CMU-CS-82-127

Learning of Construction of Finite Automata From Examples Using Hill-Climbing

RR: Regular set Recognizer

Masaru Tomita
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pennsylvania 15213
May 1982

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Abstract

The problem addressed in this paper is heuristically-guided learning of finite automata from examples. Given positive sample strings and negative sample strings, a finite automaton is generated and incrementally refined to accept all positive samples but no negative samples. This paper describes some experiments in applying hill-climbing to modify finite automata to accept a desired regular language. We show that many problems can be solved by this simple method. We then describe the method how to "re-construct" a finite automaton if the positive and/or negative samples are slightly altered, without starting from the beginning. Finally, we have an actual system, RR: Regular set Recognizer, that learns to recognize a regular set from the samples that are given by a human teacher one by one.

Acknowledgements

I would like to thank Herbert A. Simon and Jaime Carbonell, who are my thesis advisers, for supervising my work; Masakazu Nakanishi, who was my previous adviser when I was at Keio University; Yuichiro Anzai, Takeo Kanade and Pat Langley, for thoughtful comments on an earlier version of this work; and Cynthia Hibbard for helping to produce this document.

Table of Contents

1. Introduction

Consider the following problem:

Describe the property that all strings in the right-list have but no string in the wrong-list has. Does a string (1 1 0 1) have this property? You may answer the question by using any of the following: English, a regular expression, or a finite automaton.[1]

right-list	wrong-list
()	(1 0)
(1)	(1 0 1)
(0)	(0 1 0)
(0 1)	(1 0 1 0)
(1 1)	(1 1 1 0)
(0 0)	(1 0 1 1)
(1 0 0)	(1 0 0 0 1)
(1 1 0)	(1 1 1 0 1 0)
(1 1 1)	(1 0 0 1 0 0 0)
(0 0 0)	(1 1 1 1 1 0 0 0)
(1 0 0 1 0 0)	(0 1 1 1 0 0 1 1 0 1)
(1 1 0 0 0 0 0 1 1 1 0 0 0 0 1)	(1 1 0 1 1 1 0 0 1 1 0)
(1 1 1 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0)	

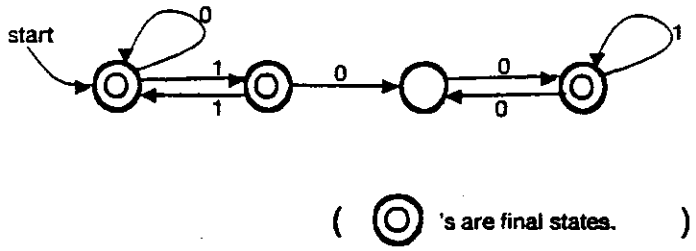
It might be possible to construct the finite automaton by a "typical" schema-filling method (i.e., finding rough property in the samples first, comparing these strings carefully). However, in this paper, we try to construct the finite automaton directly by searching in the problem space (i.e., the set of all finite automata) using hill-climbing, rather than by analyzing the samples carefully. One of the biggest advantages of hill-climbing is its simplicity, that is, we do not have to know our problem space well, while a "typical" schema-filling method requires us to provide all possible schemas, and therefore to know everything about our problem space. We shall see that hill-climbing works much better than expected in our problem space, and in fact solved most of the problems.

1.1 The finite automata used in this paper

We restrict our problem domain to be only over $\{1,0\}^*$. Furthermore, since every non-deterministic finite automaton has an equivalent deterministic finite automaton (see [Hopcroft 79]), we deal only with deterministic finite automata, that is, there is at most one 1-arrow and one 0-arrow from each state. Thus, in this paper, the terms "finite automaton", "automaton" or "machine" all mean "deterministic finite automaton". Given a string s, if there is a transition from the *initial state* to any of the *final states*, then s is *accepted* by the machine, otherwise s is *rejected*. For example, the machine of the sample problem is shown in figure 1-1.

[1] The answer is strings over $(1 + 0)^*$ without an odd number of consecutive 0's AFTER an odd number of consecutive 1's. Therefore (1 1 0 1) has the property.

Figure 1-1: The machine of the sample problem



Each machine with n states is denoted by the following form:

$((A_1, B_1, F_1)\ (A_2, B_2, F_2) \ldots . (A_n, B_n, F_n))$.

Each (A_i, B_i, F_i) corresponds to the state i, and A_i and B_i indicate the destination states of the 0-arrow and the 1-arrow from the state i, respectively. If A_i or B_i is zero, then there is no 0-arrow or 1-arrow from the state i, respectively. F_i indicates whether state i is one of the final states or not. If F_i is equal to 1, the state i is one of the final states. The initial state is always state 1. For instance, figure 1-1 is represented as follows:

((1 2 1)(3 1 1)(4 0 0)(3 4 1)).

1.2 The problem

We now are ready to describe the problem precisely. Given a *right-list* (a set of positive sample strings) and a *wrong-list* (a set of negative sample strings), we can think of the following three tasks:

1. To find a machine that accepts all strings in the right-list but none in the wrong-list.
2. To find a machine with n states that accepts all strings in the right-list but none in the wrong-list.
3. To find the machine with fewest states (*simplest machine*) that accepts all strings in the right-list but none in the wrong-list.

The first task is trivial because one can easily construct a *trivial machine* that accepts exactly all strings in the right-list but nothing else.[2] The second task and the third task are shown to be NP-complete problems by [Gold 74]. We call the second task *construction of finite automata*, and the third task *simplification of finite automata*.

1.3 Past Work

Feldman, Gips, Horning and Reder [Feldman 67] [Feldman 69] built a system that constructs a grammar in BNF from given examples. It takes only positive examples, and its problem domain is context-free languages. We quote a couple of sample runs of this system from [Feldman 69] , to make clear how their system worked.

[2] An example of the trivial machine will be found in section 3-4-2.

Figure 1-2: Sample Strings and BNF grammar produced by Feldman's system

```
------------------------
(b)
(a m b)
(a m a m b)
(a m a m a m b)

S <- b | S1mS
S1 <- a
------------------------
(c d)
(a b d)
(a c b d)
(a a b b d)
(a a c b b d)
(a a a b b b d)
(a a a c b b b d)

S <- S1d
S1 <- aS2 | c
S2 <- S1b | b
------------------------
(a m a)
(l a m a r m a)
(a m l a m a r)
(l a m a r m l a m a r)
(l a m l a m a r r m a)
(l l a m a r m a r m a)
(a m l a m l a m a r r)
(a m l l a m a r m a r)

S <- S1mS1
S1 <- lS2 | a
S2 <- Sr
------------------------
```

Their system first constructs a "trivial" grammar, and then simplifies it. As we can see, their system requires us to provide nicely-chosen examples, and it cannot solve from poorly organized examples such as the problem we introduced at the beginning.

Bierman and Feldman then built a system that constructs a finite automaton from given examples. Although it takes only positive examples, they showed an application to the case where both positive and negative examples are given. Their algorithm also requires nicely-chosen examples, and they showed the method to choose the examples from a regular set "nicely", so that it always turns out the simplest machine. However if the examples are not nicely-chosen, as in the problem we introduced at the beginning, their system hardly turns out the simplest machine.

Apart from the grammatical inference, there has been a good deal of work on discovery of a regularity or a common pattern in the given examples that are not necessarily nicely-chosen ([Langley 81a] [Langley 81b] [Buchanan 76] [Hayes-roth 77] [Michalski 73] [Vere 75] [Winston 70]).

1.4 Overview of the Paper

In the rest of this chapter, we present the 7 sample problems, that we will consider throughout this paper (7 sample problems and their inverses).

In chapter 2, we present the result of an experiment in constructing finite automata with n states using hill-climbing, in particular, we let n = 8. We shall see that all 14 sample problems can be solved by this method.

In chapter 3, we present the result of an experiment in simplifying the finite automata which we have found in chapter 2, also using hill-climbing. We shall see that we can find the simplest machine for most of the problems by this method.

In chapter 4, we discuss *re-construction* of finite automata, that is, how to re-construct a finite automaton if the right-list and the wrong-list are slightly altered. We might not want to construct it from the beginning. Rather, we want to construct the new machine by modifying the previous machine.

Finally, we have an actual system called *Regular set Recognizer* [RR], using the techniques above. RR learns to recognize a regular set, given examples by a human "teacher". We present several sample runs as well as a user's manual, in chapter 5.

1.5 Sample Problems

1.5.1 Sample Problems

Throughout this paper, we consider the following 7 sample problems.

Problem 1

right-list	wrong-list
()	(0)
(1)	(1 0)
(1 1)	(0 1)
(1 1 1)	(0 0)
(1 1 1 1)	(0 1 1)
(1 1 1 1 1)	(1 1 0)
(1 1 1 1 1 1)	(1 1 1 1 1 1 1 0)
(1 1 1 1 1 1 1)	(1 0 1 1 1 1 1 1)
(1 1 1 1 1 1 1 1)	

Problem 2

right-list	wrong-list
()	(1)
(1 0)	(0)
(1 0 1 0)	(1 1)
(1 0 1 0 1 0)	(0 0)
(1 0 1 0 1 0 1 0)	(0 1)
(1 0 1 0 1 0 1 0 1 0 1 0)	(1 0 1)
	(1 0 0)
	(1 0 0 1 0 1 0)
	(1 0 1 1 0)
	(1 1 0 1 0 1 0 1 0)

Problem 3[3]

right-list	wrong-list
()	(1 0)
(1)	(1 0 1)
(0)	(0 1 0)
(0 1)	(1 0 1 0)
(1 1)	(1 1 1 0)
(0 0)	(1 0 1 1)
(1 0 0)	(1 0 0 0 1)
(1 1 0)	(1 1 1 0 1 0)
(1 1 1)	(1 0 0 1 0 0 0)
(0 0 0)	(1 1 1 1 1 0 0 0)
(1 0 0 1 0 0)	(0 1 1 1 0 0 1 1 0 1)
(1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 1)	(1 1 0 1 1 1 0 0 1 1 0)
(1 1 1 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0)	

Problem 4

right-list	wrong-list
()	(0 0 0)
(1)	(1 1 0 0 0)
(0)	(0 0 0 1)
(1 0)	(0 0 0 0 0 0 0 0 0)
(0 1)	(1 1 1 1 1 0 0 0 0 1 1)
(0 0)	(1 1 0 1 0 1 0 0 0 0 0 1 0 1 1 1)
(1 0 0 1 0 0)	(1 0 1 0 0 1 0 0 0 1)
(0 0 1 1 1 1 1 1 1 0 1 0 0)	(0 0 0 0)
(0 1 0 0 1 0 0 1 0 0)	(0 0 0 0 0)
(1 1 1 0 0)	
(0 1 0)	

[3] This problem was introduced at the very beginning.

Problem 5

right-list
()
(1 1)
(0 0)
(1 0 0 1)
(0 1 0 1)
(1 0 1 0)
(1 0 0 0 1 1 1 1 0 1)
(1 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0)
(1 1 1 1 1 1)
(0 0 0 0)

wrong-list
(1)
(0)
(1 1 1)
(0 1 0)
(0 0 0 0 0 0 0 0 0)
(1 0 0 0)
(0 1)
(1 0)
(1 1 1 0 0 1 0 1 0 0)
(0 1 0 1 1 1 1 1 1 1 1 0)
(0 0 0 1)
(0 1 1)

Problem 6

right-list
()
(1 0)
(0 1)
(1 1 0 0)
(1 0 1 0 1 0)
(1 1 1)
(0 0 0 0 0 0)
(1 0 1 1 1)
(0 1 1 1 1 0 1 1 1 1)
(1 0 0 1 0 0 1 0 0)

wrong-list
(1)
(0)
(1 1)
(0 0)
(1 0 1)
(0 1 1)
(1 1 0 0 1)
(1 1 1 1)
(0 0 0 0 0 0 0 0)
(0 1 0 1 1 1)
(1 0 1 1 1 1 0 1 1 1 1)
(1 0 0 1 0 0 1 0 0 1)

Problem 7

right-list
()
(1)
(0)
(1 0)
(0 1)
(1 1 1 1 1)
(0 0 0)
(0 0 1 1 0 0 1 1)
(0 1 0 1)
(0 0 0 0 1 0 0 0 0 1 1 1 1)
(0 0 1 0 0)
(0 1 1 1 1 1 0 1 1 1 1 1)
(0 0)

wrong-list
(1 0 1 0)
(0 0 1 1 0 0 1 1 0 0 0)
(0 1 0 1 0 1 0 1 0 1)
(1 0 1 1 0 1 0)
(1 0 1 0 1)
(0 1 0 1 0 0)
(1 0 1 0 0 1)
(1 0 0 1 0 0 1 1 0 1 0 1)

1.5.2 Solution of Sample Problems

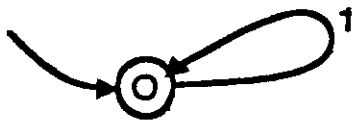
The solution of these problems are:

1. 1^*
2. $(1\ 0)^*$
3. any string without an odd number of consecutive 0's AFTER an odd number of consecutive 1's.
4. any string without more than 2 consecutive 0's.
5. any string of even length which, making pairs, has an odd number of (0 1) or (1 0)'s.
6. any string such that the difference between the numbers of 1's and 0's is 3n.
7. $0^*1^*0^*1^*$.

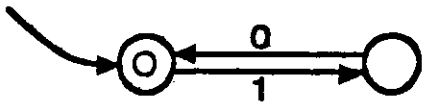
1.5.3 Finite Automata of Solutions

The machines corresponding to these solutions are as follows.

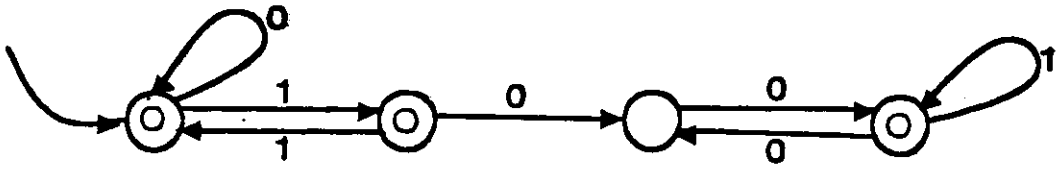
Solution of Problem 1



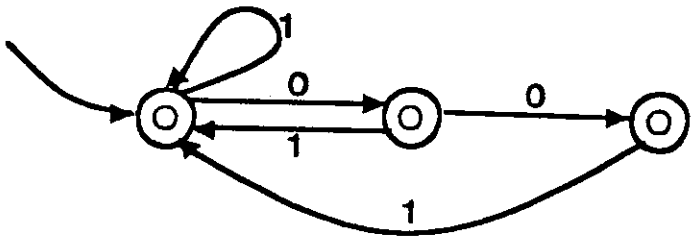
Solution of Problem 2



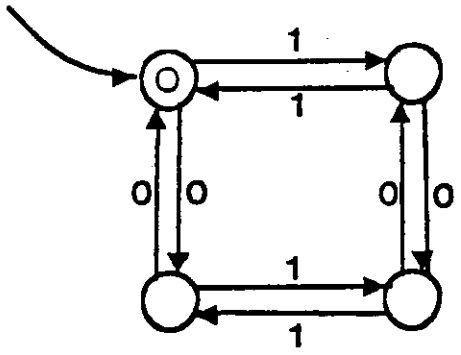
Solution of Problem 3



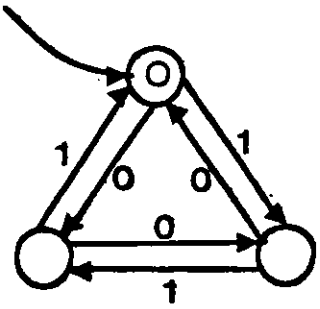
Solution of Problem 4



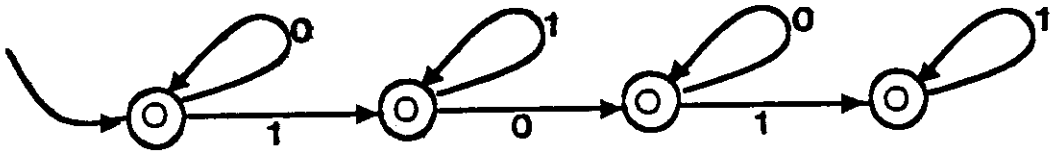
Solution of Problem 5



Solution of Problem 6



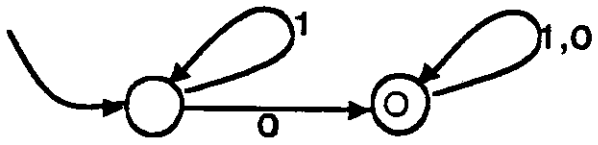
Solution of Problem 7



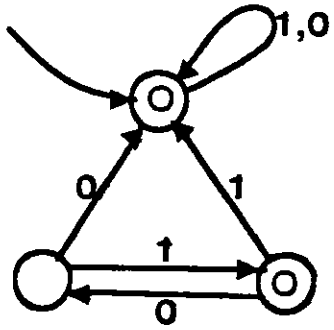
1.5.4 Inverse Problems

We also consider the *inverse problems* of these sample problems. The inverse problems are created by exchanging the right-list and wrong-list. We use these 14 problems in our experiments and refer to the inverse problem of problem 1 as problem 1-, the inverse problem of problem 2 as problem 2-, and so on.

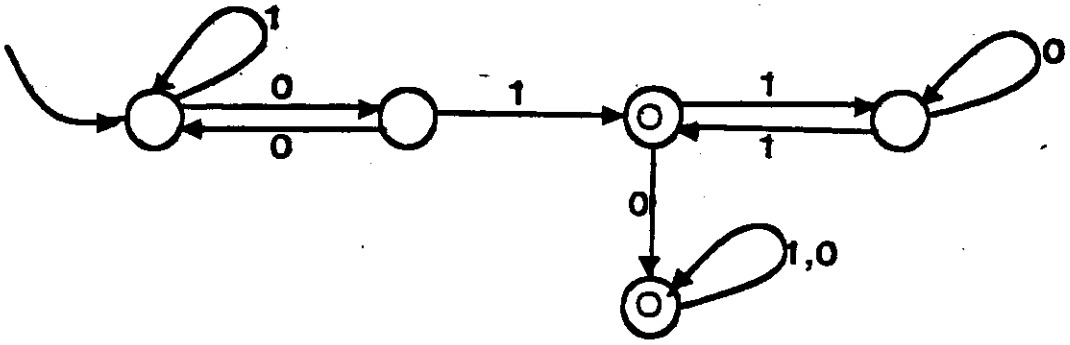
Solution of Problem 1-



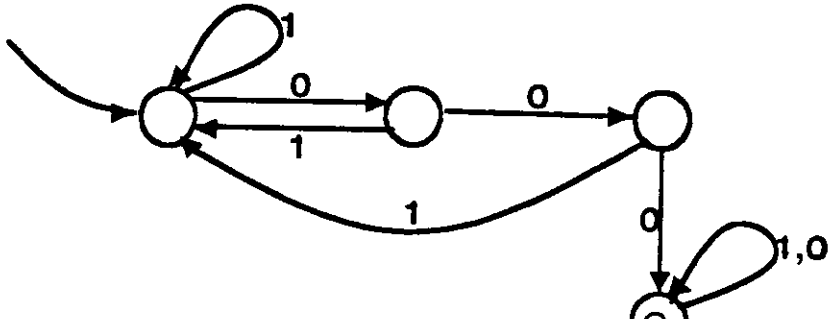
Solution of Problem 2-



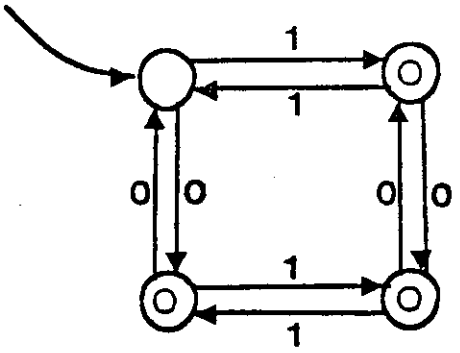
Solution of Problem 3-



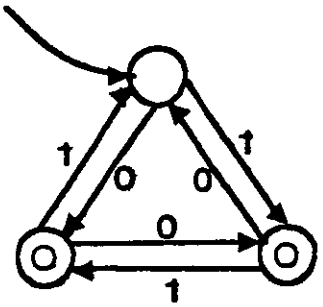
Solution of Problem 4-



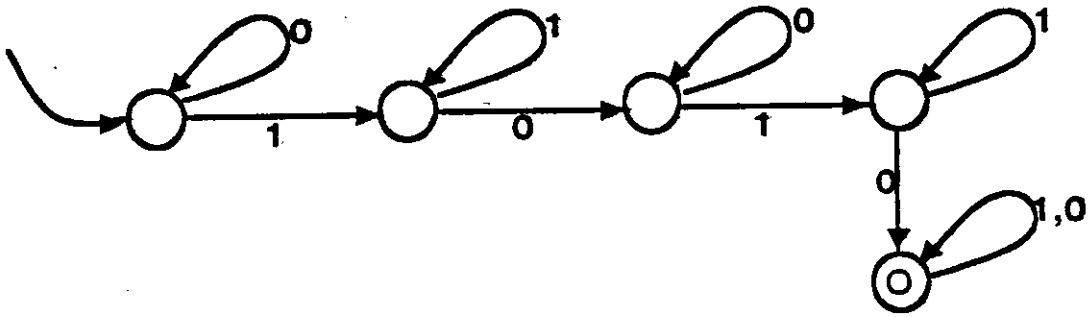
Solution of Problem 5-



Solution of Problem 6-



Solution of Problem 7-



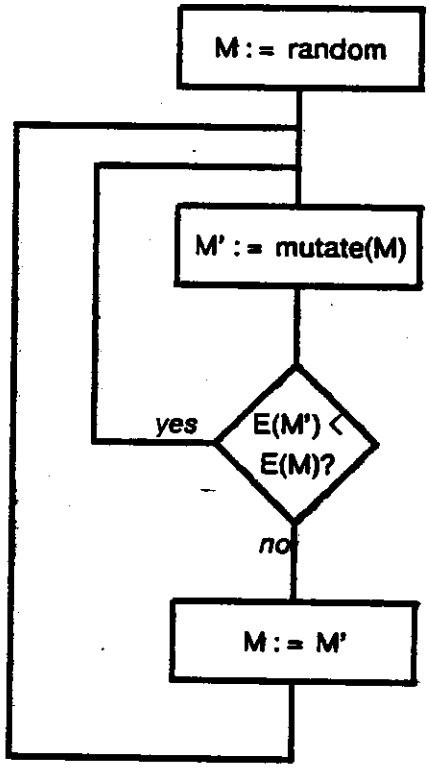
2. Construction of Finite Automata

In this chapter, we describe an experiment in constructing a finite automaton with n states from a given right-list and a wrong-list, using hill-climbing. In particular, we let n equal 8. We shall see that each of the 14 problems can be solved in at most a few thousands steps.

2.1 Algorithm

The hill-climbing algorithm of this experiment is shown in figure 2-1.

Figure 2-1: Flowchart of the Hill-Climbing



We first construct a random machine with 8 states. We next make a copy of this machine, where the copy is slightly altered from the original by an operator ***mutate***. We compare the new machine with the original by an evaluation function E. The better machine is called *current generation* and we make a copy of this machine, and so forth. The worse machine is simply discarded. The operator ***mutate*** and the evaluation function E are defined more precisely in the following.

Operator *mutate*: Taking a machine $((A_1, B_1, F_1) \ldots (A_8, B_8, F_8))$ as its argument, the operator ***mutate*** chooses one digit randomly, and replaces it by another digit.[4] That is, the mutation in our algorithm is randomly one of the following: delete an arrow, insert an arrow, change the destination

[4] $0 =< A_i =< 8$; $0 =< B_i =< 8$; and $0 =< F_i =< 1$.

of an arrow to another destination, make a non-final state into a final state, and make a final state into a non-final state.

Evaluation Function E: The evaluation function E takes a machine as its argument and returns **r** - **w**, where **r** is the number of strings in the right-list accepted by the machine, and **w** is the number of strings in the wrong-list accepted by the machine. If **r** - **w** < 0 then it returns 0.

2.2 Results

We show in this chapter the result of our experiments. We first show in figure 2-2 the trace of the experiment of problem 3, to see how our algorithm gradually refines a random machine into the desired machine. Each line corresponds to the current generation M. The column **E** indicates E(M), and **G** indicates the cumulative number of generation. The final machine of this trace accepts all strings in the right-list but none in the wrong-list of problem 3 (figure 2-3).

We show the results for the other 13 problems in figure 2-4.

Figure 2-2: Sample Trace of Problem 3

```
                                                                      E G
-------------------------------------------------------------------------------
(((1 4 0)(6 2 1)(3 3 1)(1 0 0)(0 0 1)(4 8 0)(4 1 1)(3 2 1)) 0 1)
(((1 4 0)(6 2 1)(3 3 1)(1 0 0)(0 0 1)(4 8 1)(4 1 1)(3 2 1)) 0 2)
(((1 4 0)(6 2 1)(4 3 1)(1 0 0)(0 0 1)(4 8 1)(4 1 1)(3 2 1)) 0 3)
(((1 4 0)(6 2 1)(4 3 1)(1 0 0)(0 0 1)(4 8 1)(4 1 0)(3 2 1)) 0 4)
(((1 4 0)(6 2 1)(4 3 1)(1 0 0)(0 0 1)(4 8 1)(4 1 0)(3 4 1)) 0 5)
(((1 4 0)(6 3 1)(4 3 1)(1 0 0)(0 0 1)(4 8 1)(4 1 0)(3 4 1)) 0 6)
(((1 4 0)(6 3 1)(4 3 1)(1 0 0)(0 0 1)(4 8 1)(4 1 0)(3 2 1)) 0 7)
(((1 4 0)(6 3 1)(4 3 1)(1 0 0)(0 0 1)(4 0 1)(4 1 0)(3 2 1)) 0 8)
(((1 4 0)(6 3 1)(4 3 1)(1 0 0)(0 0 1)(0 0 1)(4 1 0)(3 2 1)) 0 9)
(((1 4 0)(6 3 1)(4 3 1)(1 0 0)(0 0 1)(0 0 1)(4 0 0)(3 2 1)) 0 10)
(((1 4 1)(6 3 1)(4 3 1)(1 0 0)(0 0 1)(0 0 1)(4 0 0)(3 2 1)) 2 11)
(((1 4 1)(6 3 1)(4 4 1)(1 0 0)(0 0 1)(0 0 1)(4 0 0)(3 2 1)) 2 12)
(((1 4 1)(6 3 1)(4 4 1)(1 0 0)(0 0 1)(0 1 1)(4 0 0)(3 2 1)) 2 14)
(((1 4 1)(6 3 1)(4 4 1)(1 0 0)(0 0 1)(0 1 1)(4 5 0)(3 2 1)) 2 15)
(((1 4 1)(6 3 1)(0 4 1)(1 0 0)(0 0 1)(0 1 1)(4 5 0)(3 2 1)) 2 16)
(((1 4 1)(6 3 1)(0 4 1)(0 0 0)(0 0 1)(0 1 1)(4 5 0)(3 2 1)) 4 17)
(((1 4 1)(6 3 1)(0 8 1)(0 0 0)(0 0 1)(0 1 1)(4 5 0)(3 2 1)) 4 18)
(((1 4 1)(0 3 1)(0 8 1)(0 0 0)(0 0 1)(0 1 1)(4 5 0)(3 2 1)) 4 19)
(((1 4 1)(0 3 1)(0 8 1)(0 0 0)(0 0 0)(0 1 1)(4 5 0)(3 2 1)) 4 20)
(((1 4 1)(0 3 1)(0 8 1)(0 0 0)(0 3 0)(0 1 1)(4 5 0)(3 2 1)) 4 21)
(((1 4 1)(0 3 1)(0 8 1)(0 0 0)(0 3 0)(0 6 1)(4 5 0)(3 2 1)) 4 23)
(((1 4 1)(1 3 1)(0 8 1)(0 0 0)(0 3 0)(0 6 1)(4 5 0)(3 2 1)) 4 24)
(((1 4 1)(1 3 1)(0 8 1)(0 0 0)(0 5 0)(0 6 1)(4 5 0)(3 2 1)) 4 25)
(((1 4 1)(1 3 1)(0 8 1)(0 0 0)(0 5 0)(0 6 1)(0 5 0)(3 2 1)) 4 26)
(((2 4 1)(1 3 1)(0 8 1)(0 0 0)(0 5 0)(0 6 1)(7 5 0)(3 2 1)) 4 27)
(((1 4 1)(1 3 1)(0 8 1)(0 0 0)(0 5 0)(0 6 1)(7 5 0)(2 2 1)) 4 28)
(((1 4 1)(1 3 1)(0 8 1)(0 5 0)(0 5 0)(0 6 1)(7 5 0)(2 2 1)) 6 29)
(((1 4 1)(1 3 1)(0 8 1)(0 2 0)(0 5 0)(0 6 1)(7 5 0)(2 2 1)) 6 30)
              .                                   .
              .                                   .
              .                                   .
(((1 5 1)(5 0 0)(4 6 1)(0 0 0)(2 1 1)(2 0 0)(8 7 1)(6 0 1)) 12 2048)
(((1 5 1)(5 0 0)(4 6 1)(0 0 0)(2 1 1)(2 4 0)(8 7 1)(6 0 1)) 12 2049)
(((1 5 1)(5 0 0)(4 6 1)(4 0 0)(2 1 1)(2 4 0)(8 7 1)(6 0 1)) 12 2050)
(((1 5 1)(5 0 0)(4 6 1)(4 0 0)(2 1 1)(2 4 0)(2 7 1)(6 0 1)) 12 2051)
(((1 5 1)(7 0 0)(4 6 1)(4 0 0)(2 1 1)(2 4 0)(2 7 1)(6 0 1)) 13 2052)
-------------------------------------------------------------------------------
total runtime 129.036000 sec
```

Figure 2-3: The final machine of problem 3

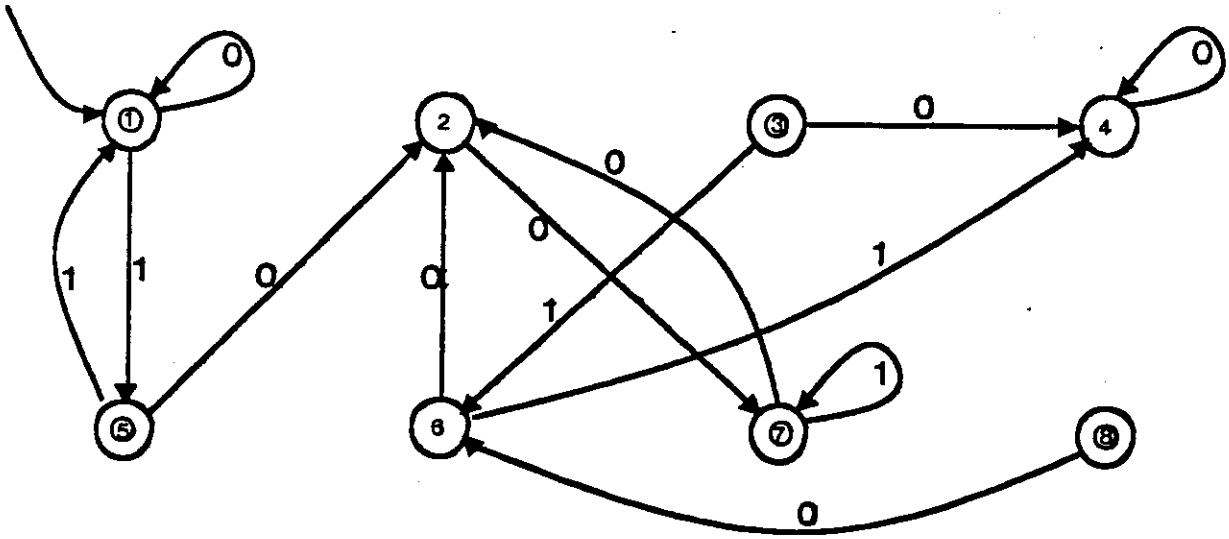
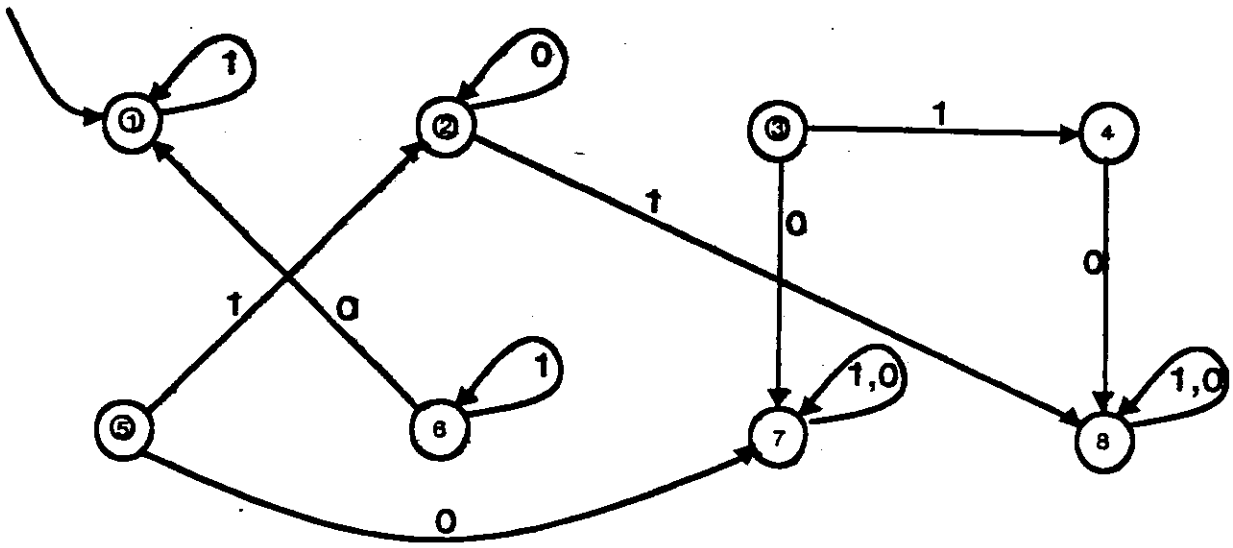


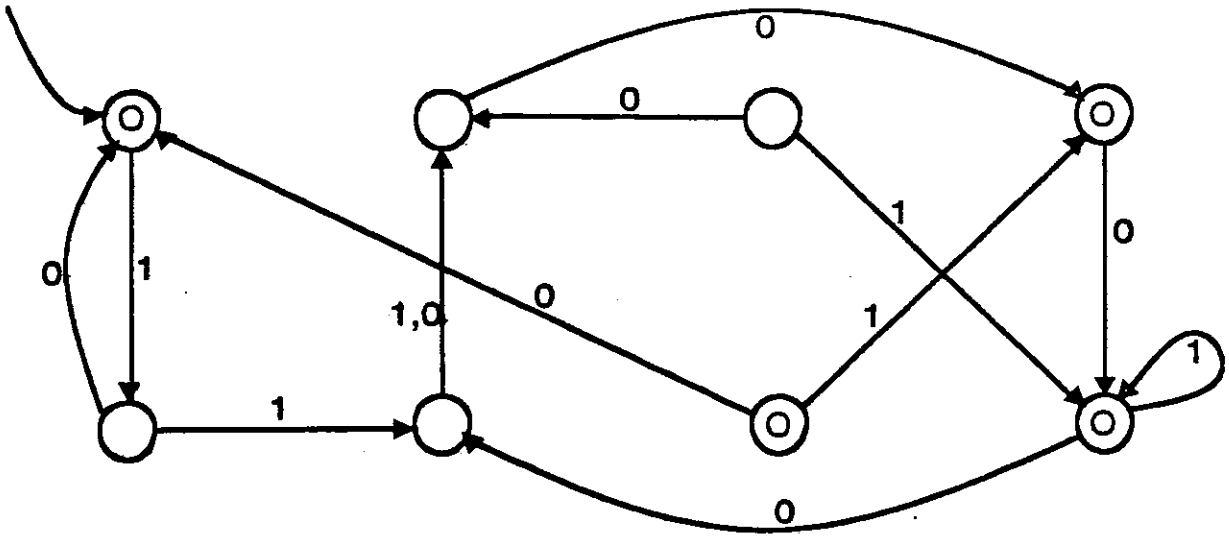
figure 2-4: The results of the other 13 problems

Final Machine of Problem 1



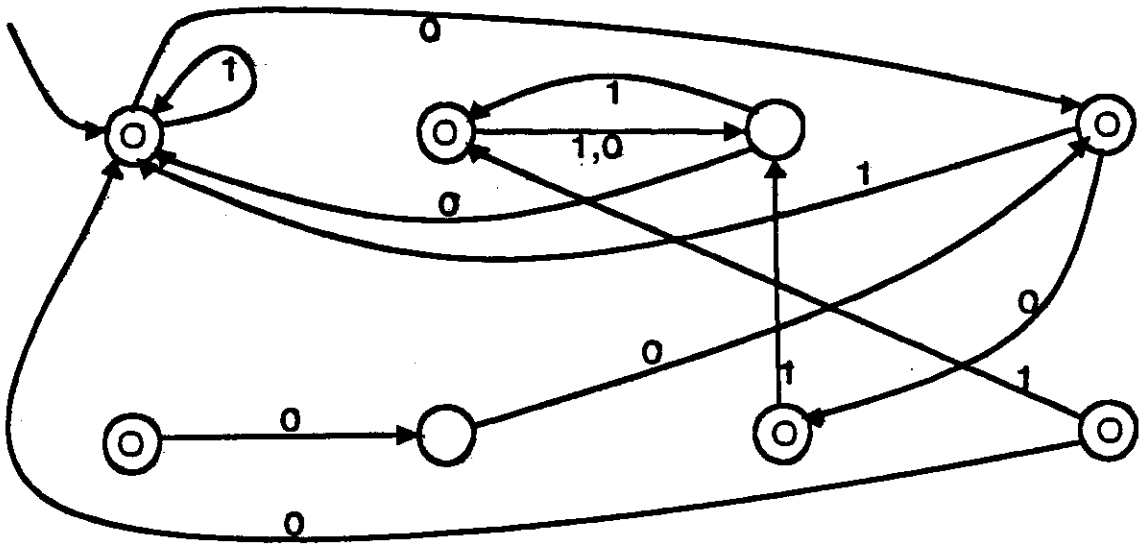
((0 1 1)(2 8 1)(7 4 1)(8 0 0)(7 2 1)(1 6 0)(7 7 0)(8 8 0)) 98

Final Machine of Problem 2



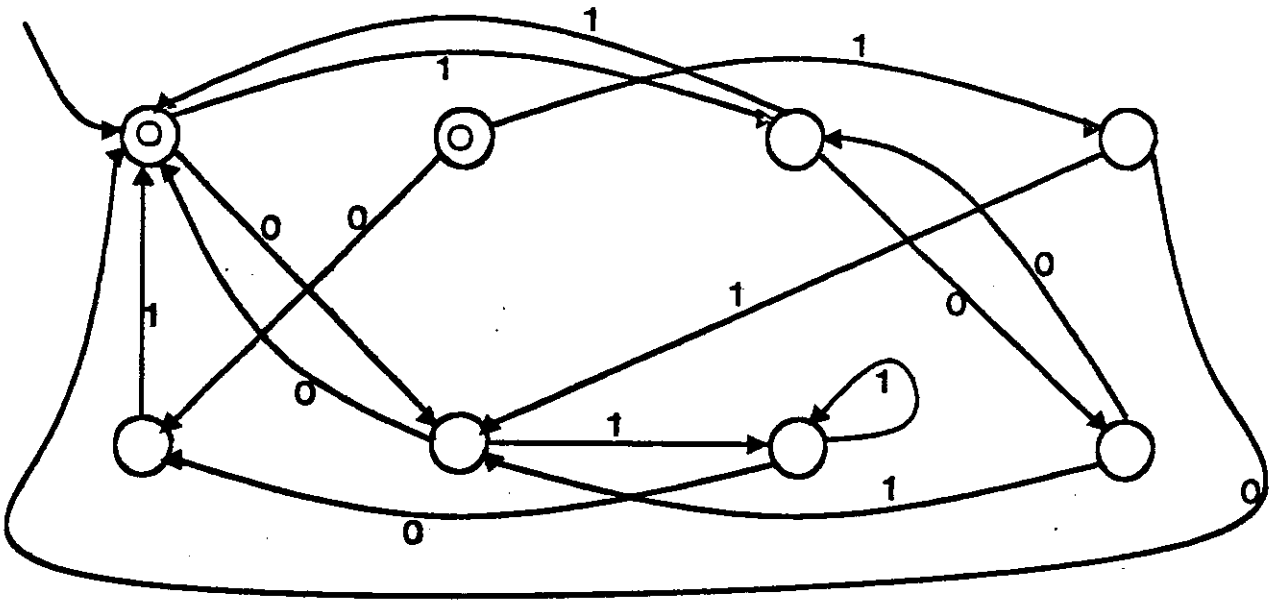
((0 5 1)(4 0 0)(2 8 0)(8 0 1)(1 6 0)(2 2 0)(1 4 1)(6 8 1)) 134

Final Machine of Problem 4



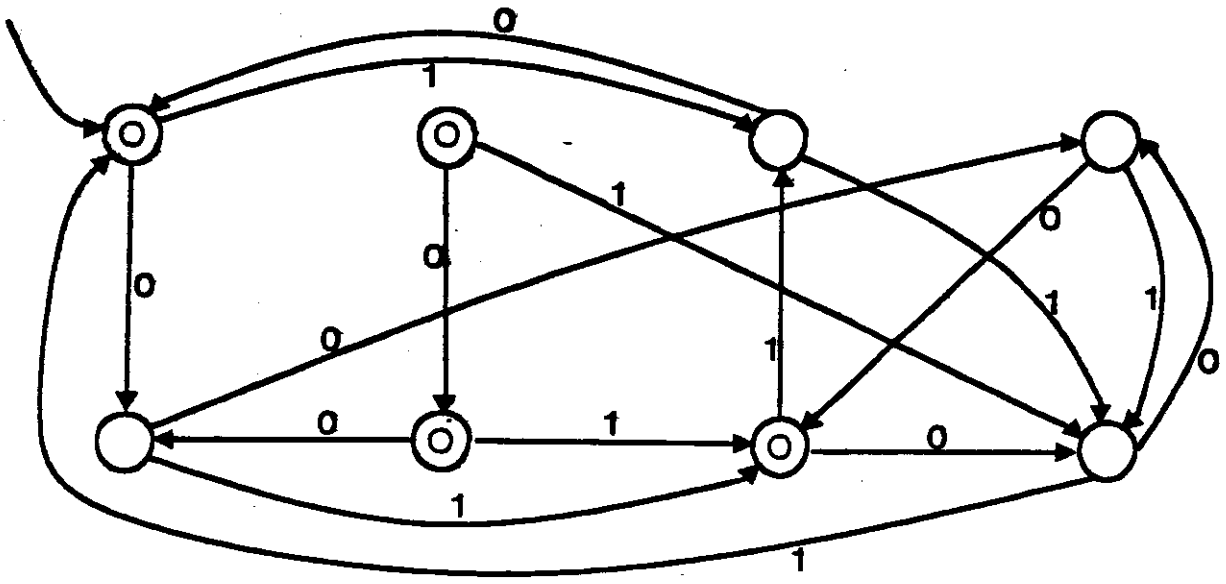
((4 1 1)(3 3 1)(1 2 0)(7 1 1)(6 0 1)(4 0 0)(0 3 1)(1 2 1)) 442

Final Machine of Problem 5



((6 3 1)(5 4 1)(8 1 0)(1 6 0)(0 1 0)(1 7 0)(5 7 0)(3 6 0)) 1768

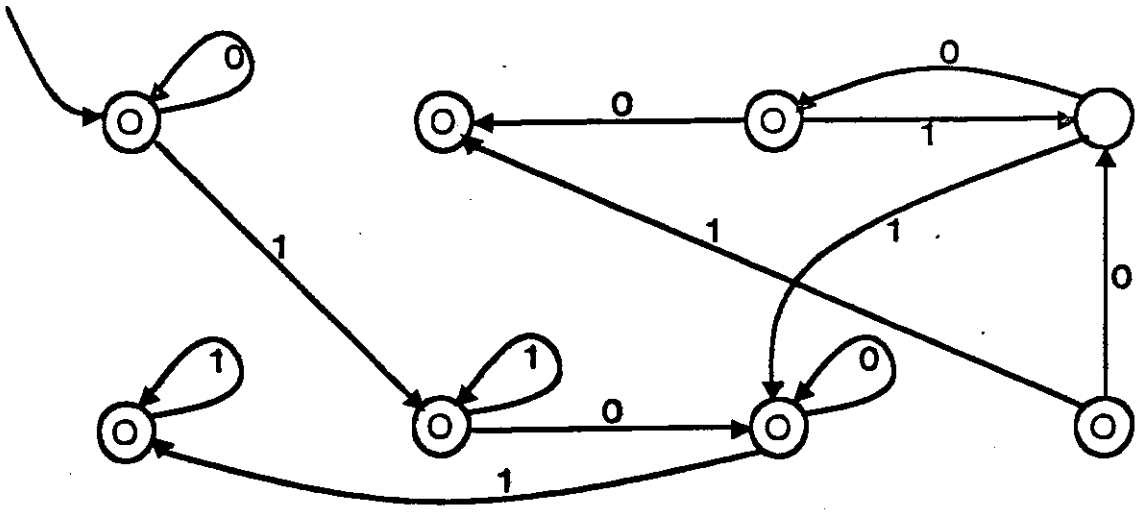
Final Machine of Problem 6



((5 3 1)(6 8 1)(1 8 0)(7 8 0)(4 7 0)(5 7 1)(8 3 1)(4 1 0)) 277

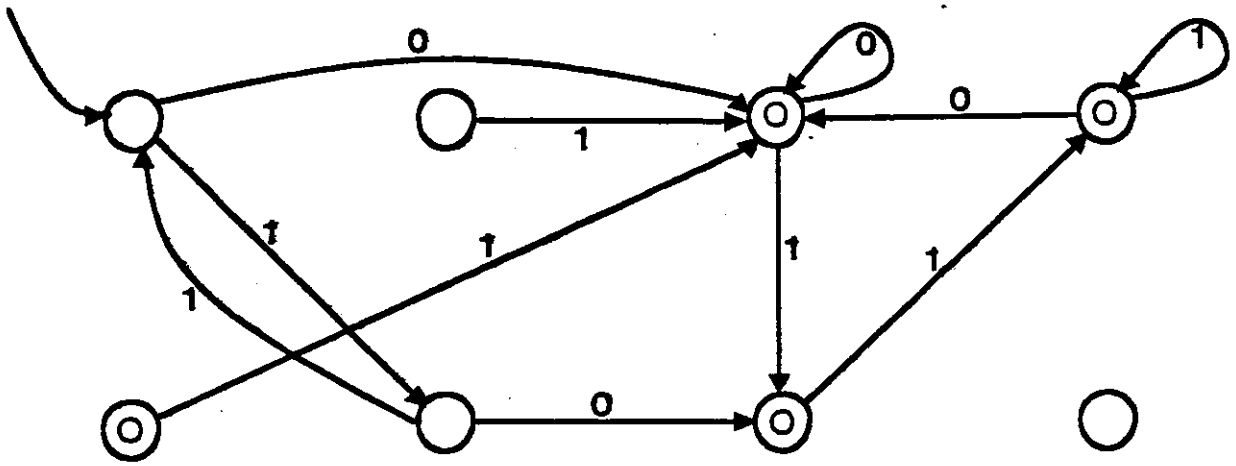
CONSTRUCTION OF FINITE AUTOMATA

Final Machine of Problem 7



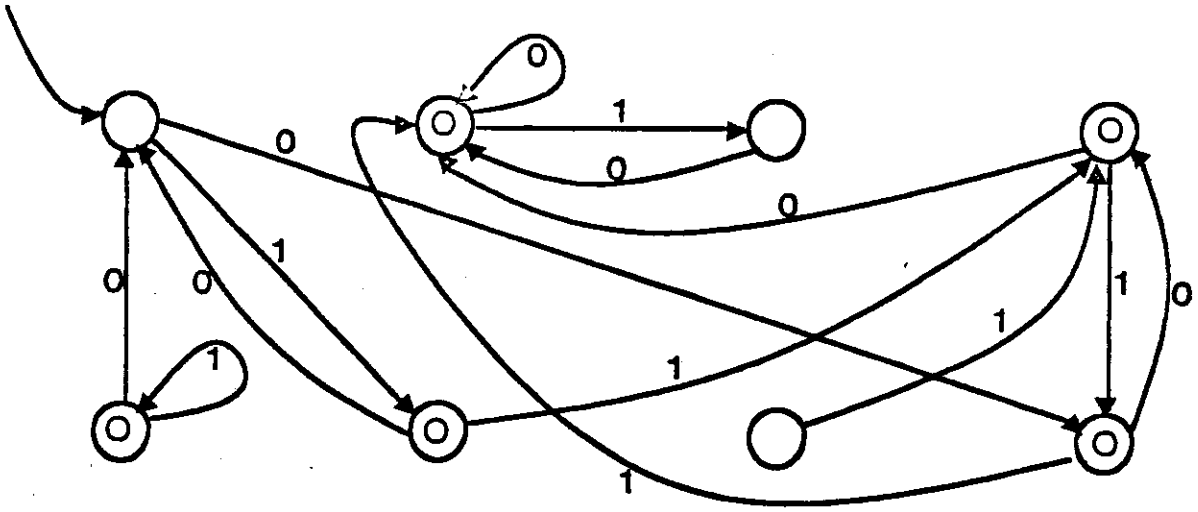
((1 6 1)(0 0 1)(2 4 1)(3 7 0)(0 5 1)(7 6 1)(7 5 1)(4 2 1)) 206

Final Machine of Problem 1-



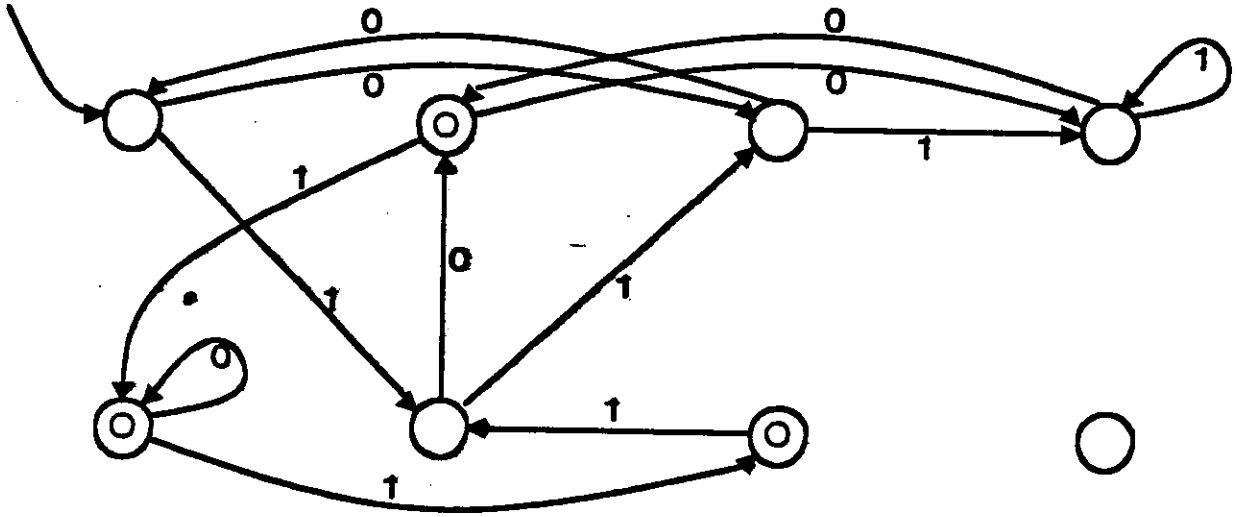
((3 6 0)(0 3 0)(3 7 1)(3 4 1)(0 3 1)(7 1 0)(0 4 1)(0 0 0)) 300

Final Machine of Problem 2-



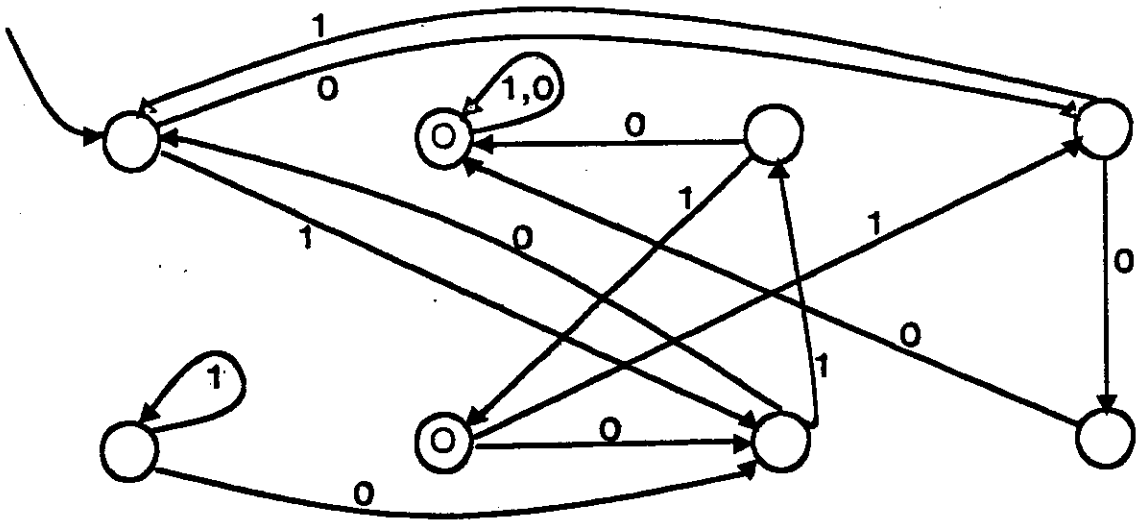
((8 6 0)(2 3 1)(2 0 0)(2 8 1)(1 5 1)(1 4 1)(0 4 0)(4 2 1)) 89

Final Machine of Problem 3-



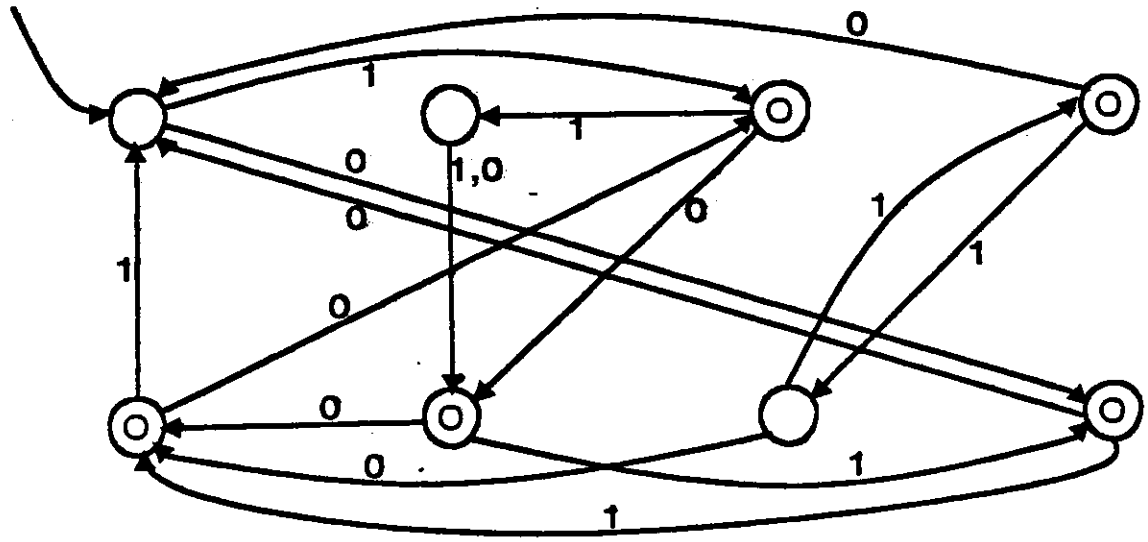
((3 6 0)(4 5 1)(1 4 0)(2 4 0)(5 7 1)(2 3 0)(0 6 1)(0 0 0)) 1939

Final Machine of Problem 4-



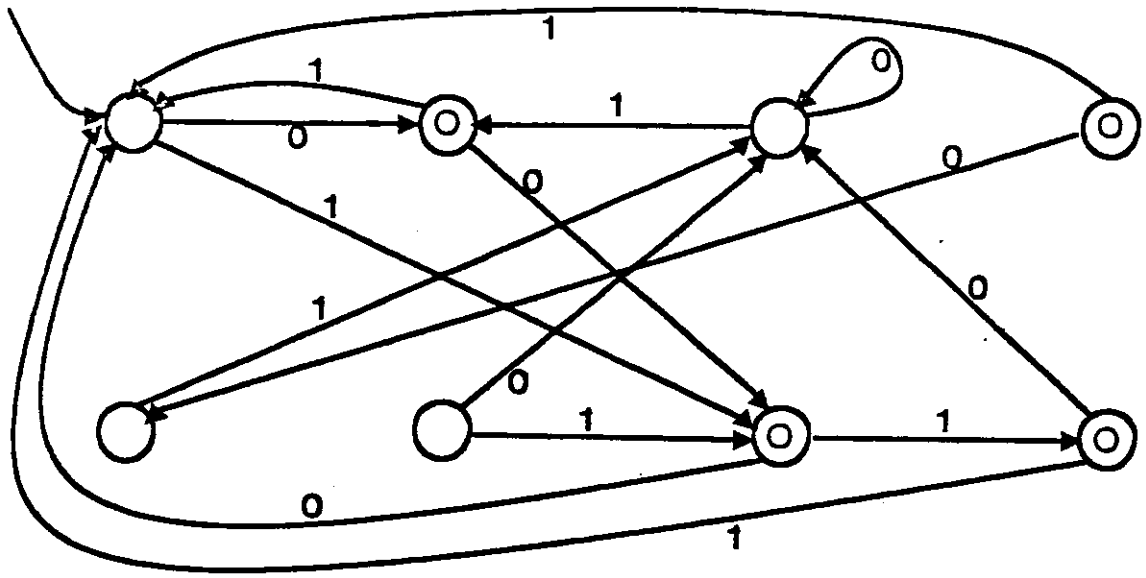
((4 7 0)(2 2 1)(2 6 0)(8 1 0)(7 5 0)(7 4 1)(1 3 0)(2 0 0)) 246

Final Machine of Problem 5-



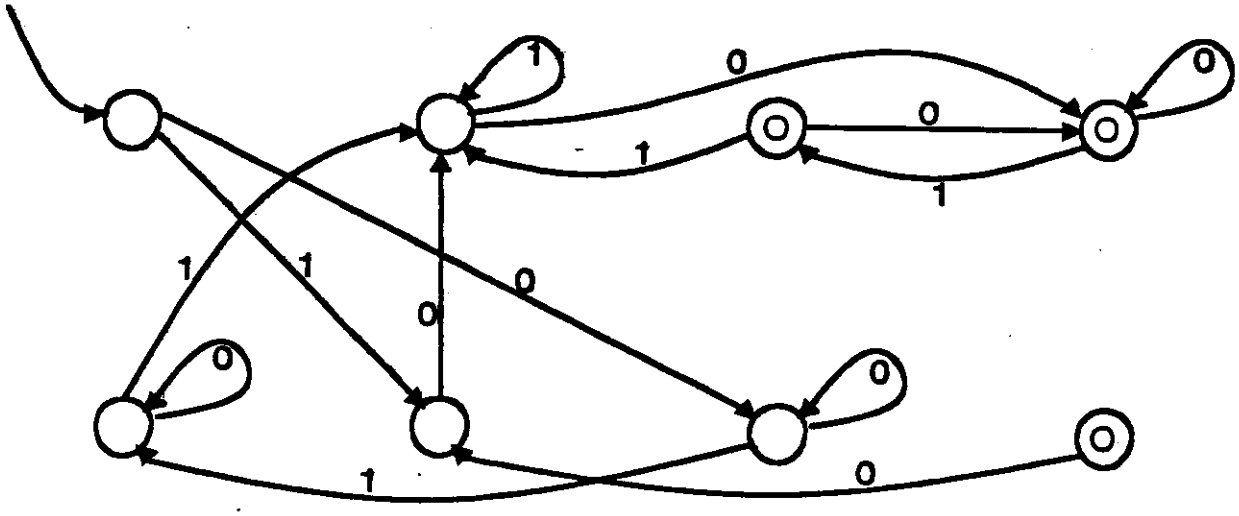
((8 3 0)(6 6 0)(6 2 1)(1 7 1)(3 1 1)(5 8 1)(5 4 0)(1 5 1)) 1844

Final Machine of Problem 6-



((2 7 0)(7 1 1)(3 2 0)(5 1 1)(0 3 0)(3 7 0)(1 8 1)(3 1 1)) 886

Final Machine of Problem 7-



((7 6 0)(4 2 0)(4 2 1)(4 3 1)(5 2 0)(2 0 0)(7 5 0)(6 0 1)) 3725

2.3 Discussion

2.3.1 Hill-Climbing vs. Exhaustive Search

To see how effectively our hill-climbing algorithm has performed, we compare our method with an exhaustive search. There are $(9 \times 9 \times 2)^8$ = about 5×10^{17} machines in our problem space. We now want to know the number of the desired machines in our problem space, so that we can calculate the expected number of steps until the exhaustive algorithm finds the first desired machine. This can be done by the following "sampling" method: take one machine in the problem space randomly, and test if this machine is the desired machine; repeat this procedure 100,000 times.

We show the expected number of steps using the exhaustive search calculated by this procedure in figure 2-5. Although the exhaustive search works better on "easy" problems, it is obvious in general that our hill-climbing works much better than the exhaustive search.

Figure 2-5: The number of Steps to get the desired machine

Problem	Hill-Climbing	Exhaustive-Search
P1	98	33
P2	134	315
P3	2052	> 50000
P4	442	12500
P5	1768	> 50000
P6	277	50000
P7	206	50000
P1-	300	167
P2-	89	1852
P3-	1939	> 50000
P4-	246	> 50000
P5-	1844	> 50000
P6-	886	> 50000
P7-	3725	> 50000

2.3.2 Result with Different Numbers of States

So far, we fixed the number of states to be 8. In this section, we shall try the same experiment with different numbers of states (4 - 10). Figure 2-6 shows the result of this experiment. In the table, "---" indicates "it could not solve within the given time". This can happen when the hill-climbing algorithm climbs a "local hill". This table implies that the number of states n should be reasonably large to avoid climbing a local hill, and we can hardly get the simplest machine by this method. We shall, however, see that we can *simplify* the machine with 8 states that we have gotten in this chapter, so that it becomes the simplest machine.

Figure 2-6: The Number of States and Runtime [sec.]

PROBLEM	NUMBER OF STATES 4	5	6	7	8	9	10
1	0.4	0.4	1.9	5.3	2.2	1.9	0.6
2	0.5	0.4	6.4	2.2	3.0	4.1	3.6
3	338.1	---	3.3	39.6	129.0	16.2	158.3
4	12.3	4.8	18.1	9.4	13.1	1.4	11.7
5	---	---	164.7	7.9	56.6	220.4	95.6
6	3.1	12.5	5.2	20.5	7.9	26.3	137.8
7	49.5	10.9	23.2	2.8	7.1	5.6	18.6
1-	2.4	0.8	1.3	4.3	8.1	1.8	2.5
2-	15.3	1.9	13.9	7.4	2.4	19.2	17.9
3-	---	---	---	76.5	78.4	---	243.6
4-	---	23.9	12.5	20.7	7.7	14.8	17.5
5-	162.0	---	---	28.8	66.5	52.1	68.4
6-	2.8	3.6	53.4	13.6	29.0	5.5	7.5
7-	---	---	263.5	---	138.7	54.5	33.3

3. Simplification of Finite Automata

In the previous chapter, we saw that our hill-climbing method successfully produced a machine that accepts all strings in the right-list but no string in the wrong-list. However, the final machine of the result of problem 2, for example, does not accept our desired regular set $(1\ 0)^*$. For instance, it does accept a string (1 1 0 0), which is not in $(1\ 0)^*$. We therefore want the machine to be "generalized" so that it accepts exactly $(1\ 0)^*$. In fact, the final machines of all problems except problem 1, 3 and 7, need to be generalized.

We define the generality of a machine in terms of its simplicity. The simplicity of a machine is determined by the number of states the machine has, and if two machines have the same number of states, a machine with fewer arrows and final states is simpler.

Our task is to simplify the machines we have obtained in the previous chapter, so that the machines become the simplest or the most general . We call this task *simplification* of finite automata, and it can be also done by using a hill-climbing method.

3.1 Minimization

Before we simplify the final machine of the previous experiment, we first remove any useless arrows and states, using a *Minimization Algorithm* (see, for example, [Hopcroft 79]). We show the result of the minimization in figure 3-1. Note that even after minimization, all problems except 1, 3 and 7 still need to be generalized.

Figure 3-1: Minimized Final Machine

Problem	Minimized Machine
P1	((0 1 1))
P2	((0 4 1)(3 0 0)(6 0 1)(1 5 0)(2 2 0)(5 6 1))
P3	((1 2 1)(3 1 1)(4 0 0)(3 4 1))
P4	((4 1 1)(3 3 1)(1 2 0)(5 1 1)(0 3 1))
P5	((6 3 1)(3 6 0)(2 1 0)(5 4 0)(0 1 0)(1 4 0))
P6	((5 3 1)(6 3 1)(1 6 0)(2 6 0)(4 2 0)(4 1 0))
P7	((1 2 1)(3 2 1)(3 4 1)(0 4 1))
P1-	((3 2 0)(5 1 0)(3 5 1)(3 4 1)(0 4 1))
P2-	((5 6 0)(2 3 1)(2 0 0)(2 5 1)(4 2 1)(1 4 1))
P3-	((3 6 0)(4 5 1)(1 4 0)(2 4 0)(5 7 1)(2 3 0)(0 6 1))
P4-	((4 7 0)(2 2 1)(2 6 0)(5 1 0)(2 0 0)(7 4 1)(1 3 0))
P5-	((4 3 0)(6 6 0)(6 2 1)(1 5 1)(3 1 1)(5 4 1))
P6-	((2 4 0)(4 1 1)(3 2 0)(1 5 1)(3 1 1))
P7-	((7 6 0)(4 2 0)(4 2 1)(4 3 1)(5 2 0)(2 0 0)(7 5 0))

3.2 Simplification Algorithm

The algorithm for simplification is similar to the algorithm described in the previous chapter. The major differences are as follows: (1) the evaluation function E(M) returns a higher value if the machine M is simpler; (2) if M does not accept some strings in the right-list, or does accept some

strings in the wrong-list, E(M) returns minus infinity; (3) the algorithm starts with the minimized final machine of the previous experiment instead of a random machine; (4) whenever a "useless state" (i.e. a non-final state with neither 0-arrow nor 1-arrow) is found, delete it.

3.3 Results

A sample trace of problem 2- is shown in figure 3-2. Each line corresponds to current generation M, and the right-most number is the cumulative number of steps. The final machine of this trace is the desired simplest machine.

The final machines of all 14 problems are shown in figure 3-3. We see that some problems could not be simplified completely within the given time, probably because the search was climbing a local hill.

3.4 Discussion

3.4.1 Hill-Climbing vs. Exhaustive Search

We compare our method with an exhaustive search. The exhaustive search enumerates all machines in the order of simplicity, and the first machine that accepts all strings in the right-list but none in the wrong-list is considered the simplest machine. Thus we can calculate the expected number of steps until the exhaustive search finds the desired machine[5]. The result is shown in figure 3-4.[6]

---

[5] Let n be the number of states of the simplest machine. Then the expected number of steps S_n is:

$$S_n = [\Sigma_{i=1 \text{ to } n-1} U_i] + [U_n / (2 \times (n-1)!)],$$

where U_j is the number of all possible machines with j states, that is,

$$U_j = (j+1)^{2n} \times 2^n.$$

[6] The number of steps using hill-climbing in this figure is the sum of the number of steps to construct the 8 state machine and the number of steps to simplify it into the simplest machine.

Figure 3-2: Sample Trace of Problem 2-

```
(((5 6 0) (2 3 1) (2 0 0) (2 5 1) (4 2 1) (1 4 1)) 0)
(((5 6 0) (2 3 1) (2 0 0) (2 0 1) (4 2 1) (1 4 1)) 8)
(((5 6 0) (2 3 1) (2 0 0) (2 0 1) (5 2 1) (1 4 1)) 23)
(((5 6 0) (2 3 1) (6 0 0) (2 0 1) (5 2 1) (1 4 1)) 41)
(((5 6 0) (2 3 1) (2 0 0) (2 0 1) (5 2 1) (1 4 1)) 59)
(((5 6 0) (2 3 1) (2 0 0) (2 0 1) (5 2 1) (1 2 1)) 75)
(((5 6 0) (2 3 1) (2 0 0) (2 0 0) (5 2 1) (1 2 1)) 82)
(((4 5 0) (2 3 1) (2 0 0) (4 2 1) (1 2 1)) 89)
(((4 5 0) (2 3 1) (2 0 0) (5 2 1) (1 2 1)) 109)
(((4 5 0) (2 3 1) (4 0 0) (5 2 1) (1 2 1)) 123)
(((4 5 0) (2 3 1) (4 0 0) (2 2 1) (1 2 1)) 138)
(((4 5 0) (2 4 1) (4 0 0) (2 2 1) (1 2 1)) 158)
(((3 4 0) (2 3 1) (2 2 1) (1 2 1)) 165)
(((3 4 0) (2 3 1) (4 2 1) (1 2 1)) 181)
(((3 4 0) (2 3 1) (4 3 1) (1 2 1)) 199)
(((3 4 0) (4 3 1) (4 3 1) (1 2 1)) 213)
(((3 4 0) (4 0 1) (4 3 1) (1 2 1)) 221)
(((3 4 0) (4 0 1) (4 2 1) (1 2 1)) 233)
(((3 4 0) (4 0 1) (3 2 1) (1 2 1)) 245)
(((3 4 0) (4 0 1) (2 2 1) (1 2 1)) 258)
(((3 4 0) (3 0 1) (2 2 1) (1 2 1)) 284)
(((3 4 0) (3 0 1) (3 2 1) (1 2 1)) 295)
(((3 4 0) (3 0 1) (2 2 1) (1 2 1)) 308)
(((3 4 0) (3 0 1) (3 2 1) (1 2 1)) 330)
(((3 4 0) (3 0 1) (4 2 1) (1 2 1)) 343)
(((3 4 0) (4 0 1) (4 2 1) (1 2 1)) 355)
(((3 4 0) (3 0 1) (4 2 1) (1 2 1)) 368)
(((3 4 0) (4 0 1) (4 2 1) (1 2 1)) 384)
(((3 4 0) (4 0 1) (2 2 1) (1 2 1)) 404)
(((3 4 0) (4 0 1) (3 2 1) (1 2 1)) 434)
(((3 4 0) (4 0 1) (2 2 1) (1 2 1)) 461)
(((3 4 0) (3 0 1) (2 2 1) (1 2 1)) 468)
(((3 4 0) (3 0 1) (4 2 1) (1 2 1)) 488)
(((3 4 0) (3 0 1) (4 2 1) (1 3 1)) 499)
(((3 4 0) (4 0 1) (4 2 1) (1 3 1)) 518)
(((3 4 0) (4 0 1) (4 2 1) (1 2 1)) 534)
(((3 4 0) (3 0 1) (4 2 1) (1 2 1)) 546)
(((3 4 0) (3 0 1) (3 2 1) (1 2 1)) 566)
(((3 4 0) (3 0 1) (4 2 1) (1 2 1)) 577)
(((3 4 0) (3 0 1) (4 2 1) (1 3 1)) 588)
(((3 4 0) (3 0 1) (4 2 1) (1 2 1)) 609)
(((3 4 0) (4 0 1) (4 2 1) (1 2 1)) 635)
(((3 4 0) (4 0 1) (4 2 1) (1 3 1)) 649)
(((3 4 0) (4 0 1) (3 2 1) (1 3 1)) 664)
(((3 4 0) (4 0 1) (3 2 1) (1 2 1)) 677)
(((3 4 0) (4 0 1) (3 2 1) (1 3 1)) 695)
(((3 4 0) (4 0 1) (4 2 1) (1 3 1)) 712)
(((3 4 0) (3 0 1) (4 2 1) (1 3 1)) 727)
(((3 4 0) (3 0 1) (4 2 1) (1 2 1)) 743)
(((3 4 0) (4 0 1) (4 2 1) (1 2 1)) 756)
(((3 4 0) (4 0 1) (3 2 1) (1 2 1)) 772)
(((3 4 0) (4 0 1) (3 2 1) (1 3 1)) 784)
(((3 4 0) (4 0 1) (4 2 1) (1 3 1)) 798)
(((3 4 0) (4 0 1) (3 2 1) (1 3 1)) 811)
(((3 4 0) (4 0 1) (4 2 1) (1 3 1)) 822)
(((3 4 0) (4 0 1) (4 2 1) (1 2 1)) 854)
(((3 4 0) (4 0 1) (3 2 1) (1 2 1)) 866)
(((3 4 0) (3 0 1) (3 2 1) (1 2 1)) 883)
(((3 4 0) (3 0 1) (3 2 1) (1 3 1)) 894)
(((3 4 0) (3 0 1) (3 2 1) (1 2 1)) 908)
(((3 4 0) (3 0 1) (3 2 1) (1 3 1)) 919)
(((3 4 0) (4 0 1) (3 2 1) (1 3 1)) 935)
(((3 4 0) (3 0 1) (3 2 1) (1 3 1)) 948)
(((3 4 0) (3 0 1) (3 3 1) (1 3 1)) 957)
(((3 4 0) (3 0 0) (3 3 1) (1 3 1)) 964)
(((2 3 0) (2 2 1) (1 2 1)) 971)
```

Figure 3-3: The Result of Simplification

```
[P2]   ((0 2 1)(1 0 0)) 7
[P4]   ((2 1 1)(3 1 1)(0 1 1)) 68
[P5]   ((4 3 1)(3 4 0)(2 1 0)(1 2 0)) 42
[P6]   ((3 2 1)(1 3 0)(2 1 0)) 174
[P1-] ((2 1 0)(2 2 1)) 145
[P2-] ((2 3 0)(2 2 1)(1 2 1)) 971
[P3-] ((1 5 0)(3 4 1)(2 3 0)(2 4 1)(2 1 0)) 363
[P4-] ((3 5 0)(2 2 1)(4 1 0)(2 0 0)(1 1 0)) <NOT-SIMPLEST>
[P5-] ((4 3 0)(6 6 0)(6 2 1)(1 5 1)(3 1 1)(5 4 1)) <NOT-SIMPLEST>
[P6-] ((2 3 0)(3 1 1)(1 2 1)) 44
[P7-] ((1 5 0)(4 6 0)(4 2 1)(4 3 1)(5 2 0)(4 0 0)) <NOT-SIMPLEST>
```

Figure 3-4: The Number of Steps to obtain the simplest machine

Problem	Hill-Climbing	Exhaustive-Search
P1	98	4
P2	141	170
P3	2052	553933
P4	510	8524
P5	1810	553933
P6	451	8524
P7	206	553933
P1-	445	170
P2-	1060	8524
P3-	2302	46593884
P4-	---	553933
P5-	---	553933
P6-	930	8524
P7-	---	46593884

3.4.2 Simplification from Trivial Machine

We have seen that our hill-climbing works rather successfully, although some problems could not be simplified completely. Our method consists of 2 parts, the construction process (chapter 2) and the simplification process (chapter 3). That is, we first construct a machine with 8 states and then simplify it. One might suppose that we could get the simplest machine using only the construction process, by choosing the number of states sufficiently small. Unfortunately, in the previous chapter, we showed that the number of states should be reasonably large, and we cannot do that. One might also notice that we would not need any construction process, because we can easily construct a *trivial machine*, which accepts exactly all strings in the right-list but nothing else. Figure 3-6 is an example of the trivial machine. In this section, we describe some experiments to try to simplify from the trivial machine. We shall see that to simplify from the trivial machine is much less effective than our construction-simplification method. The result of the experiments is shown in figure 3-7. When we compare figure 3-3 and figure 3-7, it is obvious that our construction-simplification method is more effective than the second method.

Figure 3-6: Trivial Machine of Problem 5

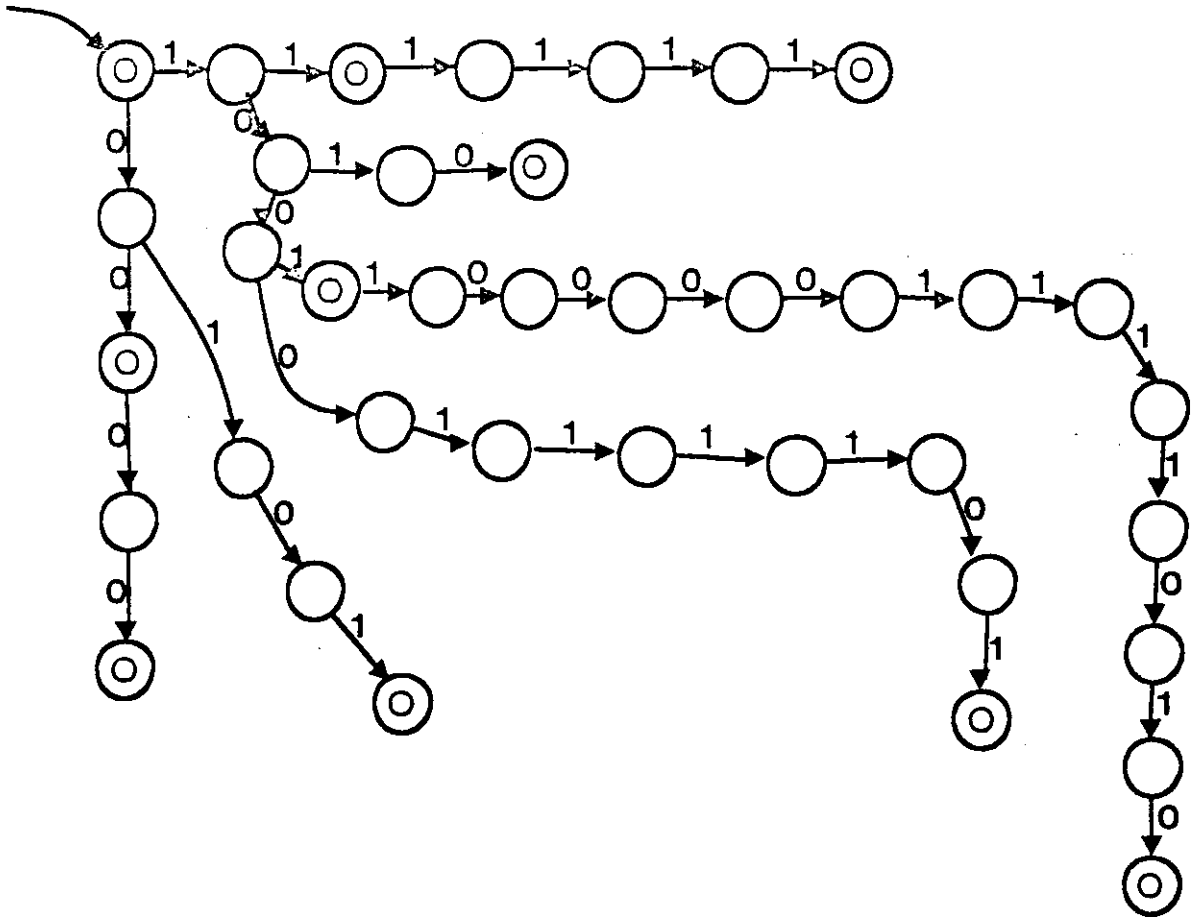


Figure 3-7: Result of Simplification from Trivial Machine

```
-------------------------------------------------------------------
[P1]  ((0 1 1)) 2 107
[P2]  ((0 2 1)(1 0 0)) 3 132
[P3]  ((1 2 1)(5 4 1)(3 6 0)(7 8 1)(6 0 0)(0 2 1)(3 0 1)(0 9 1)(10 0 0)
       (0 11 0)(0 12 0)(13 0 0)(14 0 0)(2 0 0)) 24 <NOT-SIMPLEST>
[P4]  ((3 2 1)(1 5 1)(4 2 1)(0 5 1)(0 2 0)) 12 <NOT-SIMPLEST>
[P5]  ((3 2 1)(4 1 0)(1 4 0)(2 3 0)) 9 1879
[P6]  ((3 2 1)(1 3 0)(2 1 0)) 7 1801
[P7]  ((3 2 1)(4 6 1)(6 5 1)(0 0 1)(1 2 1)(3 3 0)) 15 <NOT-SIMPLEST>
[P1-] ((2 1 0)(2 2 1)) 5 446
[P2-] ((3 2 0)(1 3 1)(3 3 1)) 8 1249
[P3-] ((1 2 0)(3 1 0)(5 4 1)(4 3 1)(3 5 0)) 12 <NOT-SIMPLEST>
[P4-] ((3 1 0)(2 2 1)(4 1 0)(2 1 0)) 9 3692
[P5-] ((3 2 0)(2 4 1)(1 6 1)(0 5 0)(8 0 1)(7 0 1)(0 10 1)(9 0 0)(0 3 0)
       (0 11 0)(0 12 0)(0 13 0)(0 14 0)(0 9 0)) 22 <NOT-SIMPLEST>
[P6-] ((3 2 0)(5 3 1)(4 1 1)(1 0 1)(1 2 0)) 12 <NOT-SIMPLEST>
[P7-] ((3 7 0)(2 2 1)(4 7 0)(0 5 0)(0 6 0)(7 0 0)(8 0 0)(2 8 0)) 13
       <NOT-SIMPLEST>
-------------------------------------------------------------------
```

4. Re-construction of Finite Automata

So far, we have described a method for constructing the simplest Finite Automaton from given examples. Suppose we have solved one problem, and are given another problem whose examples are very close to the previous one. To solve this new problem starting from the beginning is rather tedious because we already have some information about the solution. In this chapter, we describe how to re-construct a finite automaton if the right-list and/or wrong-list is slightly altered.

After the sample lists are altered, if the machine still accepts all strings in the right-list but no strings in the wrong-list, the previous solution is the new solution. If the machine does not accept some strings in the right-list, and/or does accept some strings in the wrong-list, we refer to such strings as *inconsistent strings*. Whenever we find an inconsistent string in the right-list, we call a procedure, *add-trivially*, which revises the machine, so that it accepts all strings in the right-list. On the other hand, whenever we find an inconsistent string in the wrong-list, we call a procedure, *cut-wrong-arrow*, which revises the machine, so that it accepts no string in the wrong-list. Although after calling add-trivially there is no inconsistent string in the right-list, there may now be another inconsistent string(s) in the wrong-list. In this case, we call cut-wrong-arrow. Similarly, although after calling cut-wrong-arrow there is no inconsistent string in the wrong-list, there may now be another inconsistent string(s) in the right-list. In this case, we call add-trivially. Thus, we call add-trivially and cut-wrong-arrow again and again.

We first define add-trivially and cut-wrong-arrow, and then we show that our process always terminates, producing the desired machine that accepts all strings in the right-list but no string in the wrong-list, although the machine is not the simplest.

4.1 Add-trivially

The purpose of this add-trivially routine is to accept an inconsistent string in the right-list, no matter how many strings in the wrong-list the machine comes to accept. We first define *trivial state* and *trivial path*, then finally we define add-trivially.

Definition: In each machine, we consider that there is a special arrow named *starting arrow*, which always points to the initial state q_1.

Definition: If more than one arrow (including the starting arrow and the one from q itself) point to a state q, then q is called a *non-trivial state*. If only one arrow points to q, then q is called a *trivial state*.

Definition: A sequence of states $q_{i(1)},q_{i(2)},\ldots,q_{i(k)}$ is called a *path* of a string $\alpha_1,\alpha_2,\ldots,\alpha_{k-1}$, where each α_i is in {1,0}, iff for all j such that $1 \leq j \leq k\text{-}1$, if $\alpha_j = 0$ then $A_{i(j)} = i(j+1)$ else $B_{i(j)} = i(j+1)$.

Definition: A sequence of states $q_{i(1)},q_{i(2)},\ldots,q_{i(k)}$ is called a *trivial path*, iff this sequence is a path, and for all j such that $2 \leq j \leq k$, $q_{i(j)}$ is a trivial state, and for all j such that $2 \leq j \leq k\text{-}1$, $q_{i(j)}$ is a non-final state, and $q_{i(k)}$ is a final state. This path accepts only one string.

That the machine M does not accept a string $\alpha_1,\alpha_2,\ldots,\alpha_k$ means either of the followings:

1. There is a path of $\alpha_1,\alpha_2,\ldots,\alpha_k$, but the last state is a non-final state.
2. There exists an integer j such that there is a path of $\alpha_1,\ldots,\alpha_{j-1}$, but the last state of this path does not have an α_j-arrow.

where each α_i is in {1,0}.

For each inconsistent string in the right-list, add-trivially works as follows: in case 1, let the last non-final state be the final state; in case 2, create a trivial path from the last state so that the machine accepts the whole string.

It is easy to show that after calling add-trivially the machine accepts all strings in the right-list. However, it also may come to accept some strings in the wrong-list, as we mentioned before. In this case, we call cut-wrong-arrow defined below.

4.2 Cut-wrong-arrow

If there are some inconsistent strings in the wrong-list (i.e. the machine does accept the strings), we call cut-wrong-arrow so that the machine comes to accept none of these strings, no matter how many strings in the right-list the machine comes to reject.

For each inconsistent string in the wrong-list, cut-wrong-arrow works as follows: Let $q_{i(1)},q_{i(2)},\ldots,q_{i(k)}$ be a path of the string w that should not be accepted. To reject w, one of the arrows of the path must be cut. Let $q_{i(n)}$ be one of the non-trivial states in the path.[7] Cut the arrow from $q_{i(n-1)}$ to $q_{i(n)}$. If q_1 (initial state) is the only non-trivial state, then let the machine M be **((0 0 0))**, which does not accept anything.

It is easy to show that after calling cut-wrong-arrow all strings in the wrong-list are rejected, although the machine may come to reject some strings in the right-list. In this case, we call add-trivially.

4.3 Termination

In this section, we show that the algorithm above always terminates.

Theorem: The algorithm above always terminates.

Proof: Consider the following partial ordering:

non-triviality of state: the number of arrows which point to the state.

non-triviality of machine: total of non-triviality of all non-trivial states.
We denote this by $nt(M)$, where M is a machine. Note that $nt(M) = 0$, iff M is a trivial machine.

Let M' be the result of adding-trivially to M, then $nt(M') = nt(M)$, because add-trivially adds only a trivial path. Next, let M' be a result of cut-wrong-arrow over M, then $nt(M') < nt(M)$, because we always cut the arrow that points to a non-trivial state q, and non-triviality of the state q decreases, and therefore non-triviality of machine also decreases. Thus, we cannot have an infinite loop, add-trivially, cut-wrong-arrow, add-trivially, cut-wrong-arrow, add-trivially,..... , because $nt(M)$ always decreases but $nt(M) \geq 0$. **<end of proof>**

---

[7] Such a non-trivial state always exists if the original machine has been simplified, and throughout this paper, we deal only with the re-construction of a simplified machine.

5. RR: Regular set Recognizer

Finally, we describe an actual system, RR, that learns to construct finite automata. RR is running in MACLISP either on CMU-20C or CMU-10A.

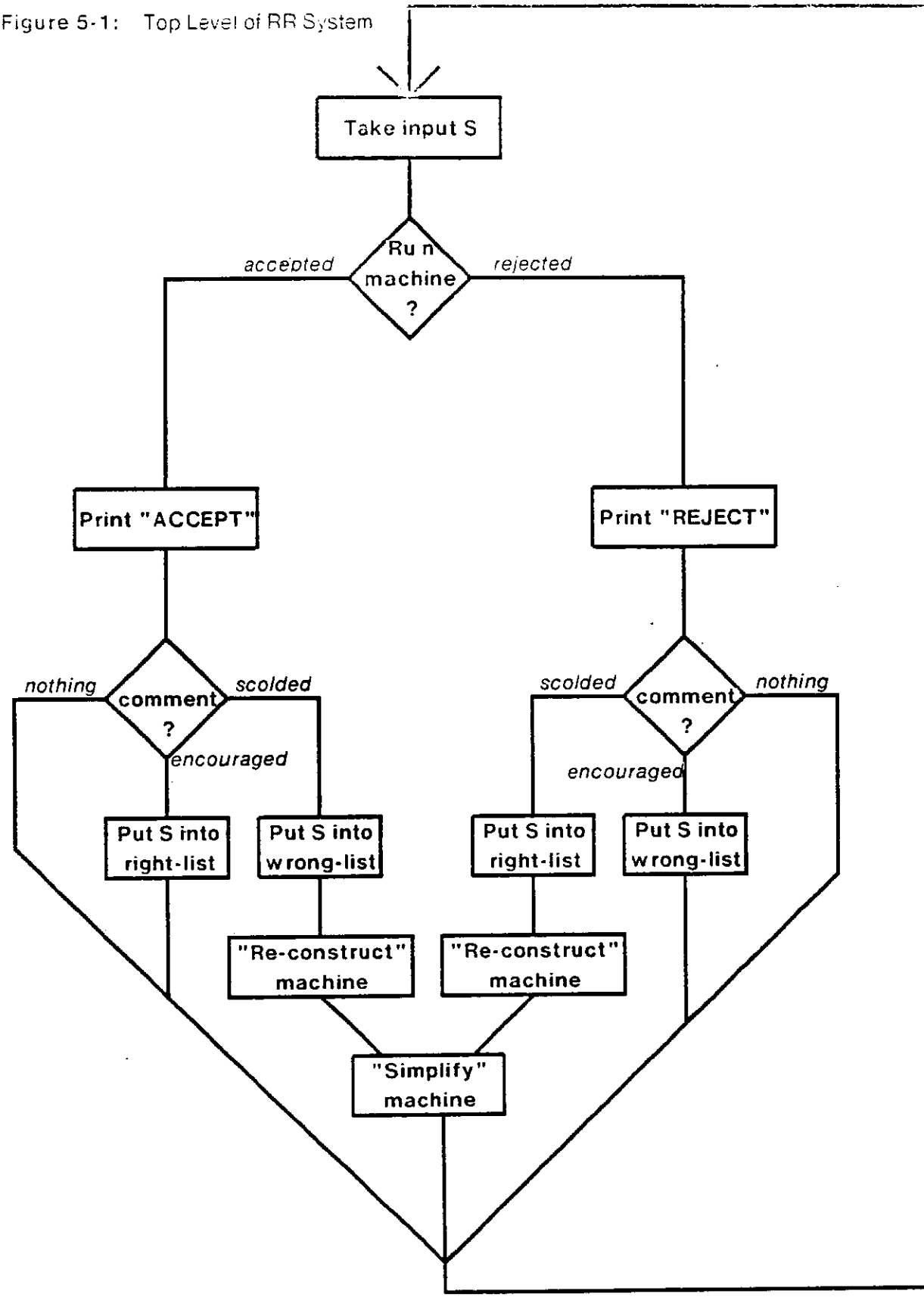
RR has a machine (finite automaton) and each time RR is given a string in $(1 + 0)^*$ as its input, RR runs the machine with the string given. If the machine accepts the string, RR answers ACCEPT, otherwise it answers REJECT. At the very beginning, RR has a null machine, which accepts nothing, and therefore RR does not accept any string at all. Now, consider some regular set $\mathbb{R}$ that we want to teach to RR. When we input a string s to RR, it should accept s if and only if s is in $\mathbb{R}$. If s in not in $\mathbb{R}$, RR should reject it. Whenever RR answers incorrectly, we *scold* it. When RR answers correctly and we think this example is important[8] , we *encourage* it. When RR is scolded or encouraged, it memorizes the fact that the string must be accepted or rejected, that is, if it is the case that the string must be accepted, RR puts it into right-list, which is a set of strings that must be accepted, and similarly, if the string must be rejected, RR puts it into wrong-list. After memorizing, RR re-constructs[9] the machine in the way described in chapter 4, so that it accepts all strings in the right-list and none in the wrong-list. After each re-construction, RR simplifies the machine in the way described in chapter 3.

Figure 5-1 shows a flow chart of the RR system.

[8] We do not need to encourage it every time it answers correctly.

[9] Only when it has been scolded.

Figure 5-1: Top Level of RR System



5.1 How to execute the RR system

In this section, we describe how to execute the RR system, and in the following section, we show several sample runs.

5.1.1 Getting Started

RR runs in MACLISP either on CMUC or CMUA. In MACLISP, type

`(slurp <tommy> rr)` (CMUC)

or

`(slurp c410mt80 rr)` (CMUA).

And call function:

`(main)` (both CMUA and CMUC).

Then you get prompt ">>>" and are in the RR system.

5.1.2 How to teach

- Giving example: The format for giving an example to RR is the following:

  (0-or-1 <space> 0-or-1 <space> . . . . . <space> 0-or-1)

  Typical input is:

  `>>> (1 0 1 0 1 0 1 0)`

  RR then outputs the answer, either **ACCEPTED** or **REJECTED**.

- Scolding: To scold for a wrong answer, input **n** right after the wrong answer.

  `>>> n`

- Encouraging: To encourage RR, input **y** right after the answer.

  `>>> y`

- Anyway-accept: If the example string starts with **+**, this means : *if this string is accepted then encourage; otherwise scold.* Typical input is:

  `>>> (+ 1 0 1 0 0 1)`

- Anyway-reject: If the example string starts with **-**, this means: *if this string is rejected then encourage; otherwise scold.* A typical input is:

  `>>> (- 0 0 0 1 0 0 1 0)`

To give the null string, use **( )** or **(+)** or **(-)**.

5.1.3 Other Commands

- **r**: show present right-list.
- **w**: show present wrong-list.

- m: show present machine.
- l: show last input.
- o: show order of memorized strings.
- t: show runtime of each step and total runtime.
- ?: show every thing above.
- new: initialize.
- ↑G: quit.

5.2 Sample Runs

5.2.1 Sample Run 1:

As the simplest example, let us teach the regular set 1^* to RR. The desired machine is:

((0 1 1)).

The underlined strings are user's inputs, and the Italic strings are comments.

```
[PHOTO:  Recording initiated  Thu 4-Mar-82 2:38PM]
 TOPS-20 Command processor 4(723)-7
@lisp
[Keeping]
MacLisp for TOMMY
*
(slurp <tommy> rr)
(<TOMMY> RR FASL)
(main)
>>> new                                  Initialization.
>>> ()                                   Input null string as an example.
REJECTED                                The null string was rejected.
>>> n                                    Since null should be accepted, scold it.
MODIFYING *                             It is trying to modify.
>>> (1)                                  Next, input (1).
REJECTED                                (1) was rejected.
>>> n                                    Since (1) should be accepted, scold it.
MODIFYING **                            It is modifying itself.
>>> (1 1 1)                              Next try (1 1 1).
ACCEPTED                                This was accepted, all right, no scolding.
>>> (0)                                  Next try (0), which should not be accepted.
REJECTED                                This was rejected, all right, no scolding.
>>> (1 0 1 1 1)                          Next try (1 0 1 1 1), which should not be accepted.
REJECTED                                Rejected, all right, no scolding.
>>> (1 1 1 1 1 1 1 1 1 1 1 1 1 1)        Next try this.
ACCEPTED                                Accepted, all right, it should be accepted.
>>> ?                                    Maybe we've got 1*, let us look inside the machine.
RIGHT-LIST
```

```
(NIL (1))                                  Right-list contents null string and (1).
WRONG-LIST
NIL                                        Wrong-list contents nothing.
PRESENT-MACHINE
((0 1 1))                                  Present machine is, yes, 1*.
LAST-INPUT
(+ 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
ORDER
((+) (+ 1))                                We taught it in this order.  + means "in right-list".
TIME
(0.019 0.048)                              Time spent to teach (+) and (+ 1).
TOTALTIME
(0.067)                                    Total time in seconds to learn 1*.
```

5.2.2 Sample Run 2:

Let us try to teach a harder automaton, problem 4. This regular set is:

The difference between the number of 0's and the number of 1's is divisible by 3.

For instance, the string **(1 0 1 1 1)** should be accepted because 4-1 = 3 is divisible by 3. The desired machine is as follows:

((3 2 1)(1 3 0)(2 1 0)).

```
>>> new
>>> ()                                     First, let us try null, which should be accepted.
REJECTED
>>> n                                      No, null should be accepted.
MODIFYING *
>>> m                                      Show the present machine.
((0 0 1))                                  This machine accepts nothing but a null string.
>>> (1 1 1)
REJECTED
>>> n                                      No, this should be accepted.
MODIFYING **
>>> m
((0 1 1))                                  This machine is 1*.
>>> (1)
ACCEPTED
>>> n                                      No, this should be rejected.
MODIFYING **
>>> m
((0 2 1) (0 3 0) (0 1 0))                  This machine is (1 1 1)*.
>>> (1 1 1 1 1 1)
ACCEPTED                                   All right, it should be accepted.
>>> (0)
REJECTED
>>> y                                      Yes, it should be rejected. Particularly, encourage it.
REJECTED
```

```
>>> (0 0 0)
REJECTED
>>> n                                        No. this should be accepted.
MODIFYING **
>>> (0 0 0 0 0 0)
ACCEPTED                                    All right.
>>> (0 0)
REJECTED                                    All right.
>>> m
((4 2 1) (0 3 0) (0 1 0) (5 0 0) (1 0 0))      (111 + 000)*.
>>> (1 0)
REJECTED
>>> n                                        No, this should be accepted.
MODIFYING ***
>>> m
((4 2 1) (1 3 0) (0 1 0) (2 0 0))
>>> (0 1)
REJECTED
>>> n                                        No, this should be accepted.
MODIFYING ***
>>> m
((3 2 1) (1 3 0) (2 1 0))                   Now, we get the desired machine.
>>> (1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0)
REJECTED                                    Ok.
>>> (1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 1 1)
ACCEPTED                                    Ok.
>>> ?
RIGHT-LIST
(NIL (1 1 1) (0 0 0) (1 0) (0 1))
WRONG-LIST
((1) (0))
PRESENT-MACHINE
((3 2 1) (1 3 0) (2 1 0))
LAST-INPUT
(+ 1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 1 1)
ORDER
((+) (+ 1 1 1) (- 1) (- 0) (+ 0 0 0) (+ 1 0) (+ 0 1))
TIME
(0.014 0.087 0.117 -0.01 0.364 0.564 1.413)
TOTALTIME
(2.549)
```

5.2.3 Sample Run 3:

The total run time to learn the desired machine depends very much on the order of input examples. We now try the previous sample again but with a different order.

```
>>> new
>>> ()
```

```
REJECTED
>>> n
MODIFYING *
>>> m
((0 0 1))
>>> (1 0)
REJECTED
>>> n
MODIFYING **
>>> (0 1)
REJECTED
>>> n
MODIFYING **
>>> m
((3 2 1) (1 0 0) (0 1 0))                    Present machine is (10 + 01)*.
>>> (1 1 1)
REJECTED
>>> n
MODIFYING ***
>>> m
((3 2 1) (1 3 0) (0 1 0))                    (10 + 01 + 111)*
>>> (1 1)
REJECTED
>>> (1 1 1 1 1 1)
ACCEPTED
>>> (0 0 0)
REJECTED
>>> n
MODIFYING ***
>>> m
((3 2 1) (1 3 0) (2 1 0))                    This is the desired machine.
>>> (0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1)
ACCEPTED
>>> (1 0 1 0 1 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1)
REJECTED
>>> ?
RIGHT-LIST
(NIL (1 0) (0 1) (1 1 1) (0 0 0))
WRONG-LIST
NIL
PRESENT-MACHINE
((3 2 1) (1 3 0) (2 1 0))
LAST-INPUT
(- 1 0 1 0 1 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1)
ORDER
((+) (+ 1 0) (+ 0 1) (+ 1 1 1) (+ 0 0 0))
TIME
(0.014 0.085 0.068 0.21 0.875)
TOTALTIME
(1.252)                                      The total time is much shorter.
```

5.2.4 Sample Run 4:

We next try problem 3, which is very hard. This regular set is:

Any strings without odd number of consecutive 0's AFTER odd number of consecutive 1's.

```
>>> new
>>> (+)(+ 1 )(+ 0)(- 1 0)(+ 0 1)(+ 1 1)(+ 0 0)(- 1 0 1)(- 0 1 0)(+ 1 0 0)
(+ 1 1 0)(+ 1 1 1)(+ 0 0 0)(- 1 0 1 0)(- 1 1 1 0)(- 1 0 1 1)(- 1 0 0 0 1)
(- 1 1 1 0 1 0)(- 1 0 0 1 0 0 0)(- 1 1 1 1 1 0 0 0)
(- 0 1 1 1 0 0 1 1 0 1)(- 1 1 0 1 1 1 0 0 1 1 0)
(+ 1 1 0 0 0 0 0 1 1 1 0 0 0 0 1)(+ 1 1 1 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0)
MODIFYING *
>>>
MODIFYING **
>>>
MODIFYING **
>>>
MODIFYING **
>>>
ACCEPTED
>>>
MODIFYING **
>>>
ACCEPTED
>>>
REJECTED
>>>
REJECTED
>>>
MODIFYING **
>>>
ACCEPTED
>>>
ACCEPTED
>>>
ACCEPTED
>>>
REJECTED
>>>
REJECTED
>>>
REJECTED
>>>
MODIFYING ********
>>>
REJECTED
>>>
MODIFYING ************
>>>
```

```
REJECTED
>>>
REJECTED
>>>
REJECTED
>>>
MODIFYING **
>>>
MODIFYING *****
>>> ?
RIGHT-LIST
(NIL(1)(0)(0 1)(1 1)(0 0)(1 0 0)(1 1 0)(1 1 1)(0 0 0)(1 1 0 0 0 0 0 1
1 1 0 0 0 0 1)(1 1 1 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0))
WRONG-LIST
((1 0)(1 0 1)(0 1 0)(1 0 1 0)(1 1 1 0)(1 0 1 1)(1 0 0 0 1)(1 1 1 0 1 0)(
1 0 0 1 0 0 0)(1 1 1 1 1 0 0 0)(0 1 1 1 0 0 1 1 0 1)(1 1 0 1 1 1 0 0 1 1 0))
PRESENT-MACHINE
((1 2 1)(3 1 1)(4 0 0)(3 4 1))          This is the desired machine.
LAST-INPUT
NIL
ORDER
((+)(+ 1)(+ 0)(- 1 0)(+ 0 1)(+ 1 1)(+ 0 0)(- 1 0 1)(- 0 1 0)(+ 1 0 0)(
+ 1 1 0)(+ 1 1 1)(+ 0 0 0)(- 1 0 1 0)(- 1 1 1 0)(- 1 0 1 1)(- 1 0 0 0 1)(
- 1 1 1 0 1 0)(- 1 0 0 1 0 0 0)(- 1 1 1 1 1 0 0 0)(- 0 1 1 1 0 0 1 1 0 1)(-
1 1 0 1 1 1 0 0 1 1 0)(+ 1 1 0 0 0 0 0 1 1 1 0 0 0 0 1)(+ 1 1 1 1 0 1 1 0 0
0 1 0 0 1 1 1 0 0))
TIME
(0.014 0.042 0.08 0.078 8.0E-3 0.116 8.0E-3 0.013 0.011 0.357 0.011 0.012
 8.0E-3 9.0E-3 0.01 0.011 2.066 0.012 3.686 0.014 0.018 0.02 0.283 3.735)

TOTALTIME
(10.282)
```

5.2.5 Sample Run 5:

We now try the previous run again with a more effective ordering.

```
>>> new
>>> (- 1 0)(- 1 0 0 1 0)(- 1 0 0 0)(- 1 0 0 1 1 0)(+)m(+ 0)m(+ 0 1)m
(+ 0 1 1 0)m(+ 1 0 0)m(+ 1 1)m(+ 1 0 0 1)m(+ 1 0 0 0 0)
REJECTED
>>>
REJECTED
>>>
REJECTED
>>>
REJECTED
>>>
MODIFYING *
```

```
>>>
((0 0 1))
>>>
MODIFYING **
>>>
((1 0 1))
>>>
MODIFYING *
>>>
((1 2 1) (0 0 1))
>>>
MODIFYING **
>>>
((1 2 1) (0 1 1))
>>>
MODIFYING *
>>>
((1 2 1) (3 1 1) (4 0 0) (0 0 1))
>>>
ACCEPTED
>>>
((1 2 1) (3 1 1) (4 0 0) (0 0 1))
>>>
MODIFYING **
>>>
((1 2 1) (3 1 1) (4 0 0) (0 4 1))
>>>
MODIFYING ***
>>> m
((1 2 1) (3 1 1) (4 0 0) (3 4 1))
>>> ?
RIGHT-LIST
(NIL (0) (0 1) (0 1 1 0) (1 0 0) (1 1) (1 0 0 1) (1 0 0 0 0))
WRONG-LIST
((1 0) (1 0 0 1 0) (1 0 0 0) (1 0 0 1 1 0))
PRESENT-MACHINE
((1 2 1) (3 1 1) (4 0 0) (3 4 1))      This is the desired machine.
LAST-INPUT
NIL
ORDER
((- 1 0)(- 1 0 0 1 0)(- 1 0 0 0)(- 1 0 0 1 1 0)(+)(+ 0)(+ 0 1)(+ 0 1 1 0)
(+ 1 0 0)(+ 1 1)(+ 1 0 0 1)(+ 1 0 0 0 0))
TIME
( 9.0E-3 9.0E-3 0.017 9.0E-3 0.016 0.047 0.029 0.091 0.042 0.013 0.38 0.342)

TOTALTIME
(0.89)                                  This is much faster than the previous run.
```

5.3 Discussion

We saw in the previous section that the run-time of sample run 3 is much shorter than the run-time of sample run 2, and also sample run 5 is much faster than sample run 4. Thus, RR is very sensitive to what is given as examples, and how these are ordered. In this section, we are interested in how to teach RR effectively.

First, we consider the worst case and the best case of re-construction. In the worst case, RR calls add-trivially and cut-wrong-arrow again and again, and eventually its machine becomes the trivial machine.[10] We know that a trivial machine can be constructed easily without such a special technique as re-construction.

On the other hand, the best case is that RR calls add-trivially once but no further cut-wrong-arrow. Thus, in order to "teach" the RR system effectively, we have to choose the examples nicely so that RR can re-construct its machine only by add-trivially. For instance, the example inputs of sample run 3 and sample run 5 are so chosen, and their run-time is in fact very short. Also, to avoid calling cut-wrong-arrow, we had better give the negative examples earlier.

[10] A trivial machine is a machine that accepts exactly all strings in the right-list and nothing else. See chapter 3.

6. Concluding Remark

Our new approach to construction of finite automata from given examples has been shown to work very nicely, despite the fact that its algorithm is quite simple. In chapter 2, we saw that construction of finite automata with n states can be nicely done using hill-climbing if n is a reasonable number. In chapter 3, we saw that we could often simplify the resulting machine of chapter 2 also using hill-climbing, although some problems could not be solved. In chapter 4, we discussed how to utilize past work, if a given problem is very close to the past problem. The RR system, which uses these techniques, was introduced and described in chapter 5. Finally, we enumerate several extensions of this work.

- Our hill-climbing algorithm sometimes climbs a local hill, and therefore fails to find a correct solution. There are several ways to avoid climbing a local hill, and one of them is *adaptive search* [Cavicchio 70], [Holland 75]. Adaptive search can be considered as a powerful version of hill-climbing. There are not only one "current generation", but usually a population of 20-30 . The best five or so are chosen as winners (the others are discarded) and 15-25 slightly-altered copies of them are made as the new population. Altering way is not only mutation, but also *cross-over* (mix two and produce one), *inverse* (inverse a certain part of one) ,[11] and so on. This approach becomes really powerful if parallel computation is available.
- Our finite automata have been deterministic, that is, arrows either exist or do not exist. The operator create-arrow or delete-arrow often makes too much difference to climb hill smoothly. The idea is to let our finite automata be *probabilistic*, that is, an arrow exists partially with a real number between 0.0 and 1.0, which indicates a probability of existence of the arrow. (See [Rabin 63].) In this case, we increase or decrease the real numbers, rather than create or delete an arrow. This method might help to climb hills smoothly.
- Our mutation function might be modified so that the mutation does not take place completely randomly, but somewhat "cleverly". For instance, if the machine accepts a string in the wrong-list, then delete-arrow or decrease-prob-of-arrow should take place more often on this wrong path than on others. Our idea becomes more concrete if we deal with the probabilistic automata described in the previous paragraph. If the machine somehow accepts a string in the wrong-list, then we should decrease all probabilities of the arrows on this path. If the machine accepts a string in the right-list, we increase the probabilities on this path, etc.
- Our problem domain in this paper has been regular sets. It might be possible to extend it to context-free sets by constructing *Push-Down Automata* (finite automata with stack, see [Hopcroft 79]). Since construction of Push-Down Automata must be much harder than finite automata, we would definitely need techniques just listed.
- A finite automaton can be viewed as a program that takes a string as its argument and outputs TRUE or FALSE. Therefore we might be able somehow to apply our method to automatic programming from specification by examples.

---

[11] The cross-over operator acts on a pair of strings by breaking each string at some point and rejoining the subsegments from different strings. The inversion operator makes two breaks, inverts the inner segment and then rejoin the string.

References and Bibliography

[Biermann 70] Biermann, A. W. and Feldman, J. A.
On the Synthesis of Finite-State Acceptors.
AI Memo 114, Stanford University, April, 1970.

[Buchanan 76] Buchanan, B. G.; Smith, D. H.; White, W. C.; Gritter, R. J.; Feigenbaum, E. A.; Lederberg, J.; and Djerassi, C.
Automatic rule formation in mass spectrometry by means of the Meta-DENDRAL program.
Journal of the American Chemical Society 98(6168), 1976.

[Cavicchio 70] Cavicchio, D. J.
Adaptive Search Using Simulated Evolution.
PhD thesis, University of Michigan, 1970.

[Elschlager 79] Elschlager, R. and Phillips, J.
Automatic Programming.
Report STAN-CS-79-758, Computer Science Department, Stanford University, August, 1979.

[Feldman 67] Feldman, J. A.
First Thoughts on Grammatical Inference.
AI Memo 55, Stanford Universy, Aug, 1967.

[Feldman 69] Feldman, J. A.; Gips, J.; Horning, J. J.; Reder, S.
Grammatical Complexity and Inference.
AI Memo CS125, Stanford Universy, June, 1969.

[Fogel 66] Fogel, L. J.; Owens, A. J. and Walsh, M. J.
Artificial Intelligence Through Simulated Evolution.
Wiley, New York, 1966.

[Gill 62] Gill, A.
Introduction to the Theory of Finite-State Machines.
Mcgraw-Hill Book Company, Inc., New York, 1962.

[Gold 74] Gold, E. M.
Complexity of automaton identification from given data.
1974.

[Hayes-roth 77] Hayes-Roth, F. and McDermott, J.
Knowledge acquisition from structural descriptions.
In *Proceeding of IJCAI-5*, pages 356-362. 1977.

[Holland 75] Holland, J. H.
Adaptation in Natural and Artificial Systems.
The University of Michigan Press, 1975.

[Hopcroft 79] Hopcroft, J. E. and Ullman, J. D.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, 1979.

[Hunt 66] Hunt, E.; Marin, J.; Stone, P.
Experiments in Induction.
Academic Press, New York, 1966.

[Langley 81a] Langley, P., Bradshaw, G. L., and Simon, H. A.
Rediscovering Chemistry With BACON.4.
CIP Working Paper 423, Carnegie-Mellon University, June, 1981.

[Langley 81b] Langley, P., Bradshaw, G. L., and Simon, H. A.
The Discovery of Conservation Laws.
CIP Working Paper 430, Carnegie-Mellon University, June, 1981.

[Lindsay 68] Lindsay, R. K.
Artificial Evolution of Intelligence.
Contemporary Psychology 13(3), March, 1968.

[London 64] London, R.
A Computer Program for Discovering and Proving Sequential Recognition Rules for BNF Grammers.
Technical Report, Carnegie Tech, May, 1964.

[Michalski 73] Michalski, R. S.
Discovering classification rules using variable-valued logic system VL1.
In *Proceeding of IJCAI-3*, pages 162-172. 1973.

[Rabin 63] Rabin, M. O.
Probabilistic automata.
Inform. Control 6:230-245, 1963.

[Tomita 82] Tomita, M.
Dynamic Construction of Finite Automata From Examples Using Hill-Climbing.
In *Proceedings of 4-th Annual Conference of the Cognitive Science Society.*
Cognitive Science Society, August, 1982.

[Vere 75] Vere, S. A.
Induction of concepts in the predicate calculus.
In *Proceeding of IJCAI-4*, pages 281-287. 1975.

[Winston 70] Winston, P. H.
Learning structural descriptions from examples.
PhD thesis, MIT, 1970.