AN INTERACTIVE FINITE-STATE LANGUAGE LEARNER

Alan W. Biermann

(Ohio State University, Columbus, Ohio)

I. INTRODUCTION

The grammatical inference problem has been studied by a number of individuals [1,3,5,6,7,11,13, 14,15,16,18,19] as a model for theory formation or inductive behavior. Briefly stated, the problem is to find the grammar for a formal language L from observations of a finite number of strings in L and perhaps a finite number of strings not in L. A number of decidability results, definitions of learnability, and grammatical complexity measures have developed from this work, and a summary of some of these results appears in the survey paper by Biermann and Feldman [3]. However, because of the difficulty of the problem very few practical methods for actually doing grammatical inference have been forthcoming even for the simplest classes of grammars. Enumerative methods have been tried but the amount of computation involved is astronomical. This paper gives two practical solutions to the inference problem for finite-state languages, one straightforward method which follows directly from well known finite-state machine theory and one far more efficient technique which has been devised specifically for this application.

For convenience, we will be inferring finite-state acceptors in this paper rather than finite-state grammars. The theoretical difference between the two is for our purposes negligible. The model considered is for an interactive learning situation where the inference system has the ability to question its environment concerning the memberships of certain strings in the unknown language. Thus the problem in designing such a system involves finding a good question generating mechanism as well as a good synthesis or induction method, and a major measure of efficiency will be the number of questions asked before a correct answer is found. A system can be devised which simply asks all questions of length 2n-2 or less, $\sum_{i=0}^{2n-2} m^i$ questions, and then uses a standard finite-state machine synthesis technique [8,15] to produce the answer. We are assuming that the unknown language has a minimal acceptor with n states which are k-distinguishable (as defined below) and that the alphabet has m symbols. Such a system can be improved to the point where it asks far fewer questions, specifically $\sum_{i=0}^{k} m^i + nm^{k+1}$ questions, as shown below, but even this proves to be many more questions than necessary. We will argue that a correct inference should require only about $\log_2(n^{nm}2^n)$ questions and will present an algorithm which attempts to approach this bound.

The algorithms given here have the properties that they can efficiently discover the acceptor for any finite-state language and we can compute approximately the number of questions which will be asked in the process. Other algorithms have been given by Feldman, et.al. [7], Miller [14], Pao [16], and others but none can make these two claims. The first two references used heuristic techniques which are not known to be capable of learning all finite-state languages, and none of the papers gave general estimates of how many questions will be asked. There are a number of papers on interactive sequential machine synthesis [9,12,17] but the model differs from the one discussed here in that sequences of input-output pairs are given. In their model, much information is available about the states passed through during the computation whereas in the acceptor synthesis problem, no such information is known.

Finally, we will discuss the finite-state language learning capabilities of human beings and compare their performance with that of the algorithms presented here.

II. COVER-TREES

A finite-state acceptor A is a five-tuple $[Q, \Sigma, f, q_o, F]$ where Q is a finite nonempty set of states, Σ is a finite nonempty set of input symbols, f is the transition function which maps a subset of $Q \times \Sigma \rightarrow Q$, $q_o \in Q$ is the initial state, and $F \subset Q$ is the set of final states.

The function f is extended to map a subset of $Q \times \Sigma^* \rightarrow Q$ by the recursive definition $f(q,\Lambda) = q$ where Λ is the string of length zero and $q \in Q$, and $f(q,wa) = f[f(q,w),a]$ where $w \in \Sigma^*$ and $a \in \Sigma$. We will consider in this paper only acceptors with the property that for each $q_i \in Q$ there exists a $w \in \Sigma^*$ such that $f(q_o,w) = q_i$.

The language L(A) of the finite state acceptor A is

$$L(A) = \{w \in \Sigma^* \mid f(w,q_o) \in F\}.$$

We will be interested in constructing finite trees with labeled nodes and directed branches. Each node

will be given the name of a state beginning with the root node labeled q_o. A directed branch from a node labeled q_i to a node labeled q_j will be constructed and labeled a if $f(q_i,a) = q_j$. To expand a node labeled q_i will be to construct, for each input symbol a such that $f(q_i,a)$ is defined, a node labeled q_j (where $f(q_i,a) = q_j$) and a branch to the new node labeled a.

A cover-tree for a finite-state acceptor is constructed with q_o as its initial node as follows:

Algorithm 1. Cover-tree constructor.

Step 0. Construct the initial node and expand it.
Step i. Consider each node created in step i-1 and expand it unless another node with the same label has already been expanded either in the current step or a previous step. If any nodes were expanded in this step, go to step i+1.

It is clear that the algorithm for cover-tree construction terminates in a finite amount of time since for each state q_i there is at most one expanded node. In fact, for each state there is exactly one expanded node. This can be seen by assuming the contrary that there is a state q_j which corresponds to none of the nodes expanded in the cover tree. Assume that of the states that are not expanded in the cover tree, q_j is the one with the shortest possible $w \in \Sigma^*$ such that $f(q_o,w) = q_j$. Let $w = ua$, $a \in \Sigma$. Then $f(q_o,u) = q_h$ is expanded in the cover tree. But this implies that q_h was expanded in some step i showing that q_j was expanded in step i+1. This contradiction completes the proof.

Since every state of A appears in a cover tree (for A) expanded with all of its successors, it follows that the state set Q, the initial state q_o, the alphabet Σ, and the function f for A can all be constructed from its cover-tree. An example of an acceptor and its cover-tree appear in Figure 1.

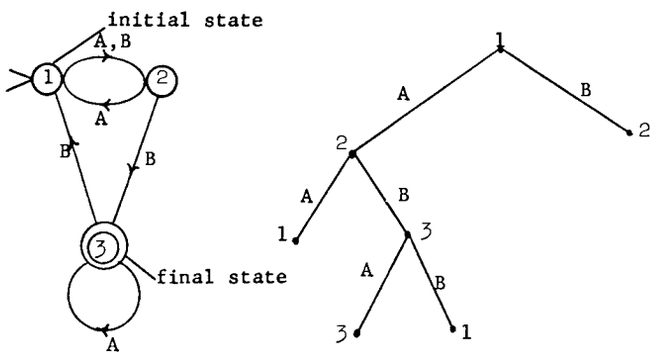


Figure 1. An acceptor and its cover tree.

The states of acceptor A will be said to be k-distinguishable if for each pair of distinct states q_i and q_j in Q there exists a $w \in \Sigma^*$ with length $(w) \leq k$ such that exactly one of $f(q_i,w)$ and $f(q_j,w)$ is in F. (We will consider minimal finite-state acceptors only here (8).)

The k-tail of $z \in \Sigma^*$ with respect to set of strings S from Σ^* is defined to be $g(z,S,k) = \{w \in \Sigma^* \mid zw \in S, \text{ length } (w) \leq k\}$.

Consider some particular cover tree for acceptor A. Let T_A be the set of strings (of length zero or more) representing paths on the tree which begin at the root node. A characteristic set C_A for A is defined as

$$C_A = \bigcup_{x \in T_A} x \cdot g(x,L(A),k)$$

where k is the smallest integer such that the states of A are k-distinguishable.

Referring to the cover-tree of Figure 1, we find $T_A = \{\Lambda,A,B,AA,AB,ABA,ABB\}$ and its associated characteristic set is $\{AB,ABA,ABAA,BB\}$. The reader may also construct a second cover-tree for the example problem and its corresponding characteristic set $\{AB,BB,BBA,BBAA\}$.

If we are given a characteristic set for an acceptor A with k-distinguishable states, we can easily construct the corresponding cover-tree using the approach of Algorithm 1. The only problem in constructing the cover-tree is in naming and expanding the nodes when C_A is given rather than A. In the current situation, we will name the states of the machine and their corresponding nodes with the function $g(w,C_A,k)$ where, for each node, w is the string representing the path from the root node to itself. Since $g(w,C_A,k)$ completely characterizes the state $q=f(q_o,w)$ out to its distinguishability length k, it is a reasonable name for that state. Thus, to expand the node $g(w,C_A,k)$, we construct for each $a \in \Sigma$ a new node $g(wa,C_A,k)$ and a branch to it labeled a.

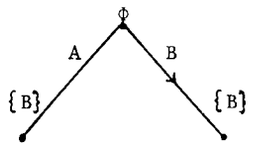
If the same node label is created more than once by some step i in the algorithm, we must be sure to expand the right node when processing step i+1. The right node is the node which was expanded when the characteristic set was created and it is easily recognized because it will be the only node j among the several identical nodes $g(w_1,C_A,k)$, $g(w_2,C_A,k)$, $\ldots,g(w_e,C_A,k)$ with the property that $w_j z \in C_A$ for some z, length $(z) > k$. If none of the identical nodes have this property, any one can be chosen as the right node.

We can thus find the cover-tree for A with k-distinguishable states from its characteristic set C_A by making the mentioned modifications in Algorithm 1. The nodes are labeled with the new state names $g(w,C_A,k)$ and the concept of node expansion is appropriately modified. Furthermore, only right nodes as defined above are expanded when two or more nodes in the same step have identical labels.

As an example, the cover-tree of Figure 1 can be constructed from its corresponding characteristic set $\{AB,ABA,ABAA,BB\}$ as shown in Figure 2.

Step 0.

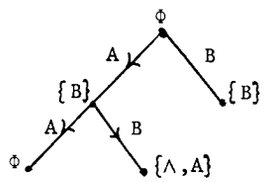
Construct the initial node and expand it.
Initial node = $g(\wedge,C_A,1) = \Phi$ (the empty set)
First successor = $g(A,C_A,1) = \{B\}$
Second successor = $g(B,C_A,1) = \{B\}$



Step 1.

Here we have two nodes from the previous level with identical labels. Choose the right node, $g(A,C_A,1)$, and expand.

$g(AA,C_A,1) = \Phi$
$g(AB,C_A,1) = \{\wedge,A\}$



Step 2.

We do not expand the node Φ from the previous step because a node with the same label was expanded in step 0. Expanding the node $\{\wedge,A\}$:

$g(ABA,C_A,1) = \{\wedge,A\}$
$g(ABB,C_A,1) = \Phi$

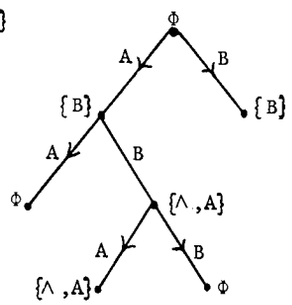


Figure 2. Synthesizing a cover-tree from its characteristic set.

In order to construct an acceptor from its characteristic set, we first construct the corresponding cover-tree and then synthesize the components Q,Σ,f,q_o, and F.

$Q = \{g(w,C_A,k) \mid w$ begins at the root node and ends on another node of the cover-tree.$\}$

Σ is the observed alphabet in C_A

$f(g(w,C_A,k),a)=g(wa,C_A,k)$ where $g(w,C_A,k)$ is an expanded node on the cover-tree.

$q_o = g(\wedge,C_A,k)$

$F=\{q \in Q \mid \wedge \in q\}$

Figure 3 shows the result of this procedure for the cover-tree of Figure 2.

Therefore an acceptor can be constructed if its characteristic set is known. We can thus design an interactive finite-state language learner if we can design a question asking routine which can construct

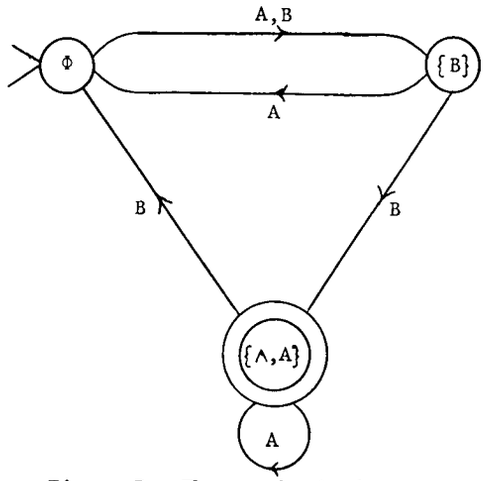


Figure 3. The synthesized acceptor.

the characteristic set. This approach yields Algorithm 2 given below which builds a characteristic set as it proceeds by interrogating its environment. The algorithm assumes that the input alphabet and the distinguishability length k are known initially. We will consider ways of removing these requirements in the following section.

Before giving Algorithm 2, it is helpful to define a question asking function $U(x,L,i)$ where $x \in \Sigma^*$, L is the unknown finite-state language, and i is a nonnegative integer. If the algorithm is to interrogate the environment to determine which strings $xy \in \Sigma^*$ with length (y) = i are in L, it employs the function $U(x,L,i)$.

$$U(x,L,i) = \{xy \in L \mid \text{length } (y) = i\}$$

Note that if Σ contains m symbols, invoking $U(x,L,i)$ implies the asking of exactly m^i questions.

Algorithm 2. Finite-State Language Learner.

Construct a set variable S which initially equals the empty set.

Step 0. $S \leftarrow \bigcup_{i=0,1,2\ldots,k} U(\wedge,L,i)$. Construct the initial node $g(\wedge,S,k) = S$.
Compute $S \leftarrow S \cup U(\wedge,L,k+1)$ and expand the initial node $g(\wedge,S,k)$ using the set S.

Step i. Consider each node $g(w,S,k)$ created in step i-1 which has the property that no other node with the same label has been previously expanded in the current step or a previous step. Compute $S \leftarrow S \cup U(w,L,k+1)$ and then expand $g(w,S,k)$. If any nodes were expanded in this step, go to step i+1.

Final step. Construct the acceptor from the synthesized cover-tree.

This algorithm can be illustrated using the example discussed above. Initially it is known that $\Sigma = \{A,B\}$ and k=1.

Step	Nodes Expanded, Questions/Answers	Nodes Created
0	Λ,A,B/no,no,no	Φ
0	$g(\Lambda,S,k)=\Phi$ AA,AB,BA,BB/no,yes,no,yes	{B},{B}
1	$g(A,S,k)=\{B\}$ AAA,AAB,ABA,ABB/no,no,yes, no	Φ,{Λ,A}
2	$g(AB,S,k)=\{\Lambda,A\}$ ABAA,ABAB,ABBA,ABBB/ yes, no,no,no	{Λ,A},Φ

Figure 4. Algorithm 2 applied to the example.

Clearly the cover-tree is identical to that in Figure 2 and so the acceptor of Figure 3 results.

One interesting point should be made about Algorithm 2; it always asks exactly $N(n,m,k)=\sum_{i=o}^{k} m^i+nm^{k+1}$ questions where n is the number of states, m is the size of the input alphabet, and k is the smallest integer such that the states are k-distinguishable. The algorithm must construct the initial node $\left(\sum_{i=o}^{k} m^i \text{ questions}\right)$ and expand exactly one node for each of the n states (m^{k+1} questions per expansion). Unfortunately, this is far too many questions considering the complexity of the problem being solved. There are less than $n^{mn}2^n$ different n-state acceptors and we can get approximately one bit of information from each question asked. Therefore, we should not expect an efficient algorithm to ask too many more than $\log_2(n^{mn}2^n)$ questions. The sample figures given in Figure 5 show that Algorithm 2 is far from being an optimum performer if n is very large.

Number of states n	Number of questions asked by Algorithm 2 N(n,m,k)	Approximate number of questions asked by an efficient learner. $\log_2(n^{mn}2^n)$
2	5	6
3	15	13
4	39	20
5	95	29
6	223	37

Figure 5.
Number of questions asked assuming k=n-2 and m=2.

III. A FINITE-STATE LANGUAGE LEARNER

In the previous section, a state q was characterized by the set $g(w,L,k)$ where $f(q_o,w)=q$. This requires that all $\sum_{i=0}^{k} m^i$ of strings wx (length (x) $\leq$ k) be classified concerning their memberships in L before q can be characterized. But it is important to remember that the only fundamental question which needs to be answered in Algorithm 2 is whether the states $f(q_o,w_1)$ and $f(q_o,w_2)$ are identical or different. This can often be determined after asking far fewer than $\sum_{i=0}^{k} m^i$ questions.

As an illustration, suppose the strings w_1AB, w_1BB,w_2BB are known to be in language L, and it is desired to know whether the states $f(q_o,w_1)$ and $f(q_o,w_2)$ are identical. Algorithm 2 above would have asked the questions (assuming k = 2) w_1,w_1A,w_1B, w_1AA,w_1BA,w_2,w_2A,w_2B,w_2AA,w_2AB, and w_2BA, and would have consequently completely resolved the problem. (Construct $g(w_1,L,2)$ and $g(w_2,L,2)$ and compare them.) The approach of this section will be to ask all of the questions in the set $w_2\cdot[g(w_1,L,2)-g(w_2,L,2)]$ and all the questions in the set $w_1\cdot[g(w_2,L,2)-g(w_1,L,2)]$. If the two states appear to be the same after these questions, they are assumed to be the same for the time being, and if they are proven to be different, the problem is resolved and the questioning is terminated. In this example, $w_2\cdot[g(w_1,L,2)-g(w_2,L,2)] = w_2\cdot(\{AB,BB\}-\{BB\}) = \{w_2AB\}$ and the other set yields no questions at all. So we ask only the question w_2AB and classify $f(q_o,w_1)$ and $f(q_o,w_2)$ as identical or different depending on whether w_2AB is in or not in L. Whereas Algorithm 2 asked eleven questions, our current approach is to ask only one.

This method has the pitfall that it will sometimes classify states as being identical when they are not. It, however, never classifies states different when they are, in fact, the same. The consequence is that constructed machines during the learning process never have more states than the final solution machine. After a machine is constructed using this imprecise state classification scheme, it is tested to check whether it accepts all of the strings known to be in L and none of the strings known to be not in L. If it passes both tests, it is assumed to be correct, and if it fails, more questions are asked to fill in gaps in information.

Algorithm 3 will begin assuming that a scattering of strings from L are available. If none are, a few random questions can be asked until such a collection is obtained.

Algorithm 3 will, therefore, function as follows: Set k = 0 and do a construction much like Algorithm 2 but with minimum question asking as illustrated above. If the resulting machine is compatible (A is compatible with the set of known strings if it accepts all strings known to be in L and none of the strings known to be not in L.) with all known strings in and not in L, stop. Otherwise, ask more questions to try to discover wrongly identified states, and when an additional string in L is found, enter Algorithm 2 again with minimum question asking. Repeat this loop until either a solution is found or until a complete characteristic set is constructed for the current value of k. If the resulting machine is still not correct, check whether any transitions to empty states have

been omitted. (A peculiarity of the above construction is that under certain conditions it omits such transitions.) If repair of all such omissions still does not produce a compatible acceptor, increment k and begin again.

Because of the increased complexity of this method, additional notation must be introduced. The inputs to Algorithm 3 will be a scattering of strings (at least one) from the unknown language and perhaps some samples of strings not in the language. These two sets, the first of which is nonempty, will be denoted S and B, respectively. The cover-tree which is constructed will be stored as a set R of pairs (x,i) where $x \in \Sigma^*$ and i is an integer, the name assigned to the state $f(q_o,x)$. Thus the cover tree of Figure 1 would be stored as {(∧,1), (A,2), (AA,1), (AB,3), (ABA,3), (ABB,1), (B,2)}. If a node is labeled i and another node labeled .i has been previously expanded, the string x associated with this node is stored in T. Thus T is the set of strings x whose associated nodes are not to be expanded. Again referring to Figure 1, the strings AA, ABA, ABB, and B would be included in T by the time the algorithm terminated. V will denote a set of strings from which the questions to be asked will be taken.

Consider all the prefixes x of strings in S. (x is a prefix of z if there is a $y \in \Sigma^*$ such that xy = z.) Order them according to length, shortest to longest, and then alphabetize the strings of equal length with respect to each other. (Example: ∧,A,B,AA,AB,BA,BB,---) p(i,S) will be defined to be the i^{th} string in such an ordering of prefixes of S.

The dot operation · denotes concatenation so that $x \cdot Y = \{xy \mid y \in Y\}$ where $x \in \Sigma^*$ and Y is a set of strings $y \in \Sigma^*$. Also $\Sigma^i = \{x \in \Sigma^* \mid \text{length}(x) \leq i\}$.

If a new string x in L is found during the interrogation process, the cover-tree constructed up to the current point is invalid, and the algorithm must backtrack to the point where x would affect the computation. This backtracking is implemented by the function reset which is defined using an ALGOL-like format.

```
procedure reset;
begin
t ← max {i|p(i,S) = p(i,Su{x})};
t ← max {i,0| length(p(i,S))<length(p(t,S))-k}
T ← T - {p(i,S)|i > t};
R ← R - {(y,i) ∈ R|p(j,S) = y implies j > t};
S ← S u {x};
end;
```

The function question (V) chooses by some criterion a string from the set V and calls it the string x. The environment is interrogated concerning whether $x \in L$ or $x \notin L$, and question returns the value true or false, respectively. If $x \notin L$, then $B \leftarrow B \cup \{x\}$.

If $f(q_o,x)$ is being identified as a new state, it is given a new name, and if it is being assumed to be identical to some other node $f(q_o,y)$, it is given the same name. These two actions are performed by the functions newname and samename as defined here.

```
procedure newname (x);
begin
R ← R ∪ {x,nextname};
nextname ← nextname + 1;
end;

procedure samename (x,y);
begin
T ← T ∪ {x};
R ← R ∪ {(x,j)} where (y,j) ∈ R;
end;
```

The function compatible will check whether the acceptor A(R,S) constructed from the cover-tree represented by R accepts all of the strings in S and none of the strings in B. If so, compatible (A(R, S),S,B) returns value true; otherwise, it returns false. We will assume that R is an ordered set so that we can speak of its i th element R_i. The i th element is a pair (x,j) whose first and second elements will be referred to as car(R_i) and cadr(R_i), respectively. The function number(S) gives the number of elements in the set S.

These definitions make it possible to specify Algorithm 3 very precisely using the ALGOL-like form. The inputs are the initial sets S and B, and the output after sufficient environment interrogation is an acceptor A which is compatible with the initial S and B and all answers to subsequent questions.

Algorithm 3. Improved Finite-State Language Learner

Comment: Apply Algorithm 2 with minimum question asking.

```
k ← 0;
beginning T ← Φ;
R ← {(∧,1)};
V ← Φ;
nextname ← 2;
r ← number {y|y = p(i,S),i=integer};

t ← 2;
go to start;
backtrack: reset;
start:
for i ← t step 1 until r do
      begin
      if p(i,S) = yz such that y∈T then go to
                                          incrementi;
      for j ← 1 step 1 until i-1 do
            begin
            V ← p(j,S)·[g(p(i,S),S,k)-g(p(j,S),
                                          S,k)];
            if V = Φ then go to continue;
            V ← V - B;
            if V = Φ then go to incrementj;
            yes ← question (V);
            if yes then go to backtrack;
            go to incrementj;
            continue:
            V ← p(i,S)·[g(p(j,S),S,k)-g(p(i,S),
                                          S,k)];
            if  V=Φ then go to equal;
            V ← V-B;
            if V=Φ then go to incrementj;
            yes ← question (V);
```

I

```
                if yes then go to backtrack;
                incrementj:  end;

            newname(p(i,S));
            go to incrementi;
            equal:  samename (p(i,S),p(j,S));
            incrementi:  end;
```

Comment: Check compatibility.

```
II  If compatible (A(R,S),S,B) then halt (yielding
    result A(R,S));
```

Comment: Generate additional questions.

```
III c ← number (R);
    V ← Φ;
    for i ← 1 step 1 until c do
          begin
          x ← car (R_i);
          if x=yz such that y∈T then go to proceed;
          V ← V ∪ x·(Σ^{k+1} - g(x,S,k+1));
          proceed:  end;
    V ← V - B;
    nextquestion:  if V=Φ then go to finalcheck;
    yes ← question (V);
    if yes then go to backtrack;
    V ← V - {x};
    go to nextquestion;
```

Comment: Add omitted transitions into empty states.

```
IV  finalcheck:
    for i ← 1 step 1 until c do
          begin
          x ← car (R_i)
          if x=yz such that y∈T then go to next1;
          if g(x,S,k)=Φ then
               begin
               j ← cadr (R_i)
               go to emptystate;
               end;
          next1;  end;

    k ← k+1;
    go to beginning;
    emptystate:
    for i ← 1 step 1 until c do
          begin
          x ← car (R_i)
          if x=yz such that y∈T then go to next2;
          for each a∈Σ do
               if (xa,h) ∉ R for all h then
                    R ← R ∪ {(xa,j)};
          next2:  end;
    if compatible (A(R,S),S,B) then halt (yielding
    result A(R,S));

    k ← k +1
    go to beginning;
```

Let A_o be the minimal acceptor for the unknown language L. This algorithm keeps building cover-trees until it finds one which yields an acceptor compatible with S and B. If k is too small, the algorithm will either yield a wrong answer (which is not A_o but is compatible with S and B) or it will increment k. If k equals the smallest integer k_o such that the states of A_o are k_o-distinguishable, the algorithm will halt in a finite time with either the right answer A_o or a wrong answer, and it will not increment k. In the case where the algorithm halts with a wrong answer A_1, one or more additional samples from L (or its complement) which were wrongly classified by A_1 can be added to S (or B), and the algorithm can be restarted with the assurance that the next acceptor produced will not err on the newly presented strings. In practice, this process always converges relatively quickly to the correct answer A_o.

Program Segment	k	T	R, Questions,Answers
			begin
I	0	Φ	(∧,1)
		A	(∧,1),(A,1)
II			A(R,S) not compatible
III			∧? yes
			reset
I	0	Φ	(∧,1)
			(∧,1),(A,2) A? No
		AA	(∧,1),(A,2),(AA,2) AA? No
II			A(R,S) not compatible
III			B? No AB? yes
			reset
I	0	AA	(∧,1),(A,2),(AA,2)
		AA,AB	(∧,1),(A,2),(AA,2),(AB,1)
II			A(R,S) not compatible
III			No questions
IV		AA,AB	(∧,1)...,(AB,1),(B,2)
			A(R,S) compatible, halt
			Add string BBB to set S.
			begin
I	0	Φ	(∧,1)
			(∧,1),(A,2)
		B	(∧,1),(A,2),(B,2)
		B,AA	(∧,1),(A,2),(B,2),(AA,2)
		B,AA,AB	(∧,1),...,(AA,2),(AB,1)
II			A(R,S) not compatible
III			No questions
IV			
			begin
I	1	Φ	(∧,1)
			(∧,1),(A,2)
			(∧,1),(A,2),(B,3) BB? No
		AA	(∧,1),(A,2),(B,3),(AA,3)
			AAB? No
		AA,AB	(∧,1),...,(AA,3),(AB,1)
		AA,AB,BB	(∧,1),...(AB,1),(BB,2)
II			A(R,S) not compatible
III			BA? Yes
			reset
I	1	Φ	(∧,1),(A,2)
			(∧,1),(A,2),(B,3)
		AA	(∧,1),(A,2),(B,3),(AA,3)
			AAA? Yes
			reset
I	1	AA	(∧,1),(A,2),(B,3),(AA,3)
		AA,AB	(∧,1),...,(AA,3),(AB,1)
		AA,AB,BA	(∧,1),...,(AB,1),(BA,1)
		AA,AB,BA,BB	(∧,1),...,(BA,1),(BB,2)
II			A(R,S) compatible halt

Figure 6
Example computation by Algorithm 3. Initial S={AAABBB} and the string BBB is added when the first hypothesized inference is incorrect.

	Problem	Human Performance: Per Cent Solved	Human Performance: Average* time(secs.)	Human Performance: Average number* of questions asked	Computer Performance: Per Cent Solved	Computer Performance: CPU time (sec.)	Computer Performance: Number of questions asked	Computer Performance: Total No. of strings available	Computer Performance: Was solution exact?
1.	B+AA*BB+A	100	422	15.3	100	4	10	15	no
2.	AA(BBA*)*	78	504	15.6	100	4	8	13	yes
3.	AA+A+A*AB	87	342	12.3	100	4	11	25	yes
4.	A*(A+BA)A	87	523	21.8	100	4	20	25	yes
5.	BB*A(AA+ABA)	87	580	18.1	100	4	11	16	yes
6.	A+A(A+B*)BA*	50	907	24.7	100	4	14	19	yes
7.	B*B(ABBA+A)*	57	825	24.4	100	4	11	26	no
8.	AA*(ABAB)*A*	37	346	12.3	100	4	15	20	no

*Averages do not include subjects who failed to obtain a perfect test score.

Figure 7.
Finite-State Language Learning Performance

In order to demonstrate the operation of Algorithm 3, consider the finite-state language L on the alphabet {A,B} which has the property that w∈L if and only if the number of A's minus the number of B's in w is divisible by three. If the algorithm is given the sample initial string AAABBB, it asks five questions and halts with the wrong answer as shown about half way down Figure 6. So we give the algorithm one more string, BBB, and restart it. This time it asks four questions and halts with the correct answer. Thus the total number of strings available to the system was eleven at the time that it produced its correct solution whereas Algorithm 2 would have required fifteen. The difference between the performance of the two methods, of course, becomes much more dramatic in larger problems.

IV. SOME EXPERIMENTAL RESULTS

The algorithm described in the previous section is significantly better than Algorithm 2 in terms of numbers of questions asked because of its efficient use of the string information. A version of this algorithm was programmed, tested extensively, and found to be an efficient language learner. We will examine here a series of tests in which the algorithm was asked to learn finite-state languages in competition with human beings.

Eight regular expressions were generated using a random method, four of length nine and four of length twelve. The subject, man or machine, was given five strings from the regular set, one chosen from each of the lowest five lengths in the set. He was then free to ask as many questions as desired concerning the membership of other strings in the unknown language. When he was satisfied that he knew the language, he was given a set of ten strings to classify. If he failed to classify all strings correctly, he was allowed to ask more questions and later take one or even two more ten-question tests. The subject was assumed to have learned the language when he could obtain a perfect score on such a test. Twelve subjects and the computer participated in the experiment although not all subjects had time to complete all problems. The psychological test was computer administered in a carefully controlled environment with all possible precautions taken to obtain accurate measurements. Subjects were rewarded with a ringing bell, twenty-five cents, and congratulations from the experimenter for each language learned. See Figure 7 for a summary of results.

The human subjects were not able to solve all of the problems that they were given and they seemed to find more difficulty with the longer regular expressions. The computer tended to ask fewer questions than the humans and usually (five times) obtained a perfect score on the first test. Only about one third of human successes came on the first test.

The criterion of success was a perfect score of ten on a test. In contrast to the case with human beings, it was possible to check whether the machine really had the right answer when it obtained its perfect score. In five of the eight problems, it did produce a solution which was exactly correct.

Correctly classified strings are available during the learning process from the initial set of strings, the questions asked, and the tests taken. If we note the figures for number of strings required by the computer to learn the language and remember that all of these languages have minimal acceptors of about four to six states, we see that the performance of this program is about the best that can be hoped for. Needless to say, it soundly outperforms Algorithm 2 as shown in Figure 5. The average CPU time required for the solution of each problem was less than four seconds using the Stanford Artificial Intelligence Language SAIL on a PDP-10 computer.

The program discussed in this section differed from Algorithm 3 in that it increased k after checking less stringent criteria then does Algorithm 3. This version seemed to ask fewer questions but it is not known whether it will always halt after asking a finite number of questions as will Algorithm 3.

V. SUMMARY

A finite-state language learner designed along the straightforward lines of Algorithm 2 will ask on the

order of $\sum_{i=1}^{k} m^i + nm^{k+1}$ questions in the process of learning a language. Such a method can be improved greatly as exemplified by Algorithm 3, perhaps to the point of learning after only $\log_2(n^{mn}2^n)$ questions. Some experimental evidence that this goal can be approached is presented.

A comparison is made of the finite-state language learning abilities of humans and a computer program with the conclusion that the program was slightly better on the set of problems used.

VI. ACKNOWLEDGEMENT

The first interactive finite-state language learner of the style described here was written by one of my students, Malcolm Newey, in a gradute research seminar at Stanford.

I am also deeply indebted to Professor J.A. Feldman who first introduced me to the grammatical inference problem and has encouraged me in my studies ever since.

Finally, I would like to thank my wife, Dr. Alice M. Gordon, and Dr. L.M. Horowitz for help and consultation on the psychological experiments.

The research reported here was done partly in the Computer Science Department, Stanford University and was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183)and the National Science Foundation Grant No. GJ-776. Part of the work was done in the Computer and Information Science Department at The Ohio State University and was supported by Grant Number GN-534.1 from the Office of Science Information Service, National Science Foundation.

REFERENCES

[1] A.W. Biermann, "A Grammatical Inference Program for Linear Languages," Fourth Hawaii International Conference on System Sciences, Honolulu, Hawaii, Jan. 12-14, 1971.

[2] A.W. Biermann and J.A. Feldman, "On the Synthesis of Finite-State Acceptors," A.I. Memo No. AIM-114, Computer Science Department, Stanford University, April 1970.

[3] A.W. Biermann and J.A. Feldman, "A Survey of Results in Grammatical Inference", International Conference on Frontiers of Pattern Recognition, University of Hawaii, Honolulu, Hawaii, Jan. 18-20, 1971.

[4] A.W. Biermann and J.A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior", to appear, IEEE Transactions on Computers, 1972.

[5] J.A. Feldman, "First Thoughts on Grammatical Inference", Stanford A.I. Memo No. 55, August 1967.

[6] J.A. Feldman, "Some Decidability Results on Grammatical Inference and Complexity", Information and Control, Vol. 20, pp. 244-262, 1972.

[7] J.A. Feldman, J. Gips, J.J. Horning, S. Reder, "Grammatical Complexity and Inference," Technical Report No. CS125, Computer Science Dept., Stanford University, June 1969.

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

[9] A. Gill, "Realization of Input-Output Relations by Sequential Machines", Journal of the Association for Computing Machinery, Vol. 13, No. 1, pp. 33-42, 1966.

[10] S. Ginsburg, "Synthesis of Minimal-State Machines", IRE Transactions on Electronic Computers, EC8, pp. 441-449, 1959.

[11] M. Gold, "Language Identification in the Limit", Information and Control, Vol. 10, pp. 447-474, 1967.

[12] J.N. Gray and M.A. Harrison, "The Theory of Sequential Relations," Information and Control, Vol. 9, pp. 435-468, 1966.

[13] J.J. Horning, "A Study of Grammatical Inference", Technical Report No. CS139, Computer Science Department, Stanford University, August 1969.

[14] G.A. Miller, "Finite-State Machine Induction", M.S. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, May 1969.

[15] A. Nerode, "Linear Automaton Transformations, "Proceedings of the American Mathematical Society, IX, pp. 541-544, 1958.

[16] T.W.L. Pao "A Solution of the Syntactical Induction-Inference Problem for a Non-Trivial Subset of Context-Free Languages," Report No. 70-19, The Moore School of Electrical Engineering, University of Pennsylvania, August, 1969.

[17] A.A. Tal, "Questionnaire Language and the Abstract Synthesis of Minimal Sequential Machines", Avtomatika i Telemekhanika, Vol. 25, No. 6, pp. 946-962, 1964.

[18] S. Crespi-Reghizzi, "The Mechanical Acquisition of Procedence Grammars," Report No. UCLA-ENG-7054, School of Engineering and Applied Science, University of California at Los Angeles, June 1970.

[19] R. Solomonoff, "A Formal Theory of Inductive Inference", Information and Control, Volume 7, pp. 1-22, 224-254, 1964.