STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-152

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-241

ON THE INFERENCE OF TURING MACHINES FROM SAMPLE COMPUTATIONS

BY

A. W. BIERMANN

OCTOBER 1971

COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-152

OCTOBER 1971

ON THE INFERENCE OF TURING MACHINES FROM SAMPLE COMPUTATIONS

by

A.W. Biermann

ABSTRACT: An algorithm is presented which when given a complete description of a set of Turing machine computations finds a Turing machine which is capable of doing those computations. This algorithm can serve as the basis for designing a trainable device which can be trained to simulate any Turing machine by being led through a series of sample computations done by that machine. A number of examples illustrate the use of the technique and the possibility of its application to other types of problems.

This research was supported by the Advanced Research Projects Agency of the Department of Defense (SD-183) U.S.A., and the National Science Foundation, Grant No. GJ-776.

1. Introduction

The traditional means for obtaining the desired performance from a computer is to write a program which specifies in abstract notation and in complete detail exactly what is wanted. This paper will be concerned with the problem of obtaining this performance from the machine by giving it examples of the desired computation and having it program itself. We will be concerned with designing a trainable Turing machine although the concepts presented are applicable in a much more general context as discussed in Section 4.

The Turing machine to be discussed here will have an infinite one dimensional tape and will have the capability in one move to read a symbol on the tape, print a new symbol to replace the one just read, and step right or left one increment on the tape. It will have a deterministic finite-state controller with a designated initial state which will upon receiving an input symbol read from the tape, yield the symbol to be printed and the step direction (right or left) to be made. A computation will be defined to be the complete sequence of moves which are executed by a machine starting in its initial state with its head on the left-most nonblank symbol of the tape and ending at a halting condition with the device reading a symbol and in a state such that no next move is defined. Initial tapes will be assumed to have only a finite number of nonblank symbols, and we will be interested only in computations of finite length. A particular Turing machine will be said to be able to <u>execute</u> a particular computation if when given the initial tape associated with that computation, it goes through the sequence of moves in the computation and halts after the last move.

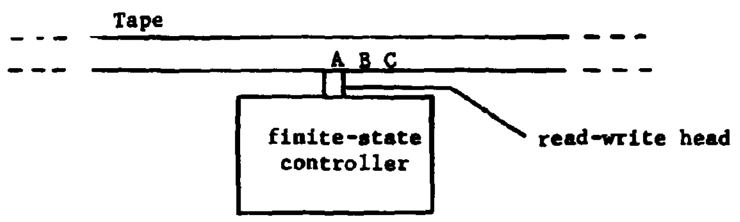


Figure 1. A Turing Machine

A move will be written as a three symbol string with the symbols representing, respectively, the symbol read, the symbol printed, and the step direction (L or R). A computation involving j moves will be written as a j-tuple with the i-th move listed in the i-th position. Thus a computation in which a machine reads an A, prints a B, and steps right, and repeats this move three times before halting will be represented as (ABR,ABR,ABR).

We will be studying the following training model: A finite set of computations which can be executed by some Turing machine M_o are given to the trainable system, and this system finds a Turing machine M_1 which will correctly execute all of the given computations. Hopefully, if the trainable system is given enough sample computations, it will find the correct machine so that M_1 is behaviorably equivalent to M_o for all finite computations which M_o can execute. That is, M_1 will exactly mimic M_o in all of its moves in any finite computation starting with any initial tape. If this occurs, we will say that the trainable system has learned the function computed by M_o.

The existence of such a trainable computer is not surprising since it is only necessary for it to begin enumerating the class of all Turing machines until it finds one which can execute the given finite-set of computations. If it yields a machine M_1 which is not equivalent to M_o, we need only give it an additional sample computation for M_o which it cannot execute to cause the enumeration to continue. Since M_o is one of the machines which will be eventually enumerated, we can be sure that we can force the system to eventually enumerate either M_o or some machine equivalent to it (for all finite computations). When it does, the system will have learned the function computed by M_o and additional sample computations from M_o will not cause it to ever yield any other machine. This learning model has been studied by others and this type of argument has been given a number of times, particularly in papers on grammatical inference [4,6,7,8,13,14,22].

From a practical point of view, on the other hand, we might expect this type of learning by enumeration to be useless for two reasons. First of all, in order to learn any function it is necessary to check all of the functions which precede it in the enumeration, and this is likely to involve an astronomical amount of computation even for very modest problems. Secondly, it appears at first glance that a huge number of sample computations may be required before the system will ever enumerate a correct answer. It is the purpose of this paper to deal with both of these objections.

We will exhibit an algorithm which enumerates not Turing machines but parts of Turing machines and which carefully guides its search by intelligently using information from the sample computations. The algorithm finds a machine which can execute the first i moves in the samples and searches for a change which will enable it to execute the first $i+1$ moves. The process is repeated for increasing i with backtracking when necessary. We will demonstrate that very large solution spaces can be searched with only a few seconds or minutes of computer time, and furthermore, that relatively few sample computations are needed before a correct answer is found. For example, in the next section, we search for and find a three state machine with a three symbol alphabet from a space of approximately $6^9 = 10{,}077{,}696$ machines. We find that it only takes one sample computation involving eleven moves to force the search to a correct answer, and the computer finds this answer in just over three seconds.

The research reported here is an outgrowth of studies in grammatical inference where the problem is to infer a grammar from a finite number of samples from its language. Many of the results and ideas presented in Biermann and Feldman [4], Feldman [7], Feldman, et.al. [8], Gold [13], Horning [14], Solomonoff [22] and others are directly applicable to the current problem although their emphasis is on grammar discovery. These papers contain a number of results concerning enumeration methods and techniques for choosing a "best" answer.

One might also look for related research among the papers which have been written on automatic computer program synthesis (Amarel [1,2], Manna and Waldinger [15], Slagle [21], Waldinger and Lee [23]) but most of these deal with a different formulation of the problem: Given a formal description of a task to be performed, how can the formalism be translated into a computer program? This paper is concerned with problems of inference from examples rather than a translation between formalisms.

Most of the previously studied trainable systems have utilized the technique of basing decisions on the values of certain stored parameters and then have exhibited adaptive behavior by varying these parameters. The perceptron [18], many pattern recognition systems [16,19], and many game playing programs [20] are examples of this type of learning system. The system described here uses an entirely different approach to learning, finite-state machine synthesis, and the nature of its performance is consequently dramatically different.

In the next sections, an algorithm for finding a Turing machine capable of executing a given set of computations is given and a number of examples demonstrating its performance are presented. In Section 4, the generality of the approach will be demonstrated by solving a program synthesis problem for a modern computer. In Section 5, the problem of computer program synthesis from input-output information only is discussed.

2. The Algorithm

The algorithm for finding a Turing machine which executes a given set of computations is given in Figures 4, 5, and 6. We will study an example before describing it in detail. Suppose it is desired to find a machine which sorts A's and B's; that is, the machine will begin with its head at the left end of a randomly arranged string of A's and B's and will rearrange the symbols until all of the A's precede all of the B's. Our sample computation will sort the string BAA and will proceed as follows: The head moves right until it finds an A. It replaces the A with a B and then moves left until it finds either the left end of the tape or another A. It moves right one step, puts the newly found A there, and then proceeds off to the right looking for another A. The computation is shown in Figure 2 and is described by the sequence (BBR,ABL,BBL,_R,BAR,BBR,ABL,BBL,AAR,BAR,BBR). A blank symbol on the tape is written as _ .

CURRENT TAPE	NEXT MOVE
ḄAA	BBR
BẠA	ABL
ḄBA	BBL
.BBA	_R
ḄBA	BAR
AḄA	BBR
ABẠ	ABL
AḄB	BBL
ẠBB	AAR
AḄB	BAR
AAḄ	BBR
AAB.	(halt)

Figure 2.
An example computation. The position of the head on the current tape is indicated by a dot.

For the moment, it will be assumed that we know that the desired Turing machine has three states, and the strategy for finding it will be to try to guess which of these states the machine is in after each move in the computation. Beginning in state 1 (see Figure 3), we guess that the machine goes to state 1 after the move BBR. After ABL, we might again guess the device will go to state 1 except that this would yield a contradiction with the next move BBL. (State 1 makes the move BBR instead of BBL.) So we guess the device will be in state 2 after ABL. After similar arguments we decide the device may go to states 1 and 3 after moves BBL and __R. However, attempts to find the state after BAR all yield contradictions causing a revision in the guesses. Perhaps the device goes to state 2 after move BBL. Then states 3 and 1 are the next noncontradictory choices to be made after moves __R and BAR. At this point, the next three choices become fixed as a logical consequence of previous decisions so they are included and are parenthesized to indicate this fact. After AAR, the only noncontradictory choice is 3 and the rest of the table follows immediately. The final machine (Figure 3, bottom) is the correct answer, a Turing machine which sorts A's and B's. Thus the trainable computer can learn to sort on the basis of one sample computation.*

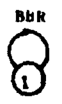
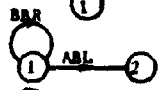
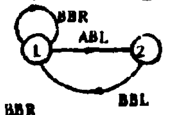
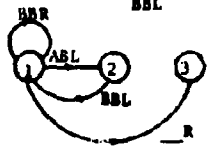
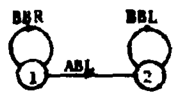
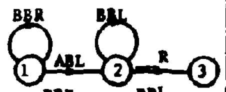
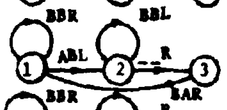
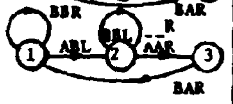
Notice that at each point the guessed state is the lowest number which does not yield a contradiction with the immediate next moves. If a contradiction is found at any time for all possible choices 1, 2, and 3, then the search is backed up to the last arbitrary choice, it is

*One can show that if this sort of procedure is executed on any string which begins with B and has at least two A's in it, then the resulting computation is satisfactory for training the machine to sort.

Figure 7

The search for a Turing Machine.

Moves with guessed states listed below them.

Current version of Turing Machine		BBR	ABL	BBL	__R	BAR	BBR	ABL	BBL	AAR	BAR	BBR
	(1)											
	(1)	1										
	(1)	1	[illegible]									
	(1)	1		1								
	(1)	1	2	1	3							
	(1)	1	2	2								
	(1)	1	2	2	3							
	(1)	1	2	2	3	1	(1)	(2)	(2)			
	(1)	1	2	2	3	1	(1)	(2)	(2)	3	(1)	(1)

incremented by one, and the search proceeds. In this way the space of all possible three state machines is searched until the correct answer is found. If no three state machine can perform the computation, then the back up will eventually reach the first move indicating that the class of four state machines should be examined.

The notation of the algorithm must be defined. INPUT is an array which holds sequentially each of the moves in each sample computation. The last symbol read before a computation halt appears with an exclamation point to indicate the end of the computation. Thus in the example above, the entries BBR,ABL,BBL,----,BBR would appear in positions 1 through 11 and _! would appear in position 12. Other sample computations would have been entered in locations 13 and beyond.

The array STATE holds the guessed sequence of states with the nonarbitrary choices enclosed in parentheses. The array TRAN holds a complete description of the momentarily guessed Turing machine and is updated continuously as changes are made in STATE. Its exact form need not be considered.

FUT(I,LEVEL) is a function which yields the list of states which the current machine in TRAN will go through beginning in state I if it makes the moves INPUT(LEVEL),INPUT(LEVEL+1),-----. Often FUT will yield an empty list because TRAN will not have transitions corresponding to the given sequence of moves. In the example above, FUT (1,6)= (1,2,2) after move BBR and FUT(3,10) = (1,1) after move BAR. It may be that TRAN is in contradiction with the given sequence of moves either because it indicates the wrong print or step right or left instruction or because INPUT indicates a computation termination (exclamation point)

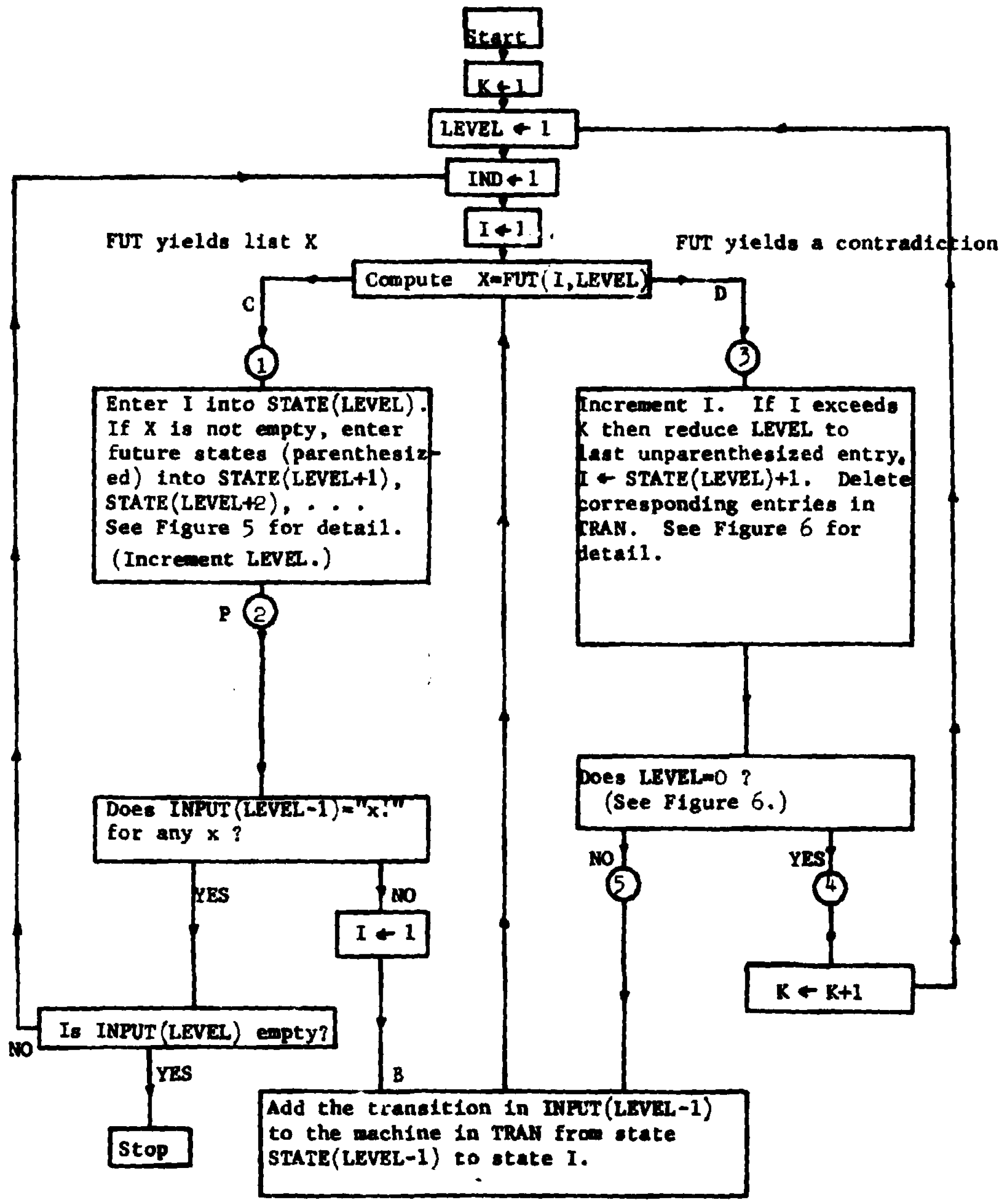

Start
K ← 1
LEVEL ← 1
IND ← 1
I ← 1
FUT yields list X
FUT yields a contradiction
Compute X=FUT(I,LEVEL)
C
D
1
3
Enter I into STATE(LEVEL). If X is not empty, enter future states (parenthesized) into STATE(LEVEL+1), STATE(LEVEL+2), . . . See Figure 5 for detail. (Increment LEVEL.)
Increment I. If I exceeds K then reduce LEVEL to last unparenthesized entry, I ← STATE(LEVEL)+1. Delete corresponding entries in TRAN. See Figure 6 for detail.
P
2
Does LEVEL=0 ? (See Figure 6.)
Does INPUT(LEVEL-1)="x:" for any x ?
YES
NO
I ← 1
NO
YES
5
4
K ← K+1
Is INPUT(LEVEL) empty?
NO
YES
Stop
B
Add the transition in INPUT(LEVEL-1) to the machine in TRAN from state STATE(LEVEL-1) to state I.

Figure 4. The algorithm

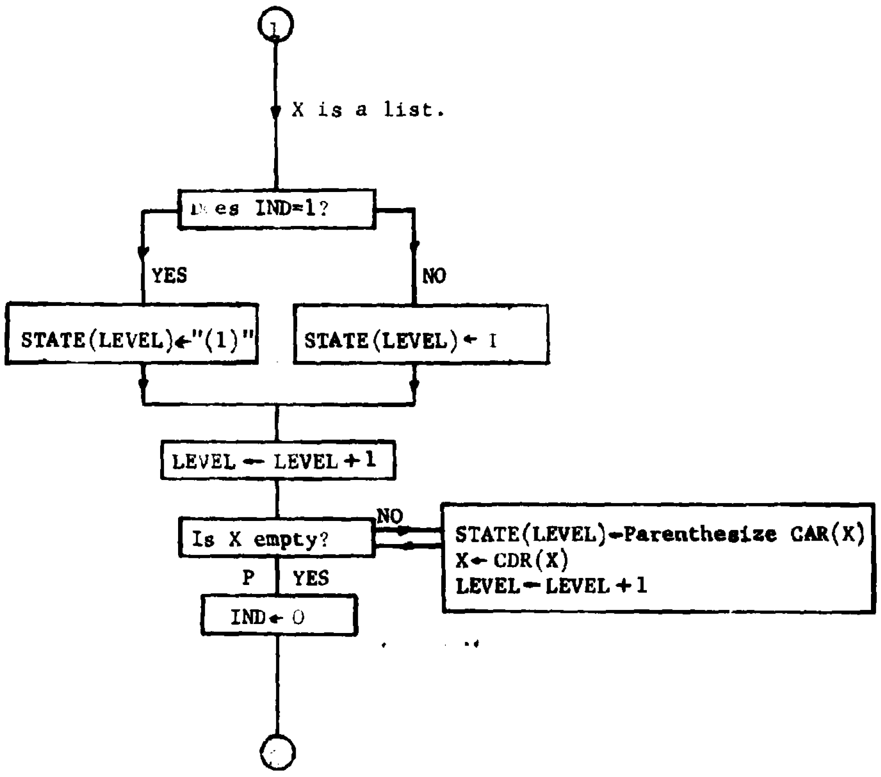


Figure 5. Enter newly proposed next state into STATE. (If X is a list, CAR(X) is the first element of the list and CDR(X) is a copy of X with the first element deleted.)

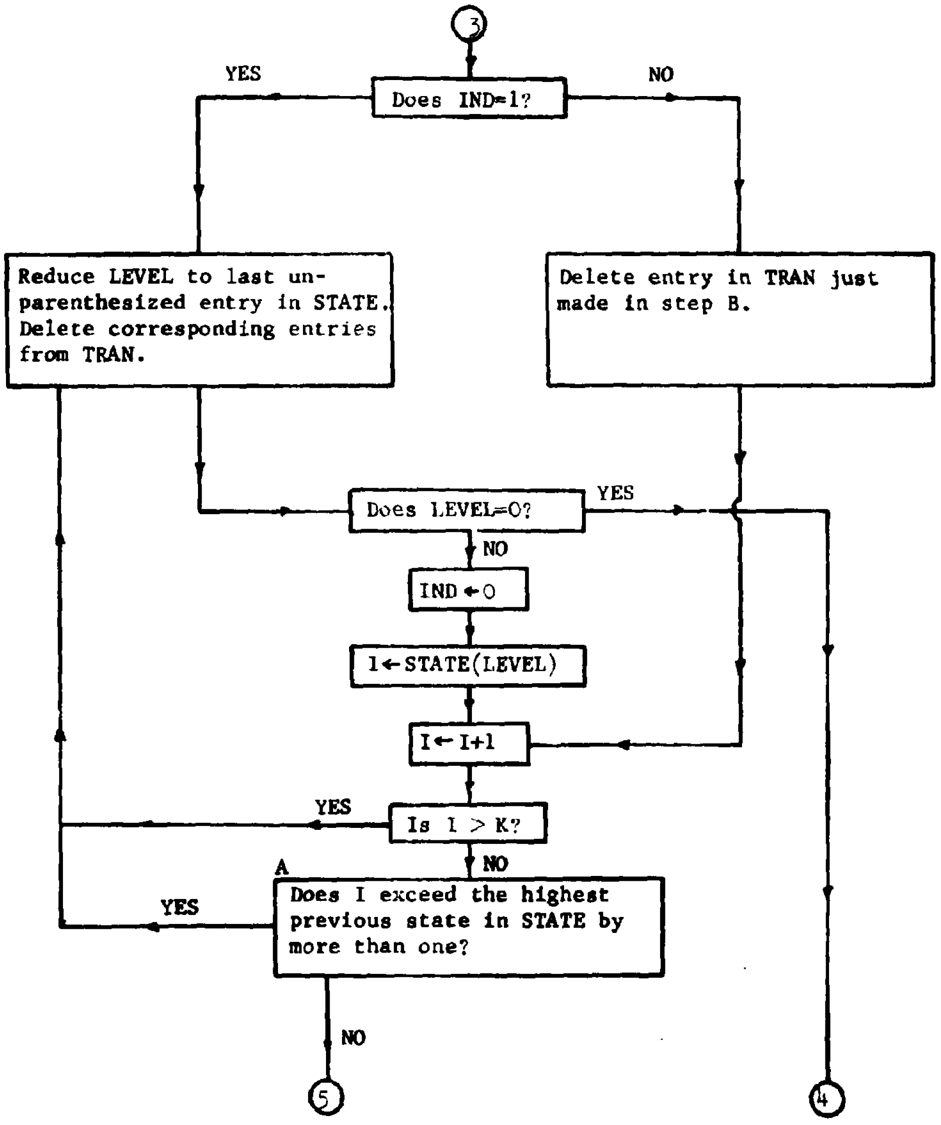

3
YES
Does IND=1?
NO
Reduce LEVEL to last unparenthesized entry in STATE. Delete corresponding entries from TRAN.
Delete entry in TRAN just made in step B.
Does LEVEL=0?
YES
NO
IND ← 0
I ← STATE(LEVEL)
I ← I+1
YES
Is I > K?
NO
A
YES
Does I exceed the highest previous state in STATE by more than one?
NO
5
4

Figure 6. I increment and backtrack logic.

and TRAN does not. In either case, FUT yields a special symbol meaning "contradiction".

K is the currently hypothesized number of states, is initially set at 1, and is incremented until an acceptable machine is found. IND is an indicator which is set at 1 if the currently considered move is the first in a computation. IND is 0 otherwise. LEVEL is the index of arrays INPUT and STATE telling which entry is currently being considered, and I is the proposed new state name to be entered into STATE(LEVEL).

Figures 4, 5, and 6 in conjunction with these definitions completely describe the algorithm for the trainable computer. If the contents of STATE are printed out each time the algorithm passes point P, the entries in Figure 3 result for the example. Notice that the search can be greatly reduced at point A in Figure 6 by requiring that each previously unused state name I exceed the highest previously used state name by exactly one. For example, if the first two entries in STATE are ((1),2) and the search for a machine has failed, there is no need to try ((1),3) since 3 is simply a new name for the state 2.

Another important way to increase efficiency which is not shown in Figure 4 is to include a test at point C which works as follows: Compute FUT(1,J) for each J>LEVEL such that INPUT(J) is the beginning of a computation. If in any case FUT(1,J) yields a contradiction, go to D. This helps to prevent the algorithm from making hypotheses on the basis of one computation which will be found to be wrong later when other computations are examined. This feature was included in the program which is discussed in the next Section.

The algorithm thus exhaustively searches the space of K state machines for K = 1,2,3,---- until a machine is found which can execute the given sample computations. If the algorithm yields a machine which is only capable of doing the sample computations correctly but is not really the "right answer", it can be given additional sample computations causing it to resume its search. Since a correct Turing machine exists somewhere in the enumeration, it will be eventually found if enough such additional computations are included.

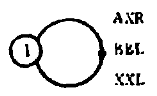
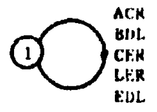
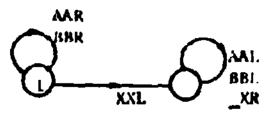
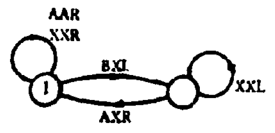
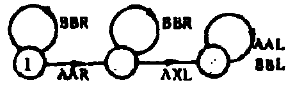
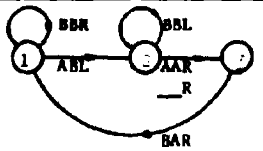
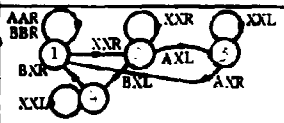
The efficiency of the algorithm can probably be improved by processing the sample computations in parallel. The method would be to examine all of the computations which have been assumed to be in some particular state and to look for the next transition from that state using the information from all of the samples simultaneously. This method has the advantage that it would not be dependent on the order in which the samples are presented and it would probably find cutoffs at an earlier time in the search.

3. Some Experiments with the Algorithm

The algorithm described above was programmed in the Stanford LISP 1.6 language, compiled and used to find Turing machines which solve various problems. The results are summarized in Figure 7 where each problem is described and its solution given. The computations in column four are represented by their initial tapes. Thus the string BAA in problem 6 represents the complete computation described in the previous section. The amount of PDP-10 CPU time required to do the search in each case is given in the last column. These times do not necessarily represent the best possible performance since no unusual efforts were made to write optimal code and LISP does not typically yield fast executions. Another thing that should be mentioned is that repetitions of the same computation did not necessarily yield the same computation time because the number of internal garbage collections would vary from one test to the next. So these times should be considered to be only a kind of rough estimate of the amount of effort required to obtain a solution.

The first set of computations in each problem was obtained as follows. The first i initial tapes from the set of allowed tapes (see column three) were used to generate i sample computations. These i computations were input to the algorithm and a solution was produced. This process was completed for i = 1,2,3,--- until a correct answer was found. The first set of computations given for each problem is thus minimal in the sense that if the last computation were deleted, the set would no longer be adequate for inferring the correct answer.

The answers to problems one through six could be inferred from just one sample computation and the shortest such computation was found

PROBLEM DESCRIPTION	TURING MACHINE WHICH CORRESPONDS TO THE SOLUTION	SET OF ALLOWED INPUT TAPES **	SET OF COMPUTATIONS REQUIRED TO DIS-COVER TURING MACHINE	CPU TIME REQUIRED (SECONDS)
1. Mark X's out to the first B. Return to the beginning of the tape.		$\{A,B\}^+$	{A, B, AA, AB} {AB}	0.45 0.65
2. Change A's to C's. For each B left on the tape, cross off one C.		$A^*B^*-\{\Lambda\}$	{A, B, AA, AB, BB, AAA, AAB, ABB} {ABB}	2.15 0.73
3. Search for an X. If X is found, type an X at the beginning of the tape.		$\{A,B,X\}^+$	{A, B, X, AA, AB, AX, BA, BB, BX XA, XB, XX, AAA, AAB, AAX, ABA, ABB, ABX} {ABX}	2.17 1.55
4. For each B, mark off one A.		$A^*B^*-\{\Lambda\}$	{A, B, AA, AB, BB, AAA, AAB, ABB} {ABB}	1.28 1.52
5. Print an X on the second A and return to the beginning of the tape.		$\{A,B\}^+$	{A, B, AA, AB, BA, BB, AAA, AAB, ABA, ABB, BAA} {BABA}	7.06 3.23
6. Sort A's and B's.		$\{A,B\}^+$	{A, B, AA, AB, BA, BB, AAA, AAB, ABA} {BAA}	5.83 3.31
7. For input tape uXv, check whether u equals the reverse of v.	 AAR BBR XXR XXR XXL AXL BXR BXL AXR XXL	$\{A,B\}^+ X\{A,B\}^+$	{AXA, AXB, BXA, BXB, AAXA, AAXB, ABXA, ABXB, BAXA, BAXB, BBXA, BBXB, AXAA} {ABXBA, AXB, BAXAB}	13.13 22.58

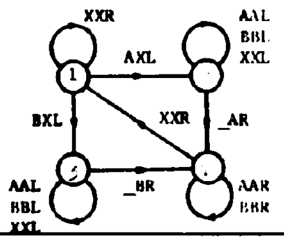
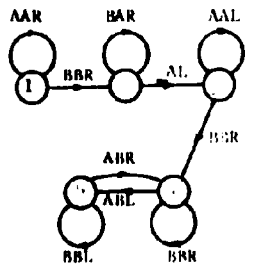
PROBLEM DESCRIPTION	TURING MACHINE WHICH CORRESPONDS TO THE SOLUTION	SET OF ALLOWED INPUT TAPES	SET OF COMPUTATIONS REQUIRED TO DIS-COVER TURING MACHINE	CPU TIME REQUIRED (SECONDS)
8. **Reverse the input string.**	 XXR AAL BBL XXL AXL 1 BXL XXR _AR AAL BBL XXL _BR AAR BBR	$\{A,B\}^+$	{A,B,AA,AB,BA,BB,AAA,AAB,ABA,ABB,BAA,BAB}	11.[illegible]
			{ABA,ABB,BAA,BAB,AABB}	544.[illegible] *
			{AAB,ABA,ABB,BAA,BAB,AABB}	[illegible]1[illegible].05
			{AAA,AAB,ABA,ABB,BAA,BAB,AABB}	[illegible]17.[illegible]
9. **Using an alphabet of only two symbols, for each B mark off one A.**	 AAR BAR AAL 1 BBR AL BBR ABR ABL BBL BBR	$A^*B^*-\{\Lambda\}$	{A,B,AA,AB,BB,AAA,AAB,ABB,BBB,AAAA,AAAB}	[illegible]1.[illegible]*
			{AABB,AAABBB}	[illegible].[illegible]*

Figure 7. Experiments on the trainable computer.

* The solution found was correct although not identical to the machine given in column two.

** S^+ is defined to be the set of all strings of symbols from set S including the string of length zero (denoted Λ). $S^+ = S^* - \{\Lambda\}$

in each case. These are included in the table along with their computation times. The algorithm usually found the answer in less time then in the first experiment. Surprisingly, in several cases of the first experiment, the first $i-1$ of the i sample computations could be deleted without affecting the ability of the system to find a correct answer.

In the other problems, the second set of sample computations is simply representative and not necessarily minimal in any sense. Occasionally the algorithm produced an answer which was different from the one given but which was still correct. These instances are so marked. The amount of search time required to find a solution is not an easily predicted quantity as indicated in problem eight. Adding a sample computation to a set of computations which is already adequate for inferring a correct answer can increase the total search time because each newly proposed transition must be checked for compatibility with this computation as well as the others. This addition can also decrease the search time by enabling the algorithm to discover that it has made a wrong decision at an earlier time.

These problems were not chosen using any particular criterion and are representative of all of the experience gained with this algorithm. One can expect similar performance on any problem which involves about four states or less in the control as long as the total number of transitions is not great. Some searches for four state and larger machines were terminated after about ten minutes of CPU time without an answer. Machines with a large number of states can be found in a reasonable amount of time if the number of transitions is sufficiently small. For example, the machine which starts with a blank tape and types out sequentially the twenty-six letters of the alphabet has twenty-seven states and was found in 104 seconds. The total search time is a function of number of states, size of alphabet, number of transitions, the order of the sample computations, and the order of the transitions within the computations.

When training the system to do a computation, it is necessary to have a systematic algorithm in mind. There are an infinite number of ways to get from any initial tape to any final tape, and a method must be chosen which results in a finite-state control. Clearly, it is easy to find a Turing machine which when given the number 11 yields the number 13. However, it is not so easy to find a machine which when given any prime number will find the next prime number. If the sample computations involve a naive scheme for getting from the initial tape to the final tape, the resulting machine may never have the desired capability although it will always be able to reproduce the sample computations.

From a practical point of view, it is quite helpful to choose a method for doing the desired computation which the system can easily learn. This usually involves finding a scheme which requires a small number of states. Notice that problems two, four, and nine involve essentially the same computation but alphabets of different size were used. The machines tended to be more difficult to find if they had more states even if their alphabets were significantly smaller.

4. On the Design of an Autoprogrammer

The algorithm described in this paper is designed to find a finite-state control from sample input-output sequences and can be used to find a controller or program for any computer. In order to illustrate the general applicability of this technique, we will consider the problem of writing a program which factors any natural number into its prime factors, and we will use a modern computer with registers and arithmetic operations.

It is first necessary to find a sample computation and in this example, we will factor 12 into its prime factors 2,2,3. We will store the number to be factored in register R1, the number to be divided into R1 in R2, and the remainder and quotient for the division in registers R3 and R4, respectively. The method will be to divide R1 by R2 and then either print or increment R2 depending on whether the remainder is zero or not. If a prime factor is found, the new quotient is entered into R1 and the process is continued. The sample computation is traced in Figure 8. Certain steps are taken only if some particular condition holds, and in such cases, that condition is indicated.

Condition	Command	R1	R2	R3	R4
	R1 ← read	12	0	0	0
	R2 ← 2	12	2	0	0
	R4 ← R1÷R2, R3 ← remainder	12	2	0	6
R3 = 0	R1 ← R4	6	2	0	6
	print R2	6	②	0	6
	R4 ← R1÷R2, R_3 ← remainder	6	2	0	3
R3 = 0	R1 ← R4	3	2	0	3
	print R2	3	②	0	3
	R4 ← R1÷R2, R3 ← remainder	3	2	1	1
	R2 ← R2 + 1	3	3	1	1
	R4 ← R1÷R2, R3 ← remainder	3	3	0	1
R3 = 0	R1 ← R4	1	3	0	1
	print R2	1	③	0	1
R1 = 1	halt	1	3	0	1

Figure 8

A sample computation: Factoring 12 into primes.

Since the only changes in the flow of the program result from conditional tests, the inputs to the finite-state control are the results of these tests. If no condition is listed, we will let S be the standard input symbol. The finite-state control which solves the problem thus will yield a series of commands like those in the figure altering the command sequence appropriately when the conditional tests so indicate.

Since the algorithm of Section 2 is designed to find Turing machines, it requires in each move description a step left or a step right command. This is not applicable to the current problem and will simply always be listed as R.

Now if we let each of the conditions (e.g., $R3 = 0$) and each of the commands (e.g., $R1 \leftarrow$ read) be an abstract symbol, we can submit the sample computation directly to the algorithm without change:

$[(S, R1 \leftarrow \text{read}, R), (S, R2 \leftarrow 2, R),$ ------

------, $(R1 = 1, \text{halt}, R)]$.

In this case, the LISP program computed for about eight seconds and produced the finite-state controller of Figure 9.

It is a small change to make the finite-state controller into a computer program, and the flow diagram for this program appears in Figure 10. If input symbols appear at a node which are not the standard S, a conditional test must be inserted to implement the branching. The resulting computer program will correctly extract the prime factors from any positive integer. Since there is nothing special about either the computer or the example problem, this method is clearly quite general.

Summarizing, we can find a computer program for executing some algorithm from example computations which employ that algorithm. We simply list sequentially, for each example, the commands executed by the algorithm including with each command any conditions which must be checked before its execution. These sequences with the modifications described above are then submitted to the algorithm of Section 2, and the resulting finite-state controller can be directly converted into a flow diagram for the computer program. The only alteration required is the inclusion of conditional tests to check for conditions listed as input symbols on the finite-state controller. The resulting computer program

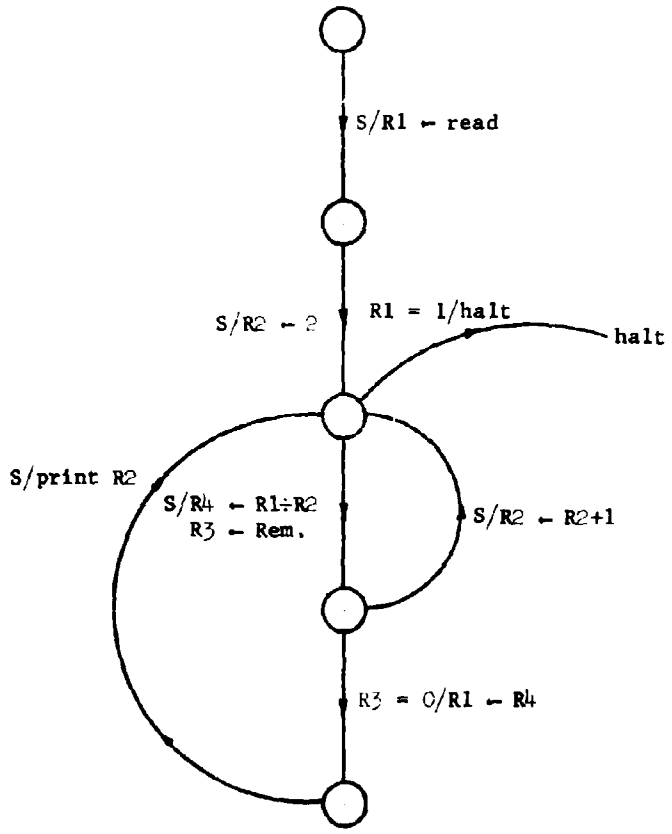


Figure 9

Finite-state controller for finding prime factors.

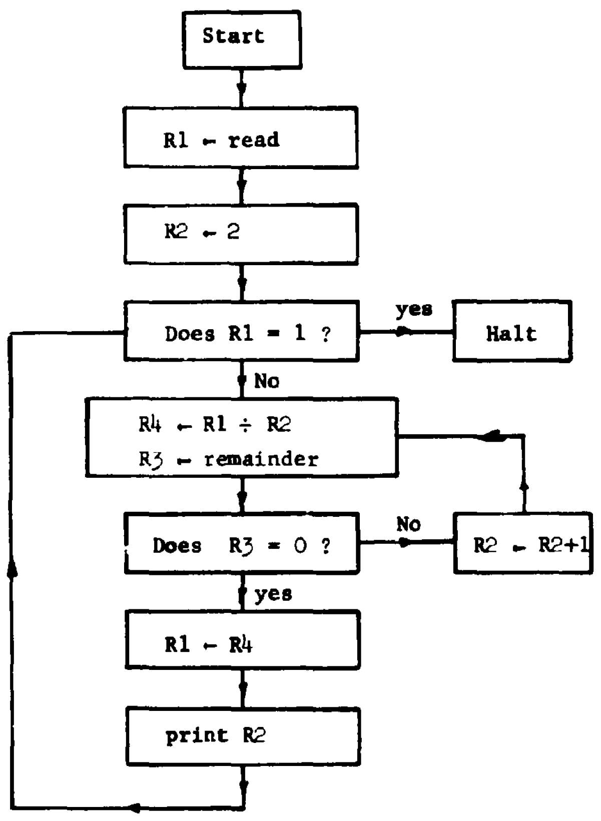


Figure 10

Flow diagram for a computer program.

will correctly execute all of its given example computations and will correctly execute the desired algorithm if it has been given enough examples. We will call a system which carries out the above process an autoprogrammer.

It is important to note that the autoprogrammer as we define it here does not actually create an algorithm for solving a problem where none existed before. The algorithm must be implicitly contained in the sample computations, and the method described here simply finds and makes explicit that algorithm by constructing a satisfactory flow diagram. It may be true that the autoprogrammer concept will not prove to be a useful aid to traditional computer programmers because one must essentially write the program as he is doing the examples. However, there are situations such as in a desk calculator where an autoprogrammer might be quite useful. In this case, the operation codes correspond to keys on the machine and the user may not be a computer programmer. With an autoprogrammer built into the device, the user could begin doing his repetitive task in the usual way but at some point he could stop and allow the keys to go on pushing themselves.

5. Computer Program Synthesis from Input-Output Pairs

Perhaps the next logical step after the autoprogrammer is the development of a system which synthesizes computer programs from input-output information only. The system would have no information concerning how each output is obtained from its corresponding input and would be faced with the problem of filling in all of the steps as well as finding the program. The system could enumerate the set of all possible sets of intermediate steps, find a program corresponding to each set of computations, and use some criteria for choosing one of the programs as its answer. This all appears to be well beyond the range of possibility for any general class of functions.

One approach to this problem was seriously investigated. Suppose we are interested in automatically synthesizing computer programs which use only finite memory and which yield as outputs, n-place binary numbers where n is fixed for any particular program. Then it is only necessary to find n finite-state automatons each of which corresponds to one of the n binary places. Here we consider an automaton to be a device which scans an input string once from left to right and returns an output of 0 or 1. We synthesize for the i^{th} place an automaton which is capable of scanning each of the sample inputs and returning in each case the i^{th} place of the corresponding output. Finite-state automata theory is well understood, and it is possible to obtain convergence to the correct machine if enough input-output pairs are known. The program synthesizer constructs the n automatons and produces a program which simulates them.

Such a system was developed and was very efficient as a program synthesizer because finite-state machine synthesis is very direct and fast. Every program produced was completed in a few seconds of CPU time and was executable on the PDP-10 system. However the requirements for input-output data were very great because of the number of strings required for synthesis of the individual automatons was large. For an m-state automaton, most of the strings of length $2m-2$ were needed before the machine could be synthesized and this amounts to $\sum_{i=0}^{2m-2} r^i$ strings for an r symbol input alphabet. As an example, the program which counts the number of A's modulo 4 (thus $n=2$) in strings of A's and B's required nearly all of the 127 possible input strings of length six or less before it could be produced. Recent developments in grammatical inference using man-machine interactive capabilities [4] have shown how to reduce this input requirement, but the fundamental difficulty is likely to remain.

Another great disadvantage of this system was its finite-state memory limitation. Although every program we write uses a finite memory, it is usually written as if the memory were infinite. We would never be able to claim that we had, say, written a program which enumerates the prime numbers if we did not make this assumption.

It may be that in terms of input requirements, the best program synthesizer will be some compromise between this approach and the auto-programmer which requires every intermediate step in each sample computation. Perhaps a method can be developed for giving hints as to how to do the computation without having to include the actual steps.

6. Conclusion

In this paper, we think of a trainable machine as a manipulative system with a finite-state controller, and the learning process for the machine involves finding the correct controller. The approach is quite general as demonstrated by the fact that it has been applied to very different types of problems. It is also important to note that the controller can be found either by a traditional finite-state machine synthesis method or by some kind of search. The philosophy of the paper may be applied in many ways, and the specific systems discussed here should be thought of as examples of a general approach.

7. Acknowledgement

The author is greatly indebted to Professor J. A. Feldman for many invaluable discussions during the period of this research.

BIBLIOGRAPHY

[1] S. Amarel, "On the Automatic Formation of a Computer Program which Represents a Theory", in Self Organizing Systems - 1962, editors: Yovits, Jacobi, and Goldstein, Spartan Books, New York, 1962.

[2] S. Amarel, "Representations and Modelling in Problems of Program Formation", in Machine Intelligence 6, editors: Meltzer and Michie, American Elsevier Publishing Company, Inc., New York, 1971.

[3] A.W. Biermann and J.A. Feldman, "On the Synthesis of Finite-State Acceptors", A.I. Memo Nol 114, Computer Science Department, Stanford University, April 1970.

[4] 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, January 18-20, 1971.

[5] M. Davis, Computability and Unsolvability, McGraw-Hill Book Company, Inc., New York, 1958.

[6] J.A. Feldman, "First Thoughts on Grammatical Inference", A.I. Memo No. 55, Computer Science Department, Stanford University, August 1967.

[7] J.A. Feldman, "Some Decidability Results on Grammatical Inference and Complexity", A.I. Memo No. 93.1, Computer Science Department, Stanford University, May 1970.

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

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

[10] 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.

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

[12] S. Ginsburg, An Introduction to Mathematical Machine Theory, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1962.

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

[14] J.J. Horning, "A Study of Grammatical Inference", Technical Report No. CS 139, Computer Science Department, Stanford University, August 1969.

BIBLIOGRAPHY
(Continued)

[15] Z. Manna and R.J. Waldinger, "Toward Automatic Program Synthesis", Communications of the Association for Computing Machinery, Vol. 14, No. 3, pp. 151-165, 1971.

[16] J.M. Mendel and K.S. Fu, Adaptive, Learning, and Pattern Recognition Systems, Academic Press, New York, 1970.

[17] M.L. Minsky, Computation: Finite and Infinite Machines, Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[18] M.L. Minsky and S. Papert, Perceptrons: An Introduction to Computational Geometry, MIT Press, Cambridge, Massachusetts, 1969.

[19] N.J. Nilsson, Learning Machines, McGraw-Hill Book Company, Inc., New York, 1965.

[20] A.L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers, II - Recent Progress", IBM Journal of Research and Development, Vol. 11, No. 6, pp. 601-617, 1967.

[21] J. Slagle, "Experiments with a Deductive Question-Answering Program", Communication of the Association for Computing Machinery, Volume 8, No. 12, pp. 792-798, 1965.

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

[23] R.J. Waldinger and R.C.T. Lee, "PROW: A Step Toward Automatic Program Writing", Proceedings of the International Joint Conference on Artificial Intelligence, Washington, D.C. 1969.