checker

**Task** To design a Mealy Machine which takes binary input and produces output 0 when the current string (so far) has odd parity.

**Training data** Training data is presented in table 6.7.

Table 6.7: Parity checker. Training data

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010101</td>
<td>10011001</td>
</tr>
<tr>
<td>10101010</td>
<td>00110011</td>
</tr>
<tr>
<td>00110011</td>
<td>11011101</td>
</tr>
<tr>
<td>00111100</td>
<td>11010111</td>
</tr>
<tr>
<td>01001100</td>
<td>10001000</td>
</tr>
<tr>
<td>10011001</td>
<td>00010001</td>
</tr>
<tr>
<td>11001001</td>
<td>01110001</td>
</tr>
</tbody>
</table>

**Parameters** Mutation: One point mutation; Mutation probability: 0.04; Size of population: 30; Number of generations: 100; Fitness function type: Hamming distance; Selection type: Roulette Wheel Selection; Crossover type: One point crossover; Generating: Smart generation; Number of states 2;

**Results** Solution was found in 3 generations.

### 6.3 Learning Mealy machine. Two unit delay

**Task** To construct a Mealy machine that behaves as a two–unit delay.

\[
r(t) = \begin{cases} 
  s(t - 2) & : t > 2 \\
  0 & : otherwise 
\end{cases}
\]
Figure 6.2: Parity checker. Transition diagram

Sample input/output session:

<table>
<thead>
<tr>
<th>time</th>
<th>123456789</th>
</tr>
</thead>
<tbody>
<tr>
<td>stimulus</td>
<td>000110100</td>
</tr>
<tr>
<td>response</td>
<td>000001101</td>
</tr>
</tbody>
</table>

**Training data** The training data is presented in table 6.8.

Table 6.8: Two unit delay. Training data

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010101</td>
<td>00010101</td>
</tr>
<tr>
<td>10101010</td>
<td>00101010</td>
</tr>
<tr>
<td>00110011</td>
<td>00001100</td>
</tr>
<tr>
<td>00111100</td>
<td>00001111</td>
</tr>
<tr>
<td>01001100</td>
<td>00010011</td>
</tr>
<tr>
<td>10011001</td>
<td>00100110</td>
</tr>
<tr>
<td>11001001</td>
<td>00110010</td>
</tr>
</tbody>
</table>

**Parameters** Mutation: One point mutation; Mutation probability: 0.04; Size of population: 90; Number of generations: 500; Fitness function type: Hamming distance; Selection type: Roulette Wheel Selection; Crossover type: One point crossover; Generating: Smart generation; Number of states 6;

**Results** Solution was found in 79 generations. Evolutionary trace: figure 6.3.
6.4 Learning Moore machine. Pattern recognizer

**Task** To design a Moore machine that functions as pattern recognizer for "aab".

**Training data** The training data is presented in table 6.9.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>babaabaabaab</td>
<td>00000001001001</td>
</tr>
<tr>
<td>bbaabaaaaba</td>
<td>00000100000100</td>
</tr>
<tr>
<td>bbbaabbbaabb</td>
<td>00000100000100</td>
</tr>
<tr>
<td>bbaabbaabaa</td>
<td>00000100000100</td>
</tr>
<tr>
<td>aabbaaabbba</td>
<td>00010000010000</td>
</tr>
</tbody>
</table>

Figure 6.3: Two unit delay. Evolutionary trace

Figure 6.4: Two unit delay. Transition diagram
Parameters  Mutation: One point mutation; Mutation probability: 0.04; 
Size of population: 90; Number of generations: 201; Fitness function type: 
Hamming distance; Selection type: Roulette Wheel Selection; Crossover 
type: One point crossover; Generating: Smart generation; Number of states 
4;

Results  Solution was found in 77 generations. Evolutionary trace: fig-
ure 6.5.

Figure 6.5: Pattern recognizer for ”abb”. Evolutionary trace

The solution generated by the algorithm (figure 6.6) is consistent with the 
training data (table 6.9). But this is not a correct pattern recogniser for 
”aab” (for example for input ”aabab” it will output ”000101”). It happened 
because the training data does not describe such situation. The correct 
solution is represented on figure 6.7.

Figure 6.6: Pattern recognizer for ”abb”, incorrect. Transition diagram
6.5 Learning Moore machine. Up–down counter

**Task**  Design a Moore Machine that will analyze input sequence in binary alphabet \( \Sigma = \{0, 1\} \). Let \( w = s_1 s_2 \ldots s_t \) be an input string

\[
N_0(w) = \text{number of 0 in } w; \quad N_1(w) = \text{number of 1 in } w;
\]

then we have \( |w| = N_0(w) + N_1(w) = t \). The output of the machine should equal: \( r(t) = \lfloor N_1(w) - N_0(w) \rfloor \text{mod} 4 \). The output alphabet \( \Delta = \{0, 1, 2, 3\} \).

Sample input/output session:

- **stimulus** 11011100
- **response** 012123032
Table 6.10: Up–down counter. Training data

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010101</td>
<td>0303030303</td>
</tr>
<tr>
<td>10101010</td>
<td>0101010101</td>
</tr>
<tr>
<td>00110011</td>
<td>0323032303</td>
</tr>
<tr>
<td>00111100</td>
<td>0323012101</td>
</tr>
<tr>
<td>01001100</td>
<td>0303230323</td>
</tr>
<tr>
<td>10011001</td>
<td>0103010301</td>
</tr>
<tr>
<td>11001001</td>
<td>0121010301</td>
</tr>
</tbody>
</table>

**Training data**  Training data is presented in table 6.10.

**Parameters**  Mutation: One point mutation; Mutation probability: 0.04; Size of population: 100; Number of generations: 201; Fitness function type: Hamming distance; Selection type: Roulette Wheel Selection; Crossover type: One point crossover; Generating: Smart generation; Number of states 6;

**Results**  Solution was found in 104 generations. Evolutionary trace: figure 6.10.

Figure 6.9: Up–down counter. Generated machine. Transition diagram
The algorithm generated the machine on figure 6.9. We can minimize it (see example 2.3). As a result we will get the original machine (figure 6.8).

Figure 6.10: Up–down counter. Evolutionary trace
Chapter 7

Conclusions and future work

7.1 Conclusions

In this thesis we have introduced methods for FSM inference based on GA.

First, we presented the basic theory of FSM (chapter 2) and GA (chapter 3). The main definitions and algorithms were described.

Second, we defined several methods for inference of different kinds of FSM: inference of Moore machine with constant number of states, inference of Mealy machines with constant number of states, also a method for adjusting the number of states for the Moore machine was introduced. Different fitness functions were used.

Also several improvements to the algorithms were made: to solve the problem of invalid individuals during initialization stage we used the inner decimal coding (section 4.3), problem of unaccessible states was solved at post–processing stage (section 5.3), for problem of non–complete transitions (appears if we use the method for adjusting number of states for the Moore machine) the method based on partial solution was introduced (section 4.2.2).

An essential part of this work is the implementation of the described methods (chapter 5) using Java. The process of evolution has been illustrated with the evolutionary trace diagram.

The tool GeSM was used for experiments (chapter 6). Several experiment on random data were made to show how this method works for different kinds of FSM, also how the number of states affects the process of evolution. To
show how this tool works with actual tasks from the real world we made experiments with several FSM like parity checker, two unit delay machine, pattern recognizer, up–down counter. During those experiments we tried to reconstruct the FSM having only external description.

The results of the work are the following:

• Method for Mealy and Moore machine inference based on GA. Individual representations and fitness functions were introduced.

• Improvement of initialization process — use of inner decimal representation to reduce invalid individuals.

• Method for solving problem of non–complete transitions based on partial solution.

• Post–processing stage which allows to solve problem of unaccessible states.

• Implementation of presented method in Java.

• Successful testing of method using random data and real–world examples.

7.2 Future work

More work needs to be done to improve the quality of the process. Future research should continue to improve the implementation. Parameters of the evolution and their affect on the presented methods must be explored. Also some other genetic operators should be implemented. The effect of using different operators must be investigated.

To improve the process of adjusting the number of states in case of Moore machines the fitness functions must be modified to take into account the number of states.

The post–processing stage can be improved by adding implementation of the minimization process.

Future work should also include experiments with more sophisticated examples.
Lõplike olekumasinate geneetiline tuletamine
Margarita Spitšakova

Resüümee

Mudeli identifitseerimine on protsess, mis võimaldab tuletada süsteemi sisemise kirjelduse tema välise käitumise põhjal. Süsteemi kirjeldamiseks kasutatakse tõiti lõplikkule olekumasinate.


Käesolev töö koosneb viiest peatükist.


Teises peatükis kirjeldatakse geneetiliste algoritmide põhiprintsiipe.

Kolmas peatükk on töö põhiosa. Selles peatükis käsitletakse geneetiliste algoritmide modifitseerimist, selleks et tuletada lõplikke olekumasinaid ainult nende sisendi–väljundi paaride ja masina tüübi põhjal. Esimeses osas kirjeldatakse funktsioone, mis võimaldavad hinnata konkretse olekumasina sobivust mudeliks. Teises ja kolmas osas vaadeldakse olekumasina kodderimist bitijadana ja sellega kaasnevaid probleeme (vigased inviviidid, saavutamatud olekud ja vigased üleminekud).

Neljandas peatükiks esitatakse lõplike olekumasinate genereerimise prototüüpi.
Nimetatud programm on kirjutatud Javas, ning tema kirjeldamiseks on kasutatud UML diagramme.

Viimases peatükis antakse ülevaade teostatud eksperimentidest: iga ülesande korral kirjeldatakse selle püstitust ja tulemusi. Esimeses osas käsitleetakse eksperimente juhuslike andmetega, teises — reaalse maailma näiteid.
Bibliography


