Computational Intelligence, Volume 19, Number 2, 2003

INDUCTIVE INFERENCE BY USING INFORMATION COMPRESSION

Ben Choi

Department of Computer Science, Louisiana Tech University

Inductive inference is of central importance to all scientific inquiries. Automating the process of inductive inference is the major concern of machine learning researchers. This article proposes inductive inference techniques to address three inductive problems: (1) how to automatically construct a general description, a model, or a theory to describe a sequence of observations or experimental data, (2) how to modify an existing model to account for new observations, and (3) how to handle the situation where the new observations are not consistent with the existing models. The techniques proposed in this article implement the inductive principle called the minimum descriptive length principle and relate to Kolmogorov complexity and Occam's razor. They employ finite state machines as models to describe sequences of observations and measure the descriptive complexity by measuring the number of states. They can be used to draw inference from sequences of observations where one observation may depend on previous observations. Thus, they can be applied to time series prediction problems and to one-to-one mapping problems. They are implemented to form an automated inductive machine.

Key words: finite state machine, inductive inference, Kolmogorov complexity, learning mechanism, minimum descriptive length, Occam's razor.

1. INTRODUCTION

Inductive inference is of central importance to all scientific inquiries. Automating the process of inductive inference is the major concern of machine learning researchers (Angluin and Smith 1983). This article proposes inductive inference techniques to address three inductive problems: (1) how to automatically construct a general description, a model, or a theory to describe a sequence of observations or experimental data, (2) how to modify an existing model to account for new observations, and (3) how to handle the situation when the new observations are not consistent with existing models. Before providing detailed constructive procedures to solve these problems, their overview and their background will first be described.

(1) *How to automatically construct a general description or a model to describe a sequence of observations or experimental data:* The techniques proposed in this article implement the inductive principle called the minimum descriptive length principle (Rissanen 1978, 1989, 1999) and relate to minimum message length (Wallace and Boulton 1968; Wallace and Georgeff 1983; Wallace and Freeman 1987), Kolmogorov complexity (Li and Vitányi 1997), and Occam's razor (Blumer et al. 1987; Domingos 1998). The idea of the inductive principle was described by Solomonoff (1964). It was formulated independently by Kolmogorov (1965) and Chaitin (1966), and is also called algorithmic information theory. Wallace and Dowe (1999), Vitányi and Li (1996), and Baxter and Oliver (1994) described the relations between minimum descriptive length, minimum message length, and Kolmogorov complexity. Vapnik (1999) provided a formal proof to justify the principle for classification problems. Although much work remains to be done on effective constructive procedures under the principle (Cherkassky and Mulier 1998), it has been applied for machine learning (Yamanishi 1992; Zemel 1993; Pfahringer 1995; Rissanen and Yu 1996), model selection (Hansen and Yu 1998), coding decision trees and graphs (Wallace and Patrick 1993; Oliveira and Sangiovanni-Vincentelli 1995), and even for classification of protein structure (Edgoose, Allison, and Dowe 1998).

Address correspondence to the author at Computer Science Department, College of Engineering and Science, Louisiana Tech University, Ruston, LA 71272; e-mail: pro@BenChoi.org

The basic idea of this principle is to measure descriptive complexity by measuring the length of the descriptions that are used to describe a sequence of observations. Consider tossing a coin 200 times (Chaitin 1975). One description is to list the outcome of each trial. Using T to represent tails and H to represent heads, the list will consist of a string of 200 H's and T's. Suppose for some reason (such as, the coin is biased, has heads on both sides, or it is just pure luck) that the outcome of each of the 200 trials comes out heads. Then, this unique information can be encoded by (or compressed to form) more compact descriptions. One description could be "two hundred H's." Another description could be "all H's." Comparing these descriptions, the last description uses the minimum number of characters and is the simplest description. It also makes a generalization that can be used to predict future outcomes. After seeing 200 consecutive H's, it could be concluded that all future outcomes would be H's. Then, Occam's razor provides guidance to make the choice among the descriptions. In essence it specifies that the simplest description is the best choice.

Choosing the simplest description was formulated into selecting the minimum program length (Kolmogorov complexity), selecting the minimum message length, or selecting the minimum descriptive length. For a sequence of observations or data encoded with a string *S*, Kolmogorov complexity selects a program, having minimum code length, which can be executed by a Turing machine and produces the string *S* as output (Wallace and Dowe 1999). Minimum message length consists of the minimum length of a two-part message; one part encodes a selected model and the other encodes the information of *S* that has not yet been implied by the model. Minimum descriptive length, like minimum message length, can consist of two parts but can also consist of only one part (Baxter and Oliver 1994). A two-part minimum descriptive length consists of the minimum length of the encoding of a class of selected models and the encoding of the information of *S* that has not yet been implied by the class of models. A one-part minimum descriptive length consists of the minimum length of the encoding of the information of *S* that has not yet been implied by a class of models that is implicitly assumed but not explicitly encoded in the description.

The techniques proposed in this article use, in principle, a special two-part message where the second part is null. The information of *S* is encoded in the specifications of a finite state machine that serves as a model. The second part is not needed because the model implies all the information of *S*. The techniques differ from minimum descriptive length principle in two aspects: (1) Instead of describing the compressed information using bit string, they use finite state machines. (2) Instead of minimizing the number of bits in the string, they minimize the number of states in the machines. In these two aspects, they also differ from other related techniques such as those of Rissanen (1989) and Quinlan and Rivest (1989) for inferring decision trees, Gains (1976), Biermann and Feldman (1972), Porat and Feldman (1988), Rivest and Schapire (1989), and Rouvellou and Hart (1995) for inferring finite automata, and Angluin (1987) for inferring regular sets. Our techniques emphasize inferring a usually nondeterministic finite state machine from initial observations and then adaptively modify the finite state machine to account for additional observations as they become available. In this aspect, they differ from other related works that merely concern themselves with constructing a deterministic finite automaton (Biermann and Feldman 1972; Lang 1999; Oliveira, Marques, and Joao 2001) or a probabilistic deterministic finite automaton (Wallace and Georgeff 1983; Raman and Patrick 1997; Patrick, Raman, and Andreae 1998) from samples of its behaviors.

Although the idea of information compression is usually viewed as a process for reducing storage space or communication data, it was recently viewed as a process for computing in general (Wolff 1993, 1996). It was discussed in relation to neural network learning (Schmidhuber 1992) and probably approximately correct (PAC) learning (Takimoto and Maruoka

1993; Floyd and Warmuth 1995). It was applied to natural language processing (Grzymala-Busse and Than 1993) and to learn sequential tasks (Choi 1998, 2002).

The techniques proposed in this article can be used to draw inference from sequences of observations where one observation may depend on previous observations. Thus, the techniques can be applied to time series prediction problems and to one-to-one mapping problems where one observation does not depend on any previous observations. Other works for time series prediction can be found in Laird and Saul (1994), Mier (2000), and others.

To solve the first inductive problem, the techniques proposed in this article are outlined as follows. First, an algorithm is proposed to transform a sequence of observations into a finite state machine. The finite state machine can exactly reproduce the sequence of observations. Then, our proposed algorithms and existing algorithms found in automata theory are used to compress the finite state machine by minimizing the number of states. In terms of data compression, this compression is lossless, i.e., the minimized machine can still reproduce the sequence of observations. The process of this compression also makes a generalization (in the same sense as discussed above for describing the outcomes of the coin tossing by "all H's"). Thus, the minimized machine can be used to predict future observations.

(2) *How to modify an existing model to account for new observations:* The first inductive problem discussed above is concerned with how to automatically create a new general description, a new model, or a new theory. Now, the second inductive problem is concerned with how to modify, evolve, or adapt the existing model or theory to account for new sequences of observations or new experimental data. In general there are two ways to modify a model: to expand the scope or to restrict the scope of the model.

To solve this inductive problem, this article proposes new algorithms to automatically expand or restrict the scope of the existing model. The algorithms will expand the scope of the model when the model is not general enough to account for new observations. They will restrict the scope of the model when the model is too general, covering too many cases or containing too many possible outcomes, and the new observations provide information to specify certain cases or certain possible outcomes. In terms of modeling using finite state machines, as proposed in this article, possible outcomes or possible cases are represented by a number of possible next states, while expanding the scope of the model is represented by adding new input symbols (as detailed in Section 4).

One of the risks of modifying an existing model or theory is that the resulting model might no longer account for the previous observations. The techniques proposed in this article insure this will not happen.

(3) *How to handle the situation when the new observations are not consistent with existing models:* The second inductive problem discussed above is concerned with the cases when the existing model or theory can be modified to account for new sequences of observations. Now, the third inductive problem is concerned with the cases when the existing model cannot be modified, for some reason, to account for the new sequences of observations. One of the reasons might be that the new observations contradict, or are not consistent with, the existing model. Another reason might be that there is simply no known method to modify the existing model to accommodate the new observations. In terms of scientific inquiries, these are the cases that demand a new theory, which is also the approach that is implemented by this investigation.

To solve this inductive problem, this article proposes to use multiple models, each of which associates with a confidence factor. When an existing model can account for new observations, its confidence factor is raised. When an existing model can be modified to account for new observations, its confidence factor is also raised. When no existing model can account for or can be modified to account for new observations, a new model is created and its confidence factor is assigned an initial value.

The models evolve over time as they see more observations. When asked to predict future outcomes, our proposed system will choose to apply the most appropriate model that has the highest confidence factor.

1.1. Organization

This article is organized into seven sections. Section 1 provides an overview of the rationale, the background, and the objectives of this investigation. Section 2 provides formulations of the problems by defining how to represent observations and models. Section 3 describes the solution to problem (1) for creating a new model based on a sequence of observations. Section 4 describes the solution to problem (2) for modifying an existing model to account for new information. Section 5 describes the solution to problem (3) for using multiple models to account for inconsistent information. Section 6 provides experimental results and performance analysis of our automated inductive machine that implemented the proposed techniques. Section 7 discusses conclusions and future research.

2. FORMULATIONS OF THE PROBLEMS

Before automating the process of inductive inference, we formulated the specification of a sequence of observations and the representation of the inferred model, as provided in the following.

2.1. Observations

An observation is specified by a stimulus and a response, an input and an output, or a condition and an action. A sequence of observations is an ordered set of observations. The order is important to capture our dynamic physical world because one observation may depend on previous observations. For our formulation, a sequence of observations is specified by an input/output sequence.

Definition 1. Let I be a nonempty set of *inputs* and O be a nonempty set of *outputs*. An *input sequence* is defined to be $(x_0, x_1, \ldots, x_{m-1}) \in \mathrm{I}^m$. An *output sequence* is defined to be $(y_0, y_1, \ldots, y_{m-1}) \in \mathrm{O}^m$. And, an *input/output sequence* is defined to be $((x_0, y_0), (x_1, y_1), \ldots, (x_{m-1}, y_{m-1})) \in (\mathrm{I} \times \mathrm{O})^m$.

An observation may depend on previous observations. That is, for $0 \leq k < m$, the output y_k may be a function of inputs $x_0, x_1, \ldots, x_k$, i.e., perhaps there exists a function F with $y_k = F(x_0, x_1, \ldots, x_k)$. There are special cases that are less complex than this. One special case is when an observation does not depend on any previous observations, that is, there exists a function f with $y_k = f(x_k)$. Another special case is when all the inputs are equal; in this case only the output sequence is of concern. For example, when repeatedly tossing a coin, we are only concerned with the outcomes $(y_0, y_1, \ldots, y_{m-1})$.

2.2. Inferred Models

This article employs finite state machines as models to describe sequences of observations. Finite state machines are particularly well suited for capturing the information of dynamic systems. In this investigation, they not only serve as compact ways to encode sequences of observations, but also serve as general models that can be used to predict future outcomes.

Definition 2. A *finite state machine* is defined to be a six-tuple (I, O, S, s_0, δ, λ), where I, O, and S are nonempty sets, $s_0 \in \mathrm{S}$, $\delta : \mathrm{I} \times \mathrm{S} \rightarrow 2^{\mathrm{S}}$, and $\lambda : \mathrm{I} \times \mathrm{S} \rightarrow \mathrm{O}$.

Here 2^{S} is the power set of S, $\{\mathrm{A} \mid \mathrm{A} \subseteq \mathrm{S}\}$. The sets I, O, and S are interpreted as the set of inputs, the set of outputs, and the set of states, respectively. The state s_0, called the *initial state,* is defined as the starting state of the machine. The functions δ and λ are called the *state transition function* and the *output function,* respectively.

The functions δ and λ are usually defined by a table (called a *state table*) having each input in I as a column and each state in S as a row. If all entries in the table are defined then this finite state machine is called *completely specified,* otherwise it is called *incompletely specified*. If an entry is not defined, it is an unspecified entry that consists of an *unspecified next state* and an *unspecified output*. In general, this definition of a finite state machine is a *nondeterministic* finite state machine because, for $i \in \mathrm{I}$ and $s \in \mathrm{S}$, the *next states* $\delta(i, s)$, is defined as the power set of S. One special case is when $\delta(i, s) = \mathrm{S}$ that specifies the next states can be any state in S including the *present state* s. Another case is when, for a state $t \in \mathrm{S}$, $\delta(i, s) = \{t\}$, which specifies one single next state. In what follows, we will write $\delta(i, s) = \{t\}$ as $\delta(i, s) = t$ for short. If all entries in the state table are specified and there is only one single next state in each entry, then the finite state machine is a *deterministic* finite state machine.

A deterministic finite state machine will produce one output sequence when given an input sequence. A nondeterministic finite state machine may produce many output sequences when given an input sequence. The finite state machine starts at its initial state s_0, when given an input $i \in \mathrm{I}$, it produces an output $o = \lambda(i, s_0)$; it goes to next states $\mathrm{T} = \delta(i, s_0)$. For each state $t \in \mathrm{T}$, it now starts at t; when given an input $j \in \mathrm{I}$, it produces an output $p = \lambda(j, t)$; it goes to next states $\mathrm{U} = \delta(j, t)$, and so on.

There are two special characters "?" and "*" which may or may not be elements of O. If they appear in O, they are interpreted as follows. The character "?" indicates a "don't care" character and the character "*" indicates an unspecified output.

3. CREATING A NEW MODEL

Based on the formulations provided in the previous section, a sequence of observations is specified by an input/output sequence and an inferred model is presented by a finite state machine. Thus, the process to create a new model involves generating a finite state machine based on a given input/output sequence. This article proposes to measure the descriptive complexity by measuring the number of states in the finite state machine. Based on Occam's razor, the simplest model is the best choice. Thus, the process of creating a new model consists of a process to minimize the complexity of the finite state machine, which minimizes the number of states.

The minimized machine serves two essential purposes for the inductive system. First, it records in a compact way a given finite input/output sequence. When it receives the same input sequence, it will reproduce the same output sequence. Secondly, it represents infinite sequences that are the results of generalization. Thus, when it receives a new input sequence, it will be able to produce an output sequence that is based on the result of generalization.

Process for creating a new model
1. Building an incompletely specified finite state machine from a given sequence
2. Partitioning the states into pairs of compatible states
3. Finding maximal compatibility classes from the compatible states
4. Finding a set of minimal consistent compatibility classes
5. Building a minimal finite state machine consistent with the given sequence

FIGURE 1. Steps for creating a new model.

The process for creating a new model is outlined in Figure 1. The first step transforms an input/output sequence to an incompletely specified finite state machine, in which some next states and some outputs were not specified. Then, the states in the machine are partitioned into pairs of compatible states. Some pairs of compatible states are grouped to from maximal compatibility classes. All the maximal compatibility classes are then used in a search procedure to find a minimal set of compatibility classes that are consistent with the given sequence. Then, each compatibility class in the minimal set of compatibility classes is considered as a single state for the final minimal finite state machine. Details of each step are provided in the following.

3.1. Building an Incompletely Specified Finite State Machine from a Given Sequence

This section describes a process for converting an input/output sequence to an incompletely specified nondeterministic finite state machine. Given an input/output sequence $((x_0, x_1), (x_1, x_2), \ldots, (x_{m-1}, y_{m-1}))$, the following procedure will construct a finite state machine (I, O, S, s_0, δ, λ) that will produce the output sequence $(y_0, y_1, \ldots, y_{m-1})$ when given the input sequence $(x_0, x_1, \ldots, x_{m-1})$:

(1) Let I be the set of distinct entries in the input sequence $(x_0, x_1, \ldots, x_{m-1})$.
(2) Let O be the set of distinct entries in the output sequence $(y_0, y_1, \ldots, y_{m-1})$ and also let O contain the special symbol "*" indicating unspecified output.
(3) Let S be the set of distinct states $\{0, 1, \ldots, m-1\}$ and s_0 be state 0.
(4) The function δ is defined as, $\forall i \in \mathrm{I}, \forall s \in \mathrm{S}$,

$$\delta(i, s) := \begin{cases} s+1, & \text{if } x_s = i \quad \text{and} \quad 0 \le s < m-1 \\ \mathrm{S}, & \text{otherwise} \end{cases} .$$

(5) The output function λ is defined as, $\forall i \in \mathrm{I}, \forall s \in \mathrm{S}$,

$$\lambda(i, s) := \begin{cases} y_s, & \text{if } x_s = i \quad \text{and} \quad 0 \le s < m \\ *, & \text{otherwise} \end{cases} .$$

Starting at the initial state s_0, state 0, when given the input x_0, based on the definition of $\lambda(x_0, 0)$, the finite state machine will produce y_0. Based on the definition of $\delta(x_0, 0)$, it will go to state 1. At state 1, when given x_1, it will produce y_1 and go to state 2, and so on to state $m-1$. Thus, the machine will produce the output sequence when the input sequence is given. The worst-case complexity for constructing the machine is $\theta(m)$. With the constructive process, we have just proved the following theorem.

Theorem 1. Given an input/output sequence $((x_0, x_1), (x_1, x_2), \ldots, (x_{m-1}, \mathrm{y}_{m-1}))$, there exists a finite state machine (I, O, S, s_0, δ, λ) that will produce the output sequence $(y_0, y_1, \ldots, y_{m-1})$ when given the input sequence $(x_0, x_1, \ldots, x_{m-1})$.

The following subsections will show how to reduce the descriptive complexity by special processes for minimizing the number of states in the incompletely specified finite state machine. General techniques for minimizing the number of states in an incompletely specified finite state machine have been well studied (see, for example, Grasselliand and Luccio 1965; Kohavi 1978; Kam et al. 1994; Pena and Oliveira 1998). However, general techniques require exponential worst-case complexity $O(x^m)$, where $x \geq 2$. Subsections 3.2–3.5 present new and special algorithms that make use of special properties of the machine constructed in this section. Our special algorithms require worst-case complexity $\theta(m^2)$ by finding a near-minimal finite state machine, which usually has a minimal number of states or else it has a number of states close to the minimum.

3.2. Partitioning the States into Pairs of Compatibility States

To generalize and to compress the incompletely specified finite state machine defined in the last subsection, we will first compare any two states to determine whether they are compatible. To enable the comparison, we will first define when two outputs are functionally equivalent as follows.

Definition 3. An output c is *functionally equivalent* to an output d, denoted as $c \approx d$, if $c = {}^*$ or $d = {}^*$ or $c = d$.

If the output c is represented by a string of characters c_p for $p = 0, 1, \ldots, h$, and the output d is represented by a string of characters d_q for $q = 0, 1, \ldots, k$, then we say that $c = d$ if $h = k$ and, for all p, $c_p =$ “?” or $d_p =$ “?” or $c_p = d_p$.

Definition 4. A state s is *compatible* to a state t, denoted as $s \sim t$, if, for all $i \in$ I, each output $\lambda(i, s)$ is functionally equivalent to $\lambda(i, t)$, and each next state in $\delta(i, s)$ is compatible to each next state in $\delta(i, t)$. That is,

$$s \sim t \text{ if } \forall i \in \mathrm{I} \quad \lambda(i, s) \approx \lambda(i, t) \wedge \delta(i, s) \sim \delta(i, t).$$

This definition of compatibility is iterative. The next states $\delta(i, s) \sim \delta(i, t)$ if $\delta(i, s) = \delta(i, t)$ or $\delta(i, s) = \mathrm{S}$ or $\delta(i, t) = \mathrm{S}$; otherwise, if $\lambda(i, s) \approx \lambda(i, t)$, then $s \sim t$ is said to *require* $\delta(i, s) \sim \delta(i, t)$, denoted by

$$s \sim t \sim> \delta(i, s) \sim \delta(i, t).$$

Based on the machine constructed in Section 3.1, the iterative steps will end at state $m - 1$ if the process does not end in other states. One of the special properties for the machine is that $\delta(i, s) = s + 1$ or $\delta(i, s) = \mathrm{S}$, and $\delta(i, t) = t + 1$ or $\delta(i, t) = \mathrm{S}$. As a result, for the machine, $s \sim t$ can only *require* $s + 1 \sim t + 1$, i.e.,

$$s \sim t \sim> s + 1 \sim t + 1.$$

Based on this property a special procedure for checking compatibility is developed in this investigation. The procedure consists of two passes through all pairs of states. For the first pass, the procedure checks each pair of states, in order of $(0, 1), (0, 2), \ldots, (0, m - 1)$; $(1, 2), (1, 3), \ldots, (1, m - 1); \ldots, (m - 2, m - 1)$, where there are a total of m states. The first pass determines whether the pair of states is compatible, is not compatible, or requires another pair of states. For the second pass, the procedure checks each pair of states in the order of $(m-2, m-1), (m-3, m-2), \ldots, (0, 1); (m-3, m-1), (m-4, m-2), \ldots, (0, 2); \ldots, (1, m-1), (0, m-2)$. If states (p, q) are not compatible but the states $(p-1, q-1)$ require (p, q), then the states $(p-1, q-1)$ are now also marked as not compatible; otherwise no change is made. The worst-case complexity of this algorithm is $\theta(m^2)$ because there are $\frac{1}{2}m(m-1)$ pairs of states.

3.3. Finding Maximal Compatibility Classes from the Compatible States

In the previous subsection we considered whether two states are compatible while in this subsection we consider whether three or more states are compatible. We will define a set called a maximal compatibility class (Kohavi 1978) and derive a procedure to find all the maximal compatibility classes.

Definition 5. A subset of set S having all its states pairwise compatible is called a *compatibility class*. A *maximal compatibility class* is a compatibility class that is not a subset of any other compatibility class.

The procedure developed in this investigation for deriving maximal compatibility classes uses a binary search tree. The root of the tree consists of the set S of all states in the machine. The procedure starts with the set S. It checks any two states in S. If two states j and k are not compatible, then it splits S into two sets: one set contains all states except j, and the other contains all states except k. Next, it checks the two sets, one at a time using the same method. The procedure continues. If a set is checked but not split, then it is a compatibility class. If a set has already been checked, then it is not checked again. All the compatibility classes are then collected and checked for containment: a maximal compatibility class is one that is not a subset of any other compatibility classes.

The worst-case complexity of the algorithm is $\theta(m^2)$, where m is the number of steps in the input/output sequence. Two special cases are when all states in S are compatible and when all states in S are not compatible. In the former, it requires $\frac{1}{2}m(m-1)$ comparisons to find that all states are pairwise compatible. In the latter, it requires $\frac{1}{2}m(m-1)$ splits to separate all states. Between these two special cases, less than $\frac{1}{2}m(m-1)$ splits are required but each split may require more than one comparison. It amounts to $\theta(m^2)$.

3.4. Finding a Set of Minimal Consistent Compatibility Classes

To derive a near-minimal finite state machine, we will use a concept called a minimal closed cover (Kohavi 1978). We will show in this subsection that we can find a minimal closed cover in the set of all maximal compatibility classes that were derived in the previous subsection. In general, the set of all maximal compatibility classes and some of its subsets called prime compatibility classes are needed to find a minimal number of states (Grasselliand and Luccio 1965). Because we only use the set of all maximal compatibility classes, our result is near minimal, that is, equal to or close to a minimum.

Definition 6. A compatibility class C *requires* a compatibility class P, written C $\sim>$ P, if there exist two states s and t in C and there exists an input i, such that $\delta(i, s)$ and $\delta(i, t)$ are elements of P. P is said to be a *required* class for C. That is,

$$\mathrm{C} \sim> \mathrm{P} \quad \text{if} \quad \exists s, t \in C, \exists i \in \mathrm{I} : \delta(i, s), \delta(i, t) \in \mathrm{P}.$$

Definition 7. A set C of compatibility classes is *closed* if for each compatibility class c in C such that c requires p, p is in C. That is,

$$C \text{ is } \textit{closed} \text{ if } p \in C \text{ whenever } c \sim> p, \forall c \in C.$$

If C is not closed, then it requires some other class.

Definition 8. A compatibility class C *covers* a state s if s is in C. A set of compatibility classes covers a finite state machine if all states of the machine are covered. A set of compatibility classes is called a *closed cover* if it is closed and it covers the machine. A *minimal closed cover* of a machine is a closed cover that has the minimal number of compatibility classes.

To find a minimal closed cover, a breadth-first search procedure is used. The procedure described here is based on that proposed by Meisel (1967) but which is modified here to become a breadth-first search procedure. As the search progresses, a tree is generated. The search begins at the root node. The root node generates its child nodes, and checks for closed and cover conditions. Then, each of its child nodes generates its child nodes, and checks for closed and cover conditions. The process continues until both closed and cover conditions are satisfied.

For our search tree, the root node is called level 0; its child nodes are called level 1; its grandchild nodes are called level 2, and so on. Each node contains three items: a maximal compatibility class, a list called *pending list* (that contains required classes that remain to be checked), and a list called missing list (that contains states that are not yet covered). Let the root node contain an empty compatibility class, an empty pending list, and a missing list that contains all states.

There is a unique path from the root node to any other node. A path from the root node to a node in level 2, for example, connects the root node, a node in level 1, and a node in level 2. There is a compatibility class in each of the connected nodes. The collection of all those compatibility classes, except the one in the root node, is called a *path set*. The path set for a node in level 2, for example, contains 2 compatibility classes. In general, the path set for a node in level h contains h compatibility classes.

For any node, the pending list determines whether its path set is closed. If the pending list is empty, then the path set is closed. If it is not empty, then it contains required classes that have not yet been closed. Also, for any node the missing list determines whether its path set covers all states. If the missing list is empty, then the path set covers all states. If it is not empty, it contains states that have not yet been covered. If a node has both an empty pending list and an empty missing list, then its path set is a closed cover.

How a node generates its child nodes depends on whether its pending list is empty. If its pending list is not empty, then it takes a required class out of the pending list, and generates a node for each compatibility class that contains the required class. If its pending list is empty, then it finds from its missing list a state that is covered by a minimal number of compatibility classes, and it generates a node for each compatibility class that covers the state.

A new child node contains a different compatibility class than its parent-node. It inherits the pending list and the missing list, but updates them if needed. If its compatibility class has required classes, then it adds those required classes to its pending list. However, it removes from its missing list states that are covered by its compatibility class.

Starting from the root node, the procedure generates new child nodes and checks for closed and cover conditions in a breadth-first search manner. The procedure terminates when we find a node that satisfies both the conditions. The path set of the node contains a minimal closed cover set of compatibility classes that is consistent with the given sequence.

The worst-case complexity of the algorithm is $O(m^2)$, where m is the number of steps in the input/output sequence. Two special cases are when all states in S are compatible and when all states in S are not compatible. In the former, the search tree only has two nodes, the root node and one node at level 1. In the latter, the search tree has $m + 1$ nodes, the root node and one node at each level up to level m. Between these two cases, there are n levels for $1 < n < m$. As n is closer to 1, the number of states that are compatible increases. A state is more likely to be covered by several compatibility classes. Thus, the number of nodes in a level will increase. As n is closer to m, the number of states that are compatible decreases. A state is more likely to be covered by fewer compatibility classes. Thus, the number of nodes in a level will decrease. These amount to a worst-case complexity bounded by $O(m^2)$.

3.5. Building a Minimal Finite State Machine Consistent with the Given Sequence

This section describes a procedure for deriving a near-minimal finite state machine $M' = (I', O', S', s'_0, \delta', \lambda')$ from a minimal closed cover $\mu = \{C_0, C_1, \ldots, C_{n-1}\}$ and the original finite state machine $M = (I, O, S, s_0, \delta, \lambda)$.

The process is described as follows. First let $I' = I$, and let $S' = \{0, 1, \ldots, n-1\}$, where a state k is assigned to one compatibility class $C_k \in \mu$. The initial state s'_0, in this case state 0, is assigned to a compatibility class C_0 and $s_0 \in C_0$. Without loss of generality, we let a compatibility class that contains s_0 be C_0.

Here we define the function δ' for the minimal machine. For an $i' \in I'$ and a state $s' \in S'$, the next state $\delta'(i', s')$ is specified if there exists a state $t \in C_{s'}$ in the original finite state machine and, for $i = i'$, such that $\delta(i, t)$ is specified in the original finite state machine. If $\delta'(i', s')$ is specified, it is equal to state x such that $\delta(i', t)$ is in C_x. If it is not specified, it is equal to the set S'. That is,

$$\delta'(i', s') = \begin{cases} x, & \text{if } \exists\, t \in C_{s'}\colon \delta(i', t) \in C_x \\ S', & \text{otherwise} \end{cases} .$$

The function λ' is defined as follows:

$$\lambda'(i', s') = \begin{cases} \lambda(i, t)\,, & \text{if } i = i', \exists\, t \in C_{s'} \text{ and } \lambda(i, t) \neq * \\ *, & \text{otherwise} \end{cases} .$$

And finally, let O' be all the distinct outputs specified by λ'. The worst-case complexity of the algorithm is $\theta(m)$, where m is the number steps in the input/output sequence, because $m \geq n$. This completes the process for creating a minimal model to describe a sequence of observations.

4. UPDATING AN EXISTING MODEL

The previous section described the proposed processes for creating a new minimal model. This section will describe how to modify the model to incorporate the new information provided by a new sequence of observations. It also describes how to check whether a model is consistent with the new sequence of observations. The process outlined in Figure 2 is developed in this investigation for both updating a model and checking consistency. The process incorporates new information into a model by adding new input symbols, defining previously unspecified outputs, and removing possible next states. Checking consistency insures that the updated model will remain consistent with previous observations.

```
Updating a model and checking consistency

  Compare the given input sequence to a finite state machine, starting at state 0
  IF a given input is not in the input set I
  THEN add the input to the input set I.

  In responding to the given input sequence, the finite state machine generates many output sequences.
  FOR each output sequence generated by the finite state machine
      Compare it with the given output sequence and define previously unspecified outputs
      IF they are functionally equivalent
      THEN keep the generated output sequence and keep the newly defined outputs
      ELSE    remove the generated output sequence by removing a possible next state that is
              responsible for generating the output sequence, and undo the defined outputs.

  IF no update is made to the finite state machine
  THEN     the finite state machine is consistent with the input/output sequence
  ELSE IF the updated finite state machine contains an entry that has all its possible next states removed
  THEN     both the updated finite state machine and the original finite state machine
           are not consistent with the input/output sequence
  ELSE     the updated finite state machine is consistent with the input/output sequence.
```

FIGURE 2. Updating a model and checking consistency.

4.1. Checking Consistency

To determine what to do with the new sequence of observations, our inductive system must have a method to compare the new observations with the existing models. The new observations come to the inductive system in the form of input/output sequences, while the system stores its models in the form of finite state machines. Thus, we propose a method for comparing an input/output sequence with a finite state machine.

The proposed method for checking consistency is outlined in Figure 2. The following provides explanations and definitions. The stored finite state machines can be deterministic or nondeterministic. A deterministic finite state machine produces one output sequence in responding to one input sequence. A nondeterministic finite state machine, on the other hand, can produce many output sequences in responding to one input sequence.

Definition 9. A finite state machine is *consistent* with an input/output sequence if in responding to the input sequence, the finite state machine produces at least one output sequence that is functionally equivalent to the given output sequence, and produces no output sequence that is not functionally equivalent to the given output sequence.

Definition 10. An output sequence $(p_0, p_1, \ldots, p_{m-1})$ is *functionally equivalent* to another output sequence $(q_0, q_1, \ldots, q_{h-1})$ if $m = h$ and for $k = 0, 1, \ldots, m-1$, each output p_k is functionally equivalent to output q_k.

The definition for two outputs to be functionally equivalent was provided in Definition 3. Based on the above definitions, our automated inductive machine will try to modify a model to make the model consistent with the new observations, while insuring that the updated model will remain consistent with previous observations.

State \ Input	0	1
0	1, 1?	1, 00
1	0, 10	0, 01

Next State, Output

(a) Original state table.

Given Inputs	Sequence (i)	Sequence (ii)	Required Outputs
0	0, 1?	=	11
n	1, 01	=	01
1	0, 00	1, 01	?1

Current state, Output

(b) Sequences produced after adding new input column.

State \ Input	0	1	*n*
0	1, 11	1, 00	S, *
1	0, 10	0, 01	1, 01

Next state, Output

(c) Resulting state table.

FIGURE 3. Example for updating a model.

4.2. Adding New Input Symbols

To allow our model to expand its scope, to generalize, or to cover more instances, an input symbol is added to the input set I of the finite state machine when the input is in the input sequence but not in I. The process for adding a new input is illustrated by an example.

The following example also illustrates the process outlined in Figure 2 for updating a model and checking consistency. As an example an input/output sequence (0/11, *n*/01, 1/?1) is compared to the machine defined by the state table shown in Figure 3(a). The result of each step of the comparisons is shown in Figure 3(b) while the resulting modified state table is shown in Figure 3(c). The steps of the comparison are as follows:

(1) Starting at state 0, the finite state machine receives an input 0. It produces an output 1? (shown in sequence (i) in Figure 3(b)) that is compared with the required output 11. To make the outputs functionally equivalent, the produced output 1?, by the specified "don't care" character "?" to 1, is now specified to 11. Because the next state is state 1, the machine goes to state 1.

(2) In state 1, the machine receives an input *n* that is not in the set I. A new input column is created, which is labeled *n* (shown in Figure 3(c)). All the entries in the new column are initialized to unspecified next state and unspecified output, shown as S, *. The entry at the current state that is state 1 is then set to S, 01 in which the next state is the set S while the value of the output is that of the input/output pair *n*/01 because at current state the machine is required to produce an output 01. The next state S is treated as

next states 0 and 1. Thus, there are two *possible next states* for the current state. The next state 0 is checked first while next state 1 is saved in a stack to be checked later. Now the machine goes to state 0.

(3a) In state 0, the machine receives an input 1. It then produces an output 00 which is compared to required output ?1. The first characters of the two outputs can be functionally equivalent if the don't care character is set to 0, but the second characters cannot be functionally equivalent. Thus, the newly produced output sequence (shown in sequence (i) in Figure 3(b)) is not functionally equivalent to the required output sequence. So the required output is restored back to ?1, and the newly produced output sequence needs to be removed. To do so, the possible next state 0, which is the result of step (2), is removed.

(3b) To generate sequence (ii) (see Figure 3(b)), the process performs backtracking and finds that in step (2) there is another possible next state that needed to be checked. It retrieves from the stack the possible next state, that is, state 1. The machine now goes to state 1 in which it receives input 1 and then produces output 01 which is compared to the required output ?1. These two outputs are functionally equivalent and the required output is now specified to 01. This completes sequence (ii). In the figure, an equal sign "=" in an entry indicates that the value of that entry is equal to that on the left. To complete sequence (ii) the process does not need to regenerate those values that are equal to that already generated in the sequence (i). Because sequence (ii) is functionally equivalent to the required output sequence, the possible next state 1 is kept. This completes the search process and the final result is shown in Figure 3(c).

The resulting machine has a new input symbol as shown in Figure 3(c). The "don't care" output character "?" in the original finite state machine is now specified to 1.

4.3. Defining Previously Unspecified Outputs

When a new sequence of observations contains new or more specific information, this information will be incorporated into the existing model. One way to do so is by defining previously unspecified outputs in the finite state machine. The proposed method is explained and illustrated by an example.

To incorporate new information into the existing model requires comparing the new information with information contained in the models. In our case this is partially accomplished by comparing the given output sequence with the output sequence generated by the finite state machine. Comparing two output sequences in which any output may contain "don't care" characters or may be an unspecified output requires a method for comparing two outputs and a method for keeping track of which don't care character or which unspecified output has been defined.

An example shown in Figure 4 will illustrate the proposed method. There are two output sequences, say sequence O (o1, o2, o3, o4) and sequence P (p1, p2, p3, p4). To complicate the matter, sequence O consists of only three distinct output variables namely a, b, and c; those values will change as the result of the comparisons. Sequence O = (a, c, b, c). The initial values are $a = 11?$, $b = ?10$, and $c = *$, where "*" denotes an unspecified output and "?" denotes a don't care character. Sequence P consists of only two distinct output variables, namely x and y. The initial values are $x = ?10$ and $y = 1?0$. Sequence P = (x, y, x, y).

1. Start the comparison with the first pair, o1 to p1. Compare o1 ($a = 11?$) to p1 ($x = ?10$); the result is that this pair is functionally equivalent and that the values of a and x are both changed. Now they are $a = 110$ and $x = 110$ as well. The original don't care character in a is now specified to 0 while that of x is specified to 1.

```
Sequence O (o1, o2, o3, o4)
       O = (a,  c,  b,  c )
Initial value: a = 11?  b = ?10  and  c = *

Sequence P (p1, p2, p3, p4)
       P = (x,  y,  x,  y )
Initial value: x = ?10  and  y = 1?0

Compare o1: a=11? to p1: x=?10  =>  a=110  x=110
Compare o2: c= *  to p2: y=1?0  =>  c=1?0  y=1?0
Compare o3: b=?10 to p3: x=110  =>  b=110  x=110
Compare o4: c=1?0 to p4: y=1?0  =>  c=1?0  y=1?0
```

FIGURE 4. Example for comparing two output sequences.

2. Next compare o2 ($c = {}^*$) to p2 ($y = 1?0$); the result is that they are functionally equivalent and the value of c is changed while that of y is unchanged. Now they are $c = 1?0$ and $y = 1?0$; the original unspecified output c is now specified to the specific value.
3. Next compare o3 ($b = ?10$) to p3 ($x = 110$), where the value of x is the result of the first comparison. The result of the comparison is that they are functionally equivalent and the value of b is changed. Now the values are $b = 110$ and $x = 110$.
4. Finally compare o4 ($c = 1?0$) to p4 ($y = 1?0$); the result is that they are functionally equivalent and both the values of c and y are unchanged.
5. Because all the corresponding pairs of sequence O and P are functionally equivalent, the final result is that these two sequences are functionally equivalent.

In this investigation, it is necessary to compare a given output sequence to many output sequences generated by a finite state machine. For the purpose of creating a modified finite state machine that is consistent with the given input/output sequence, a generated output sequence will be removed if it is not functionally equivalent to the given output sequence. The specified don't care characters or previously unspecified outputs, associated with the removed output sequence, will be restored. As described in the last example (Figure 4) some of the don't care characters may be specified to specific values and some unspecified outputs may be specified. The results of the specification are kept if the two sequences turn out to be functionally equivalent.

4.4. Removing Possible Next States

Another way to incorporate new information into an existing model is by removing possible next states in the finite state machine. The new information helps the inductive system to distinguish possible outcomes, to limit the choices of many outcomes, or to eliminate nondeterministic states. This will restrict the scope of the model. The model becomes more specific and better defined.

Removing the next state corresponding to an output sequence will remove the output sequence. To produce a modified finite state machine that can be consistent with a given input/output sequence, the output sequence generated by a finite state machine will be

State \ Input	x	y	z
0	0,1, L	1, L	0, M
1	0, K	0,2, L	1, N
2	1, M	0, K	0, M

Possible next states, ..., Output

(a) State table.

Inputs	(i)	(ii)	(iii)	Outputs
x	0, L	0, L	0, L	L
y	0, L	1, L	1, L	L
z	1, N	0, M	2, M	N

(b) Sequences produced.

FIGURE 5. Choosing possible states to be removed.

removed if it is not functionally equivalent to the given output sequence. The proposed process is illustrated by an example.

As an example the input/output sequence $(x/L, y/L, z/N)$ is compared to the finite state machine defined by the state table shown in Figure 5(a). In responding to the input sequence (x, y, z), the machine produces three state sequences and output sequences as shown in Figure 5(b). Compare the required output sequence (L, L, N) to the three produced output sequences resulting in that output sequence: (i) is functionally equivalent while output sequences (ii) and (iii) are not functionally equivalent. Both sequence (ii) and sequence (iii) must be removed to produce a modified machine that is consistent with the given input/output sequence.

There are two possible next states that are responsible for producing sequence (ii): the possible next state 1 of present state 0 and input x, and the possible next state 0 of present state 1 and input y. Removing any one of the two possible next states will remove the possibility to produce sequence (ii). Similarly there are two possible next states that are responsible for producing sequence (iii): the possible next state 1 of present state 0 and input x, and the possible next state 2 of present state 1 and input y. Removing any one of the two possible next states will remove sequence (iii).

To remove both sequence (ii) and sequence (iii), there are two choices: (1) remove the possible next state 1 of present state 0 and input x (that possible next state is responsible for producing both the sequences); or (2) remove both possible next states 0 and 2 of present state 1 and input y. If choice (1) is chosen, then the resulting modified machine will be consistent with the given input/output sequence and the modified machine will remain consistent with those input/output sequences that are consistent with the original machine. If choice (2) is chosen, then the resulting modified machine will not be consistent with those input/output sequences that are consistent with the original machine. The reason for the latter is that the modified machine will now consist of an undefined next state (for present state 1 and input y), in which all possible next states have been removed.

A general rule for choosing which possible next states to remove is: (1) remove first those next states that are responsible for as few output sequences as possible; if the result of (1) is that all the possible next states of a particular present state and input have been removed, then (2) restore all those possible next states and then remove those next states that are responsible for the whole set of output sequences.

The reason for the general rule is that during the process for creating an updated model, it is necessary to ensure that (1) the updated model is consistent with the new input/output sequence and (2) the updated model remains consistent with those original input/output sequences that are consistent with the original model. To ensure these, the updated machine must have remaining at least one next state for every pair of present states and inputs.

5. USING MULTIPLE MODELS

The previous subsection described how to update an existing model based on the information provided by new sequences of observations. This subsection will describe methods that will be used when none of the existing models can be modified to account for the new observations or when the new observations contradict all existing models. The proposed method is to use multiple models. Each model is associated with a confidence factor called weight.

5.1. Assigning a Confidence Factor to Each Model

The proposed method is outlined as shown in Figure 6. A number called weight is assigned to each model. The weight for a newly created model equals 1 (as outlined in Figure 6). Whenever the inductive system receives a new input/output sequence, it will check for consistency of the new sequence against all the existing models (step (1) in Figure 6). If a model is consistent with the new sequence, then its weight is increased by 1; otherwise its weight is unchanged. (2) If a model can be modified so that the modified model is consistent with the new sequence and that remains consistent with the old sequences, then the newly modified model is added and its weight is 1 plus the weight of the original model. (3) Otherwise, a completely new model is constructed based on the information provided by the new input/output sequence, and its weight is 1.

This process uses other processes described earlier: (1) It detects any inconsistency by using the process for checking consistency (described in Section 4.1). (2) It incorporates new useful information into the existing models by using the processes for updating models (outlined in Figure 2). (3) It records inconsistent information by using the process for creating a new model (described in Section 4.1).

If inconsistent information occurs less frequently than useful information, then the weights of the models that result from inconsistent information will be lower than the weights of the good models. The model having the highest weight is the most favorable model.

```
Assigning weight to each model
    The weight for a newly created model equals 1
    FOR each model
        IF (1) an input/output sequence *is consistent with* the model
        THEN increase the weight of the model by 1
        ELSE IF the input/output sequence *is consistent with* an updated model
        THEN    (2) store the updated internal model;
                its weight equals 1 plus that of the original model.
    IF the input/output sequence *is not consistent with* any existing models
    THEN    (3) create a new model;
            its weight equals 1.
```

FIGURE 6. Process for assigning weight to each model.

5.2. Choosing the Best Model to Predict Future Outcomes

This subsection describes our proposed method for choosing a model, among all existing models, to predict the future outcomes. When our inductive system is asked to predict the future outcomes, to produce an output sequence based on an input sequence, it must choose the best model to apply. We proposed two criteria for making the best choice: (1) measuring similarity between input symbols in the input sequence and that in the models, and (2) using confidence factor.

Using the two criteria, the proposed method is described as follows. Assuming the input sequence consists of inputs x_k, for $k = 0, 1, \ldots, m - 1$. Let a set J be the set of distinct inputs in the input sequence. In our system, a model is represented by a finite state machine. Assume there are M_t finite state machines, for $t = 0, 1, \ldots, h - 1$. Each M_t consists of a set of input symbols I_t, and associates with a confidence factor W_t (weight as described in previous subsection). Let $|\mathrm{X}|$ denote the number of elements in the set X.

- Step 1: We will choose a set of machines M_t such that the set I_t has maximal similarity to the set J, that is, max $|\mathrm{I}_t \cap \mathrm{J}|$, for $t = 0, 1, \ldots, h - 1$.
- Step 2: From the chosen set of machines in step 1, we then choose those machines such that $|\mathrm{I}_t| \geq |\mathrm{J}|$. If there exists a machine such that this condition is satisfied, then this new chosen set is used in the next step; otherwise the chosen set from step 1 is used in the next step.
- Step 3: from the resulting set of machines in step 2 we choose those machines that have the maximal confidence factor W_t.

After going through the three-step process described above, there might be more than one machine in the chosen set. Thus, there might be more than one set of predictions, i.e., more than one output sequences generated based on a given input sequence. In the case when there is only one chosen machine, there might still be more than one set of predictions. However, if the only chosen machine is deterministic, then there will be only one set of predictions.

6. EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS

The constructive procedures proposed in this article have been implemented to form an *automated inductive machine*. The automated inductive machine written in C++ implements the ideas proposed in the article. The system can create finite state machines based on information provided by any input/output sequences. It can modify the existing finite state machines to account for new observations. In addition, it can use multiple finite state machines to capture inconsistent observations.

The automated inductive machine was tested by many inductive problems including, for example, a parity checker, an up-down counter, a sequence detector, a task involving many states, and a simple code breaker. The experimental results of the examples are provided in Tables 1 and 2, and the performance analysis is provided below.

6.1. Performance Analysis

We have an $\theta(m^2)$ algorithm for constructing a new model from m observations. As detailed in Section 3, each of the five steps for creating a new model requires a worst-case complexity less than or equal to $\theta(m^2)$. The new model is encoded using a near-minimal finite state machine that is usually less compact and represents a less general

TABLE 1. Summary of Simulation Results for Creating New Models.

Name of application examples	Number of steps in the input/output sequence (m)	Number of states in the generated state table (n)
A parity checker	11	2
An up-down counter	10	4
A sequence detector	5	3
	8	3
A simple code breaker	9	1
	8	1
A task involving many states	30	30
	100	100

description compared to a minimal finite state machine. However, the inductive process remains sound.

The process for updating a model and checking consistency (described in Section 4) has a worst-case complexity $O(n^m)$, where n is the number of states in the model and m is the number of observations in a given input/output sequence. The complexity depends on the degree of nondeterminism of the model. There are two extreme cases: one when the model is fully deterministic and the other when the model is generated from a input/output sequence that cannot be compressed. In the former, the process requires order of m steps. In the latter, because most of the next states are considered to be the state set S having n states, the process requires order of n^m steps.

To reduce the search space for updating a model and checking consistency, two methods are used to eliminate branches in the search tree. The first method uses a depth-first search process. The depth-first search process is stopped as soon as an inconsistency is found. Thus, not all the output sequences are required to be generated or compared. The second method is incorporated in the process for removing possible next states (described in Section 4). Removing a possible next state corresponding to an output sequence will remove the possibility for generating the output sequence. Because a next state is usually responsible

TABLE 2. Summary of Simulation Results for Updating Models and Checking Consistency.

Name of application examples	Number of states in an existing model for updating (n)	Number of steps in the input/output sequence (m)	Number of nodes used by the process for updating the model
A sequence detector	3	10	22
	3	8	4
	3	8	13
	3	8	4
A simple code breaker	1	9	9
	1	8	6
	1	8	5
A task involving many states	30	7	7470
	100	7	94690

for generating a subtree, removing one next state will prevent many other corresponding output sequences from being generated. Our test results showed that these two methods work well; in some cases only 9.5×10^{-8} percent of the n^m steps are required.

The proposed automated inductive machine has several advantages over other systems: (1) The system is often capable of inferring a reasonable finite state machine from just one sequence of examples. On the contrary, recurrent neural networks (Jordan 1986; Giles, Kuhn, and Williams 1994; Tsoi 1998; Teuscher 2001) that are the types of neural networks required for learning a sequence of steps, often require thousands of sequences (Bengio, Frasconi, and Simard 1993; Bengio, Simard, and Frasconi 1994). Other related artificial intelligence systems have their limitations, such as producing only accepted or rejected outputs (Angluin 1981, 1987; Porat and Feldman 1988), restricting allowed inputs or outputs (Biermann and Feldman 1972; Barto and Anandan 1985; Lanctot and Oommen 1992; Rouvellou and Hart 1995), or passing computational burden to their teachers (Angluin 1981, 1987; Ibarra and Jiang 1988; Marron 1988; Rivest and Schapire 1989). (2) The system can handle sequential tasks involving long-term dependencies for which recurrent neural networks have been shown to be inadequate (Bengio et al. 1993, 1994). (3) The system represents learned information by finite state machines that are easy to implement in digital hardware. (4) The system has a wide range of applications including but not limited to (a) sequence detection, prediction, and production, (b) an intelligent macro system that can learn rather than simply record sequences of steps performed by a computer user, and (c) a design automation system for designing finite state machines or sequential circuits (Choi 2002).

7. CONCLUSIONS AND FUTURE RESEARCH

This article described our proposed constructive procedures to solve three inductive problems: (1) how to automatically construct a model to describe a sequence of observations, (2) how to modify an existing model to account for new observations, and (3) how to handle situations where the new observations are not consistent with existing models. Our proposed techniques encode a sequence of observations by using input/output sequences, represent models by using finite state machines, and minimize descriptive complexity by using the minimal descriptive length principle.

The proposed ideas and constructive procedures were implemented to form an automated inductive machine. Test results indicated that our machine is often able to generate a deterministic finite state machine from just one input/output sequence. Other times the system will first generate a nondeterministic finite state machine, but as more input/output sequences are provided to the system, the model will evolve to become a deterministic finite state machine. This mimics the scientific progress from a hypothesis having many possible alternatives to a well-defined theory having deterministic results. When no existing models can account for or can be modified to account for new observations, our system will create a new model to capture the new information. This mimics the cases in scientific inquiries when a new theory is called for. Each model is assigned a confidence factor. As more information is provided, models are updated and confidence factors are changed. When asked to predict the future outcomes, the system will employ our proposed method to choose and to use the best model.

Our proposed system has potential for many applications. One future research direction will be to apply the system to applications such as data mining and bioinformatics for DNA sequence detection and prediction.

ACKNOWLEDGMENTS

The author thanks Professor Dick Greechie and anonymous reviewers for proofreading and suggesting many changes to make this article more readable.

REFERENCES

ANGLUIN, D. 1981. A note on the number of queries needed to identify regular languages. Information and Control, **51:**76–87.

ANGLUIN, D. 1987. Learning regular sets from queries and counter examples. Information and Computation, **75**(2):87–106.

ANGLUIN, D., and C. H. SMITH. 1983. Inductive inference: Theory and methods. Computing Surveys, **15**(3):237–269.

BARTO, A. C., and P. ANANDAN. 1985. Pattern-recognizing stochastic learning automata. IEEE Transactions on Systems, Man, and Cybernetics, **15**(3):360–375.

BAXTER, R. A., and J. J. OLIVER. 1994. MDL and MML: Similarities and Differences. Technical Report TR 207, Department of Computer Science, Monash University, Clayton, Victoria, Australia.

BENGIO, Y., P. FRASCONI, and P. SIMARD. 1993. The problem of learning long-term dependencies in recurrent networks. *In* IEEE International Conference on Neural Networks, San Francisco, pp. 1183–1195.

BENGIO, Y., P. SIMARD, and P. FRASCONI. 1994. Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, **5**(2):157–166.

BIERMANN, A. W., and J. A. FELDMAN. 1972. On the synthesis of finite-state machines from samples of their behavior. IEEE Transactions on Computers, **21**(6):592–597.

BLUMER, A., A. EHRENFEUCHT, D. HAUSSLER, and M. K. WARMUTH. 1987. Occam's razor. Information Processing Letters, **24:**377–380.

CHAITIN, G. J. 1966. On the length of programs for computing finite binary sequences. Journal of the ACM, **13:**547–569.

CHAITIN, G. J. 1975. Randomness and mathematical proof. Scientific American, **232:**47–52.

CHERKASSKY, V., and F. MULIER. 1998. Learning from Data: Concepts, Theory, and Methods. John Wiley and Sons, New York.

CHOI, B. 1998. Automata for learning sequential tasks. New Generation Computing, **16:**23–54.

CHOI, B. 2002. Applying learning by example for digital design automation. Applied Intelligence, **16**(3):205–221.

DOMINGOS, P. 1998. Occam's Two Razors: The Sharp and the Blunt. *In* Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. AAAI Press, New York, pp. 37–43.

EDGOOSE, T., L. ALLISON, and D. L. DOWE. 1998. An MML classification of protein structure that knows about angles and sequence. *In* Pacific Symposium on Biocomputing '98. World Scientific, Singapore, pp. 585–596.

FLOYD, S., and M. WARMUTH. 1995. Sample compression, learnability, and the Vapnik-Chervonenkis dimension. Machine Learning, **21:**269–304.

GAINS, B. R. 1976. Behaviour-structure transformations under uncertainty. International Journal of Man-Machine Studies, **8:**337–365.

GILES, C. L., G. M. KUHN, and R. J. WILLIAMS, editors. 1994. IEEE Transactions on Neural Networks, Special Issue on Dynamic Recurrent Neural Networks, **5**(2).

GRASSELLIAND, A., and F. LUCCIO. 1965. A method for minimizing the number of internal states in incompletely specified sequential networks. IEEE Transactions on Electronic Computers, **14**(3):350–359.

GRZYMALA-BUSSE, J. W., and S. THAN. 1993. Data compression in machine learning applied to natural language. Behavior Research Methods, Instruments, and Computers, **25**(2):318–321.

HANSEN, M., and B. YU. 1998. Model selection and the principle of minimum description length. Technical Memorandum, Bell Labs, Murray Hill, NJ.

IBARRA, D. H., and T. JIANG. 1988. Learning regular languages from counterexamples. *In* Proceedings of the 1988 Workshop on Computational Learning Theory, MIT, pp. 371–385.

JORDAN, M. I. 1986. Attractor dynamics and parallelism in connectionist sequential machine. *In* Proceedings of the Eighth Annual Conference of the Cognitive Science Society, pp. 521–545.

KAM, T., T. VILLA, R. BRAYTON, and A. SANGIOVANNI-VINCENTELLI. 1994. A fully implicit algorithm for exact state minimization. *In* Proceedings of the thirtyfirst Annual Conference on Design Automation.

KOHAVI, Z. 1978. Switching and Finite Automata Theory. McGraw-Hill, New York.

KOLMOGOROV, A. N. 1965. Three approaches to the quantitative definitions of information. Problems of Information Transmission, **1:**1–7.

LAIRD, P., and R. SAUL. 1994. Discrete sequence prediction and its applications. Machine Learning, **15:**43–68.

LANCTOT K., and B. J. OOMMEN. 1992. Discretized estimator learning automata. IEEE Transactions on Systems, Man, and Cybernetics. **22**(6):1473–1483.

LANG, K. 1999. Faster algorithms for finding minimal consistent DFAs. *http://citeseer.nj.nec.com/353128.html.*

LI, M., and P. VITÁNYI. 1997. An Introduction to Kolmogorov Complexity and Its Applications. Springer-Verlag, New York.

MARRON, A. 1988. Learning Pattern Languages form a Single Initial Example and from Queries. *In* Proceedings of the 1988 Workshop on Computational Learning Theory, MIT, pp. 345–352.

MEISEL, W. S. 1967. A note on internal state minimization in incompletely specified sequential networks. IEEE Transactions on Electronic Computers, 508–509.

MIER, R. 2000. Nonparametric time series prediction through adaptive model selection. Machine Learning, **39:**5–34.

OLIVEIRA, A. L., and A. SANGIOVANNI-VINCENTELLI. 1995. Using the minimum description length principle to infer reduced ordered decision graphs. Machine Learning, **12:**1–32.

OLIVEIRA, A. L., S. MARQUES, and P. JOAO. 2001. Efficient algorithms for the inference of minimum size DFAs. Machine Learning, **44**(1):93–119.

PATRICK, J., A. RAMAN, and P. ANDREAE. 1998. A bean search algorithm for PFSA inference. Pattern Analysis and Applications, **1:**121–129.

PENA, J. M., and A. L. OLIVEIRA. 1998. A new algorithm for the reduction of incompletely specified finite state machines. *In* Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design, San Jose, CA, pp. 482–489.

PFAHRINGER, B. 1995. Practical Uses of the Minimum Description Length Principle in Inductive Learning. Ph.D. Thesis, Institut fur Med.Kybernetik u. AI, Technische Universitat Wien.

PORAT S., and J. A. FELDMAN. 1988. Learning automata from ordered examples. *In* Proceedings of the First Annual Workshop on Computational Learning Theory. Morgan Kaufmann, San Mateo, CA, pp. 386–396.

QUINLAN, J., and R. RIVEST. 1989. Inferring decision trees using the minimum description length principle. Information and Computation, **80:**227–248.

RAMAN, A., and J. PATRICK. 1997. The sk-strings method for inferring PFSA. *In* Proceedings of the Fourteenth International Conference on Machine Learning—ICML'97, Nashville, TN.

RISSANEN, J. 1978. Modeling by shortest data description. Automatica, **14:**465–471.

RISSANEN, J. 1989. Stochastic Complexity and Statistical Inquiry. World Scientific, Singapore.

RISSANEN, J. 1999. Hypothesis selection and testing by the MDL principle. Computer Journal, **42**(4):260–269.

RISSANEN, J., and B. YU. 1996. MDL learning. *In* Learning and Geometry: Computational Approaches, Progress in Computer Science and Applied Logic. *Edited by* D. Kueker and C. Smith. Birkhäuser, Boston, pp. 3–19.

RIVEST, R. L., and R. E. SCHAPIRE. 1989. Inference of finite automata using homing sequences. *In* Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, pp. 411–420.

ROUVELLOU, I., and G. W. HART. 1995. Inference of a probabilistic finite state machine from its output. IEEE Transactions on Systems, Man, and Cybernetics, **25**(3):424–437.

SCHMIDHUBER, J. 1992. Learning complex, extended sequences using the principle of history compression. Neural Computation, **4:**234–242.

SOLOMONOFF, R. J. 1964. A formal theory of inductive inference. Information and Control, **7**(1-22):224–254.

TAKIMOTO, E., and A. MARUOKA. 1993. Relationships between learning and information compression based on PAC learning model. Systems and Computers in Japan, **24**(8):47–58.

TEUSCHER, C. 2001. Turing's Connectionism: An Investigation of Neural Network Architectures. Springer, New York.

TSOI, A. C. 1998. Recurrent neural network architectures: An overview. Lecture Notes in Computer Science, vol. 1387. Springer-Verlag, New York.

VAPNIK, V. N. 1999. The Nature of Statistical Learning Theory. Springer-Verlag, New York.

VITÁNYI, P., and MING LI. 1996. Minimum Description Length Induction, Bayesianism, and Kolmogorov Complexity. *http://citeseer.nj.nec.com/89617.html.*

WALLACE, C. S., and D. M. BOULTON. 1968. An information measure for classification. Computing Journal, **11:**185–195.

WALLACE, C. S., and D. L. DOWE. 1999. Minimum message length and Kolmogorov complexity. Computer Journal, **42**(4):270–283.

WALLACE, C. S., and P. R. FREEMAN. 1987. Estimation and inference by compact coding. Journal of the Royal Statistical Society, Series B, **49:**240–265.

WALLACE, C. S., and M. P. GEORGEFF. 1983. A general objective for inductive inference. Technical Report No. 32, Department of Computer Science, Monash University.

WALLACE, C. S., and J. D. PATRICK. 1993. Coding decision trees. Machine Learning, **11:**7–22.

WOLFF, J. G. 1993. Computing, cognition, and information compression. AI Communications, **6**(2):107–127.

WOLFF, J. G. 1996. Information compression by multiple alignment, unification, and search as a general theory of computing. School of Electrical Engineering and Computer Science Report, February.

YAMANISHI, K. 1992. A learning criterion for stochastic rules. Machine Learning, **9:**165–203.

ZEMEL, R. S. 1993. A minimum description length framework for unsupervised learning. Ph.D. Thesis, University of Toronto.