4-1984

An Improvement in the Performance of a Learning System for Finite State Machines

Lea F. Fuller

AN IMPROVEMENT IN THE PERFORMANCE OF A LEARNING SYSTEM FOR FINITE STATE MACHINES

by

Lea F. Fuller

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
April 1984

AN IMPROVEMENT IN THE PERFORMANCE OF A LEARNING SYSTEM FOR FINITE STATE MACHINES

Lea F. Fuller, M.S.

Western Michigan University, 1984

A learning system is examined which is capable of learning a finite state machine from a class of finite state machines based on the observed behavior of the machine. The size of the search space becomes very large as the number of states in the machine increases. The size of the search space quickly becomes the limiting factor in the size of the class of machines which may be learned. An investigation is made of methods to improve the performance of the learning system. The application of a depth first approach to the development of the search space is shown to provide a significant reduction in the amount of space required to identify a machine. An heuristic estimator is used to guide the depth first search and is shown to have a potential for reducing the time required to identify the machine.

ACKNOWLEDGMENTS

When undertaking an endeavor such as this, there are invariably people whose assistance is worthy of special note. My gratitude goes first to Dr. Dionysios Kountanis for setting me on the path of this research and whose guidance and infinite patience kept me there. My appreciation is also extended to Clark Equipment Company for the generous use of their equipment in the implementation and documentation of this research. My thanks go also to numerous people on the Clark staff for assistance in the production of this document. And finally, my special thanks go to James W. Herrick. Without his support I would have never survived the ordeal.

Lea F. Fuller

FULLER, LEA FERN

AN IMPROVEMENT IN THE PERFORMANCE OF A LEARNING
SYSTEM FOR FINITE STATE MACHINES

WESTERN MICHIGAN UNIVERSITY M.S. 1984

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

CHAPTER I

INTRODUCTION

A current area of research in artificial intelligence is the study of learning, both how humans learn and how computers can be endowed with the ability to learn. The use of computers to study the learning process itself represents an attempt to understand the way in which humans learn. The learning ability of a computer system may be considered from several different perspectives. One is to declare that a system has the ability to learn if it can in some way improve its performance. This may be done by applying a new method for solving a problem or improving an existing one. A second approach is to amass large amounts of data on a particular subject and then apply this data in the form of an expert system. By manipulating rules defined on the collected data, such a system has the ability to reach an intelligent conclusion when posed a question pertaining to the subject about which it is knowledgeable. Yet another approach to learning is the area of inductive inference. This involves learning general properties of a subject based on examples.

Research Goals

In this work, attention is focused on the area of inductive inference. The particular application under consideration is that of deducing the structure of a finite state machine based on its observed behavior. The machine may be considered as being contained within a "black box" which prohibits the observer from knowing the structure of the machine. The inputs to the machine and the outputs from the machine are available for study and provide the primary means of the observer for deducing the structure of the machine. Restrictions will be placed on the machine which may be contained in the black box to reduce the problem of identification to a solvable one. The goal of this research is to study the performance of a particular learning system and to devise a method for improving its performance.

Chapter Review

Chapter II provides a review of selected literature pertaining to the concept of learning structure from behavior. Chapter III is a review of finite automata theory and provides some basic definitions used throughout this work. In chapter IV, the particular learning system under consideration is reviewed and the groundwork is laid for a discussion of performance improvement. Chapter V reverts to a more global level for a discussion of different search methodologies and a review of selected works in this area. A method for improving the performance of the learning system being examined using an informed depth first search guided by an heuristic is developed in chapter VI. In chapter VII, an implementation of the learning system with and without the informed depth first search method is discussed and the results are presented. Chapter VIII provides a summary of this work and areas of further research are proposed.

CHAPTER II

REVIEW OF SELECTED LITERATURE

Gold (1967) investigates the problem of learning a formal language by examining samples of the language. A language is defined to be a set of strings on some finite alphabet. A class of languages is defined in which all members of the class are defined over the same alphabet. Next a method for presenting information to a learner about an unknown member of the class of languages is defined. At quantized time intervals, a unit of information about the language is presented to the learner and he/she is asked to make a guess as to the identity of the language. The class of languages will be considered learnable (with respect to the method of presentation of information) if there exists an algorithm that can be applied by the learner which after some finite time allows all guesses to be the same and correct. Two methods of presentation are considered. In the first, all strings of the language are enumerated and presented to the learner in any order. In the second, there exists an informant who chooses the order that the strings are presented to the learner and who is able to provide data as to whether or not a string is in the language. Gold determines that the class of context sensitive languages is learnable through the use of an informant but not even the class of regular languages may be learned if only the method of enumerating strings from the language is used.

Biermann, Baum and Petry (1975) study the problem of synthesizing a program from a trace of its behavior. A computer display system is devised which allows the user to view any data structures which have been defined and the commands available for performing a calculation. The user manipulates commands and their operands through the use of a light pen on a screen just as if the calculations were being made on a scratch pad. The display is constantly updated to allow the user visibility of the results of calculations. While a sample calculation is performed by the user, the computer records the sequence of steps taken as well as any information which the user supplies as a determining factor for some decision made, i.e. A is less than B. After one or more examples have been recorded, a program is created which performs the same calculations. If additional examples are then provided which contradict or augment the program already defined, the program is refined to accomodate the new data. As the development of the program requires a search of the possible alternatives which describe the observed calculations, the synthesis of non trivial programs could command a significant amount of time. Pruning techniques are applied to reduce the size of the search space. A mechanism termed "failure memory" is implemented which allows the program synthesizer to remember previous steps in the synthesis which led to failure and the conditions which were in effect at the time. If the same conditions are detected at some other stage of the search, the path under consideration is dismissed as a failure and is not investigated further. The failure memory is "initialized" before the synthesis begins by examining the set of sample computations and identifying conditions which will lead to a contradiction. The failure memory is also updated on a dynamic basis as the synthesis proceeds to include any failure conditions incurred so far. With the use of a pruning technique, the autoprogramming system has proved to be a practical tool.

Biermann and Krishnaswamy (1976) provide more information on the work done by Biermann, Baum and Petry (1975) with regards to the autoprogramming system. The computational environment of the user is described and the program synthesis technique is shown to be both sound and complete. The synthesized program is guaranteed to correctly execute the examples provided (soundness) and it is shown that every possible program (or its equivalent) can be created by the autoprogramming system (completeness).

Manna and Waldinger (1975) investigate the roles played by a priori knowledge and reasoning in program synthesis. Their work proposes that a great deal of knowledge about the world in general and the specific problem to be solved is required by the program synthesizer. The ability of the synthesizer to reason is central to the process of constructing program. During the program construction formation of program branches, loops and the handling of statements with side effects are given special attention. Manna and Waldinger show that if there exists a program which can be modified to meet the newly specified needs, the strength of the program synthesis method is greatly enhanced. The tactics employed by the program synthesizer and the reasoning approach both build upon the available knowledge stored by the synthesizer in the construction of a program.

The work done by Moore (1956) represents the foundation for determining information about the structure of a finite state machine from its behavior. Moore develops the concept of an experiment as follows: The machine under investigation is said to reside within a black box, thus the experimenter is prevented from directly determining the answer to any questions regarding the structure of the machine. This may be considered a realistic restriction on the learning process. For example, if the object whose structure is being investigated is a bomb which may explode if tampered with, any knowledge of the internals of the bomb must be obtained though external observations. After all, once the bomb has exploded, nothing more can be learned about it. The' experimenter is allowed to apply values from the input alphabet of the machine to the black box and observe the resulting output. From the observed behavior of the machine, the experimenter is able to deduce information about the structure of the machine or about some particular state of the machine.

Moore defines two types of experiments which may be performed. An experiment is said to be "simple" if only one copy of the machine is available. (This also presumes that the machine may not be reset to its initial state). The experimenter chooses a sequence of input symbols to be applied to the machine. This sequence may be completely determined before the beginning of the experiment or it may be created as the experiment progresses, allowing the input values to be chosen as a result of some previously received output values. Figure 1 gives a pictorial representation of a simple experiment.

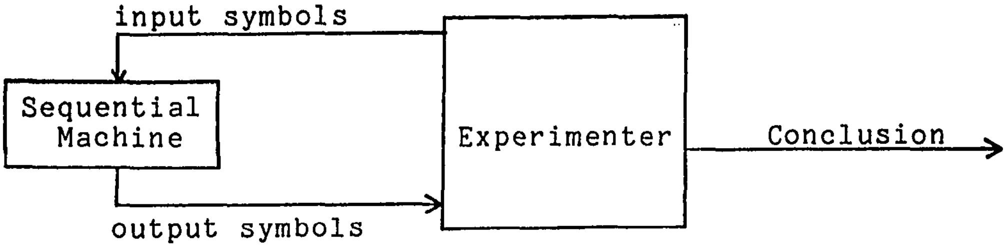

Figure 1. Moore's Simple Experiment

An experiment is called a "multiple" experiment if more than one copy of the machine is available for testing. (All copies of the machine are assumed to be equivalent and initialized to the same state at the beginning of the experiment). The experimenter then sends copies of the input sequence to each of the machines and receives the corresponding output. Figure 2 gives a pictorial representation of a multiple experiment with k copies of a machine.

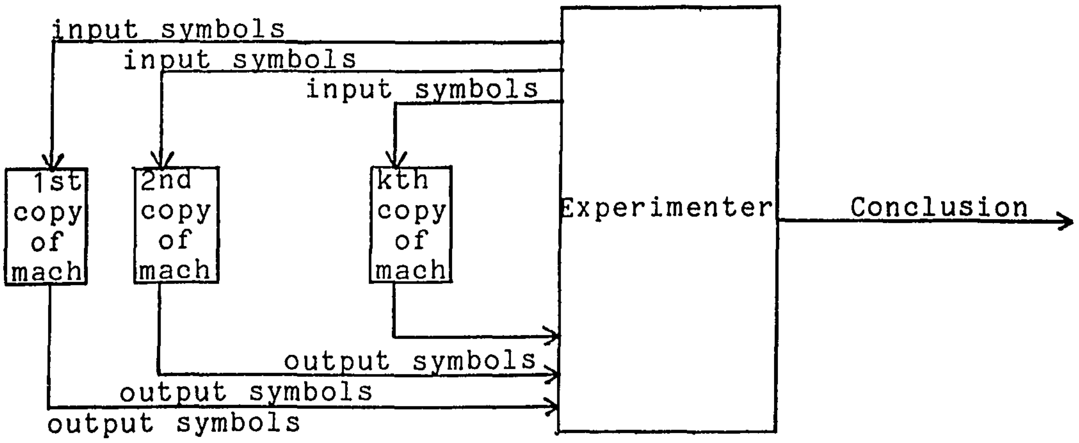

Figure 2. Moore's Multiple Experiment

Moore defines several restrictions on the structure of the machines to be studied. Each machine is restricted to have a finite number of states, a finite number of input values and a finite number of output values. The output of a machine at each step of the experiment is determined solely from the current state of the machine and is not dependent on the previous input value. The behavior of the machine is strictly deterministic.

Moore investigates the concept of distinguishability in depth. He proves, in Theorem 2, that it is not possible to uniquely identify a machine in a black box if no bound is imposed on the number of states the machine may have. The number of input and output values must also be restricted if the machine is to be identified. Several theorems are proven with respect to:

1) The distinguishability of the states within a machine.

2) The distinguishability of two machines from the same class of machines.

3) The ability of the experimenter to identify the state the machine is in at the end of the experiment.

Moore also investigates the length of an experiment required to determine information regarding the internal structure of the machine or its current state.

Gill's (1961) investigations based on the work of Moore and others further clarifies the problem of state identification. He classifies experiments into categories based on how the input sequence is constructed, the number of copies of the machine available for testing and the ability of the experiment to identify a minimal solution.

An experiment is said to be "preset" if the input sequence is completely designed in advance and is valid regardless of the initial state of the machine. It is said to be "adaptive" if the structure of the input sequence is dependent on the previously received output sequence. A simple experiment is one in which only a single copy of the machine is involved whereas a multiple experiment utilizes more than one copy of the machine. An experiment is called "optimal" if it is the shortest one which will yield the desired solution and "regular" if it does not guarantee a minimal solution.

Gill proposes a structure called a "successor tree" to be used during the process of identifying the initial state of a machine. Each level of the tree represents a refinement in the solution and is created as a result of a new value in the observed input/output sequence of the machine. The root of the tree contains a node representing the set of all possible states of the machine and each successive entry is revised to contain a subset of all of the states based on the partitioning effect of the input/output values.

The two major contributions of Gill's work are in refinements to the distinguishing problem; trying to determine which state the machine was in at the beginning of the experiment, and the homing problem; determining the sequence which will deliver the machine to its home state.

Trakhtenbrot and Barzdin (1973) provide an extensive development on the analysis of the behavior of finite state automata and their synthesis. The behavior of different types of automata, both outputless and automata with output, are studied. Treatment of the input sequences supplied to the machines (and output sequences received from the machines) is studied both as finite words over the input and output alphabets and w-words which represent infinite sequences of values from the input and output alphabets. The definition of a language which allows the construction of an automaton to model a problem is pursued in depth. A precise specification for such a language, termed a meta-language, is developed. The meta-language I is developed and shown to be as expressive as possible, allowing for the greatest possible freedom in the definition of the problem without endangering the ability to synthesize an automaton to model the problem. Trakhtenbrot and Barzdin also address the topic of identifying the structure of an automaton from its observed behavior. Algorithms are developed for the identification of a machine within a black box based on the different types of experiments which may be applied and the restrictions placed on the machine to be identified.

Kountanis (1977) defines a learning system which is capable of learning the structure of a deterministic finite state machine from a class of finite state machines. The machines are restricted to have a bound on the number of states and a finite set of input and output values. The input and output alphabets are the same for all machines in a class. The machines are further restricted to be deterministic, completely specified, connected and to have only one initial state. The experiment applied to a machine is simple and the machine is assumed to begin the experiment in its initial state. The learning system, which itself takes the form of a nondeterministic finite state machine, focuses attention on the learning portion of the experiment defined by Moore, as opposed to the teaching portion. The learner is passive and only observes the input/output sequences applied to the machine. The learner has no say in determining the values of the input sequence. This function is the responsibility of the teacher and is not investigated in detail. The learning system is proven to converge to the machine to be identified provided the input/output sequence meets certain necessary and sufficient conditions. A tree structure similar to that defined by Gill (1961) is used to represent the identification process. The tree developed by Kountanis differs from that of Gill in that the root of the tree represents a machine with only one state and no transitions. Each successive level represents the possible machines which would explain the behavior observed so far. This difference in the interpretation of the tree structure is due to the nature of the problem to be solved. Gill attempts to determine a particular state of a machine which meets some criterion whereas Kountanis is attempting to identify the structure of the entire machine.

Kountanis and Mitchell (1979) evaluate the complexity of Kountanis' learning system with respect to the implementation used. Homomorphic images of the learning system are developed in an effort to reduce the size of the problem space.

Reibling (1983) extends the learning system defined by Kountanis to determine the structure of a probablistic automaton from a class of probablistic automata with stationary transition probabilities. This is accomplished by relaxing the restriction that the machines be deterministic and the existence of a reset function is added to allow the automaton to be reset to its initial state. The reset function allows multiple experiments to be performed on the machine to provide for the computation of probabilities for the state transitions. The complexity of the new learning system is investigated and is found, as expected, to be more complex than the learning system for deterministic machines.

CHAPTER III

DEFINITIONS FROM AUTOMATA THEORY

In this chapter, several definitions about finite state automata are provided which are pertinent to this research.

A finite state automaton is defined as a system

$$M(S, I, f, q_0, F)$$

where

S is a finite, nonempty set of states

I is a finite input alphabet

f is the next state function

q_0 is the initial state of M and is a member of S

F is a subset of S and is the set of final states

The next state function f is defined as

$$f : S \times I \dashrightarrow S$$

A sentence x over the alphabet I is said to be accepted by M if when the sentence is applied to M, with M starting in its initial state, the resulting state of M is a member of the set of final states F. This may be represented as

$$f(q_0,x) = p, \ p \text{ in } F$$

A finite state automaton may be defined by listing the states, input alphabet and set of final states and then enumerating all of the possible values for the next state function. Such a definition is presented in Figure 3. Another method of listing the values for the next state function is to create a table, indexed by the states of the machine and the values of the input alphabet. The entries in the table correspond to the values of the next state function. Figure 4 shows such a table for the machine defined in Figure 3.

$S = \{q_0, q_1\}$ $\qquad f : f(q_0, a) = q_0$

$I = \{a, b\}$ $\qquad f(q_0, b) = q_1$

$F = \{q_1\}$ $\qquad f(q_1, a) = q_0$

$f(q_1 b) = q_1$

Figure 3. A Finite State Automaton M

	a	b
q_0	q_0	q_1
q_1	q_0	q_1

Figure 4. Tabular Representation of Next State Function

Another means of representing a finite state automaton is through the use of a state diagram. In this type of representation, a directed graph is used to depict the structure of the automaton. Each node represents a state and is labelled with the name of the state. An arc between two nodes, say q and p, with the label i, indicates that if the automaton is in state q and receives an input value i, control of the machine is transferred to state p. Figure 5 shows the state diagram of the finite state automaton defined in Figure 3.

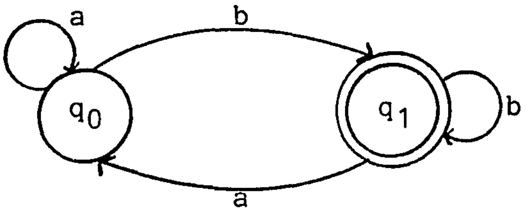

Figure 5. A State Diagram Representation

By listing the states of an automaton, its input alphabet and the set of final states and then enumerating all of the values of the next state function, one is said to have defined the behavior of the automaton. In the same way, a state diagram can be said to define the structure of an automaton. In the discussion to follow, a state diagram will most often be used to define an automaton.

A <u>deterministic</u> finite state automaton is a finite state automaton for which the next state function f defines only one possible state as the destination from a given state for a given input value.

$$f(q,a) = p, \quad q,p \text{ in } S, \quad a \text{ in } I$$

A <u>nondeterministic</u> finite state automaton is a finite state automaton for which the next state function f defines a set of states (possibly empty) as the destination from a given state for a given input value.

$$f(q,a) = \{p_1,p_2\} \quad q,\ p_1,\ p_2 \text{ in } S, \quad a \text{ in } I$$

This allows an automaton M in state q, upon receipt of the input value a to transfer control to either state p_1 or state p_2. Figure 5 depicts a deterministic finite state automaton and Figure 6 shows a nondeterministic finite state automaton.

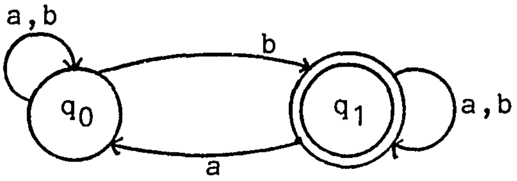

Figure 6. A Nondeterministic Finite State Automaton

A finite state machine is an adaptation of a finite state automaton in which the concept of a set of final states is removed and an output alphabet and output function defined. A finite state machine is defined as:

$$M(S, I, O, f, g, s_0)$$

where

S is a finite, nonempty set of states

I is a finite input alphabet

O is a finite output alphabet

f is the next state function

g is the output function

s_0 is the initial state of M

The next state function f is defined as

$$f : S \times I \dashrightarrow S$$

The output function g may be defined in one of several ways. The value of g may depend solely on the current state of M, in which case g is defined as

$$g : S \dashrightarrow O$$

Or, the value of g may depend on the previous state of M and the input value received. Then g is defined as

$$g : S \times I \dashrightarrow O$$

The latter definition of g will be used throughout this work. An example of a deterministic finite state machine is presented in Figure 7.

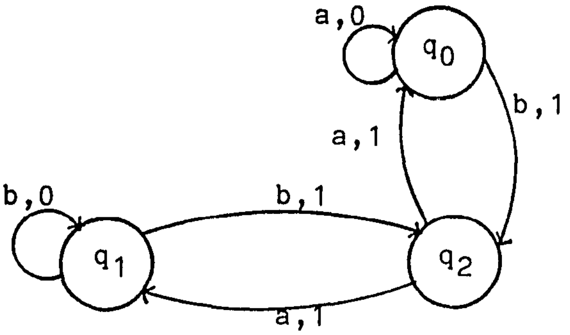

Figure 7. A Deterministic Finite State Machine

A finite state machine is referred to as a "state transducer" since it converts a sequence of values over the input alphabet into a sequence of values over the output alphabet.

A deterministic finite state machine is said to be <u>connected</u> if for every state q in S, there exists an input sequence x such that

$$f : (q_0,x) = q$$

that is, any state of the machine may be reached from the initial state given the proper sequence of input values.

A deterministic finite state machine is defined as being <u>completely specified</u> if for each state q in S, a transition from q is defined for every member of the input alphabet I.

CHAPTER IV

REVIEW OF KOUNTANIS' LEARNING SYSTEM

Kountanis (1977), in his development of a learning system, further refines the concept of the experimenter defined by Moore (1956). Moore's model of a learning system defines a black box containing a finite state machine and an experimenter which observes the behavior of the machine in the black box. Figure 8 gives an illustration of Moore's learning system.

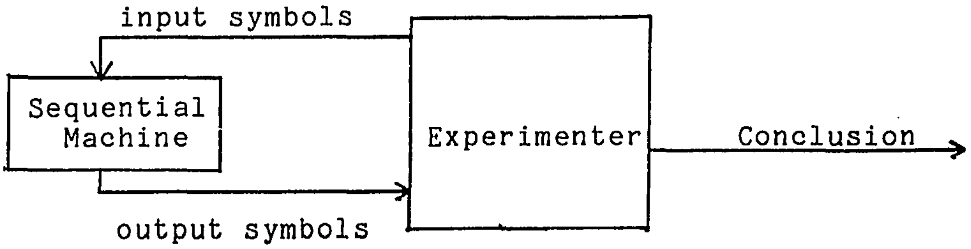

Figure 8. Moore's Learning System

Kountanis replaces the experimenter by its two major components, the teacher and the learning system. The teacher is responsible for generating the sequence of input values to be submitted to the machine in the black box. Both the input supplied to the machine and the corresponding output serve as the information supplied to the learning system. The learning system, after receiving the entire input/output sequence will produce an exact copy (up to isomorphism) of the machine in the black box, provided that the input/output sequence meets ce.-tain conditions. Kountanis' concept of a learning system is shown in Figure 9.

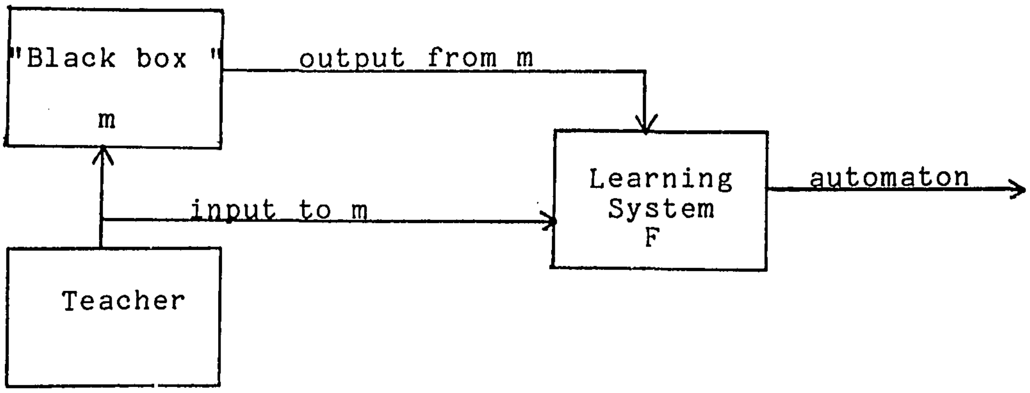

Figure 9. Kountanis' Learning System

The objective of this learning system is to uniquely identify (up to isormorphism) a finite state machine from a class of finite state machines. The class of finite state machines has several restrictions placed on it to allow the machine in the black box to be identified. It has been proven (Moore 1956) that it is impossible to identify a completely unknown finite state machine. (i.e. one in which there are no restrictions placed on its structure). The class of finite state machines will be defined as follows.

Definition 4.1: Define M(p,q,r) to be the class of finite state machines with the following characteristics

1) All machines in the class are defined over the same set of input and output sets with p and q their respective cardinalities.

2) There exists a bound r on the number of possible states

3) Each machine must be deterministic, completely specified, connected and have only one initial state.

Definition 4.2: Let m_i, a member of the class of machines M(p,q,r), be defined as follows

$$m_i = (S_i, I, O, f_i, g_i, s_{0i})$$

where

S_i is the set of states of machine m_i, cardinality $\leq$ r

I is the input set over which every machine of M(p,q,r) is defined, cardinality = p

O is the output set over which every machine of M(p,q,r) is defined, cardinality = q

f_i is the next state function of m_i

g_i is the output function of m_i

s_{0i} is the initial state of m_i

Several restrictions are placed on the learning system:

1) Only simple experiments will be performed. Only one copy of the machine is available for study and it cannot be reset to its initial state during the experiment.

2) The learning system does not participate in the

generation of the input sequence for the machine in the black box. This function is the responsibility of the teacher.

Kountanis' learning system, henceforth referred to as F, uses a tree structure during the identification of the machine. The root of the tree contains a machine with a single state and no transitions emanating from it. As each successive input/output pair is received, new machines are conjectured which could explain the behavior exhibited so far.

The following properties of the semilattice developed by the learning system F are stated in this work as definitions and where necessary are supported by an intuitive explanation. They are developed in full and proven by Kountanis in his dissertation. The reader is referred to Kountanis (1977) for a more rigorous treatment.

Definition 4.3: Let M be a deterministic finite state machine M = (S, I, O, f, g, s_0) defined (perhaps partially defined) over the input set I and the output set O. For some state s of the machine M, we say that the input/output pair (i,o) in (IxO) is permissible for M in state s if and only if either of the following conditions is true.

1) f(s,i) and g(s,i) are both defined and f(s,i) = s' for some s' in S and g(s,i) = o.

2) f(s,i) and g(s,i) are not defined.

Definition 4.4: An input/output pair (i,o) in (IxO) is not permissible for the machine M in state s if and only if both of the following are true.

1) f(s,i) and g(s,i) are both defined.

2) s(s,i) =s' for some s' in S and g(s,i) ≠ o.

The concept of permissibility may be intuitively viewed as follows. For a machine M in some state s, an (i,o) pair is permissible if a transition already exists from the state s with the label (i,o) or no transition yet exists from the state s for the input value i. The pair (i,o) is not permissible if a transition exists from the state s with an input label of i but an output label different than o. This concept is necessary in order to prevent a violation of the restriction that the machine M is a deterministic machine.

New machines are conjectured for each permissible (i,o) pair by creating all possible simple extensions of existing machines.

Definition 4.5: For two deterministic finite state machines M_1 = (S_1, I_1, O_1, f_1, g_1, s_{01}) and

$M_2 = (S_2, I_2, O_2, f_2, g_2, s_{02})$, both belonging to the class of machines $M(p,q,r)$, M_2 is said to be a simple extension of M_1 if and only if conditions 1, 2 and 3 and either 4 or 5 listed below are met. For machine M_1 in state s in S_1, for the input/output pair (i,o) in (IxO):

1) The pair (i,o) is permissible.

2) $I_2 = I_1 \cup \{i\}$.

3) $O_2 = O_1 \cup \{o\}$.

4) The machine M_1 is isomorphic to a submachine of M_2: M_2 has an additional transition from the state in M_2 which is the image of the state s in M_1 and the label of the transition is the pair (i,o). The initial state of machine M_2 is the image of the initial state of machine M_1.

5) The machine M_1 is isomorphic to a submachine of M_2: M_2 contains one more state than M_1, M_2 contains a transition from the state in M_2 which is the image of the state s in M_1 to the new state with a label of the pair (i,o). The initial state of machine M_2 is the image of the initial state of machine M_1.

A more intuitive description of definition 4.5 is the following.

1) The input/output pair (i,o) does not violate the restriction that the machine is deterministic.

2) The input set of M_2 is the same as the input set of

M_1, or is enhanced by the value of i.

3) The output set of M_2 is the same as the output set of M_1, or is enhanced by the value of o.

4) The machine M_2 contains the same number of states as the machine M_1 but a new transition is added from the state s (of M_1) to some other state in the machine with the label (i,o).

5) The machine M_2 has one more state than the machine M_1. This new state is reached by a transition from the state s (of M_1) with a label (i,o).

Note that condition 4 of the above assures that any machine is a simple extension of itself at any of its states by any (i,o) pair for which the next state and output functions are defined. The isomorphism in this case is the identity.

For the remainder of this research, the generation of simple extensions will be said to be by Rule 1 if the new machine is created through condition 4 of definition 4.5 or by Rule 2 if created through condition 5.

For each machine conjectured, the learning system F is said to remember the machine M along with a state s of M. The state s is called the remembered state of M and represents the current state of M in the processing of the input/output sequence received thus far.

To identify the machine contained in the black box, the learning system uses the following procedure. The tree of candidate machines, each of which represents a possible solution, is initialized to contain a single node called the root. This node represents a machine with only one state and no transitions. As each successive (i,o) pair is received, a new level of conjectural machines is added to the tree. For each machine in the current level (starting at level 0 with the root), the new (i,o) pair is evaluated. If the (i,o) pair is not permissible, no further development of this machine is done. If the (i,o) pair is permissible, then two cases exist. First, there already exists a transition from the remembered state of the current machine with the label (i,o). In this case, the new state of the machine indicated by f(s,i) becomes the remembered state of the machine and processing of this machine is complete. Second, no transition with the label (i,o) exists so all possible simple extensions of the machine are generated through the application of Rule 1 and Rule 2. Rule 1 generates all possible simple extensions which contain a transition from the remembered state of the current machine to all possible states of the machine. Rule 2 may generate one machine or none. If the bound on the number of states has not yet been reached, a new machine is created with an additional state and a transition from the previously remembered state to the new state with the label (i,o). If the bound on the number of states has been reached, no new machines are generated through Rule 2.

A sample of the operation of F is presented for reference.

Let M(p,q,r) = M(2,2,2)

Let $F_{(2,2,2)}$ be the learning system designed to learn M(2,2,2)

Let I = {a,b}, cardinality p = 2

Let O = {0,1}, cardinality q = 2

Let the bound on the number of states r = 2

The goal machine to be identified is presented in Figure 10.

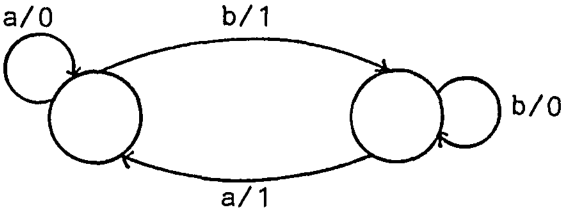

Figure 10. Goal Machine of Class M(2,2,2)

Let the input/output sequence received =

(ao)(b1)(b0)(a1)(a0)(a0)

Figure 11 shows the tree structure developed by F for learning the goal machir.e described.

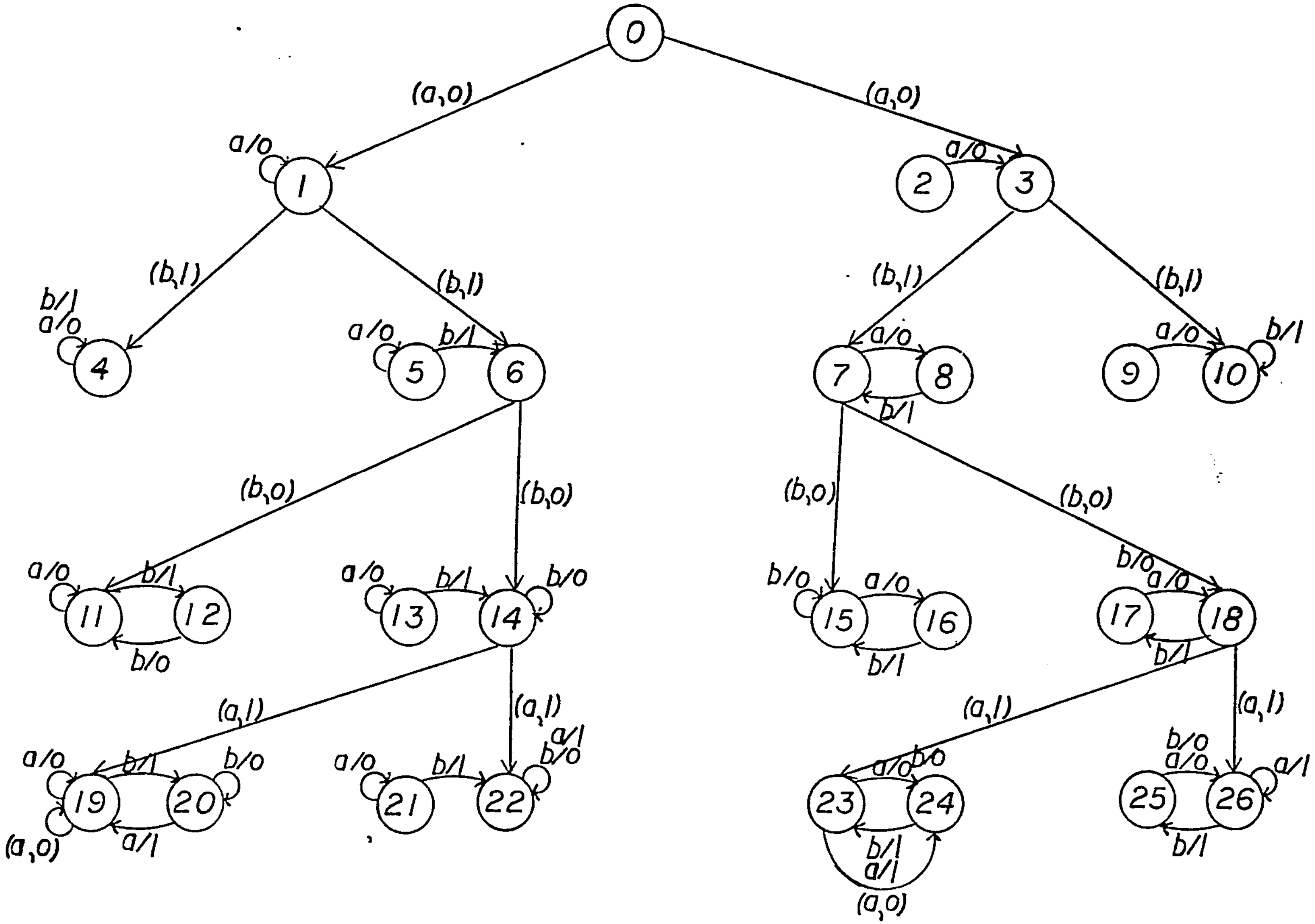

Figure 11. Sample Tree Developed by $F_{(2,2,2)}$

There are two types of labels which appear in Figure 11. The first type has the form (x,y) and represents an input to the learning system. Transitions with this type of label indicate the control structure of the tree of conjectural machines. The second type of label (x/y) is used to represent a transition within one of the conjectural machines. A transition with this type of label is said represent a control of the structure of an individual machine. This is referred to as a "local" control of the learning system. The arrows at the end of global transitions lines (those marked by (x,y) labels) indicate the state of the candidate machine remembered by the learning system along with the candidate machine.

The connection of two machines by a transition with an (x,y) label means that the machine in the lower level is an extension of the machine in the higher level. The conjectural machines of each level of the tree are simple extensions of the machines at the predecessor level.

The successive set of conjectural machines which the learning system computes in response to the received input/output pairs are listed below. Since each conjectural machine has only one initial state, we may use the name of the initial state as the name of the corresponding machine. Each machine is represented by a pair (M,s) where M is the name of the machine and s is the name of the remembered state.

Received (i,o) pair	Computed conjectural machines
(a0)	$\{(M_1,1),(M_2,3)\}$
(b1)	$\{(M_4,4),(M_5,6),(M_7,7),(M_9,10)\}$
(b0)	$\{(M_{11},11),(M_{13},14),(M_{15},15),(M_{17},18)\}$
(a1)	$\{(M_{19},19),(M_{21},22),(M_{23},23),(M_{25},26)\}$
(a0)	$\{(M_{19},19),(M_{23},24)\}$
(a0)	$\{(M_{19},19)\}$

We can see from the example that the only conjectural machine which has not been "pruned" from the tree is the machine M_{19}. This conjectural machine is the machine to which the learning system converges and is identical, up to isomorphism, to the goal machine in Figure 10.

It is worth noting that the input sequence may be divided into two parts. The head part, consisting of the (i,o) pairs (a0)(b1)(b0)(a1), covers every transition of the goal machine. All machines conjectured by F after processing the head part are completely specified. The remaining (i,o) pairs (a0)(a0) form the tail part of the sequence and serves to distinguish the goal machine from among the completely specified candidate machines.

The previous example only illustrates the development of the tree for a particular sequence of input/output pairs. It is interesting to study the tree which would result if one were to consider all possible values which could be received for a particular class of machines. This is useful in evaluating the complexity of the search space developed by F. For this example, the learning system represented will be $F_{(1,2,2)}$ which can learn any machine from the class of machines M(1,2,2).

Let I = {a}, cardinality = 1

Let O = {0,1}, cardinality = 2

Let r = 2, the bound on the number of states

The name of the initial state of each machine will be used as the name of the conjectural machine as in the previous example. Figure 12 illustrates the entire search space for the system $F_{(1,2,2)}$. The successive sets of machine state pairs created by $F_{(1,2,2)}$ are as follows:

Level 0 - $\{(M_0,0)\}$

Level 1 - $\{(M_1,1),(M_2,3),(M_4,5),(M_6,6)\}$

Level 2 - $\{(M_7,8),(M_9,9),(M_{11},12),(M_{13},13),(M_{15},16),$
$(M_{17},17,M_{19},20),(M_{21},21)\}$

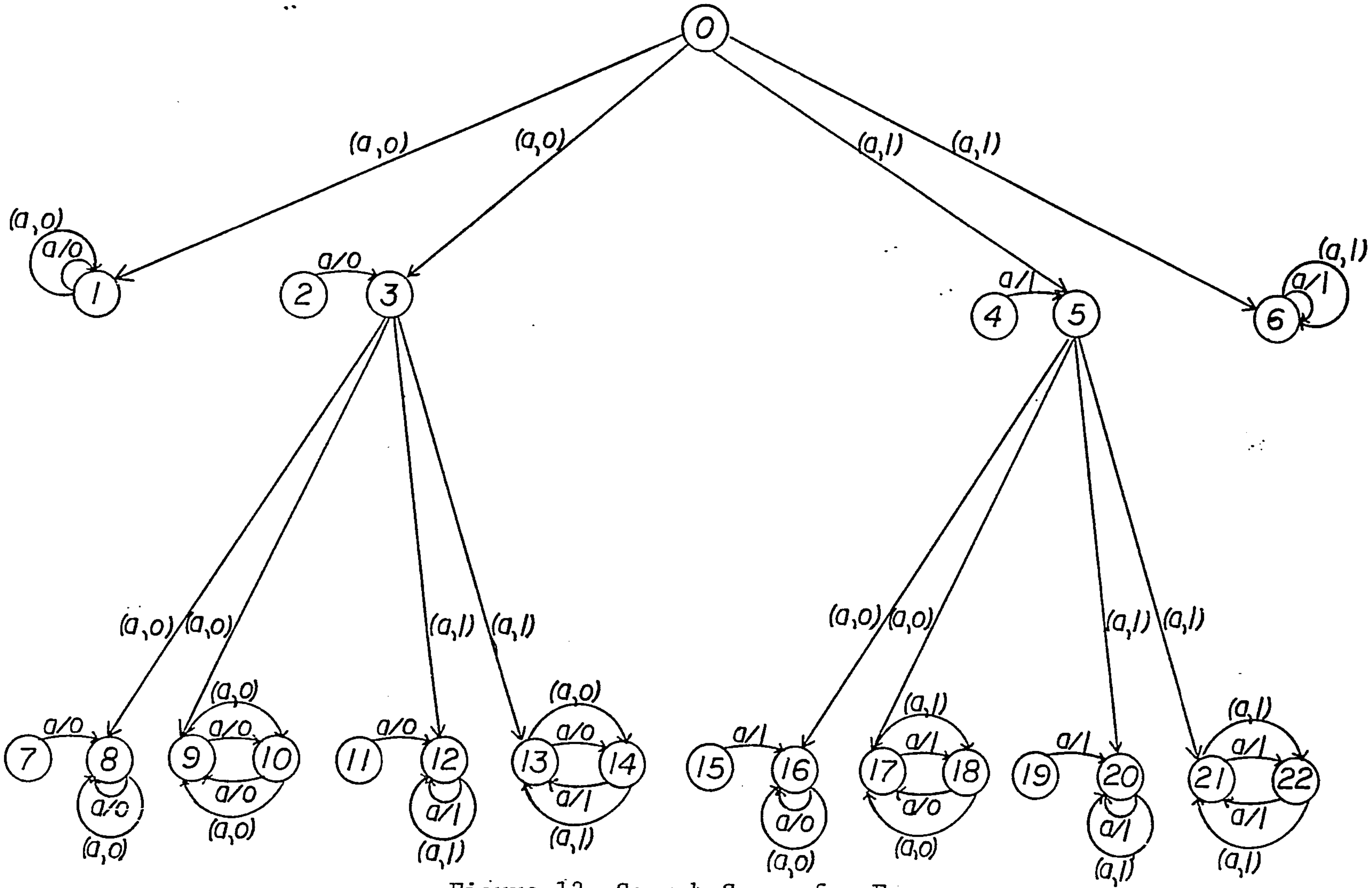

Figure 12. Search Space for $F_{(1,2,2)}$

Kountanis proves that the set of all conjectural machines of a learning system F forms a semilattice. The learning system F is also shown to converge to the goal machine provided that the input/output sequence meets the following necessary and sufficient conditions.

1) The sequence x is composed of a head part x_h and a tail part x_t.

2) x_h covers all transitions of the goal machine

3) x_t distinguishes each of the machines resulting from the processing of x_h.

Complexity of The Learning System F

In this section, several features of the complexity of the semilattice are evaluated and the size of the search space created by F is examined.

Definition 4.6: The width of the semilattice at a particular level is the number of conjectural machines at that level.

Definition 4.7: The depth of the semilattice at a particular level is the minimum number of transitions in a trace from the initial machine to a conjectural machine at that level. This is equivalent to the length of the sequence of (i,o) pairs received so far.

Definition 4.8: The depth of the semilattice is the depth of the last level.

Definition 4.9: $M_{i,j}$ is said to denote a conjectural machine M in the i^{th} level of the semilattice with j states.

Definition 4.10: When enumerating the states of a conjectural machine, s_1 will denote the remembered state of M.

Definition 4.11: Let $n_i(t_1,t_2,...,t_j)$ denote the number of conjectural machines in the i^{th} level of the semilattice which have j states, where t_1, t_2, ...t_j are the number of transitions emanating from the corresponding states s_1, s_2,...s_j of the machines. $m_i(t_1,t_2,...,t_j)$ will denote one of these machines. As an example, consider the machine M_{13} from Figure 12, reproduced below in Figure 13. This is an $M_2(1,1)$ machine, a representative of the set $N_2(1,1)$.

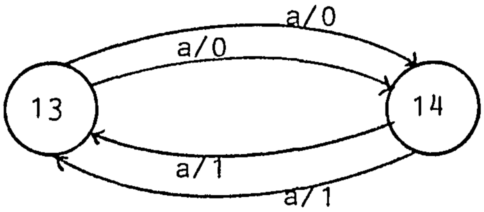

Figure 13. An $M_2(1,1)$ Machine from Figure 12

The value of $N_2(1,1)$ for Figure 12 would be 8 because there are eight such $M_2(1,1)$ machines in level 2 of the learning system $F_{(1,2,2)}$.

Definition 4.12: If j denotes the number of states of a conjectural machine in the i^{th} level of the semilattice, then min(j) = 1 and max(j) = i + 1.

The value of the minimum is easy to discern by examining the machine M_0 at level 0 of the semilattice. The maximum number of states is derivable by the observation that at each level at most one state can be added if Rule 2 is used to generate the new machine. Since the machine M_0 contains one state, the maximum for a machine a level i is i + 1.

Definition 4.13: If T_i denotes the total number of transitions of a machine at the i^{th} level of the semilattice, then $T_i = i$.

A simple extension of a machine has one more transition than its predecessor. The machine M_0 at level 0 has no transitions and at each level at most one transition may be added. Therefore $T_i = i$.

Definition 4.14: Let T = $\{t | t$ is the number of transitions emanating from a state in $m_i(t_1, t_2, \ldots, t_j)\}$. Then,

$t_{max} = \max(T) = (i-j+2)$

$t_{min} = \min(T) = 0$

The minimum value t_{min} is easy to see. For the machine M_0 has no transitions emanating from its single state. Also, a machine created through Rule 2 will have one state (the newly added state) with no transitions. The maximum, t_{max}, may be derived by considering that each of the conjectured machines is defined to be connected. Therefore, every state with the possible exception of the remembered state must have at least one state emanating from it. To obtain the maximum value, substitute the minimum values for all other states of M. This yields

$$0 + 1 + 1 + \ldots + t_{max} + \ldots + 1 = i \quad \text{or}$$

$$0 + (j-2)1 + t_{max} = i \quad \text{or}$$

$$t_{max} = (i-j+2)$$

Definition 4.15: The maximum and minimum values for t_1 for all conjectural machines of the form $m_i(t_1,t_2,\ldots,t_j)$ is given by

$$t_{1min} = 0$$

$$t_{1max} = (i-j+1)$$

This is similar to Definition 4.14 with the exception that the value of t_1 cannot be zero yielding

$$t_{1max} + 1 + 1 + \ldots + 1 = i \quad \text{or}$$

$$t_{1max} = (i-j+1)$$

Definition 4.16: The width of the semilattice at level i is given by

$$w_i = \sum_{j=1}^{i+1} N_{ij}$$

where N_{ij} is the number of machines at level i with j states.

Definition 4.17: If there exists a bound r on the number of states of each conjectural machine, the depth D of the semilattice may be defined as

$$D = pr$$

where p is the cardinality of the input set I.

Every conjectural machine in a level of the semilattice is a simple extension of a machine in the previous level. But a simple extension of a machine M has exactly one more transition than its predecessor. Therefore, the depth of the semilattice is equal to the number of possible different transitions in a completely specified, deterministic machine with r states which is defined over the input set I and the output set O. There are p possible different transitions emanating from each state of a completely specified machine over I. Thus, from r states there will be pr different transitions in the machine. Therefore, D = pr.

Definition 4.18: If there exists a bound r on the number of states allowed for a conjectural machine and the cardinality of the input set is given by p then

1) for $i \leq (r-1)$ the width of the semilattice is given by

$$w_i = \sum_{j=1}^{i+1} N_{ij}$$

where N_{ij} is defined as the number of machines in the semilattice at level i with j states.

2) for $r \leq i \leq pr$, the width of the semilattice is given by

$$w_i = \sum_{j=1}^{r} N_{ij}$$

where N_{ij} is the number of machines in the semilattice at level i with j states.

Definition 4.19: For a learning system with input sets I and O having cardinalities p and q respectively, then the number of machines a level i of the semilattice having one state with no transitions emanating from it is given by

$$n_i(0,t_2,t_3,\ldots,t_j) = (p-t_2+1)qN_{i-1}(t_2-1,t_3,\ldots,t_j)$$

where

$$1 \leq j \leq i$$

$t_2, t_3, \ldots, t_j > 0$

$N_0(0) = 1$

These machines will be the ones created from machines in level (i-1) through the application of Rule 2. Consider such a machine in level i. The remembered state t_1 will have no transitions emanating from it, hence $t_1 = 0$. The remembered state of the predecessor machine t_2 will have had a new transition added to the new state t_1, hence the value of t_2-1 in $M_{i-1}(t_2-1,\ldots,t_j)$. For each such $M_{i-1}(t_2-1,t_3,\ldots,t_j)$ machine there are $(p-(t_2-1))q = (p-t_2+1)q$ possible transitions. This is given by the fact that the number of transitions possible from the state t_2 of M_{i-1} is limited by the number of input values not yet used to label a transition from that state, $(p-(t_2-1))$, multiplied by the number of possible output values which may be associated with each input value, q. Thus the possible labellings for a transition is given by $(p-(t_2-1))q = (p-t_2+1)q$. Each of the $(p-t_2+1)q$ transitions may be attainable for each of the $N_{i-1}(t_2-1,t_3,\ldots,t_j)$ machines. This yields the value of $(p-t_2+1)qN_{i-1}(t_2-1,t_3,\ldots,t_j)$ which must be evaluated for all values of j.

Definition 4.20: For a learning system with input sets I and O having cardinalities p and q respectively, then the number of machines at level i of the semilattice having no state with zero transitions emanating from it (all states have at least one transition) is given by

$$N_i(t_1,t_2,\ldots,t_j) = \sum_{k=1}^{j} (p-t_k+1)qN_{i-1}(t_k-1,t_{k+1},\ldots,t_{k-1})$$

where the addition of indices is carried out modulus j

$t_1,t_2,\ldots,t_j > 0$

$1 \leq j \leq i$

These machines will be the ones created from machines in level (i-1) through the application of Rule 1. Each of the N_i machines will have the same number of states as its predecessor in level (i-1) but a new transition will have been added to the state t_1 of M_i from some state t_k of M_{i-1}. The new transition may carry any of the $(p-t_k+1)q$ labellings. This is derived from the number of input values not yet used to label transitions from the remembered state t_k of M_{i-1}, $(p-(t_k-1))$ or $(p-t_k+1)$ multiplied by the number of possible output values which may be applied, q. The number of machines in level (i-1) which may have created the new machines at level i is given by $N_{i-1}(t_k-1,t_{k+1},\ldots,t_{k-1})$ where a machine of the form $M_{i-1}(t_k-1,t_{k+1},\ldots,t_{k-1})$ may have any one of the forms

$$M_{i-1}(t_1-1,t_2,\ldots,t_j)$$

$$M_{i-1}(t_2-1,t_3,\ldots,t_j,t_1)$$

$$. \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad .$$

$$M_{i-1}(t_j-1,t_1,\ldots,t_{j-1})$$

Each of these machine may take any of the $(p-t_k+1)q$ labellings yielding

$$\begin{aligned} N_i(t_1,t_2,\ldots,t_j) &= (p-t_1+1)qN_{i-1}(t_1-1,\ldots,t_j) + \\ &\quad (p-t_2+1)qN_{i-1}(t_2-1,t_3,\ldots,t_j,t_1) + \\ &\quad . \quad . \quad . \quad . \quad . \quad . \quad . \\ &\quad (p-t_j+1)qN_{i-1}(t_j-1,t_1,\ldots,t_{j-1}) \\ &= \sum_{k=1}^{j} (p-t_k+1)qN_{i-1}(t_k-1,t_{k+1},\ldots,t_{k-1}) \end{aligned}$$

where the addition of the indices is carried out with modulus j arithmetic because the configured machines have j states numbered from 1 to j.

Size of The Search Space of The Semilattice

To understand the need for an improvement in the performance of the learning system F, it is useful to examine the size of the search space created by the semilattice. It is not possible to determine the exact size of a problem space unless the class of machines to be solved is known and the input/output sequence for a given machine is known. However, some information can be determined as to the maximum size of the search space.

This is directly applicable to the performance of F since in order to determine the goal machine, F must essentially examine all of the candidate machines in the search space.

First to be investigated is the rate at which the semilattice can grow.

In order to determine the growth rate of the semilattice, it is necessary to determine how many conjectural machines can be generated from a single machine if the largest possible growth rate is assumed. The number of descendants of a machine M is dependent on the (i,o) pair received. If the (i,o) pair is not permissible for the machine M in its remembered state, no successors are generated. If a transition already exists from the remembered state of M with a labelling of (i,o), then one machine is generated (a copy of the current machine). If no transition exists with the label (i,o), then two cases exist

1) The machine M does not yet contain the maximum number of states allowed ($j < r$). In this case, j machines will be added by adding a transition from the remembered state to each of the j states of M (by Rule 1). In addition, one machine will be generated by Rule 2.

2) The machine M contains the maximum number of states allowed ($j = r$). In this case, only Rule 1 may be used to

generate successors. There are r new machines created by adding a transition from the remembered state of M to each of the r states of M.

The maximum number of descendants possible for each machine upon the receipt of a new (i,o) pair is then given by r. This leads to an absolute worst case complexity of the number of machines at level i of the semilattice of r^i. This could only be the case if at each level of the semilattice, each machine generated r successors. In reality, this will never be the case. Initially, M_0 only contains one state. At each successive level, a maximum of one state may be added to a machine therefore, a machine with r states could not exist before level (r-1). However, this worst case value allows us to establish an upper bound on the size of the search space created by the semilattice.

The maximum depth of the semilattice has already been established as D = pr. This is the level at which all machines are completely specified. After this level, no new machines may be generated. A machine may be retained if it successfully accepts the tail portion of the input/output sequence or it may be "pruned" from the semilattice if the (i,o) pair represents a nonpermissible transition. At level pr, the maximum number of machines which may have been created is represented as follows.

1	machine M_0, level 0
r	machines at level 1
r^2	machines at level 2
. . . . .	
r^{pr}	machines at level pr

This yields a total of $1 + r + r^2 + \ldots + r^{pr}$ or

$$\sum_{k=0}^{pr} r^k$$

which may be expressed as

$$\frac{1 - r^{pr+1}}{1 - r}$$

One can easily see that the size of the search space grows exponentially with respect to the constraints placed on M(p,q,r). The size of the search space quickly becomes the determining factor in limiting the the class of machines for which the learning system may be implemented. It is for this reason that the rest of this research is dedicated to attempting to find a method to reduce the size of the search space which must be examined in an effort to improve the performance of the learning system.

CHAPTER V

SEARCH METHODOLOGIES

In this chapter, two types of search will be examined. Both types of search are applicable to searching a tree structure such as the semilattice developed by the learning system F. The study of search methodologies is presented in an effort to devise possible ways to improve the performance of the learning system.

As each successive input/output pair is received by F, a new level in the tree of conjectural machines is created. All possible machines are developed until the one which describes the behavior of the goal machine is identified. This is roughly equivalent to having all of the possible machines already defined and then searching through them to find the one which matches the goal machine. If a method exists for minimizing the amount of search done, the method can be converted for use in minimizing the development of the tree of conjectural machines.

Tree Searches

One method of searching a tree structure for a particular node is called the breadth first search. With this approach, all of the nodes of the tree at one level are developed or searched before any of the nodes in a succeeding level. For the tree depicted in Figure 14, the nodes would be developed by a breadth first search in the sequence A, B, C, D, E, F, G.

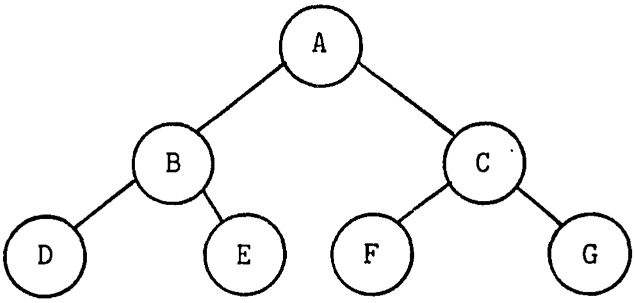

Figure 14. Breadth First Tree Traversal

The breadth first search is advantageous in that it is a conservative approach. If a goal node exists in the tree, the breadth first search is guaranteed to find it even if the depth of the tree is infinite or effectively infinite. If the goal node is likely to be found high in the tree, a breadth first search may be a good choice. Because it develops all of the nodes at one level before moving on to the next one it can be very wasteful, particularly if the tree is very broad but shallow or if the goal node is not very likely to occur high in the tree. In general, the worst case expected for the performance of a breadth first search is said to be of order N, denoted O(N), where N is the number of nodes in the tree. This implies that in the worst case, N nodes will be expanded before a solution is found. In the case of the semilattice, a breadth first search would be expected to expand the worst possible number of nodes. A solution cannot exist in the semilattice before level pr as the goal machine is defined to be completely specified and the semilattice must be expanded to level pr before all machines are completely specified. This requires the expansion (worst case) of

$$\sum_{k=o}^{pr} r^k$$

nodes.

Another method of searching a tree is the depth first search. Starting at the root of the tree, only one successor branch is pursued. At each successive level, one branch of the tree is chosen and pursued. This process continues until the search reaches the last level of the tree, or some other restriction is used to limit the search if the depth of the tree is infinite or effectively infinite. If a solution has not been found and the search reaches the bottom of the tree, it backs up to the previous level and investigates another alternative node. For the tree depicted in Figure 15, if the convention is to examine the left child of a node before the right, the nodes would be developed in the sequence A, B, D, E, C, F, G.

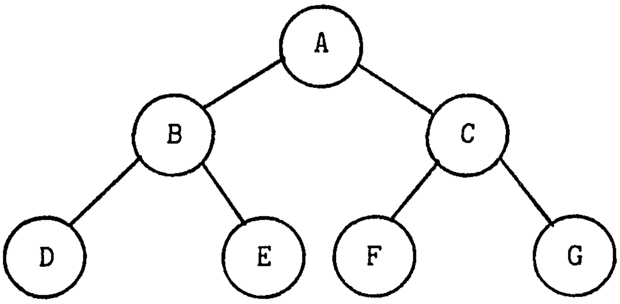

Figure 15. Depth First Tree Traversal

This method can yield a solution more quickly than a breadth first search, however, it is not guaranteed to find the shortest solution in the tree. The search terminates after the first solution has been found. This may be considered an aggressive yet dangerous type of approach. Without a limit to the depth of the search, it may continue infinitely. The worst case performance of the depth first search is said to be of order N, denoted O(N) where N is the number of nodes in the tree. The worst case performance results when the only solution in the tree is found in the last branch of the tree examined. In application to the tree of the semilattice, this worst case is no worse than the breadth first search. In the best possible case, the search will terminate with the first completely specified machine found which accepts the entire input/output sequence. This occurs by level pr. The search is guaranteed to terminate because the input/output sequence is of finite length. If a machine is located by a depth first search which accepts the entire input/output sequence, it is guaranteed to be a solution that is isomorphic to the goal machine. This is true because the sequence in input/output pairs is constrained to contain a head part which covers all transitions of the goal machine and a tail part which uniquely distinguishes the goal machine from the set of all possible completely specified machines. This analysis would seem to imply that the application of some type of depth first search to the development of the semilattice would yield an improvement in the performance of the learning system.

Search Improvements

In an attempt to reduce the amount of time and/or space required by a search, there are basically two approaches which may be taken. One is to reduce the size of the search space and the other is to reduce the number of elements examined (through some type of improvement on the search method).

To actually reduce the size of the search space, it may be possible to redefine the problem to be solved in different terms such that the size of the new search space is smaller than the original one. An example of this is the mutilated chessboard problem.

Suppose two diagonally opposed corner squares are

> removed from a standard 8 X 8 square chessboard. Can 31 rectangular dominoes, each the size of exactly two squares be so placed as to exactly cover the remaining board ?

If the search space is defined to contain the configurations which may be obtained by placing dominoes on the mutilated chessboard, the search space is very large. If, however, one observes that every dominoe placed on the board must cover both a black and a red square, and that the squares removed from the corners are both of the same color, the answer becomes obvious. The search was avoided altogether.

Probably the most interesting means of improving the efficiency of a search procedure is through the use of an heuristic. Some knowledge about the domain of the problem is brought to bear to reduce the amount of searching done. The meaning of "heuristic" is not well agreed upon. However, in the context of reducing the amount of search to be done, the following definition will suffice.

> Heuristic: A piece of knowledge capable of suggesting actions to follow or implausible ones to avoid. (Lenat 1982)

Using an heuristic to guide a search has received much attention. Two types of search which easily demonstrate the use of an heuristic are those done on adversary game trees and graph searches.

An adversary game tree is a structure used in game playing systems to model the decisions which may be made by two opponents making alternate moves. All possible choices in the game are known, there is no element of chance as in a card game. Adversary game trees are usually constructed using an AND/OR tree. The tree is drawn from one players perspective with nodes representing that players moves drawn as OR nodes, and his/her opponents moves drawn as AND nodes. Figure 16 shows a typical AND/OR tree.

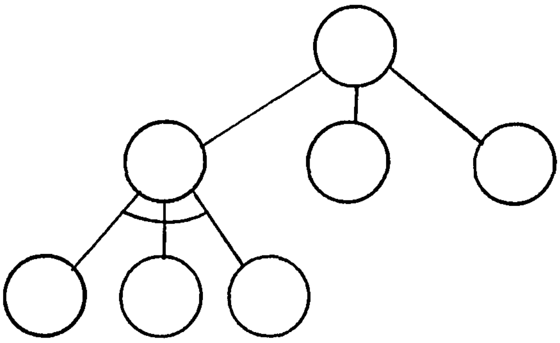

Figure 16. AND/OR Tree

Moves are evaluated by assigning a value to each of the nodes which represents the relative value of that move to the player. The player whose perspective is represented by the tree if often referred to as MAX and nodes which are favorable to him/her are given high values and nodes which are favorable to his/her opponent, named MIN are given low values. Often positive values are used for MAX and negative values are used for MIN. The tree may then be searched to the tip nodes and the best possible move determined. This can prove to be a major undertaking (or an impossible one) if the search space is large which it often is.

The Alpha-Beta pruning technique eliminates the unnecessary exhaustive search done on an AND/OR tree to evaluate each of the nodes. A tip node is evaluated as soon as it is created and its value is backed up to the preceeding nodes if appropriate. If a node which has been evaluated thus far represents the best possible choice which could be made, no more nodes at successive levels need to be investigated. A lower bound is established on MAX's nodes, said to be an alpha value, and represents the worst that MAX could do. An upper bound is established on MIN's moves, said to be a beta value, and indicates the best move that MIN could make. The bounds on the backed-up values may be revised but it may be noted that

1) The alpha value of MAX nodes (including the start node) can never decrease, and

2) The beta values of the MIN nodes can never increase.

Because of these constraints, the following rules for discontinuing the search may be stated.

1) Search can be discontinued below any MIN node having a beta value less than or equal to the alpha value of any of its MAX node ancestors. The final backed-up value of this MIN node can then be set to its beta value. This value may not be the same as that obtained by a full search, but its use results in selecting the same best move.

2) Search can be discontinued below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN ancestors. The final

backed up value of this MAX node can then be set to its alpha value.

During search, the alpha and beta values are computed as follows:

1) The alpha value of a MAX node is set equal to the current largest final backed-up value of its successors.

2) The beta value of a MIN node is set equal to the current smallest final backed-up value of its successors.

Much research has been done in the analysis and possible improvements of the Alpha-Beta pruning technique. One work of interest is that of Nau (1983). Nau investigates the effect of the depth of the search made as related to the quality of the decision made as a result of the search. It is mathematically shown, in this work, that an increase in the depth of the search does not necessarily improve the quality of the decision made. In fact, there exist a class of searches which as the depth of the search is increased, the quality of the decision can actually worsen to the point of being equivalent to a random choice. This condition is referred to as "pathology" and is caused by the fact that as the search depth increases, the likelihood of all children of a node having the same value increases (using some static evaluation function to determine the value of the nodes). This effectively reduces the choice of which path to follow to a random one. The types of searches for which this condition was found to exist are those used to evaluate game trees, such as the previously described AND/OR tree.

A classic algorithm for graph search procedures is called A as developed by Nilsson (1971). In the algorithm A, each arc of the graph is assigned a finite cost. The cost of a path in the graph is the sum of the costs of each arc in the path. As each node in the graph is developed, it is assigned a cost g(n) which represents the minimal-cost path located so far from the start node to the node n. Also associated with each node is an estimated cost h'(n) of the minimal-cost path from the node n to a goal node. The cost of the path from the start node to a goal node which passes through the node n is represented by

$$f(n) = g(n) + h'(n)$$

As each node is visited, the value of f(n) is computed. To choose the next node in the graph to search, the node with the smallest value of f(n) is used. It has been proven by Nilsson (1971) that if sufficient constraints are placed on the heuristic h'(n), the algorithm A is guaranteed to terminate and provide the cost of the minimal-cost solution path. The restriction placed on the heuristic h'(n) is that h' must satisfy the admissibility condition which states:

> The heuristic search estimate h' is called admissible if for every node n in the search graph G, $h'(n) \leq h(n)$

where h(n) is the actual cost of the minimal-cost path from a node n to a goal node. In other words, h'(n) must underestimate the true value of h(n).

Bagchi and Mahanti (1983) as well as others have scrutinized algorithm A and proposed enhancements. Two other algorithms are compared to Nilsson's and the behavior of each is studied when the admissibility condition is relaxed. Other, less stringent, restrictions on the heuristic estimator are examined. These are defined below.

> A heuristic function h' will be called nonmisleading if the following condition holds. Let m and n be any two nodes in the search graph such that $m \neq s$ and $n \neq s$ where s is the start node. Let P_1 be any path from s to m and P_2 be any path from s to n. Let $C(P_i)$ be the cost of the path P_i. Then

$$C(P_1) + h'(m) < C(P_2) + h'(n) \Rightarrow C(P_1) + h(m) \leq C(P_2) + h(n)$$

An heuristic is defined to be proper as follows

> 1) A path P_1 is a subpath of path P_2 if every arc in P_1 is present in P_2.
>
> 2) A heuristic h' is proper if the following condition holds. Let m and n be any two nodes in the search graph, such that $m \neq s$ and $n \neq s$, where s is the start node. Let P_1 be any path from s to m and P_2 any path from s to n, such that P_1 is not a subpath of P_2 and P_2 is not a subpath of P_1. Let $C(P_i)$ be the cost of the path P_i. Then

$$C(P_1) + h'(m) < C(P_2) + h'(n) \Rightarrow C(P_1) + h(m) \leq C(P_2) + h(n)$$

Wilkins (1982) provides a distinction between a parameter controlled search and a knowledge controlled search. A search is said to be parameter controlled, if the heuristic used in the search uses a parameter which is not related to the problem to terminate or direct the search. An example of such a parameter would be the depth limit imposed on a depth first search. A search is said to be knowledge controlled if knowledge of the problem is employed as the controlling factor in limiting or directing the search. An example of this is a threshold value which must be exceeded before an alternative is considered a viable move and further exploration is terminated. The calculation of the threshold value would be dependant on the problem domain.

The application of an heuristic to prune a branch from a tree is another method of reducing the search space (as in the Alpha-Beta technique). If it can be determined, without an exhaustive search, that a node in the tree cannot possibly lead to a solution (or that a better choice already exists), the node is pruned from the tree. No further development of the node or its successors takes place. Pruning techniques usually require a great deal of knowledge of the problem being solved before they can be applied. If a branch of the tree is pruned by a not-so-well informed heuristic, a solution to the problem might never be found !

With regard to improving the performance of the learning system F, it is believed that the application of an informed depth first search which uses an heuristic to choose which node in the tree to develop next will result in an increase in the size of the class of machines M(p,q,r) which may be learned by F for a given implementation.

CHAPTER VI

PERFORMANCE IMPROVEMENT OF THE LEARNING SYSTEM

It is believed that the application of a depth first approach to the development of the tree of candidate machines will improve the performance of the learning system F. The depth first approach has two advantages over the breadth first method. First, the goal machine may be located faster, that is with the development of fewer candidate machines. In the breadth first approach, the learning system is guaranteed to create all conjectural machines up through level pr. (This is the level at which all of the machines in the tree are completely specified.) In the semilattice with a depth of D = pr, there may be as many as

$$\sum_{k=0}^{pr} r^k$$

conjectural machines. The average number of machines created is the same as the worst case. All of the machines through level pr are developed before any attempt is made to identify the goal machine. In the depth first approach, only one branch of the tree is developed at a time. The best possible performance would be achieved when the goal machine exists at the end of the first branch of the tree to be developed. This would require developing only (worst case) r machines at each level for pr levels, for a total of pr(r) machines. In the worst case, every branch of the tree would be developed, locating the goal machine in the last branch of the tree developed. However, the worst case in the depth first approach is the same as the average case in the breadth first approach. Therefore, a depth first approach can do no worse than a breadth first approach and may yield a significant improvement. Because fewer machines are likely to be conjectured, the time to identify the machine in the black box will be less.

The second way that the depth first approach can improve on the breadth first method is that fewer machines need to be remembered by the learning system at any given time. As only one branch of the tree is developed at a time, the maximum number of machines to be remembered is pr(r). When a branch of the tree is proven to not contain a goal machine, all machines in that branch of the tree may be "forgotten" by the learning system, reducing the amount of space required to represent the semilattice.

As the semilattice has been proven to converge to a goal machine, we know that a goal machine exists in the semilattice. The breadth first approach is guaranteed to locate the goal machine. Can the same be said of the depth first approach? The answer is yes as shown below.

1) The depth of the semilattice is finite, attaining a depth D = pr. The depth first development is guaranteed to proceed to level pr before a goal machine can be identified as the goal machine, constrained to be a completely specified machine, could not occur higher in the tree than level pr.

2) The input/output sequence is of finite length. The head part identifies all completely specified machines and the tail part uniquely identifies (up to isomorphism) the goal machine from the set of completely specified machines. The length of the head part is pr and the length of the tail part is guaranteed to be no greater than pr.

3) Therefore the depth first approach is guaranteed to terminate because of the finite length of the input/output sequence. If the depth first method identifies a machine which accepts the entire input/output sequence, the machine is guaranteed to be isomorphic to the goal machine because the input/output sequence is constrained to identify the goal machine from the class of machines M(p,q,r).

Heuristically Guided Search

To choose the order in which the branches of the tree are developed, an heuristic will be used to guide the depth first search. Because the search does not make its choice at random, it is called an informed depth first search. By choosing the branch of the tree which is most likely to lead to the goal machine, the number of conjectural machines developed will be kept to a minimum, hence reducing the time required to identify the machine in the black box. The heuristic to be used will be referred to as the "greatest number of descendants" or GND. It is expected that the machine with the greatest number of descendants will have the best chance of leading to a goal machine. This procedure develops the fullest branch of the tree first. If this proves unsuccessful, then the next fullest branch of the tree is investigated. The procedure for the development of the tree of candidate machines using the GND heuristic to guide the depth first search is as follows.

1) At each level of the tree accept a new (i,o) pair. If there are no more input/output pairs a solution has been found. Otherwise, prune all conjectural machines for which the (i,o) pair is nonpermissible from the remembered state.

2) Next, apply an evaluation function E which calculates the number of descendants to each of the conjectural machines in the tree at the current level.

3) Choose the machine with the largest value of E and develop this machine. That is, add all possible simple

extensions of the machine to the tree. In the event of a tie, one possible resolution is to choose the first machine generated (the left most offspring).

4) If a machine cannot be expanded, delete that machine from the tree and choose the machine with the next highest value of E. If all machines at a level have been exhausted without finding a solution, back up to the previous level and continue with the next possible machine at that level.

The application of this method to the search space illustrated in Figure 11 would result in the following sequence of machines being generated. Each machine is named using the name of its initial state. The pair (M,s) indicates a conjectural machine M and its remembered state s.

Input	Level	Machines
	0	$\{(M_0,0)\}$
(a,0)		
	1	$\{(M_1,1),(M_2,3)\}$
(b,1)		Choose M_1
	2	$\{(M_4,4),(M_5,6)\}$
(b,0)		M_4 pruned, choose M_5
	3	$\{(M_{11},11),(M_{13},14)\}$
(a,1)		M_{11} pruned, choose M_{13}
	4	$\{(M_{19},20),(M_{21},22)\}$
(a,0)		M_{19} accepts, choose M_{19}
(a,0)		M_{19} accepts, choose M_{19}
		End of inputs, goal machine = M_{19}

Only 9 conjectural machines are developed using the depth first approach as opposed to the 15 developed by the breadth first approach. Figure 17 illustrates the tree developed by $F_{(2,2,2)}$ using the informed depth first approach.

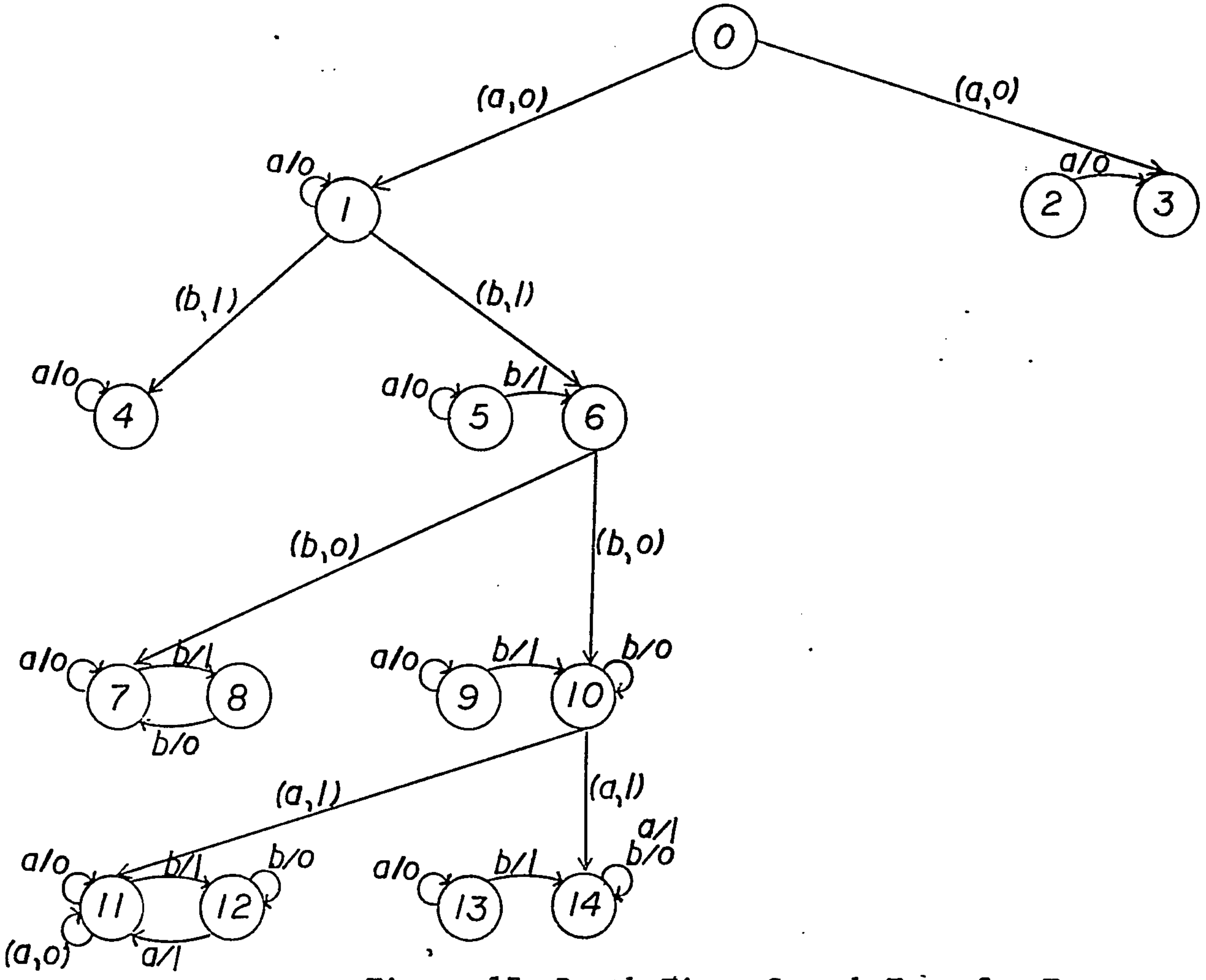

Figure 17. Depth First Search Tree for $F_{(2,2,2)}$

Calculation of The Evaluation Function

In this section, several methods for calculating the evaluation function E are examined and one is selected as being more feasible to implement than the others.

The first method of calculating the number of descendants of a machine is the direct application of the functions developed by Kountanis (1977). See Definitions 4.19 and 4.20 in this work for a definition of these functions. For a machine at level i of the semilattice, to calculate the total number of descendants, one can calculate the number of simple extensions, thus determining the number of descendants in level i+1. Next, for each machine in level i+1, calculate all possible extensions. Continue this process until the number of descendants in level pr, the last level of the semilattice, are known. The sum of all these values would be the total number of descendants of the machine at level i. This method has several drawbacks. One is that the configuration for each machine must be known in order to determine the number of simple extensions possible. More specifically, one must know the number of states of the machine, the number of transitions from the remembered state and the labels on all transitions from the remembered state. Another problem is that the entire input/output sequence must be known at the first level of the tree in order to know the configurations of the descendants of each machine (some machines may be pruned from the tree because of nonpermissible (i,o) pairs). In order to evaluate the total number of descendants of each machine, as much work must be done as in the development of the entire search tree when no evaluation function is applied. Because of this, an exhaustive evaluation of descendants to the last level of the semilattice is rejected as being too expensive and resulting in no savings of either time or space.

The next method of calculating the number of descendants of a machine involves defining a generating function which is equivalent to the recursive formulas presented in Definitions 4.19 and 4.20. A generating function g is a single quantity which represents an entire sequence of values such as

$$g(z) = a_0 + a_1 z + a_2 z^2 + \dots = \sum_{n \geq 0} a_n z^n$$

By evaluating the function g for successive values of z, G is said to generate the sequence. If such a function exists for the formulas of Definitions 4.19 and 4.20, it would allow the calculation of the number of descendants of a machine without the exhaustive development of the intervening machines.

One example of a generating function is the polynomial identity

$$\frac{1 - x^{n+1}}{1 - x} = 1 + x + x^2 + \ldots + x^n$$

This was used earlier to calculate the total number of machines in the search space of the learning system F.

Another application of generating functions is in the solution of recurrence relations. Suppose we are given the following recurrence relation $a_n = a_{n-1} + a_{n-2}$ with the initial conditions $a_0 = 0$, $a_1 = 1$. We want to find a generating function of the form

$$g(x) = a_0 + a_1 x + \ldots + a_n x^n$$

To do this, we substitute the recurrence relation for every term in g(x) except a_0 and a_1, which gives $a_n x^n = a_{n-1} x^n + n x^n$. Summing the terms, this can be rewritten as

$$g(x) - a_0 - a_1 x = \sum_{n=2}^{\infty} a_n x^n$$

$$= \sum_{n=2}^{\infty} (a_{n-1} x^n + a_{n-2} x^n)$$

$$= x \sum_{n=2}^{\infty} a_{n-1} x^{n-1} + x^2 \sum_{n=2}^{\infty} a_{n-2} x^{n-2}$$

$$= x \sum_{m=1}^{\infty} a_m x^m + x^2 \sum_{k=0}^{\infty} a_k x^k$$

$$= x(g(x)-a_0) + x^2g(x)$$

Setting $a_0 = 1$ and $a_1 = 1$ we have

$$g(x)-1-x = x(g(x)-1) + x^2g(x)$$

So

$$g(x)(1-x-x^2) = 1$$

or

$$g(x) = 1/(1-x-x^2)$$

To locate a generating function for calculating the number of descendants of a machine involves solving the following recurrence relations

$$n_i(1,t_2,t_3,...,t_j) = (p-t_2+1)qN_{i-1}(t_2-1,t_3,...,t_j)$$

where

$1 \leq j \leq i$

$t_2, t_3,...,t_j > 0$

$N_0(0) = 1$

and

$$N_i(t_1,t_2,...,t_j) = \sum_{K=1}^{j} (p-t_k+1)qN_{i-1}(t_K-1,t_{k+1},...,t_{k-1})$$

where the addition of indices is carried out modulus j

$t_1,t_2,...,t_j > 0$

$1 \leq j \leq i$

These recurrence relations do not represent a well behaved series such as $1 + x + x^2 + \ldots + x^n$ but instead are dependant on the configurations of the machines at each level of the tree. It is not possible to calculate the number of machines at level pr without knowing the configurations of the machines at level pr-1. An attempt was made to evaluate these functions for several levels of the recursion in the hope that some type of behavior could be observed which would better lend itself to the discovery of an equivalent generating function. The following shows the result of expanding the recurrence relations to 4 levels. The notation used is that presented in Definition 4.11, N represents the total number of machines in the tree developed thus far.

Level 0 $N = N_0(0)$

Level 1 $N = N_1(1) + N_1(0,1)$

Level 2 $N = N_2(2) + N_2(0,2) + N_2(1,1) + N_2(0,1,1)$

Level 3 $N = N_3(3) + N_3(0,3) + N_3(0,1,2) + N_3(0,2,1) + N_3(1,1,1) + N_3(0,1,1,1)$

Evaluating each of the above using the cardinalities of the input and output sets p and q yields

Level 0 $N = 1$ (by definition)

Level 1 $N = pq + pq$

Level 2 $N = (p-1)q(pq) + (p-1)q(pq) + 2pq(pq) + pq(pq)$

$= 5(pq)^2 - 2qpq$

Level 3 $N = (p-2)q[(p-1)qpq] + (p-1)q[(p-1)qpq] +$

$pq(p-1)qpq + (p-1)q2pqpq +$

$3pqpqpq + pqpqpq$

$= pq(q^2(p-1)(5p-3)) + 4(pq)^3$

It is fairly obvious that determining a generating function for calculating the number of machines at a given level of the tree is far from trivial. Research in this direction was abandoned.

A simpler approach does exist which has neither the complexity of attempting to determine a generating function nor the time and space required to perform an exhaustive evaluation of the number of descendants of a machine. It is this approach which will be applied. Rather than trying to find a perfect estimator of the total number of descendants, we will content ourselves with a local estimator which gives an indication of which machine is likely to have the most descendants even though the exact number of descendants is not known. This estimator will calculate the number of descendants of a machine at the next level of the tree only. The number of possible simple extensions of a machine for a given (i,o) pair is readily available. One needs to know only the following:

1) The number of states of the machine

2) The bound on the number of states

3) The transitions from the remembered state

By computing only the number of descendants at the next level, the heuristic becomes a worse estimator of the value of a path but should still provide a fairly reasonable estimate. The more descendants a machine at level i has at level i+1, the more descendants it is likely to have at level i+2, i+3, ... pr. Assuming the next (i,o) pair is known, determine if the (i,o) pair is permissible. If not, the machine is pruned from the tree. If the (i,o) pair is permissible, then the number of descendants of a machine M is calculated as follows:

$$ND_M = j + k$$

where

ND_M = Number of descendants of machine M

j = The number of states of the machine M

k = 0 if r-j = 0

1 if r-j > 0

where r is the bound on the number of states

The cardinality of the input and output sets is not a factor because the (i,o) pair is known and the new transitions will have only one possible labelling, namely (i,o).

The first term, j, calculates the number of machines generated through the application of Rule 1. A new transition, with the label (i,o), is added to each of the j states of the machine M. The second term, k, calculates the number of machines generated through the application of Rule 2. If the number of states j of the machine M has not yet reached the bound on the number of states possible, one machine is generated. If the bound has been reached, no machines are generated.

It is expected that the application of an informed depth first search using the greatest number of descendants heuristic will improve the performance of the learning system and increase the size of the class of machines M(p,q,r) which F can learn for a given implementation.

CHAPTER VII

IMPLEMENTATION

The learning system F is implemented using both the breadth first development of the tree and the informed depth first search. Two variations of the informed depth first search are included to test different methods for the resolution of ties between machines which have the same number of descendants. One form of tie resolution chooses the leftmost descendant of a machine and the other chooses the rightmost descendant. In addition, a depth first search which uses no heuristic is included to measure the validity of the greatest number of descendants heuristic. The implementation is written in Pascal and run on a Digital Equipment Corporation PDP 11/70. Several criterion are used to evaluate the relative performance of each method.

1) The execution time required to identify the goal machine is calculated for each method. The test runs were made in a controlled environment (single user) to ensure the integrity across multiple runs (by eliminating distortions in execution time caused by other users of the machine). Even though the test runs were made in a stable environment, the runtime statistics can only provide a rough evaluation of the performance. The program must still compete with system activities. Some

distortion can be caused by the time required for access of disk storage. Also, the elapsed runtime is expressed in ticks (60 ticks = 1 second) which is a rather broad estimator considering the number of machine instructions which may be executed in 1 tick.

2) The total number of machines generated is calculated. This indicates the amount of search required and is directly proportional to the time required to identify a machine.

3) The largest number of machines in existence at one time is computed. This gives a measure of the space requirements of each method.

4) The number of machines which are deleted, or "forgotten" by the learning system is also computed. This indicates the relative merit of the depth first technique. Comparison of this value between the depth first methods indicates a measure of the validity of the GND heuristic.

Some special features of the implementation are as follows:

1) The entire input/output sequence is read in and stored internally to the program before any of the methods are run. In a real application, all of this information might not be available at the beginning of the experiment. However, for the purpose of evaluating the different search techniques, this is valuable to

improve the accuracy of the comparison. The cost of accepting the input/output sequence is constant across all three methods.

2) No error checking is performed. Only when a condition is relevant to the logic flow of the program is the value of a variable checked. This is done to keep the size of the code to a minimum. On PDP 11 architecture, a task image (the executable program image) has a limited size. Both the instruction space and the data space of the program reside within the task image. This provides a boundary on the amount of space available for the development of the tree of conjectural machines. There are ways of obtaining a larger data space but these were not pursued. For this research, the space within the task image is sufficient for measuring the relative space needs of the different search techniques.

The following summarizes the results of the implementation. For a complete listing of the program code used in the implementation, the reader is referred to the Appendix.

Most of the analysis was performed on the class of machines M(3,3,3). For machines with 2 or fewer states, the search space is so small that the performance of all of the methods is indistinguishable and the runtime statistics are inconclusive. For machines with 4 or more states, the problem space becomes too large for the development of valid test data without the use of an implementation for the teaching portion of the experimenter. The size of the search space for machines with 3 states is sufficiently large to test the merits of the different search methods.

Figure 18 provides a summary of the statistics of four test runs.

	Run1 time	Run2 time	Run3 time	Run4 time	Largest # mach.	Total space (bytes)
BFS	45	45	45	45	266	25872
DFS	48	40	23	12	21	2058
IDFS (left)	50	41	24	12	18	1764
(right)	50	51	14	26	18	1764

Figure 18. Summary of Test Runs

Results

All three of the depth first methods provide a significant improvement over the breadth first method in the amount of space required to identify the goal machine. As shown in Figure 18, the breadth first method requires an order of magnitude more space than the depth first methods. For this implementation, the largest class of machines which could be learned using a breadth first search was M(3,3,3). Using a depth first search, the largest class of machines which could be learned was expanded to M(6,6,6). The space needs of all of the depth first methods are essentially equivalent, requiring only enough space to develop one branch of the tree at a time.

As expected, the time required to develop the search space using the breadth first method is usually the largest. However, the amount of time required by the depth first methods is not consistently less than that of the breadth first search. This is because the number of machines which must be developed is highly dependant on the location of the goal machine within the tree. When the goal machine resides in the first portion of the tree to be searched, the time required is reduced by a significant amount. However, if the goal machine is in the last portion of the tree to be searched, the time required increases and approaches that of the breadth first search. This performance supports the anticipated behavior.

The informed depth first search technique does not provide the expected improvement over the uninformed depth first search. Restricting the scope of the evaluation function to a local one has a limiting effect on the merit of the evaluation function. However, this is not necessarily the only reason for the observed behavior of the informed depth first search. To fully understand the limited benefit of the GND heuristic, it is necessary to study the growth pattern of the tree of candidate machines. The tree is found to grow in a very symmetric fashion. After level r, most of the conjectural mahines contain the maximum number of states. Thus, the evaluation function returns the same value for all machines in that level. This reduces the choice of the next machine to develop to an arbitrary one, the same as in the uninformed search. Also, the machines at any level tend to be pruned from the tree, as a result of nonpermissible (i,o) pairs, in a symmetrical fashion. This further reduces the distinction of the number of descendants of a machine.

The most important decision as to which machine to search first occurs at level 1 of the tree. The root node has only two descendants, regardless of the size of the class of machines to be learned. One machine is generated through the application of Rule 1 and one through the application of Rule 2. If the "correct" half of the tree, the one containing the goal machine, is chosen first at level 1, the time required to locate the goal machine is greatly reduced. If not, then the search proceeds to develop the half of the tree which does not contain the goal machine before returning to develop the half of the tree which does contains the solution. Figure 19 shows a sample execution of F when the machine to be identified lies in the first half of the tree searched. Figure 20 shows a sample execution when the goal machine is in the second half of the tree searched.

```
For the class of machines M(3,3,3)
Using heuristic 1 : Leftmost son with greatest value of E
With an input/output sequence of: (a0)(b0)(a1)(b1)(c0)
(a2)(c2)(c1)(b2)(c1)(a0)(a2)(a1)(c2)(b0)(b1)(b2)(b0)

SEARCH METHOD: Breadth First Search
Elapsed Run Time:    46 ticks
Total Machines Developed: 467
Largest Number of Machines at One Time 264
At 98 bytes per machine, total space = 25872 bytes
Number of machines deleted 310

MACHINE 1118

           a     b     c
        -----------------
 1        2,0   2,2   2,0
 2        3,2   3,0   1,1
 3        3,1   1,1   2,2

SEARCH METHOD: Informed Depth First Search
Elapsed Run Time:    13 ticks
Total Machines Developed: 111
Largest Number of Machines at One Time 18
At 98 bytes per machine, total space = 1764 bytes
Number of machines deleted 97

MACHINE 321

           a     b     c
        -----------------
 1        2,0   2,2   2,0
 2        3,2   3,0   1,1
 3        3,1   1,1   2,2

SEARCH METHOD: Depth First Search
Elapsed Run Time:    12 ticks
Total Machines Developed: 111
Largest Number of Machines at One Time 21
At 98 bytes per machine, total space = 2058 bytes
Number of machines deleted 91

MACHINE 321

           a     b     c
        -----------------
 1        2,0   2,2   2,0
 2        3,2   3,0   1,1
 3        3,1   1,1   2,2
```

Figure 19. Sample Execution 1 of F

```
For the class of machines M(3,3,3)
Using heuristic 1 : Leftmost son with greatest value of E
With an input/output sequence of: (a0)(b0)(a1)(b1)(c0)
(a2)(c2)(c1)(b2)(c1)(a0)(a2)(a1)(c2)(b0)(b1)(b2)(b0)

SEARCH METHOD: Breadth First Search
Elapsed Run Time:    45 ticks
Total Machines Developed: 467
Largest Number of Machines at One Time 264
At 98 bytes per machine, total space = 25872 bytes
Number of machines deleted 310

MACHINE 1349

          a    b    c
        ---------------
  1      1,0  2,0  3,0
  2      2,1  1,1  3,1
  3      3,2  1,2  2,2

SEARCH METHOD: Informed Depth First Search
Elapsed Run Time:    50 ticks
Total Machines Developed: 443
Largest Number of Machines at One Time 18
At 98 bytes per machine, total space = 1764 bytes
Number of machines deleted 432

MACHINE 1297

          a    b    c
        ---------------
  1      1,0  2,0  3,0
  2      2,1  1,1  3,1
  3      3,2  1,2  2,2

SEARCH METHOD: Depth First Search
Elapsed Run Time:    49 ticks
Total Machines Developed: 443
Largest Number of Machines at One Time 21
At 98 bytes per machine, total space = 2058 bytes
Number of machines deleted 426

MACHINE 1297

          a    b    c
        ---------------
  1      1,0  2,0  3,0
  2      2,1  1,1  3,1
  3      3,2  1,2  2,2
```

Figure 20. Sample Execution 2 of F

Where the informed and uninformed depth first searches both choose the same path, the execution time of the informed method is slightly higher due to the additional processing required in the calculation of the evaluation function. This increase in time however is minimal and does not represent a serious limitation of the informed method.

In summary, the depth first strategy for developing the tree of candidate machines provides a significant improvement in the amount of space required to learn a finite state machine from a class of finite state machines. The time required for a depth first search is usually no more than that of a breadth first search and can be substantially less.

The GND heuristic, at least the local version used in this implementation, does not provide a significant improvement over the arbitrary choice of which machine to develop made by the uninformed depth first search. The application of a GND heuristic with a larger scope may provide better results but the symmetry of the tree of candidate machines will restrict any such heuristic.

CHAPTER VIII

SUMMARY AND AREAS OF FURTHER RESEARCH

This research investigates an improvement of the performance of a learning system for finite state machines. Through the application of a depth first approach to the development of the search space, the size of the class of machines which can be learned, for a given implementation, is doubled. An heuristic estimator is used to guide the depth first search in its choice of which machine to expand at each stage of the process. The depth first approach is found to provide a significant savings in the amount of space required to learn a finite state machine from a class of finite state machines. The heuristic estimator can provide an improvement in the amount of time necessary to identify the goal machine depending on the location of the goal machine in the search space.

There are several areas of research that this thesis does not address or that are an extension of the work done here.

1) Can a method of calculating the total number of descendants of a machine be found which does not require an exhaustive development of all subsequent machines yet may be computed in a reasonable amount of time, thus making the GND heuristic a more valuable estimator ?

2) Is there a better heuristic than the greatest number of descendants which could be used to determine which of the conjectural machines to develop first ?

3) Can an heuristic be defined which correctly determines at the first level of the tree, which half of the tree contains the goal machine ?, If so, the unnecessary search performed when the GND heuristic chooses incorrectly could be eliminated. A pruning heuristic which could evaluate a conjectural machine and determine that it cannot possibly lead to a goal machine is one such approach. One suggestion along this line of reasoning is to evaluate the entire input/output sequence before developing any conjectural machines to identify the characteristics which lead to a contradiction. If a machine exhibiting these characteristics is detected, there would be no need to develop it further. An example of this is that if the same input value occurs consecutively in the input/output sequence with different output labels, then the machine which accepts the string must have at least two states and the transition upon receipt of the first (i,o) pair must lead to a different state. For an illustration of this, consider Figures 21 and 22. For the string (a0)(a1), the machine in Figure 21 is ruled out while the machine in Figure 22 is still a feasible candidate for further development.

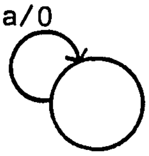

Figure 21. Implausible machine for (a0)(a1)

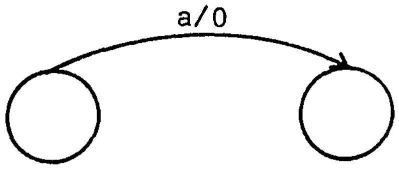

Figure 22. Plausible Machine for (a0)(a1)

An heuristic of this type is similar to the failure memory process developed by Biermann, Baum and Petry (1975) in their work on program synthesis.

4) Rather than improving the search technique employed, can the structure of the semilattice be redefined by some other structure which would reduce the amount of search needed.

APPENDIX

```
PROGRAM LEARN (INPUT, OUTPUT, INFILE, OUTFILE);

(* This program is an implementation of the learning system
   F as defined by Dionysios Kountanis. The breadth first
  development of the tree of candidate machines is performed
  and then two different depth first methods are used to
  develop the tree of candidate machines for a performance
  comparison.                                              *)

TYPE
     IO_VALS = ARRAY [1..7] OF CHAR;

CONST
     MAXP = 4;            (* Maximum cardinality of input set *)
     MAXQ = 4;           (* Maximum cardinality of output set *)
     MAXR = 4;     (* Maximum number of states for M(p,q,r) *)
     MAXIO = 32;    (* Maximum length of I/O sequence 2(pr) *)
     MAX_LEV = 32;  (* Maximum level in tree of machines pr *)
     ZERO = 0;
     INPUT_VALS = IO_VALS('a', 'b', 'c', 'd', 'e', 'f', 'g');
     OUTPUT_VALS = IO_VALS('0', '1', '2', '3', '4', '5', '6');
     BLANK = ' ';
     NOT_COMP = -1;


TYPE
    IO_PAIR = RECORD
              INPUT : CHAR;
              OUTPUT : CHAR;
              END;
    MACH_PTR = ^MACHINE;
    MACHINE = RECORD
              NUM_SMPL_EXT : INTEGER;
              NUM_STATES : INTEGER;
              NUM_DESC : INTEGER;
              REMEM_ST : INTEGER;
              INIT_ST : INTEGER;
              PARENT : MACH_PTR;
              RIGHT_NBR : MACH_PTR;
              LEFT_NBR : MACH_PTR;
              DEVELOPED : BOOLEAN;
              NUM_TRAN : ARRAY [1..MAXR] OF INTEGER;
              TRANS_STATE : ARRAY [1..MAXR,1..MAXP] OF INTEGER;
              TRANS_LABEL : ARRAY [1..MAXR,1..MAXP] OF IO_PAIR;
              DESCEN : ARRAY [1..MAXR] OF MACH_PTR;
              END;
    SEARCH_METHOD = (BFS, IDFS, DFS);
    DATA_FILE = TEXT;
    TIM_ARR = ARRAY [1..8] OF INTEGER;
```

```
VAR
    INFILE, OUTFILE : DATA_FILE;
    INFILE_NAME, OUTFILE_NAME : PACKED ARRAY [1..20] OF CHAR;
    IO_LIST : ARRAY [1..MAXIO] OF IO_PAIR;
    CUR_LEVEL, MAX_LEVEL : INTEGER;
    P, Q, R : INTEGER;
    IO_LEN : INTEGER;
    TOT_MACH, LARG_MACH, CUR_MACH, DEL_MACH : INTEGER;
    RUN_TIME_STRT, RUN_TIME_END : INTEGER;
    NEXT_INIT : INTEGER;
    LIST_HEAD : ARRAY [ZERO..MAX_LEV] OF MACH_PTR;
    LIST_TAIL : ARRAY [ZERO..MAX_LEV] OF MACH_PTR;
    ROOT : MACH_PTR;
    SEARCH_TYPE : SEARCH_METHOD;
    GOAL_MACH : MACH_PTR;
    MORE_TO_DO : BOOLEAN;
    S_TIME : TIM_ARR;
    HEUR_TYPE : INTEGER;
    SRCH_STRT, SRCH_END : INTEGER;
    SEARCHES : ARRAY [1..3] OF SEARCH_METHOD;


(* This procedure will read in the configuration for the
   class of machines M(p,q,r) which is to be learned.      *)

PROCEDURE GET_CONF;

BEGIN

  SEARCHES[1] := BFS;
  SEARCHES[2] := IDFS;
  SEARCHES[3] := DFS;
  WRITE ('Input file: ');
  READLN (INFILE_NAME);
  IF (INFILE_NAME = 'EX                  ') THEN
    MORE_TO_DO := FALSE
  ELSE
    BEGIN
      WRITE ('Output file: ');
      READLN (OUTFILE_NAME);
      WRITE ('Starting search method (1-3): ');
      READLN (SRCH_STRT);
      WRITE ('Ending search method (1-3): ');
      READLN (SRCH_END);
      WRITE ('Type of GND heuristic (1-3): ');
      READLN (HEUR_TYPE);
      WRITELN;

      RESET (INFILE, INFILE_NAME);
      REWRITE (OUTFILE, OUTFILE_NAME);
      READLN (INFILE, P);  (* Get cardinality of input set *)
      READLN (INFILE, Q);  (* Get cardinality of output set *)
      READLN (INFILE, R);  (* Get maximum number of states *)
      MAX_LEVEL := P * R;

    END;

END; (* GET_CONF *)


(* This procedure will read in the entire sequence of
   input,output pairs representing the behavior of the
   machine to be identified. This data is stored in an
   internal array IO_LIST to be used by the different tree
   development methods. Although this data may not be
   available at the beginning of an experiment in a real
   experiment, is made available for this implementation
   to minimize the run time discrepancies of the different
   methods   *)

PROCEDURE GET_IO_SEQ;

VAR
    I : INTEGER;

BEGIN
  I := 1;
  WHILE NOT EOF(INFILE) DO
    BEGIN
      READLN (INFILE, IO_LIST[I].INPUT, IO_LIST[I].OUTPUT);
      I := I + 1;
    END;
  IO_LEN := I - 1;
END; (*GET_IO_SEQ *)


(* This procedure will print the configuration of the class
   of machines for which a candidate machine is being learned.
   The behavior of the machine in the black box, represented
   by a sequence of (i,o) values which serve as inputs to the
   learning system are also printed.                        *)

PROCEDURE PRINT_CONFIG;

VAR
    I, OUT_COUNT : INTEGER;

BEGIN
  PAGE (OUTFILE);
  WRITELN (OUTFILE, 'For the class of machines M(', P:1, ',',
           Q:1, ',', R:1, ')');
  WRITELN (OUTFILE, 'Input file: ', INFILE_NAME,
           '  Output file: ', OUTFILE_NAME);
  WRITELN (OUTFILE, 'Using heuristic ', HEUR_TYPE:1);
  CASE HEUR_TYPE OF
    1 : WRITELN (OUTFILE,
                 'Leftmost son with greatest value of E');
    2 : WRITELN (OUTFILE,
                 'Rightmost son with greatest value of E');
    3 : WRITELN (OUTFILE,
             'Greatest value of E with least transitions from',
             ' remembered state');
    END;
  WRITE (OUTFILE, 'With an Input,Output Sequence: ');
  OUT_COUNT := 31;               (* Length of above literal *)
  FOR I := 1 TO IO_LEN DO
    BEGIN                          (* Just for pretty print *)
      WRITE (OUTFILE, '(', IO_LIST[I].INPUT:1, ',',
                     IO_LIST[I].OUTPUT:1, ')');
      OUT_COUNT := OUT_COUNT + 5;
      IF (OUT_COUNT > 70) THEN
        BEGIN
          OUT_COUNT := ZERO;
          WRITELN (OUTFILE);
        END;
    END;
  WRITELN (OUTFILE);
END; (* PRINT_CONFIG *)


(* This procedure prints the statistics for a tree
   development method   *)

PROCEDURE PRINT_STATS;

VAR
    I, MACH_SZ : INTEGER;

BEGIN
  WRITELN (OUTFILE);
  WRITELN (OUTFILE);
  WRITELN (OUTFILE);
  WRITE (OUTFILE, 'SEARCH METHOD: ');
  CASE SEARCH_TYPE OF
    BFS  : WRITELN (OUTFILE, 'Breadth First Search');
    IDFS : WRITELN (OUTFILE, 'Informed Depth First Search');
    DFS  : WRITELN (OUTFILE, 'Depth First Search');
    END;
  WRITELN (OUTFILE, 'Elapsed run time:   ',
                    RUN_TIME_END-RUN_TIME_STRT:3, ' ticks');
  WRITELN (OUTFILE, 'Total Machines Developed', TOT_MACH:4);
  WRITELN (OUTFILE, 'Largest Number of Machines at One Time',
                     LARG_MACH:4);
  MACH_SZ := SIZE (MACHINE);
  WRITELN (OUTFILE, 'At ', MACH_SZ:3, ' bytes per machine',
           ', required a total', ' space of ',
           MACH_SZ*LARG_MACH:5, ' bytes');
  WRITELN (OUTFILE, 'Number of Machines Deleted  ',
           DEL_MACH:4);
END; (* PRINT_STATS *)



(* This procedure will print out a machine                    *)

PROCEDURE PRINT_MACHINE (MACH : MACH_PTR);

VAR
    I, J, K, L : INTEGER;
    PT_VAL : CHAR;

BEGIN
  WRITELN (OUTFILE);
  WRITELN (OUTFILE, 'MACHINE   ', MACH^.INIT_ST:4);
  WRITELN (OUTFILE);
  WRITE (OUTFILE, '        ');
  FOR I := 1 TO P DO
    WRITE (OUTFILE, '   ', INPUT_VALS[I]:1, ' ');
  WRITELN (OUTFILE);
  WRITE (OUTFILE, '        ');
  FOR I := 1 TO P DO
    WRITE (OUTFILE, '_____');
  WRITELN (OUTFILE);
  FOR I := 1 TO MACH^.NUM_STATES DO
    BEGIN
      WRITE (OUTFILE, '    ', I:1, '   ');
      FOR J := 1 TO P DO
        BEGIN
          PT_VAL := BLANK;
          K := ZERO;
          FOR L := 1 TO MACH^.NUM_TRAN[I] DO
            IF (MACH^.TRANS_LABEL[I,L].INPUT = INPUT_VALS[J])
            THEN
              K := L;
          IF (K > ZERO) THEN
            PT_VAL := MACH^.TRANS_LABEL[I,K].OUTPUT;
          IF (PT_VAL = BLANK) THEN
            WRITE (OUTFILE, '       ')
          ELSE
            WRITE (OUTFILE, '  ', MACH^.TRANS_STATE[I,K]:1,
                  ',', MACH^.TRANS_LABEL[I,K].OUTPUT:1);
        END;
      WRITELN (OUTFILE);
    END;
END; (* PRINT_MACHINE *)


(* This is a system defined procedure which returns the
   system time.    *)

PROCEDURE GETTIME (VAR S_TIME : TIM_ARR);
  NONPASCAL;


(* This function will construct a time value from the ticks
   and seconds portion of the system time for use in
   determining the elapsed run time of each of the search
   methods.                                           *)

FUNCTION MAK_TIME( S_TIME : TIM_ARR):INTEGER;

BEGIN
  GETTIME (S_TIME);
  MAK_TIME := (S_TIME[6]*60) + (S_TIME[7]);
END;


(* This procedure will return a pointer to the goal machine. *)

PROCEDURE FIND_GOAL (VAR MACH : MACH_PTR);

BEGIN
  MACH := LIST_HEAD[MAX_LEVEL];
END; (* FIND_GOAL *)


(* This procedure will clean up the tree structure developed.
   All space used by machines of the tree is released.      *)

PROCEDURE CLEANUP (VAR MACH : MACH_PTR);

VAR
   I : INTEGER;
   T_MACH : MACH_PTR;

BEGIN
  IF (MACH^.NUM_DESC = ZERO) THEN
    DISPOSE (MACH)
  ELSE
    BEGIN
      FOR I := 1 TO MACH^.NUM_DESC DO
```

```
        CLEANUP (MACH^.DESCEN[I]);
      DISPOSE (MACH);
    END;
END; (* CLEANUP *)


(* This procedure will add a machine to the list of machines
   for the level specified. A linked list of machines is kept
   for each level of the tree to allow easy access to the
   machines at that level without having to trace from the
   root of the tree to locate a particular machine.      *)

PROCEDURE ADD_TO_LIST (MACH : MACH_PTR; LEVEL : INTEGER);

BEGIN
  IF (LIST_HEAD[LEVEL] = NIL) THEN
    BEGIN
      LIST_HEAD[LEVEL] := MACH;
      LIST_TAIL[LEVEL] := MACH;
    END
  ELSE
    BEGIN
      MACH^.LEFT_NBR := LIST_TAIL[LEVEL];
      LIST_TAIL[LEVEL]^.RIGHT_NBR := MACH;
      LIST_TAIL[LEVEL] := MACH;
    END;
END; (* ADD_TO_LIST *)


(* This procedure will remove a machine from the list of
   machines in the specified level of the tree. A linked
   list of machines is kept for each level of the tree to
   allow quick access to a machine in a specified level of
   the tree.                                              *)

PROCEDURE REM_FRM_LIST (MACH : MACH_PTR; LEVEL : INTEGER);

VAR
    T_MACH : MACH_PTR;
    HEAD : BOOLEAN;

BEGIN
  HEAD := FALSE;
  IF (LIST_HEAD[LEVEL] = MACH) THEN
    BEGIN
      LIST_HEAD[LEVEL] := MACH^.RIGHT_NBR;
      IF (LIST_HEAD[LEVEL] <> NIL) THEN
        LIST_HEAD[LEVEL]^.LEFT_NBR := NIL;
      HEAD := TRUE;
    END;
  IF (LIST_TAIL[LEVEL] = MACH) THEN
    BEGIN
      LIST_TAIL[LEVEL] := MACH^.LEFT_NBR;
      IF (LIST_TAIL[LEVEL] <> NIL) THEN
        LIST_TAIL[LEVEL]^.RIGHT_NBR := NIL;
    END
  ELSE
    IF NOT HEAD THEN
      BEGIN
        T_MACH := LIST_HEAD[LEVEL];
        WHILE (T_MACH <> MACH) DO
          T_MACH := T_MACH^.RIGHT_NBR;
        T_MACH^.LEFT_NBR^.RIGHT_NBR := T_MACH^.RIGHT_NBR;
        T_MACH^.RIGHT_NBR^.LEFT_NBR := T_MACH^.LEFT_NBR;
      END;
  MACH^.RIGHT_NBR := NIL;
  MACH^.LEFT_NBR := NIL;
END; (* REM_FRM_LIST *)


(* This procedure will create a new machine and initialize
   all fields   *)

PROCEDURE CREATE_MACH (VAR MACH : MACH_PTR);

VAR
    I, J : INTEGER;

BEGIN
  NEW (MACH);
  WITH MACH^ DO
    BEGIN
      NUM_SMPL_EXT := NOT_COMP;
      NUM_STATES := ZERO;
      NUM_DESC := ZERO;
      REMEM_ST := ZERO;
      INIT_ST := ZERO;
      PARENT := NIL;
      RIGHT_NBR := NIL;
      LEFT_NBR := NIL;
      DEVELOPED := FALSE;
      FOR I := 1 TO R DO
        BEGIN
          NUM_TRAN[I] := ZERO;
          DESCEN[I] := NIL;
          FOR J := 1 TO P DO
            BEGIN
              TRANS_STATE[I,J] := ZERO;
              TRANS_LABEL[I,J].INPUT := BLANK;
              TRANS_LABEL[I,J].OUTPUT := BLANK;
            END;
        END;
    END;
END; (* CREATE_MACH *)


(* This procedure initializes the run time statistics before
   the invocation of each method.  The tree of candidate
   machines is initialized to contain the root machine      *)

PROCEDURE INIT_STATS;

VAR
    I : INTEGER;

BEGIN
  DEL_MACH := ZERO;
  FOR I := ZERO TO MAX_LEV DO
    BEGIN
      LIST_HEAD[I] := NIL;
      LIST_TAIL[I] := NIL;
    END;

  (* Create the root of the tree *)

  CUR_LEVEL := ZERO;
  CREATE_MACH (ROOT);
  ROOT^.INIT_ST := 1;
  NEXT_INIT := 2;
  ROOT^.REMEM_ST := 1;
  ROOT^.NUM_STATES := 1;
  TOT_MACH := 1;
  CUR_MACH := 1;
  LARG_MACH := 1;

  (* Add root to list of machines at current level *)
  ADD_TO_LIST (ROOT, CUR_LEVEL);
END; (* INIT_STATS *)


(* This procedure will maintain the counts of total machines
   in existence and the largest number of machines so far.
   (Also the current number of machines in existence).     *)

PROCEDURE INC_COUNTS;

BEGIN
  TOT_MACH := TOT_MACH + 1;
  CUR_MACH := CUR_MACH + 1;
  IF (CUR_MACH > LARG_MACH) THEN
```

```
    LARG_MACH := CUR_MACH;
END; (* INC_COUNTS *)


(* This procedure will set up control to be at the next level
   of the tree. Current level is not allowed to exceed the
   maximum level.            *)

PROCEDURE NEXT_LEVEL (VAR IO_PTR : INTEGER);

BEGIN
  IF (CUR_LEVEL < MAX_LEVEL) THEN
    CUR_LEVEL := CUR_LEVEL + 1;
  IO_PTR := IO_PTR + 1;
END; (* NEXT_LEVEL *)


(* This procedure will remove a descendant machine from the
   parent machine. *)

PROCEDURE DEL_SON (PARNT, SON : MACH_PTR);

VAR
    I, J : INTEGER;

BEGIN
  I := ZERO;
  REPEAT
    I := I + 1;
  UNTIL (PARNT^.DESCEN[I] = SON);
  FOR J := I TO PARNT^.NUM_DESC-1 DO
    PARNT^.DESCEN[J] := PARNT^.DESCEN[J+1];
  PARNT^.DESCEN[PARNT^.NUM_DESC] := NIL;
  PARNT^.NUM_DESC := PARNT^.NUM_DESC - 1;
END; (* DEL_SON *)


(* This procedure will delete a machine from the tree. A
   fatal error is declared if the machine to be deleted has
   any descendants.            *)

PROCEDURE DELETE_MACH (VAR MACH : MACH_PTR);

VAR
    PARNT : MACH_PTR;

BEGIN
  PARNT := MACH^.PARENT;
  DEL_SON (PARNT, MACH);
  DISPOSE (MACH);
  DEL_MACH := DEL_MACH + 1;
  CUR_MACH := CUR_MACH - 1;
END; (* DELETE_MACH *)


(* This procedure will add a transition from the remembered
   state of the machine to the specified state with a label
   using the given input,output values.                   *)

PROCEDURE ADD_TRAN (MACH : MACH_PTR; INPUT, OUTPUT : CHAR;
                    NEW_ST : INTEGER);

VAR
    NUM_T : INTEGER;

BEGIN
  WITH MACH^ DO
    BEGIN
      NUM_T := NUM_TRAN[REMEM_ST] + 1;
      NUM_TRAN[REMEM_ST] := NUM_T;
      TRANS_LABEL[REMEM_ST,NUM_T].INPUT := INPUT;
      TRANS_LABEL[REMEM_ST,NUM_T].OUTPUT := OUTPUT;
      TRANS_STATE[REMEM_ST,NUM_T] := NEW_ST;
      REMEM_ST := NEW_ST;
    END;
END; (* ADD_TRAN *)


(* This procedure will copy one machine to another. Only the
   configuration of the machine is copied and not the
   interrelationships between machines in the tree structure. *)

PROCEDURE COPY_MACH (OLDM : MACH_PTR; VAR NEWM : MACH_PTR);

VAR
    I, J, NUM : INTEGER;

BEGIN
  WITH NEWM^ DO
    BEGIN
      NUM_STATES := OLDM^.NUM_STATES;
      PARENT := OLDM;
      REMEM_ST := OLDM^.REMEM_ST;
      INIT_ST := NEXT_INIT;
      NEXT_INIT := NEXT_INIT + NUM_STATES;
      FOR I := 1 TO NUM_STATES DO
        BEGIN
          NUM_TRAN[I] := OLDM^.NUM_TRAN[I];
          FOR J := 1 TO NUM_TRAN[I] DO
```

```
            BEGIN
              TRANS_STATE[I,J] := OLDM^.TRANS_STATE[I,J];
              TRANS_LABEL[I,J].INPUT :=
                                 OLDM^.TRANS_LABEL[I,J].INPUT;
              TRANS_LABEL[I,J].OUTPUT :=
                                OLDM^.TRANS_LABEL[I,J].OUTPUT;
            END;
        END;
    END;
  NUM := OLDM^.NUM_DESC + 1;
  OLDM^.NUM_DESC := NUM;
  OLDM^.DESCEN[NUM] := NEWM;
END; (* COPY_MACHINE *)


(* This procedure applies the static evaluation function E
   to a machine to calculate the number of descendants a
   machine can have by creating simple extensions of the
   machine.                                      *)

PROCEDURE EVALUATE (MACH : MACH_PTR);

VAR
    E, L : INTEGER;

BEGIN
  E := MACH^.NUM_STATES;
  IF (E < R) THEN
    E := E + 1;
  MACH^.NUM_SMPL_EXT := E;
END; (* EVALUATE *)


(* This function will determine if the specified (i,o) pair
   is permissible for the given machine in its remembered
   state. If a transition already exists from the remembered
   state with the (i,o) pair for a label, this fact is
   returned along with the state specified by the transition. *)

FUNCTION PERMISS (MACH : MACH_PTR; INPUT, OUTPUT : CHAR;
                  VAR TRAN_ALREADY : BOOLEAN;
                  VAR NEW_ST : INTEGER):BOOLEAN;

VAR
    R_ST, I, NUM_TRAN : INTEGER;
    STOP : BOOLEAN;

BEGIN
  PERMISS := TRUE;
  TRAN_ALREADY := FALSE;
  NEW_ST := ZERO;
  R_ST := MACH^.REMEM_ST;
  NUM_TRAN := MACH^.NUM_TRAN[R_ST];
  IF (NUM_TRAN > ZERO) THEN
    BEGIN
      STOP := FALSE;
      I := 1;
      WHILE ((NOT STOP) AND (I <= NUM_TRAN)) DO
        BEGIN
          IF (MACH^.TRANS_LABEL[R_ST,I].INPUT = INPUT)
          THEN
            IF (MACH^.TRANS_LABEL[R_ST,I].OUTPUT = OUTPUT)
            THEN
              BEGIN
                NEW_ST := MACH^.TRANS_STATE[R_ST,I];
                TRAN_ALREADY := TRUE;
                STOP := TRUE;
              END
            ELSE
              BEGIN
                PERMISS := FALSE;
                STOP := TRUE;
              END;
          I := I + 1;
        END;
    END;
END; (* PERMISS *)


(* This procedure will is used by the depth first development
   techniques to determine which of the machines at a level
   to develop first. If the informed depth first search
   method is being used, the (leftmost) machine with the
   largest value of the static evaluation function E will be
   returned. If the depth first method is being used, the
   first (leftmost) machine will be returned.             *)

PROCEDURE CHOOSE_MACH (IO_PTR, LEVEL : INTEGER; VAR BEST :
                       MACH_PTR; VAR TRAN : BOOLEAN;
                       VAR N_ST : INTEGER);

VAR
    MACH, T_MACH : MACH_PTR;
    MAX_E, NEW_ST, MIN_T, R_ST : INTEGER;
    TRAN_ALREADY : BOOLEAN;

BEGIN
  BEST := NIL;
  IF (SEARCH_TYPE = IDFS) THEN
    BEGIN
      MAX_E := ZERO;
```

```
     MIN_T := P + 1;
     MACH := LIST_HEAD[LEVEL];
     WHILE (MACH <> NIL) DO
       BEGIN
         T_MACH := MACH^.RIGHT_NBR;
         IF ((MACH^.DEVELOPED) OR (NOT PERMISS
            (MACH, IO_LIST[IO_PTR].INPUT,
            IO_LIST[IO_PTR].OUTPUT, TRAN_ALREADY, NEW_ST)))
         THEN
           BEGIN
             REM_FRM_LIST (MACH, LEVEL);
             DELETE_MACH (MACH);
           END
         ELSE
           BEGIN
             IF (MACH^.NUM_SMPL_EXT = NOT_COMP) THEN
               EVALUATE (MACH);
             CASE HEUR_TYPE OF
               1 : IF (MACH^.NUM_SMPL_EXT > MAX_E) THEN
                     BEGIN
                       MAX_E := MACH^.NUM_SMPL_EXT;
                       BEST := MACH;
                       TRAN := TRAN_ALREADY;
                       N_ST := NEW_ST;
                     END;
               2 : IF (MACH^.NUM_SMPL_EXT >= MAX_E) THEN
                     BEGIN
                       MAX_E := MACH^.NUM_SMPL_EXT;
                       BEST := MACH;
                       TRAN := TRAN_ALREADY;
                       N_ST := NEW_ST;
                     END;
               3 : IF (MACH^.NUM_SMPL_EXT >= MAX_E) THEN
                     BEGIN
                       R_ST := MACH^.REMEM_ST;
                       IF (MACH^.NUM_TRAN[R_ST] < MIN_T)
                       THEN
                         BEGIN
                           MAX_E := MACH^.NUM_SMPL_EXT;
                           BEST := MACH;
                           TRAN := TRAN_ALREADY;
                           N_ST := NEW_ST;
                           MIN_T := MACH^.NUM_TRAN[R_ST];
                         END;
                    END;
              END;
          END;
        MACH := T_MACH;
      END;
  END
ELSE (* Assume DFS *)
  BEGIN
```

```
   MACH := LIST_HEAD[LEVEL];
   WHILE ((MACH <> NIL) AND (BEST = NIL)) DO
    BEGIN
     T_MACH := MACH^.RIGHT_NBR;
     IF ((MACH^.DEVELOPED) OR (NOT PERMISS
       (MACH, IO_LIST[IO_PTR].INPUT,
       IO_LIST[IO_PTR].OUTPUT, TRAN_ALREADY, NEW_ST)))
     THEN
      BEGIN
       REM_FRM_LIST (MACH, LEVEL);
       DELETE_MACH (MACH);
      END
     ELSE
      BEGIN
       BEST := MACH;
       TRAN := TRAN_ALREADY;
       N_ST := NEW_ST;
      END;
     MACH := T_MACH;
    END;
  END;
END; (* CHOOSE_MACH *)


(* This procedure will create a duplicate of one machine at
   the specified level and set the new remembered state to
   the one indicated.          *)

PROCEDURE DUPLICATE (MACH : MACH_PTR; LEVEL, NEW_ST : INTEGER);

VAR
  NEW_MACH : MACH_PTR;

BEGIN
 CREATE_MACH (NEW_MACH);
 COPY_MACH (MACH, NEW_MACH);
 ADD_TO_LIST (NEW_MACH, LEVEL);
 NEW_MACH^.REMEM_ST := NEW_ST;
 INC_COUNTS;
END; (* DUPLICATE *)


(* This procedure will generate all simple extensions of a
   machine through the application of Rule 1. This generates
   all successors which have the same number of states as the
   parent machine and one more transition to one of the other
   states of the machine.                              *)

PROCEDURE RULE1 (VAR MACH : MACH_PTR; IO_PTR, LEVEL : INTEGER);
```

```
VAR
    I : INTEGER;
    NEW_MACH : MACH_PTR;

BEGIN
  FOR I := 1 TO MACH^.NUM_STATES DO
    BEGIN
      CREATE_MACH (NEW_MACH);
      COPY_MACH (MACH, NEW_MACH);
      ADD_TRAN (NEW_MACH, IO_LIST[IO_PTR].INPUT,
           IO_LIST[IO_PTR].OUTPUT,I);
      ADD_TO_LIST (NEW_MACH, LEVEL);
      INC_COUNTS;
    END;
END; (* RULE1 *)


(* This procedure will create the simple extension of a
   machine through the application of Rule 2. This machines
   will have one more state than the parent machine and an
   added transition from the old rmembered state to the newly
   added state.                                          *)

PROCEDURE RULE2 (VAR MACH : MACH_PTR; IO_PTR, LEVEL : INTEGER);

VAR
    NEW_MACH : MACH_PTR;

BEGIN
  IF (MACH^.NUM_STATES < R) THEN
    BEGIN
      CREATE_MACH (NEW_MACH);
      COPY_MACH (MACH, NEW_MACH);
      NEW_MACH^.NUM_STATES := NEW_MACH^.NUM_STATES + 1;
      NEXT_INIT := NEXT_INIT + 1;
      ADD_TRAN (NEW_MACH, IO_LIST[IO_PTR].INPUT,
           IO_LIST[IO_PTR].OUTPUT,
           NEW_MACH^.NUM_STATES);
      ADD_TO_LIST (NEW_MACH, LEVEL);
      INC_COUNTS;
    END;
END; (* RULE2 *)


(* This procedure will develop the tree of candidate machines
   in a breadth first fashion. This is the method defined by
   Kountanis.  The procedure terminates when the entire
   input,output sequence has been processed.  *)

PROCEDURE BREADTH_FIRST;
```

```
VAR
    IO_PTR, NEW_ST : INTEGER;
    T_MACH, MACH : MACH_PTR;
    TRAN_ALREADY : BOOLEAN;

BEGIN
  IO_PTR := 1;
  CUR_LEVEL := ZERO;
  WHILE (IO_PTR <= IO_LEN) DO
    BEGIN
      MACH := LIST_HEAD[CUR_LEVEL];
      WHILE (MACH <> NIL) DO
        BEGIN
          T_MACH := MACH^.RIGHT_NBR;
          IF (PERMISS (MACH, IO_LIST[IO_PTR].INPUT,
                       IO_LIST[IO_PTR].OUTPUT,
                       TRAN_ALREADY, NEW_ST)) THEN
            IF (TRAN_ALREADY) THEN
              IF (CUR_LEVEL < MAX_LEVEL) THEN
                DUPLICATE (MACH, CUR_LEVEL+1, NEW_ST)
              ELSE
                MACH^.REMEM_ST := NEW_ST
            ELSE
              BEGIN
                RULE2 (MACH, IO_PTR, CUR_LEVEL+1);
                RULE1 (MACH, IO_PTR, CUR_LEVEL+1);
              END
          ELSE
            BEGIN
              REM_FRM_LIST (MACH, CUR_LEVEL);
              DELETE_MACH (MACH);
            END;
          MACH := T_MACH;
        END;
      NEXT_LEVEL (IO_PTR);
    END;
END; (* BREADTH_FIRST *)


(* This procedure will perform a depth first development of
   the tree of candidate machines.  If the search method
   being performed is the informed depth first search which
   uses the greatest number of descendants heuristic, a
   static evaluation function is applied to choose which
   machine in the tree to develop next. If the search method
   is a simple depth first search, the "left most" machine
   in the current level is developed first.             *)

PROCEDURE DEPTH_FIRST;
```

```
 VAR
     TRAN_ALREADY, RE_USE : BOOLEAN;
     IO_PTR, NEW_ST : INTEGER;
     MACH : MACH_PTR;

 BEGIN
   CUR_LEVEL := ZERO;
   RE_USE := FALSE;
   IO_PTR := 1;
   WHILE (IO_PTR <= IO_LEN) DO
     BEGIN
       IF NOT RE_USE THEN
         CHOOSE_MACH (IO_PTR, CUR_LEVEL, MACH, TRAN_ALREADY,
                      NEW_ST)
       ELSE
         BEGIN
           IF NOT (PERMISS (MACH, IO_LIST[IO_PTR].INPUT,
                   IO_LIST[IO_PTR].OUTPUT, TRAN_ALREADY,
                   NEW_ST)) THEN
             BEGIN
               REM_FRM_LIST (MACH, CUR_LEVEL);
               DELETE_MACH (MACH);
               RE_USE := FALSE;
             END;
          END;
       IF (MACH <> NIL) THEN
         BEGIN
           IF (TRAN_ALREADY) THEN
             IF (CUR_LEVEL < MAX_LEVEL) THEN
               DUPLICATE (MACH, CUR_LEVEL+1, NEW_ST)
             ELSE
               BEGIN
                 MACH^.REMEM_ST := NEW_ST;
                 RE_USE := TRUE;
                 MACH^.NUM_SMPL_EXT := NOT_COMP;
               END
           ELSE
             BEGIN
               MACH^.DEVELOPED := TRUE;
               RULE2 (MACH, IO_PTR, CUR_LEVEL+1);
               RULE1 (MACH, IO_PTR, CUR_LEVEL+1);
             END;
           NEXT_LEVEL (IO_PTR);
         END
       ELSE
         BEGIN
           IF (LIST_HEAD[CUR_LEVEL] = NIL) THEN
             CUR_LEVEL := CUR_LEVEL - 1;
           IO_PTR := CUR_LEVEL + 1;
         END;
     END;
END; (* DEPTH_FIRST *)
```

```
(* Main program *)

BEGIN
  MORE_TO_DO := TRUE;
  WHILE (MORE_TO_DO) DO
    BEGIN
      GET_CONF;
      IF (MORE_TO_DO) THEN
        BEGIN
          GET_IO_SEQ;
          PRINT_CONFIG;
          FOR SEARCH_TYPE := SEARCHES[SRCH_STRT] TO
                             SEARCHES[SRCH_END] DO
            BEGIN
              INIT_STATS;
              RUN_TIME_STRT := MAK_TIME (S_TIME);
              CASE SEARCH_TYPE OF
                  BFS   : BREADTH_FIRST;
                  IDFS  : DEPTH_FIRST;
                  DFS   : DEPTH_FIRST;
                  END;
              RUN_TIME_END := MAK_TIME (S_TIME);
              FIND_GOAL (GOAL_MACH);
              PRINT_STATS;
              PRINT_MACHINE (GOAL_MACH);
              CLEANUP (ROOT);
            END;
        END;
    END;
END. (* MAIN *)
```

BIBLIOGRAPHY

Bagchi, A. and Mahanti, A. "Search Algorithms Under Different Kinds of Heuristics - A Comparative Study." Journal of The Association of Computing Machinery 30 (January 1983): 1-21.

Barr, Avron and Feigenbaum, A., eds. Handbook of Artificial Intelligence, Vol. 1. Los Altos: William Kaufmann, 1981.

Berliner, Hans. "The B^* Tree Search Algorithm: A best First Proof Procedure." Artificial Intelligence 12 (1979): 23-40.

Biermann, A. W.; Baum, R. I. and Petry, F. E. "Speeding Up the Synthesis of Programs from Traces." IEEE Transactions on Software Engineering SE-2 (September 1976): 141-153.

Cohen, Paul R. and Feigenbaum, Edward A., eds. Handbook of Artificial Intelligence. Vol. 3. Los Altos: William Kaufmann, 1982.

Gill, Arthur. "State-Identification Experiments in Finite Automata." Information and Control 4 (1961): 132-154.

Gold, E. Mark. "Language Identification in the Limit." Information and Control 10 (1967): 447-474.

Hopcroft, John E. and Ullman, Jeffrey. Formal Languages and Their Relation to Automata. Reading: Addison Wesley, 1969.

Horowitz, Ellis and Sahni, Sartaj. Fundamentals of Computer Algorithms. Potomac: Computer Science Press, 1978.

Knuth, Donald E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Reading: Addison Wesley, 1973.

Kountanis, Dionysios I. "A State Automaton as a Model of a Learning System Which Can Learn Any Machine from a Class of Machines." Ph. D. dissertation, University of Pennsylvania, 1977.

Kountanis, Dionysios I. and Mitchell, Brian T. "Affect of Representation on The Complexity of a Learning System." Computer Science Report #2. Western Michigan University, Kalamazoo, Michigan. 1979

Lenat, Douglas B. "The Nature of Heuristics." Artificial Intelligence 19 (1982): 189-249.

Manna, Zohar and Waldinger, Richard. "Knowledge and Reasoning in Program Synthesis." Artificial Intelligence 6 (Summer 1975): 175-208.

Moore, Edward F. "Gedanken-Experiments on Sequential Machines." Automata Studies, (1956): 129-153.

Nau, Dana S. "Decision Quality As a Function of Search Depth on Game Trees." Journal of The Association of Computing Machinery 30 (October 1983): 687-708.

Nilsson, Nils J. Principles of Artificial Intelligence. Palo Alto: Tioga Publishing Company, 1980.

Nilsson, Nils J. Problem Solving Methods in Artificial Intelligence. New York: McGraw Hill, Inc., 1971.

Reibling, Lyle A. "A Learning Strategy for a Class of Probablistic Automata." Masters thesis, Western Michigan University, 1983.

Trakhtenbrot, B. A. and Barzdin, Ya. M. Finite Automata behavior and synthesis. Translated from Russian by Louvish, D. New York: American Elsevier Publishing Company, 1973.

Tucker, Alan. Applied Combinatorics. New York: John Wiley and Sons, 1980.

Wilkins, David E. "Using Knowledge to Control Tree Searching." Artificial Intelligence 18 (January 1982): 1-51.

Winston, Patrick Henry. Artificial Intelligence. Reading: Addison Wesley, 1979.