Dynamic Construction of Finite Automata From Examples Using Hill-Climbing

Masaru Tomita

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

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.

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 machine by a "typical" schema-filling method (i.e., finding rough property in the samples first, comparing these strings carefully). However, in this paper,

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

we try to construct the machine directly by searching in the problem space (i.e., a set of all finite automata) using hill-climbing, rather than by analyzing the samples carefully.

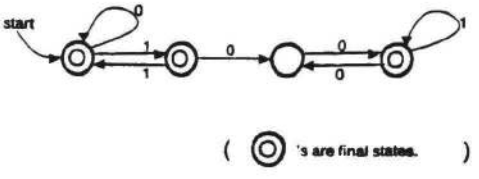
One of the biggest advantages of hill-climbing is its simplicity, that is, we do not have to know our problem space well, while a "typical" schema-filling method requires us to provide all possible schemas, and therefore to know everything about our problem space.

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 [7]), 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.

Figure 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 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. We

call the second task construction of finite automata, and the third task simplification of finite automata.

1.3. Sample Problems

Throughout this paper, we consider the particular seven problems shown in figure 2.

Figure 2: 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)
(1 0 1 0 1 0 1 0 1 0 1 0)	(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

[The problem we introduced at the beginning]

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 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 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)	

Problem 5

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

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*.

We also consider the inverse problem of those in figure 2. The inverse problems are created by exchanging the right-list and the wrong-list. We use these 14 problems in our experiments and refer to the inverse problem of problem i as i-.

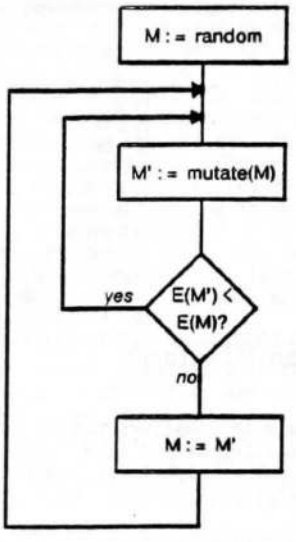
2. Construction of Finite Automata

In this section, we describe an experiment in constructing a finite automaton with n states from a given right-list and a wrong-list, using the 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 3.

Figure 3: 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.[2] That is, the mutation in our algorithm is randomly one of the following: delete an arrow, insert an arrow, change the destination of an arrow to another destination, make a non-final state 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. Result

We show in this section the result of our experiments. We first show in figure 4 the trace of the experiment of problem 3, to see how our algorithm gradually refines a random machine into the desired machine.

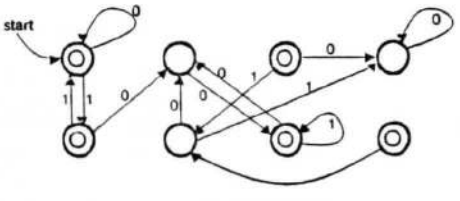
$$^{2}0 = < A_i = < 8; 0 = < B_i = < 8; \text{ and } 0 = < F_i = < 1.$$

Figure 4: 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)(5 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 2 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 22)
(((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)
(((1 4 1)(1 3 1)(0 8 1)(0 0 0)(0 5 0)(0 8 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 6 0)(0 5 0)(0 6 1)(7 6 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 6 0)(2 2 1)) 6 30)
                                    .
                                    .
                                    .
(((1 5 1)(5 0 0)(4 6 1)(0 0 0)(2 1 1)(2 4 0)(8 7 1)(5 0 1)) 12 2046)
(((1 5 1)(5 0 0)(4 6 1)(0 0 0)(2 1 1)(2 4 0)(8 7 1)(5 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)(5 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)(5 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)(5 0 1)) 13 2062)
-------------------------------------------------------------------------------------
total runtime 129.038000 sec
```

Each line corresponds to the current generation M. The column E indicates E(M), and G indicates the cumulative number of steps. The final machine of this trace accepts all strings in the right-list but none in the wrong-list of problem 3 (figure 5).

Figure 5: The final machine of problem 3



We show the result of the 13 other problems in figure 6, only by their final machines.

Figure 6: The Result of Construction

```
Problem                                                                          Steps
-------------------------------------------------------------------------------------
P1   ((0 1 1)(2 8 1)(7 4 1)(8 0 0)(7 2 1)(1 8 0)(7 7 0)(8 8 0)) 98
P2   ((0 5 1)(4 0 0)(2 8 0)(8 0 1)(1 5 0)(2 2 0)(1 4 1)(5 8 1)) 134
P4   ((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
P5   ((6 3 0)(6 6 1)(8 1 0)(1 6 0)(0 1 0)(1 7 0)(5 7 0)(3 5 0)) 1768
P6   ((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
P7   ((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
P1-  ((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
P2-  ((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
P3-  ((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
P4-  ((4 7 0)(2 7 1)(2 6 0)(8 1 0)(7 5 0)(7 4 1)(1 3 0)(2 0 0)) 246
P5-  ((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
P6-  ((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
P7-  ((7 6 0)(4 2 0)(4 2 1)(4 3 1)(6 2 0)(2 0 0)(7 5 0)(6 0 1)) 3725
-------------------------------------------------------------------------------------
```

2.3. Discussion

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 100000 times.

We show the expected number of steps using the exhaustive search calculated by this procedure in figure 7. 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 7: 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

3. Simplification of Finite Automata

In the previous section, we saw that our hill-climbing method successfully produced a machine that accepts all positive sample strings but no negative sample strings. 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 section, 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 the hill-climbing method as in the previous section.

3.1. Algorithm

The algorithm of the simplification is essentially the same as the algorithm described in the previous section. The major differences are as follows: the evaluation function E(M) returns a higher value if the machine M is simpler; 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; the algorithm starts with the result of the previous experiment instead of a random machine.

3.2. Result

The final machines of these experiments are shown in figure 8.

Figure 8: The Result of Simplification

```
[P2]  ((0 2 1)(1 0 0)) 7
[P4]  ((2 1 1)(3 1 1)(0 1 1)) 88
[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>
```

3.3. Discussion

We compare our method with an exhaustive search. The exhaustive search generates 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[3]. The result is shown in figure 9.[4] The symbol "---" indicates that the algorithm fails to find the simplest machine. This can happen when the hill-climbing algorithm climbs a "local hill".

Figure 9: The Number of Steps to obtain the simplest machine

Problem	Hill-Climbing	Exhaustive-Search
P1	98	4
P2	141	170
P3	2052	563933
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

4. Concluding Remark

Our new approach to construction of finite automata from given examples has been shown to work successfully, although it could not find the simplest machines for some problems. To avoid climbing a "local hill", it might be possible to apply adaptive search ([6], [2]) instead of our simple hill-climbing.

[3] Let n be the number of states of the desired simplest machine. Then the expected number of the 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,

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

[4] 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.

Although our problem domain has been regular languages, we might be able to extend it to context-free languages by constructing Push-Down automata (finite automata with stack, see [7]) using a similar method.

Acknowledgements

I would like to thank Herbert A. Simon and Jaime Carbonell for supervising this work; Masakazu Nakanishi, Yuichiro Anzai, Pat Langley and Takeo Kanade for thoughtful comments on an earlier version of this work; and Cynthia Hibbard for helping to produce this document.

References

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

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

[3] Feldman, J.A.
First Thoughts on Grammatical Inference.
AI Memo 55, Stanford University, August, 1967.

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

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

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

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

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